SNF: Serverless Network Functions

Arjun Singhvi
University of Wisconsin-Madison
asinghvi@cs.wisc.edu

Junaid Khalid[*]
Google
junaidkhalid@google.com

Aditya Akella
University of Wisconsin-Madison
akella@cs.wisc.edu

Sujata Banerjee
VMware Research
sujata@banerjee.net

ABSTRACT

Our work addresses how a cloud provider can offer Network Functions (NF) as a Service, or NFaaS, using the emerging serverless computing paradigm. Serverless computing has the right NFaaS building blocks - usage-based billing, event-driven programming model and elastic scaling. But we identify two core limitations of existing serverless platforms that undermine support for NFaaS - coupling of the billing and work assignment granularities, and state sharing via an external store. Our framework, SNF, overcomes these limitations via two ideas. SNF allocates work at the granularity of *flowlets* observed in network traffic, whereas billing and programming occur at a finer level. SNF embellishes serverless platforms with *ephemeral local state* that lasts for the flowlet duration and supports high performance state operations. We demonstrate that our SNF prototype matches utilization closely with demand and reduces tail packet processing latency substantially compared to alternatives.

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**.

KEYWORDS

Serverless Computing, Network Functions, Flowlets

ACM Reference Format:
Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. 2020. SNF: Serverless Network Functions. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421295

[*]Work done while at University of Wisconsin-Madison

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '20, October 19–21, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421295

1 INTRODUCTION

Recent advancements in *serverless computing* have made great strides in reducing the burden of managing, provisioning, and scaling the server/VM infrastructure underlying distributed applications. Users simply have to focus on application logic. They write and upload programs called "functions" that the platform dynamically scales based on user specified event triggers. The functions run in stateless and short-lived computing instances/containers, and the user is billed for exactly the compute cycles used for a function. While this paves the way for cost-effective application and service deployments [23], serverless computing today is narrowly focused on stateless, short-lived, batch mode, and/or embarrassingly parallelizable workloads. It leaves behind stateful streaming workloads, which form our focus.

We consider the specific example of network functions (NFs) that perform stateful operations on packet streams in enterprises and telecommunication infrastructures. These workloads are central to the Network Functions Virtualization (NFV) transformation in telco service provider networks and still face significant challenges in simultaneously achieving low cost (to both the user and the infrastructure provider), and scalable performance, while supporting programming ease. In many ways, serverless computing provides a key set of building blocks to address these issues impacting NFV, but important gaps remain (§2).

We address the challenges of supporting NF workloads atop serverless computing by adding in the missing abstractions and mechanisms to support stateful computation effectively, while preserving the unique benefits of the serverless paradigm, such as simplified management, event-driven compute triggering, usage-based pricing, and autoscaling.

NFs are a resource intensive workload with fine grained latency and throughput performance needs. We examine limitations imposed by two naive realizations of support for such workloads over today's serverless platforms—invoking a function per packet, or invoking a function per flow. We make two observations, and argue, correspondingly, for two design improvements. First, we observe that, in today's platforms, "events" (e.g., each incoming request) are used for determining both the granularity of work allocation as well as the granularity against which functions are programmed and billing is performed. This coupling imposes a hard trade-off between resource use efficiency, performance, and billing granularity for stateful applications. We argue for *breaking this coupling*, and allowing work allocation to happen at a different granularity than that at which a function and billing operates. Second, we observe that given state externalization in today's platforms (externalization is key to keeping functions stateless, enabling rapid scale out) the only way state can be shared across events and functions is by using a remote storage service. State sharing is crucial to stateful applications and, for NFs, state externalization substantially worsens packet processing latency. Thus, we advocate for *ephemeral statefulness*, where state is *logically decoupled* from computation and *physically* bound to function instances in an *ephemeral* fashion just for the duration of the computation corresponding to a single unit of work allocation.

We leverage these two ideas in designing SNF, a new serverless platform that allows cloud providers to offer NFs-as-a-service (NFaaS) wherein users can outsource NFs to enjoy the benefits of the cloud [30, 36]. Users of SNF (e.g., NFV operators) can program different NFs as functions. For a given NF, SNF transparently distributes the packet processing work in an incoming traffic stream across an elastically scalable set of compute units; each unit has a function corresponding to the given NF deployed in it. SNF maintains high utilization of compute units, and ensures that minimal number of units are used, which are attractive to the cloud provider. SNF also ensures that a given NF instance's packet processing throughput is high, tail latency is low, and that billing only captures the work done in processing traffic, all of which are appealing to users (NFV operators).

To achieve these goals, SNF relies on the following ideas:

- We note from prior work [40] that the packet processing workloads in a flow can be naturally granularized into *flowlets*. In SNF, we use flowlets as the units of our workload assignment, whereas NF programs that run in function instances operate on a packet at a time, preserving the current NF programming abstraction and ensuring that billing is only for work done.
- We store NF-internal state in the local memory of a compute unit where a given flowlet is being processed. We develop protocols that use inter-flowlet gaps to proactively replicate such ephemeral state to a new compute unit where the processing of a subsequent flowlet in the same flow is to occur, while avoiding inconsistent updates.
- Our work assignment algorithm keeps all active compute units at the maximum possible utilization using a weighted greedy bin-packing algorithm that maximally packs flowlets into few compute units, while ensuring performance targets are met, and while preferring instances to which state has been proactively replicated.

We implement and evaluate a standalone prototype of SNF on CloudLab [16] using 10Gbps and 100Gbps workloads. We use real traffic traces and five stateful NFs. We find that SNF simultaneously achieves efficiency, performance, and fault tolerance for NF processing. SNF uses compute capacity closely matching demand. It reduces 75%-ile processing latency by **2.9K-19.6Kx** over alternatives that operate at flow granularity. Our proactive state management improves the 99%-ile tail latency of NF processing by **12-15x** over state-of-the-art state management solutions. Additionally, our simple fault tolerance protocol supports fast recovery (**22.8x-183.6x** reduction in comparison to alternatives).

2 MOTIVATION

NFs and NFaaS: NFs examine and modify packets to ensure security, improve performance, and provide other diverse functionalities: examples include network address translators (NAT), intrusion detection systems (IDS), firewalls, load balancers, etc. Many NFs are stateful in nature; e.g., NAT maintains address mappings, and IDS maintains string matches.

Over the last few years, researchers have advocated outsourcing NFs to the cloud [30, 36] to realize NFaaS to enable NF users to enjoy cloud benefits such as leveraging the scale, elasticity, and availability of the cloud, pay-as-you-go billing, and the built-in management and operational expertise available at cloud providers.

However, as of today, no cloud provider offers NFaaS. There exists limited support where an end user can use specific out-of-the-box NFs provided by the cloud provider, such as a load balancer, or a firewall [6, 7]. However, the user does not have the flexibility of running custom NFs. Our work addresses how a cloud provider can offer NFaaS and realize the goal of outsourcing NFs.

Ideally, an NFaaS platform should provide an intuitive programming model to write custom NF logic and deliver low packet processing latency while automatically scaling up/down to meet the demand, and charging users only for the work performed, i.e., *usage-based billing*.

2.1 Drawbacks of "Server-full" NFaaS Realizations

It is possible for a provider to use "server-full" compute platforms that use VMs or containers as compute units to realize NFaaS. However, a key issue is that today's native VM- or container-based compute platforms' interface does not allow users to simply supply high-level functions; users are responsible for managing the lifecycle of compute units (e.g., launch the compute unit, install appropriate NF logic and other software dependencies, etc) which imposes significant management burden.

A more fundamental impediment arises in such platforms due to their *physical coupling of compute and state.* By definition, a VM or a container couples a certain amount of local memory with compute, thereby providing the abstraction of a machine/server. As the compute unit executes the processing logic (e.g., NF logic) on the incoming workload (i.e., stream of packets), local memory is read from or written to - in order to access or mutate state (i.e., NF state).

