Granular Computing and Network Intensive Applications: Friends or Foes?

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

Sujata Banerjee
VMware
sujata@banerjee.net

Yotam Harchol*
UC Berkeley
yotamhc@berkeley.edu

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

Mark Peek
VMware
markpeek@vmware.com

Pontus Rydin
VMware
prydin@vmware.com

ABSTRACT

Computing/infrastructure as a service continues to evolve with bare metal, virtual machines, containers and now serverless granular computing service offerings. Granular computing enables developers to decompose their applications into smaller logical units or functions, and run them on small, low cost and short lived computation containers without having to worry about setting up servers - hence the term serverless computing. While serverless environments can be used very cost effectively for large scale parallel processing data analytics applications, it is less clear if network intensive packet processing applications can also benefit from these new computing services as they do not share the same characteristics. This paper examines the architectural constraints as well as current serverless implementations to develop a position on this topic and influence the next generation of computing services. We support our position through measurement and experimentation on Amazon's AWS Lambda service with a few popular network functions.

1 INTRODUCTION

Computing requirements in virtually every sector of industry and society continue to grow rapidly. To meet this demand, both public and private cloud computing services are evolving with the goal to providing cheap, performant and flexible computing resources that can be dynamically scaled up and down with workload changes. There are a large variety of computing services today that enable users to rent bare-metal servers, virtual machines (VMs), or containers, at a dizzying number of sizes and pricing structures, and with varying levels of resource management burdens on the end users. The latest exciting evolution in computing is a serverless computing model also known as functions-as-a-service (FaaS), which provides computing in short-lived, stateless, bite sized chunks with highly flexible scaling and a pay-for-what-you-use price structure. Users no longer have to provision/manage servers or VMs - instead they build their applications as a set of functions with the cloud provider taking on the resource provisioning tasks and scaling.

Serverless computing today is particularly well suited for microservices that are part of a class of embarrassingly parallelizable applications. Obviously, not every application can be written in a "serverless" manner. However, given that cloud computing providers can now provide these services profitably and pass on a significant cost benefit and flexibility to the users and developers, it is our expectation that many traditional applications will evolve to make use of this new paradigm in some way or another. In this position paper, we take one such class of applications, which on the face of it, seem completely unsuitable for serverless environments and propose a workable framework with a view to generating discussion in the community. With this activity, our goal is to help serverless architectures to evolve in a direction that retains many of the current benefits while enabling a larger class of applications to use this emerging framework.

The class of applications considered here are network functions (NFs) involving packet processing functions used in both enterprise and telecommunications service provider environments. These are particularly challenging for serverless environments, as network functions may be stateful, long lived, require high packet throughput, have additional requirements such as chaining of functions, which are all at odds with what serverless environments provide today. At the same time, the long term deployment vision for network functions is a dynamic auto-scaling micro-service based architecture with a significant capital and operational cost reduction over what is possible today. This vision has alignment with the goals of serverless platforms. Thus matching the requirements of this class of applications to the features of serverless environments and identifying gaps that exist today helps