Given this physical compute-state coupling, solutions that adopt these platforms have to resort to coarse grained rules for allocating incoming workload (flow-level allocation in our case) to compute units. Doing so results in a *long-term commitment* between a flow and a compute unit, which can lead to overload, affecting performance, or under utilization, affecting efficiency, depending on the assigned flow's rate (§9). However, the advantage of coupling is that performance, especially at the tail, can be excellent, if no overload occurs (rare in practice), as the state required for processing is available in local memory. Thus, there exists a fundamental trade-off between efficiency and performance in such platforms that physically couple compute and state.

2.2 Serverless Computing

Serverless computing, or function-as-a-service, is a new cloud computing model that offers an event-driven programming model, usage based billing, and automatic compute elasticity. Additionally, serverless computing with fine-grained functions is also beneficial to the cloud provider as it can facilitate higher resource utilization. Thus, on the face of it, serverless computing seems to have the right building blocks to meet the requirements of NFaaS, which a cloud provider could leverage.

However, existing platforms' key design choices leave a substantial gap in leveraging the promise of serverless computing to supporting NFaaS. Before delving into these, we provide a quick primer on serverless computing.

Serverless Background. In FaaS or serverless computing, the user writes a function, uploads it to the serverless platform and registers for an event (e.g., object uploads, incoming HTTP request) to trigger function execution. *The event is the granularity at which the platform does work assignment, and it is also the granularity for programming functions and for billing.* When an event arrives, the platform routes the event to a compute unit that runs the function to process the event. An event may cause setting up of a compute unit from scratch which involves launching the unit and downloading the relevant runtime and the function code from a data store; alternatively, an event may be sent to an already launched and "warmed up" compute unit. Additionally, the platform elastically scales computation up/down based on incoming event rate. The user is charged for the duration that it takes a compute unit to process the event.

Gaps in serverless platforms. Serverless platforms today lie at the other end of the spectrum from "serverfull systems" in that they *physically decouple compute and state* by completely externalizing state and relegating it to a data store (e.g., S3 [3]). Functions are stateless - all state needed to process an event is read from an external store, and any generated state is written back to the store.

The physical decoupling offers the advantage that processing logic can be run wherever there is capacity - leading to simple elasticity logic and enabling arbitrary scale elastic processing and high utilization. However, its interplay with another key design aspect of serverless today – the event granularity couples work assignment, programming, and billing – imposes fundamental trade-offs for NFaaS.

One granularity for running NFs is *per-packet.* Here, the platform would assign work, and NF function execution would be triggered, per packet. This efficiently utilizes compute units, which helps the cloud provider, and also ensures that users are billed exactly for just the compute cycles used. But it causes reordering, with packets sprayed across the compute units. Also, due to the physical decoupling of compute and state, state required for processing must be accessed via an external store for each packet leading to high latencies.

The *per-flow* granularity, where the platform receives an event when a new flow arrives, triggering NF function execution, does not suffer from any packet reordering and state access overheads induced by physical decoupling. However, it places a strict constraint that all packets belonging to a flow must be processed by a single function and defaults to "serverfull"-like physical compute-state coupling. Moreover, the user is charged for periods even when no packets are being processed by the function as it busy-waits for packets pertaining to the still active "event" (flow) to arrive.

Thus, naively running NFs atop serverless platforms leads to a trade-off between performance, efficiency, and billing benefits. These trade-offs are fundamental to the two attributes of today's platforms: (a) the tight coupling between workload assignment, programming, and billing granularities, and (b) the physical decoupling of compute and state.

3 SNF IDEAS AND ARCHITECTURE OVERVIEW

In SNF, we match serverless computing with NFaaS using two key ideas.

Idea 1: Decouple work assignment and programming granularities: In SNF *packet arrivals are treated as events.* This naturally aligns with the way developers implement NFs

- take a packet as input and execute the processing function (process_pkt()).

Work assignment happens at per-flowlet granularity. A flowlet [40] is a burst of packets that is separated from other bursts of packets from the same flow by a sufficient gap called the flowlet timeout. Acting at this granularity provides more opportunities to assign/allocate work. While operating at a flow granularity, multiple flows are assigned to the same compute unit which can lead to under utilization/overload and head of line blocking (HOL): packets from an earlier elephant flow can cause those from mice flows later to wait in buffers at the unit, degrading latency (see §9.1). On the other hand, while operating at the flowlet granularity, large flows are "broken up" into many flowlets that can now be assigned at many units. This mitigates HOL blocking and over-utilization. Also, the smaller size of flowlets than flows enables better packing of work to compute units, and hence is better at avoiding under utilization.

Given the above decoupling and elevation of both packets and flowlets as first class system-level entities, *billing can become very fine grained.* A provider can charge for usage, by accounting for packet processing. Further, by tracking flowlets, the provider can also taking into account control plane actions taken on behalf of a user (e.g., work allocation). In practice, NFaaS providers can pick a combination of the two. In our tech report [39], we evaluate the trade-offs of various possibilities and show that SNF enables billing structures that are competitive compared to using alternative compute substrates (e.g., VMs).

Idea 2: Choose a middleground for compute and state (de)coupling. While efforts have been made to reduce state access overheads by using low latency networking [24, 25] to make physical decoupling viable, they still experience a hold up of packets in queues when a flow is reallocated to a new compute unit (waiting for state to be made available before processing). This worsens tail latency by 15x (§9.2).

Instead, we argue for a middle ground - *logical decoupling of compute and state, with ephemeral physical coupling.* Compute and state are *logically decoupled* from each other: meaning that processing can be launched at any compute unit untethered to state. Given the flowlet granularity of work assignment, we *ephemerally* maintain *physically local* state just for the duration of the flowlet's processing (as all packets in a flowlet are going to be processed at the same compute unit). Further, in case subsequent flowlets of the same flow are assigned to different compute units, we ensure physical ephemeral coupling by leveraging the flowlet inactivity period to opportunistically transfer ephemeral state from the current compute unit to the estimated location, helping us curtail delays and bring down tail latencies due to state unavailability. If the estimation is incorrect, we make state available in a fast, low overhead fashion for processing to resume.

This brings the best of both worlds: good compute utilization benefits of physical decoupling, while coming close to the ideal-case latency performance of physical coupling.

3.1 SNF Overview

The SNF serverless NFaaS platform has two main components – controller and NF runtime – that jointly realize the above two ideas. We outline their main functions.

Controller. The controller consists of three main components: (a) workload granularizer (WG), (b) work assigner (WA) and (c) state manager (SM). When registered events (packets) arrive at the platform, the controller's WG groups them into flowlets. Specifically, given a packet it determines which flowlet it belongs to. If it belongs to an existing flowlet, then it is routed to the appropriate compute unit where the function (NF logic) executes and the event is processed. If not, the WA determines the compute unit to which this new flowlet should be assigned, so as to ensure performance and efficiency. The SM encodes a small amount of metadata to each packet - which compute units to push/pull state to/from in support of ephemeral coupling, and a logical clock to prevent stale state updates.

NF Runtime. This is responsible for state management at each compute unit, and it implements ephemeral physical compute-state coupling by transparently handling state transfers among compute units. State ephemerality and transfers are not visible to the NF developers.

Programming Model. Given that packet arrivals are events in SNF, NF developers operate in a familiar model wherein process_packet() takes a packet as input. Also, the NF runtime exposes simple put(key, value) and get(key) APIs which the developers use to access state.

4 COMPUTE MANAGEMENT

The SNF controller is responsible for the compute management – this primarily includes assigning incoming flowlets to available computational units leveraging the logical decoupling of state. The controller also handles spinning up or decommissioning units based on demand.

We first describe how the incoming workload is organized into flowlets (§4.1) and then the algorithm for determining where to assign flowlet processing (§4.2). The core logic is in Pseudocode 1.

4.1 Workload Granularizer (WG)

When a packet arrives, a new flowlet is detected if one of the two criteria is met (§4.2) - (a) the gap between the current packet and the previous packet of the same flow (identified by the 5-tuple) exceeds the flowlet inactivity timeout; or (b)

Pseudocode 1 SNF Compute Management - Core Logic

```
1:  ▷ Given a packet P, decide which compute unit to send to
2:  procedure EventRouting(Packet P)
3:      $\mathbb{M}$                      ▷ Mapping between flowlet and compute unit
4:      T = ExtractTuple(P)
5:      if FlowletDetector(T, P) then          ▷ Detects new flowlet
6:          ▷ Call WorkloadAssigner as new flowlet is detected
7:          ComputeID = WorkloadAssigner(T, P)
8:      else
9:          ▷ Use the existing assignment as P is within current flowlet
10:         ComputeID = $\mathbb{M}$[T]
11:     end if
12:     return ComputeID
13: end procedure

14: ▷ Given a flowlet F and packet P, detect new flowlet
15: procedure FlowletDetector(Flowlet F, Packet P)
16:     if $currTime$ - F.lastPktTime > $timeout$ then
17:         ▷ New flowlet detected as timeout criteria met
18:         return True
19:     else if P.Size + F.Size > $sizeThreshold$ then
20:         ▷ New flowlet detected as size criteria met
21:         return True
22:     else
23:         return False
24:     end if
25: end procedure

26: ▷ Given a flowlet F, assign a compute unit
27: procedure WorkloadAssigner(Flowlet F)
28:     $\overrightarrow{C}$                        ▷ Candidate compute unit IDs
29:     $\overrightarrow{G}$                        ▷ Sorted active compute unit IDs
30:     for all $g \in \mathbb{G}$ do
31:         if F.DemandEstimator(F) + $g$.Load() < $g$.Capacity() then
32:             $score$ = $g$.Utilization() + $\alpha$ * $g$.StateExist(F)
33:             ▷ Add the computed score to $\overrightarrow{C}$
34:             $\overrightarrow{C}$.add($g$, $score$)
35:         end if
36:     end for
37:     ▷ Pick the compute unit ID which has the maximum score
38:     return max($\overrightarrow{C}$)
39: end procedure
```

the size of the existing flowlet exceeds a threshold (lines 14-25 in Pseudocode 1). Both the timeout and size thresholds are configurable parameters and poor choices will impact the overall efficiency and performance; we perform detailed sensitivity analysis in §9.6. If a new flowlet is not detected, the controller forwards the packet to the compute unit already associated with the packet's flowlet.

The WG is also responsible for estimating the rate of incoming new flowlets, which is needed to make work assignment decisions (line 31 in Pseudocode 1). For our prototype, we estimate the demand of the first flowlet of a flow to be the average load of all flowlets (across all flows) seen in the past. For subsequent flowlets, we find that an estimator that computes an exponentially weighted moving average (EWMA) over the previous flowlet's *measured* rate and the previous estimate is sufficient for effective workload allocation (§9).

4.2 Work Assigner (WA)

If the WG has detected the start of a new flowlet, it sends an assignment request to the WA along with the flowlet load estimate. The WA assigns the flowlet to the appropriate compute unit running NF logic. Prior to assignment, we require compute units' current load (line 31 in Pseudocode 1) which we estimate in the following low-overhead manner - given that each compute unit is handled by a single controller, the controller measures the rate at which packets are drained for a particular compute unit as its load estimate.

Our work assignment algorithm's goal is to avoid overload even in the face of highly dynamic workloads, while keeping utilization high. In assigning work, we make a practical assumption that each NF compute unit is provisioned with adequate CPU and memory resources to support a workload up to BW_{max}bps[1] ; as long as the incoming rate to the compute unit is less than BW_{max}, the NF will be able to provide the requisite performance. We greedily pack incoming flowlets to active compute units so as to maximize their utilization. This is analogous to bin packing: balls are flowlets, and bins reflect the network processing capacity at compute units. The "greedy" aspect arises from the fact that in our approach the compute units are considered in a *deterministic* sorted order of their IDs; the smallest ID unit with room is chosen, which leads to units with lower IDs being packed first (lines 26-39 in Pseudocode 1). This determinism makes the algorithm simple and easy to implement scalably.

Crucially, this determinism makes it easy to take ephemeral state's likely availability into account while making assignment decisions (line 32 in Pseudocode 1 - see §5); this helps us balance utilization against per-packet processing latency.

The controller maintains utilization by adding new compute units if the existing ones are saturated, and inactivating existing ones if they do not receive any packets for a fixed duration. To amortize compute unit start-up times we proactively start them when existing units' load all cross 90%.

Adversarial Flowlets: A key issue in packing arises when traffic demand spikes suddenly on certain, or all, flow subspaces. These "adversarial" flowlets, have actual load that is significantly higher than the estimate provided by our WG; thus, an adversarial flowlet can degrade the performance of other flowlets assigned to the same compute unit by building up queues. If flowlets are detected by just using the inactivity

[1]The per packet CPU and memory resources are specified by the user and can be used by the provider to estimate the resources needed to processed BW_{max}bps.

timeout, the impact of an adversarial flowlet can last arbitrarily. Thus, in SNF, we bound the negative impact of adversarial flowlets by forking a new flowlet from the current one if the current flowlet's size exceeds a threshold. By bounding size in this manner, we ensure that adversarial flowlets are drained quickly and their negative impact is limited. The new flowlet forked after exceeding the size threshold undergoes the process of assignment using an updated load estimate, wherein the moving average accounts for the rate spike observed in the previous adversarial flowlet.

5 STATE MANAGEMENT

A stateful NF's actions on a packet depend on the current state, and for correct and high performance operation, fast access to correct updated state is crucial. NFs may maintain per-flow or cross-flow state. We focus on per-flow state since it is the common case and plan to consider cross-flow state in the future. Also, NFs have configuration state which is often static (e.g., an IDS has string matching rules) and does not vary at packet-scale timelines. In SNF, such state is stored in an external store and is pulled during the compute unit setup phase leading to no visible overheads. Alternatively, configuration state can be packaged along with NF code while it is being uploaded to SNF.

In SNF, NF developers do not have to worry about per-flow state management as the NF runtime makes the state available in an ephemerally physically coupled fashion by transparently moving it across compute instance locations. In what follows, we first describe how ephemeral state is set up and used (§5.1). We then describe how SNF enables ephemeral physical coupling via proactive replication (§5.2). We end with how SNF prevents updates to stale state (§5.3).

5.1 Ephemeral Physically Coupled State

A compute unit in SNF maintains physically local state while processing a flowlet and thus compute and state are physically coupled ephemerally. State is bound to a compute unit from just before the first packet of the flowlet assigned to the unit is processed till the time the last packet is done being processed. Once the flowlet has ended, this state is no longer associated with its compute unit. Ephemeral physical coupling ensures that packets within a flowlet are processed quickly as state access is always local and fast for each arriving packet, but imposes no commitment between flows and compute units.

Ephemeral state is initialized when the first packet of a flowlet arrives at a compute unit as follows: if the flowlet is the first one of the flow, then the state is set to null; otherwise, if the state has already been copied over to the compute unit's memory (as described next), then this state value is used; else, the unit pulls state from the remote unit where the previous flowlet was processed. The controller sends the processing location of the previous flowlet of the same flow as metadata along with the packet belonging to the new flowlet.

5.2 Enabling Ephemeral Physical Coupling

Different flowlets of a flow may be processed by different units depending on the work assignment algorithm. This could lead to a scenario where a flowlet f_1 of a flow F arrives at a different compute unit from the one that the prior flowlet f_0 of the same flow F was processed. Relevant state after f_0's processing is needed at the new compute unit before packet processing can begin. When state is not available, packets are held up in buffers at the compute unit until the state is initialized, affecting latency.

To minimize stalls, SNF replicates ephemeral state *proactively* among compute units by leveraging the *gaps that exist between flowlets* of a flow. This solution works well if the amount of per-flow state maintained by an NF is small enough that it can be transferred during the inter-flowlet gap and not cause stalls. Luckily, prior work [26] has shown that per-flow state size in typical NFs (e.g., PRADS [4], Snort [34]) is under just a few KB for the entirety of a flow's lifetime; even smaller fraction of this may be updated per flowlet. Note, however, that proactive replication is unlikely to help with flowlets that were created from packets exceeding the size threshold (as opposed to the timeout).

One issue is that proactive replication requires compute units to communicate with each other directly. This is a substantial departure from existing serverless platforms, where units (e.g., lambdas [1]) are disallowed from communicating with each other, and all communication can happen only via the external state store. We do not view this constraint as fundamental, and for performance reasons, relax it to enable communication between cooperating compute units.

To ensure the peer-to-peer state transfers are useful and performant, two key questions need to be addressed - (1) when should a compute unit proactively initiate state transfer? and (2) where should it transfer state to?

(1) When to transfer? It is difficult to accurately predict when a flowlet will end. Replicating state whenever there is a small period of inactivity for a flow may lead to unnecessarily doing proactive state transfers if the flowlet does not end and more packets arrive. Waiting till the end of the inactivity timeout would default to reactively pulling the state, which has performance implications.

In SNF, we proactively replicate state once the period of inactivity exceeds *half* of the flowlet inactivity timeout to balance minimizing wait times against making unnecessary state transfers. In case this flowlet does not end, processing can carry on without interruption at the primary unit which still holds a copy of the latest state. However, this can lead

Pseudocode 2 SNF State Management

```
1: ▷ Given a replication factor R, decide where replication should occur

2: procedure DeterministicReplicator(ReplicationFactor R)
3:     $\overrightarrow{C}$                                    ▷ Candidate unit IDs
4:     $\overrightarrow{G}$                                    ▷ Sorted active unit IDs
5:     ▷ Return the first R active compute units
6:     return $\overrightarrow{G}[1:R]$
7: end procedure

8: ▷ Pick R units via a weighted (inversely to IDs) randomized distribution
9: procedure WeightedRandomizedReplicator(ReplicationFactor R)
10:    $\overrightarrow{G}$                                    ▷ Sorted active unit IDs
11:    $\overrightarrow{W}$                                    ▷ Weights assigned inversely to IDs
12:    $\overrightarrow{C}$                                    ▷ Candidate unit IDs
13:    while len($\overrightarrow{C}$) < R do
14:        ▷ Pick a compute unit from $\overrightarrow{G}$ where units are weighed by $\overrightarrow{W}$
15:        replicationSite = WeightedRandomizer($\overrightarrow{G}$, $\overrightarrow{W}$)
16:        $\overrightarrow{C}$.add(replicationSite)
17:    end while
18:    return $\overrightarrow{C}[1:R]$
19: end procedure
```

to inconsistent state updates, which we discuss and address in §5.3. In case a new flowlet arrives at a new compute unit before the proactive transfer begins, we first *reactively pull* relevant state (from the compute unit with state for the immediate preceding flowlet). If the flowlet arrives while the proactive transfer is occurring, we hold off processing.

(2) Where to replicate state? A strawman solution is to broadcast to all other active units but this may cause unnecessary transfers. Instead, in SNF, the controller estimates the top K units where the next flowlet of this flow could likely be assigned to and it tracks this information per flowlet. The reason for picking the top K and not the exact one is because it is not possible to know ahead of time as to where the next flowlet would be assigned, because a unit that is available currently may be saturated by the time the new flowlet arrives (due to flowlets of other flows being assigned in the interim). The question is how to pick the "top K" such that the probability of the compute unit chosen by the WA for the next flowlet already having the necessary state is high.

We could replicate to the K least loaded units, expecting that the WA would assign the next flowlet to them. But, the load can change by the time the next flowlet of this flow starts. Also, implementing a load-aware strategy is complex, as we need up-to-date load information at scale.

Since the WA deterministically processes compute units (lines 2-7 in Pseudocode 2), one simple load-unaware strategy is to pick the least K ID compute units, i.e., units with IDs from 1 to K, to replicate to (the WA would preferentially allocate a new flowlet amongst these). But doing this for every flowlet's replication would render proactive replication ineffective when the least K ID units become overloaded, which is likely especially for a small K. In such cases, future flowlets are assigned outside these K units, and thus they would have to pull state reactively.

SNF uses a simple variant of the above strategy that allows for some error in the estimated location where a future flowlet goes to. We pick the top-K compute units to replicate state to, with probability inversely proportional to the units' IDs (lines 8-19 in Pseudocode 2). Doing so ensures we pick the lower ID units' with higher probability as is done by WA.

The next question is how should the controller make the next assignment decision to account for state availability and maximize the potential benefits of proactive replication? A strawman solution would be for the controller to check if any of the K compute units (which have the required state) could handle this flowlet. If yes, the flowlet is assigned to one of the units in question and processing can proceed without any wait time. If not, then we assign the flowlet to an available compute unit and the state is pulled reactively. Unfortunately, this approach ignores load, and can cause compute units to become fragmented with many compute units poorly utilized.

Instead we extend the work assignment algorithm to make decisions using a *weighted scoring metric* (line 32 in Pseudocode 1) for choosing from the compute units accounting for both utilization and state availability. This metric is $S = utilization + \alpha \times \beta$, where β is 1 if the compute unit has the replicated state; otherwise it is 0. α is a knob between 0 and 1, and balances utilization against proactive benefits: $\alpha = 0$ results in the controller making assignment decisions to improve utilization (and ignoring state) and $\alpha = 1$ biases more in favor units where replicated state is available.

5.3 Preventing Updates on Stale State

While the above techniques minimize packet wait time, we need to ensure that a flowlet does not make updates on stale state that is present at its compute unit. This can occur when the optimistic approach of using half the flowlet timeout as the deadline to proactively replicate state was erroneous in assuming a flowlet would end. Here, the NF runtime would proactively copy state, but a few lingering packets from the original flowlet continue to arrive at the old unit and update state there. State updates due to such packets should be reflected in the state copied over to the new unit before *any* processing begins there.

To prevent a new flowlet from acting on stale per-flow state at the new unit, we introduce the notion of *monotonically increasing logical clocks* for each packet of a flow. These are assigned by the controller. Each packet carries its logical clock as metadata. This prevents flowlets from making updates on stale state in the following manner. The NF runtime tags the state that is proactively replicated with the logical clock of the last packet of this flow that was received by the old unit. When a new flowlet of this flow arrives at the new compute unit, before making updates to the state, the NF runtime verifies if the latest state is available by checking the logical clock of the packet (i.e., first packet of the new flowlet) is one more than the value with which the copied-over state is tagged; if not, state update due to the new packet is stalled, fresh state is pulled reactively, and then the update proceeds.

This technique also works in the rare event of packets arriving out of order. As is done today, if the NF logic requires packets to be processed in order, then the NF developer needs to provide appropriate reordering logic. This typically involves storing the out-of-order packet until the intermediate packet arrives and then processing them in order. Thus, out-of-order packets become a part of the ephemeral NF state (which is tagged appropriately to prevent stale updates as described above) and are processed per the logic defined.

6 FAULT TOLERANCE

In existing serverless platforms, physical decoupling of state and externalization to a persistent store offers intrinsic fault tolerance since state is always available. However, due to logical state decoupling and ephemeral state coupling, SNF as described cannot guarantee fault tolerance of NF state.

In SNF, when the originally assigned (or primary) NF compute unit fails while processing a flowlet, a recovery unit takes over the flowlet's remaining processing. The key fault tolerance property we desire here is that *the per-flow state that the recovered unit is bootstrapped with should have the same value as that with no failures.* We describe how we achieve this property. We assume the standard fail-stop model in which a compute unit can crash at any point and that the system can immediately detect the failure.

Traditional recovery mechanisms [14, 15] do not work in the NF context due to the performance constraints they impose, as well as the presence of non-deterministic state update operations in NF logic (e.g., the use of `random()`) [35]. Our solution adapts prior NF-specific work on fault tolerance [25, 35]. Notably, SNF can adopt a simpler version of such fault tolerance mechanisms, because: (a) SNF handles a single type of NF (or a composite NF), as opposed to a chain of NFs [25], and (b) SNF NFs use a single processing thread as opposed to complex multi-threaded logic [35].

Output Logger (OL): Each NF unit is coupled with a separate *output logger* (OL), which is launched on a different physical machine. Once a packet has been processed by an NF unit, the packet along with its *state delta* is sent to its OL; here, delta is the change to the state value. The OL uses the delta to locally update state it maintains for the NF, and only then forwards the packet externally. Thus, the OL maintains a consistent copy of the state of the primary NF unit. Note that the OL can also be implemented using the same packet-based programming model as NFs in SNF, with the difference being that it only performs state IO on packet arrivals. As an optimization, multiple NF units can share an OL.

NF *or* OL Failing (but not together): When an NF unit fails, a recovery unit can take over by pulling state from the associated OL. If an OL fails, then a recovery OL is initialized by pulling state from its associated NF unit. The controller provides the necessary metadata to the failover NF/OL to pull state from the relevant location.

Race conditions: While this approach generally provides the property we desire, a few race conditions can arise. An NF can crash after processing a packet but before/after transmitting that packet along with the state delta to the OL. Similarly, an OL can crash before or after transmitting the packet. In our tech report [39], we prove that the above mechanism ensures that the recovered state always has the same value as under no failure *even under such scenarios* because some participating entity will always have the correct state.

Controller failure: SNF controller is stateful – it maintains logical clock, and current/previous flowlet-to-compute unit mapping. Thus, upon failure, a recovery controller rebuilds its state by querying all the active NF runtimes where the state exists authoritatively - it obtains the clock values associated with each flow and uses them to determine the maximum clock value associated with each flow before the crash; in a similar manner, the flowlet-compute mapping is also reconstructed.

Correlated failures: To protect against simultaneous NF and OL failures, the state must be replicated at multiple OLs before the packet is released. This increases overhead and we do not consider it here; SNF by default assumes that simultaneous NF+OL failures are rare. But, SNF can handle correlated failures of an NF unit with controller as the recovery controller can build the required state pertaining to the failed NF unit from its OL. Likewise, SNF can handle correlated failures of an OL with controller as it can build the required state pertaining to the failed OL from its NF.

7 CONTROLLER SCALABILITY

The latency overhead introduced by the controller is minimal (§9.7), but as the input workload scales (e.g., 100 Gbps), a single (even powerful) controller can become a bottleneck eventually. Thus, to support large scale workloads, we would need to have multiple controllers. One approach is for controllers to operate on dedicate sets of compute units. But this impacts compute efficiency due to resource fragmentation as controllers do not share units. Alternatively, controllers can share the underlying compute units by using a state store to share compute unit load information. But this imposes the overhead of coordinating over store access for every work allocation decision, i.e., every flowlet.

SNF's compute management design (§4) makes it amenable to adding hierarchy to address scalability. In SNF, we use a global resource manager (RM) that manages a pool of compute units. The various controllers ask for the required processing capacity based on the load seen in the last epoch (say 100ms). The RM allocates the requested capacity, which can be fractional; e.g., the RM can allocate 2.5 units to a controller, which requires spinning up 3 units with the full first two units and half the capacity of the third unit allocated to the requesting controller. When load in the current epoch is nearing requested/allocated capacity (all units at > 90%; §4.2), the controller requests for more capacity to avoid performance degradation. Controllers give back resources once they become inactive (§4.2). Our RM-based design ensures that resource fragmentation across controllers is reduced as it strives to pack (fractional) units to their capacity.

8 IMPLEMENTATION

We built SNF prototype from scratch in C++ (20K LOC) rather than building off existing platforms such as AWS Lambda due to their blackbox nature [38, 43]. It consists of:

Resource Manager. Implemented as a standalone process, it establishes TCP connections with controllers and handles resource requests.

Controller. Implemented as a multithreaded process, it establishes TCP connections with compute units and runs the compute management algorithm. It measures the compute unit load by monitoring the rate at which packets are drained. It measures this load at fixed buckets (of 500us) and considers the load over multiple buckets when packing (last 200 buckets, i.e., the last 100ms). The #buckets considered indicates the minimum time for which a change in traffic pattern should exist for the system to react. We choose the above values because smaller values made our system unstable by reacting to minor bursts, and larger values cause it to react too slowly. Flowlet detection is implemented using an approach similar to one described in [11, 42].

External Datastore and NF Runtime. Implemented as multithreaded processes. The former holds NF configuration state and the latter realizes the notion of ephemerally physically coupled state. Packet reception, transmission, processing, and datastore connection are handled by different threads. Protobuf-c [5] is used to encode and decode state transferred between units. Also, the NF runtime exposes APIs using which we reimplemented five NFs of varying complexity:

NAT. Performs address translation and the list of available ports is the NF configuration state. When a new connection arrives, it obtains an available port and it then maintains the per-connection port mapping.

LB. Performs hash-based load balancing. The servers' list constitute the NF configuration state. When a new connection arrives, it obtains the server based on hash, and then maintains (a) per-connection server mapping and (b) per-connection packet count.

IDS. Monitors packets using the Aho-Corasick algorithm [9] for signature matching. The string matching rules (e.g., Snort rules [34]) constitute the NF configuration state. Also, the NF maintains per-connection automaton state mapping.

UDP Whitelister. Prevents UDP-based DDoS attacks [17] by recording clients who send a UDP request.

QoS Traffic Policer. Implements the token bucket algorithm to do per-connection traffic policing. The per-connection (a) committed rate and (b) token bucket size constitute the NF configuration state. Also, the NF maintains per-connection mapping of (a) time since previous packet and (b) current available tokens.

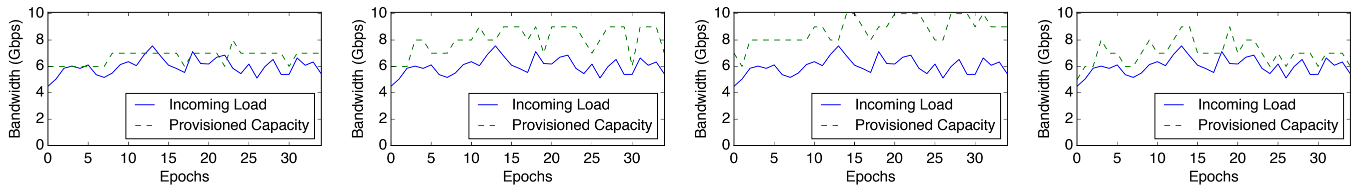
9 EVALUATION

We evaluate SNF to answer the following questions:

- Can it provision compute as per the traffic demand at fine time scales? Do we meet our goal of maximizing utilization without sacrificing performance?
- Does proactive state replication curtail tail latencies?
- How does it perform when adversarial flowlets occur?
- How quickly can it recover in the presence of failures?
- Is it able to reduce resource fragmentation when multiple controllers are being used?
- How does it perform with different system parameters?

Experimental Setup: We use 30 CloudLab [16] servers each with 20-core CPUs and a dual-port 10G NIC. The SNF RM and controller run on dedicated machines. The controller receives the replayed traffic from traces (details below) while the compute units run within LXC containers [8] on the remaining machines. For our experiments, we use one controller and each compute unit is configured to process packets at BW_{max}=1 Gbps, enabling 10 compute units per machine. The default parameters are: flowlet inactivity timeout $T = 500\mu s$, the flowlet size threshold $B = 15KB$, the balancing knob $\alpha = 0.25$ and the replication factor $K = 3$. We evaluate the sensitivity to these parameters in §9.6.

Real Packet Traces: We use two previously collected packet traces on the WAN link between our institution and AWS EC2 for a trace-driven evaluation of our prototype. One trace has 3.8M packets with 1.7K connections whereas the other trace has 6.4M packets with 199K connections. The median packet sizes are 368 Bytes and 1434 Bytes. All the experiments were conducted on both the traces with similar results; we only show results from the latter trace for brevity. Given that the load of the collected traces was not high, we scale the trace files by reducing the packet inter-arrival times.



(a) Vanilla Flow (b) Smart Flow (100ms) (c) Smart Flow (50ms) (d) Flowlet

Figure 1: Compute provisioning across various work allocation modes.

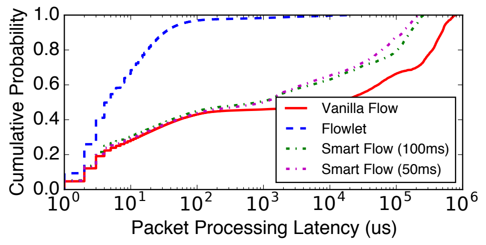


Figure 2: Packet processing latencies (NAT) across various work allocation modes.

9.1 Compute Management Performance

We first evaluate SNF's approach of flowlet-level work allocation. We measure (1) the provisioning efficiency with changes in traffic demands by recording the number of active compute units at 100 ms time intervals (referred to as an epoch hereafter), (2) the NF packet processing latencies which reflect the time spent in the input queue at the compute unit (queuing delay) plus packet processing time and (3) compute unit utilizations.

We compare against two state-of-the-art baselines:
(1) **Vanilla Flow Allocation:** work allocation is done when flows arrive, and once a flow is assigned to a compute unit, it is associated with that unit for its entire lifetime. This mimics work allocation techniques used by today's serverfull compute alternatives (when optimized state reallocation schemes [20] are not used). (2) **Smart Flow Allocation (X ms):** work allocation is done when flows arrive, and if required, flows are reallocated every X ms to avoid overload-/underutilization at any compute unit. This reflects state-of-the-art work allocation schemes NF solutions use - it involves using state reallocation mechanisms and thus supports flow migration [19, 20, 25, 44].

Figs. 1a-1d show a runtime snapshot of SNF's provisioned bandwidth and the packet processing demand[2]. We see that acting on flowlets enables SNF to closely match the incoming load, which is not the case at the flow granularity, irrespective of whether flow migration is supported or not.

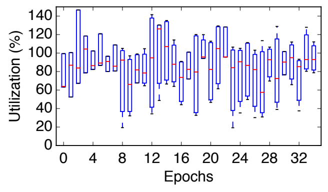
In the vanilla allocation mode, we see that the system does not adapt well to the incoming load as it can react only when new flows arrive. In the smart flow allocation mode, when we reallocate, if required, every X ms (X being 50ms or 100ms) the system is more adaptive in comparison to the vanilla flow mode as it gets more opportunities to reallocate flow. Acting at the flowlet granularity gives us **3.36X** more opportunities to assign work as compared to the alternatives, enabling SNF to better react to incoming load variations.

Additionally, when operating in the smart flow mode, there is over provisioning of units (greater when we are more aggressive to reallocate, i.e., smart flow (50ms)) due to the poor packability of flows which are larger work allocation units (in comparison to flowlets). However, even when acting in the flowlet mode, at times, additional 1-2 compute units are used in order to extract the benefits of proactive replication (§9.2).

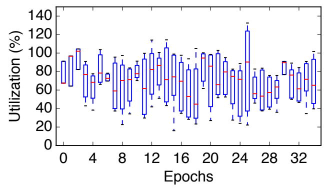
Packet Processing Latencies. Fig. 2 shows that the packet processing latency for the NAT NF while using the vanilla flow mode is significantly worse in comparison to the flowlet mode: the 75th%-ile latency is 275.4ms, which is **19.6K** times worse than for flowlet mode. Once flows are pinned to a compute unit, their association continues until the flows end and because flow rates vary during their lifetime (at times more than the estimated rate during work allocation) we observe input queues at compute units to build up. This is further exacerbated by the presence of elephant flows (5% of the flows in our trace have a size greater than 10KB). The trends for the other NFs are similar.

In the smart flow mode when we reallocate every 100ms (50ms), the latency is still worse than flowlets: the 75th%-ile latency is 64.5ms (41.1ms), which is **4.6K (2.9K)** times worse than the flowlet mode. This is due to (1) the mode being unable to handle overloads that occur at lower time-scales than the reallocation frequency (50ms or 100ms) causing input queues at the compute units to build up and (2) hold up of packets once reallocated until the relevant state is pulled from the prior compute unit. In the flowlet case the 99%-ile latency is 2.8ms, while the median is 5μs. The tail is contributed by micro bursts as the prototype cannot detect changes that last $<$ 100ms, leading to queuing occurring at the units and also due to flowlets for which the NF runtime

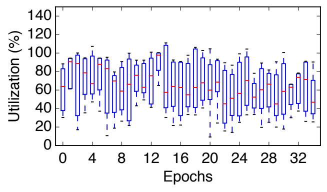
[2]Since each compute instance has BW_{max}=1 Gbps, the provisioned bandwidth is 1x #instances.



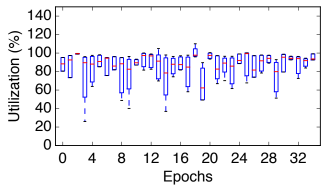
(a) Vanilla Flow



(b) Smart Flow (100ms)



(c) Smart Flow (50ms)



(d) Flowlet

Figure 3: Epoch-wise utilization distribution of the active compute units across various work allocation modes.

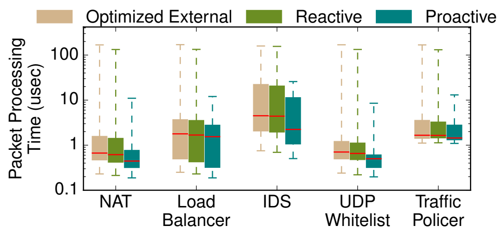


Figure 4: Packet processing latencies (1-25-50-75-99%-iles) for different storage modes.

has to reactively pull state from the previous units where the prior flowlet in the flow was processed.

Utilization. While we have seen that operating in the flowlet mode has the best performance, we need to verify that this improved performance is not coming at the cost of simply using more compute units. To do so, we delve deeper and look at the epoch-wise distribution of the active compute unit utilizations under the various modes (see Figs. 3a-3d).

As expected, while operating in the vanilla flow mode we experience maximum number of overloaded compute units as the system is the least reactive. Interestingly, in the smart flow (100ms) mode, even though over-provisioning occurs, we do see certain compute units being overloaded and this is due to the fact that the system can react only every 100ms. Consequently, in the smart flow (50ms) mode, we see lesser overload. Moreover, in all these modes wherein we act the flow level, we do experience more underutilization as well due to the poor packability of flows.

On the other hand, operating at the flowlet granularity rarely experiences overload as we get far more opportunities to react and have lesser underutilization as flowlets being smaller work units pack better in comparison to flows.

9.2 State Management Performance

We now evaluate SNF's approach of proactively replicating ephemeral state. We compare it against two state-of-art alternatives: (1) Optimized External: state is proactively pushed (rather than waiting for the flowlet end) to an external in-memory store and is read at the beginning at the flowlet. This baseline is an optimization to how state is transferred across compute units in today's serverless platforms[3]. (2) Reactive: state is pulled on the arrival of a flowlet from the previous compute unit that the flow was processed at. This represents existing NF state managements solutions [20, 25, 44].

We measure the per-packet processing latencies for the various NFs (Figure 4). For the NAT, the median latencies across the three modes, external, reactive and proactive are more or less similar (0.67μs, 0.61μs and 0.44μs) due to the fact that state for most flowlets is eventually locally available in all the three models. However, the tail latencies improve while shifting from the external to reactive and finally to proactive (168.18μs, 132.74μs and 11.01μs respectively). In both the baselines, upon arrival of the new flowlet, the updated state needs to be pulled from the external store and the previous compute unit respectively, thus the latencies are dominated by the network RTT. The tail latencies in case of using an external store are slightly higher than when reactively pulling state in a peer-peer fashion as there may be scenarios wherein a flowlet makes a reactive request to the external store and its state is not available, and thus has to wait longer. In the proactive mode, state is made available prior to the arrival of a new flowlet (unless there is a delay due to network anomalies or the flowlet has been scheduled to a unit which does not have replicated state) due to which processing is not stalled due to state unavailability. Similar latency trends are noticed for the other NFs.

Proactive State Replication: We carry out deeper analyses to understand where the benefits of proactive replication arise from. For the flowlets that were assigned to different units than the previous unit for the various NFs, proactive replication ensured that 90.43% of flowlets (on average across the NFs) were able to proceed seamlessly without any wait time whereas the remaining 9.57% reactively pulled state. This indicates that proactive replication comes into effect the majority of time helping to vastly reduce the tail.

Thus, with state optimizations, SNF achieves median latencies similar to when state is maintained locally which reduces the tail in comparison to existing alternatives.

[3]Today serverless platform don't write to external stores by default; such stores would have to be provisioned by the application.

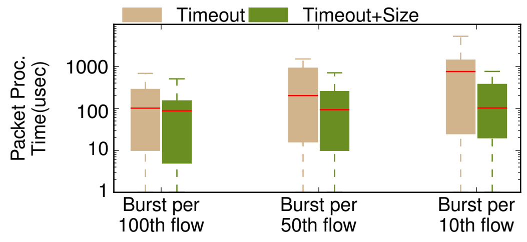


Figure 5: Packet processing latencies (1-25-50-75-99%-iles) in the presence of adversarial flowlets.

9.3 Tackling Adversarial Flowlets

To evaluate if SNF can tackle adversarial flowlets we use three synthetic workloads with varying frequency of such flowlets: we create these flowlets every 100th, 50th and 10th flow by adding bursts of 20 packets (~1400 bytes) on average. Other aspects of the experimental setup remain the same. For brevity, we only present results for NAT.

Recall that SNF should be able to mitigate the impacts of adversarial flowlets as it uses a size threshold in addition to inactivity timeout to detect flowlets. To study whether this helps, we compare detecting flowlets using both the criteria (timeout + size) with a baseline mode of using just the timeout (timeout) in terms of the packet processing latencies.

We see in Fig. 5, that it is beneficial to use both timeout and size in comparison to just using the timeout - from the least aggressive to the most aggressive workload we observe that the median (tail) latency reduces from 101μs (677.9μs) to 87μs (504.4μs), 201.2μ (1.5ms) to 93μs (702.4μs) and 752μ (5.2ms) to 102μs (756.2μs). Using both helps SNF bound the impact of adversarial flowlets by starting a new flowlet as soon as the size threshold has been met, which happens quickly for an adversarial flowlet and gives us the opportunity to reallocate such flowlets (does not occur while using just timeout) leading to reduced packet processing latencies.

9.4 Fault Tolerance

We study the performance of SNF under failure recovery and compare it against state of the art NF fault tolerance solutions - FTMB and CHC [25, 35]. The main metric of interest is the recovery time, i.e., the amount of time it takes to ensure that a new NF unit is available with up to date state. We fail a single NAT unit and measure the recovery times for FTMB, CHC and SNF at 50% load. We assume that the failover compute unit is launched immediately in all cases.

In case of FTMB, the recovery time is 25.7ms (assuming that FTMB does checkpointing every 50ms) and includes the time taken to load the latest checkpoint as well as the time taken to process the packets that need to be replayed to bring the new NAT unit up to date. CHC under the same failure scenario takes 3.2ms during which the latest state is fetched from the datastore and the in-transit packets are replayed. On the other hand, in SNF the recovery time is 140μs which accounts for the amount of time taken to transfer the state from the OL of the failed NAT unit to this newly launched NAT unit. Unlike FTMB and CHC, given that SNF stores a copy of the latest state at the OL, it does not need to replay packets during recovery leading to a faster recovery time.

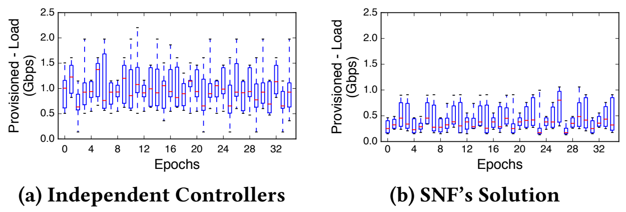


Figure 6: Comparison of epoch-wise overprovisioning distribution by each controllers while using independent controllers and our approach.

9.5 Multiple Controllers

To evaluate SNF at scale when using multiple controllers we compare our approach of using a RM to the baseline mode of operating the controllers independently. We use 10 controllers and the cumulative load is on average 93.5 Gbps. We record the per-controller provisioned capacity and actual load received and look at their difference (see Fig. 6a-6b). The Y-axis value being 0 represents the ideal case (provisioned capacity equals load), > 0 indicates over provisioning (reducing efficiency).

As seen in Figs. 6a-6b, the amount of over provisioning is minimal in case of SNF as opposed to using independent controllers. The reason being that in the baseline mode there is more resource fragmentation due to controllers provisioning compute units independently. With SNF, since the RM "leases out" capacity of compute units, resource fragmentation is reduced as multiple controllers can send traffic to the same shared compute unit (up to their allocated share).

9.6 Sensitivity Analysis

Flowlet Inactivity Timeout (T): Setting the flowlet inactivity timeout plays a crucial role in SNF as the value decides how closely SNF can adapt to traffic changes. Additionally, it impacts the efficiency of proactive replication. We consider two timeout thresholds: $T = 100\mu s$ and $T = 500\mu s$. In comparison to allocation at the per flow level, we see 5.18X and 4.66X more opportunities to do work allocation when $T = 100\mu s$ and $T = 500\mu s$, respectively. While $T = 100\mu s$ clearly has benefits, we choose $T = 500\mu s$ in SNF. This improves the benefits of proactive replication, given that to replicate state of our NFs takes about 160μs.

Flowlet Size Threshold (B): We consider multiple thresholds to study the impact on both performance and utilization

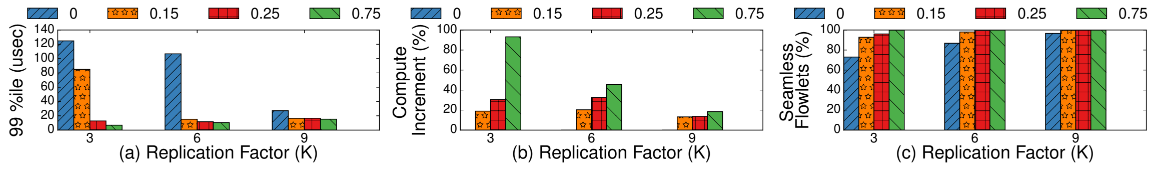


Figure 7: Impact of varying K and α on (a) packet processing time, (b) compute instances and (c) flowlets that can be processed seamlessly.

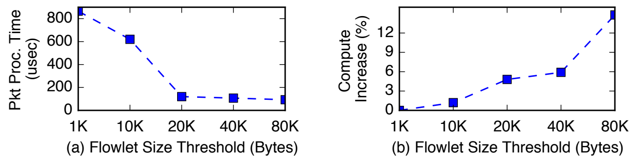


Figure 8: Impact of size threshold on latency and compute provisioning.

for NAT. As the flowlet size increases, the processing latencies improve (Fig. 8a) primarily because the number of reactive state pulls decrease. On the other hand, this decrease in latency comes at the cost of increased usage of compute units (Fig. 8b) due to poor packability of the larger work allocation units.

Replication Factor (K) and Balancing Knob (α): Figs. 7a-7c show the impact of changing the value of K and α while having a maximum of 15 compute units in use. Here the NF capacity used is 500 Mbps. For a given K, on increasing α, the number of flowlets that can be processed without state unavailability delays increases, as our scoring metric gives more importance to units that have the state (§5). However, this comes at the cost of using some amount of additional compute units. Needless to say, the tail latencies improve as the value of α increases as more flowlets are scheduled on units where their state is present (e.g., for $K = 3$, the latency decreases from 107μs to 6.8μs when α changes from 0 to 0.75). While for smaller values of α, the latencies are dominated by reactive state pulls, for larger values of α we see that as K increases, the latency increases from 6.8μs to 15.2μs (when K changes from 3 to 9) reflecting the overhead involved in proactively replicating state.

9.7 Overheads

Work Allocation Overhead. The controller calls into the work allocation algorithm for every new flowlet. This adds a latency of 1μs, but this is once per flowlet and hence is amortized across the packets of the flowlet.

Proactive Replication. In our current prototype, we proactively replicate state to K compute units every 250us (half of the flowlet timeout). For the said trace with K = 3, the proactive replication for NAT, LB, IDS, UDP Whitelister and QoS Traffic Policer uses up an additional bandwidth of 3.62 Mbps, 4.13 Mbps, 3.12 Mbps, 2.9 Mbps and 4.8 Mbps respectively.

10 OTHER RELATED WORK

Some recent studies have show the benefits of using *existing* serverless platforms in unmodified form for "non-standard" applications, e.g., scalable video encoding [18], and parallelized big data computations [22, 33]. Other works instead focus on improving key aspects of serverless computing, e.g., reducing container start-up times [12, 31], improved system performance [10], new storage services [27, 28], which proposed elastic ephemeral storage for serverless, and security [13]. Our work falls into this second category, where we provide additional support for high performance stateful applications such as NFs. [37] proposes a high-performance stateful serverless runtime using software-fault isolation which avoids expensive data movement when functions are co-located on the same machine. However, it defaults to reactively pulling state when functions are on different machines leading to high processing latencies. SNF adds to the long line of literature on NFs and NFV. Improving performance in standalone software environments is the goal of several papers [2, 21, 32, 41]. There has also been significant efforts in failure resiliency for NFV environments [25, 29, 35].

11 CONCLUSIONS

We show the benefits of leveraging serverless computing for streaming stateful applications, using NFs. SNF effectively matches varying NF demands, while ensuring good efficiency and packet processing performance, and offering fine-grained resource tracking and billing. SNF decouples the functions operation unit and serverless work allocation unit, using flowlets for the latter. The SNF runtime proactively replicates state during inter-flowlet gaps, realizing ephemeral physical coupling between compute and state.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers of SoCC'20 and the members of WISR Lab for their insightful comments and suggestions. This research was supported by NSF Grants CNS-1565277, CNS-1719336, CNS-1763810, CNS-1838733 and by gifts from Google and VMWare.

REFERENCES

[1] 2017. AWS Lambda. https://aws.amazon.com/lambda.

[2] 2017. Berkeley Extensible Software Switch (BESS). http://span.cs.berkeley.edu/bess.html.

[3] 2018. AWS S3. https://aws.amazon.com/s3/.

[4] 2018. PRADS. https://gamelinux.github.io/prads/.

[5] 2018. Protobuf-c. http://lib.protobuf-c.io/.

[6] 2019. AWS Elastic Load Balancing. https://aws.amazon.com/elasticloadbalancing/.

[7] 2019. AWS Firewall Manager. https://aws.amazon.com/firewall-manager/.

[8] 2019. LXC Containers. https://linuxcontainers.org/lxc/introduction/.

[9] Alfred V Aho and Margaret J Corasick. 1975. Efficient string matching: an aid to bibliographic search. *Commun. ACM* 18, 6 (1975), 333–340.

[10] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 923–935.

[11] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 503–514.

[12] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 645–650.

[13] Stefan Brenner and Rüdiger Kapitza. 2019. Trust More, Serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. 33–43. https://doi.org/10.1145/3319647.3325825

[14] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. San Francisco, 161–174.

[15] YaoZu Dong, Wei Ye, YunHong Jiang, Ian Pratt, ShiQing Ma, Jian Li, and HaiBing Guan. 2013. Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.

[16] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14.

[17] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. 2015. Bohatei: Flexible and Elastic DDoS Defense. In *24th USENIX Security Symposium (USENIX Security 15)*. 817–832.

[18] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 363–376.

[19] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Vyas Sekar, and Aditya Akella. 2013. Stratos: A network-aware orchestration layer for virtual middleboxes in clouds. *arXiv preprint arXiv:1305.0209* (2013).

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/2619239.2626313

[21] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 445–458.

[22] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[23] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley.

[24] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 97–112.

[25] Junaid Khalid and Aditya Akella. 2019. Correctness and Performance for Stateful Chained Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 501–516.

[26] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 239–253.

[27] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 789–794.

[28] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA, 427–444.

[29] Sameer G Kulkarni, Guyue Liu, K. K. Ramakrishnan, Mayutan Arumaithurai, Timothy Wood, and Xiaoming Fu. 2018. REINFORCE: Achieving Efficient Failure Resiliency for Network Function Virtualization Based Services. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, 41–53.

[30] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. 2016. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16)*. USENIX Association, Berkeley, CA, USA, 255–273.

[31] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 57–70.

[32] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 203–216.

[33] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation*

(NSDI 19). 193–206.

[34] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks *(LISA '99)*. 229–238.

[35] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 227–240.

[36] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 13–24.

[37] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433.

[38] Arjun Singhvi, Sujata Banerjee, Yotam Harchol, Aditya Akella, Mark Peek, and Pontus Rydin. 2017. Granular Computing and Network Intensive Applications: Friends or Foes?. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 157–163.

[39] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. 2019. SNF: Serverless Network Functions (Tech Report). arXiv:1910.07700 [cs.DC]

[40] Shan Sinha, Srikanth Kandula, and Dina Katabi. 2004. Harnessing TCP's burstiness with flowlet switching. In *Proceedings of the 3rd ACM Workshop on Hot Topics in Networks*.

[41] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. ACM, New York, NY, USA, 43–56. https://doi.org/10.1145/3098822.3098826

[42] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 407–420.

[43] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.

[44] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 299–312.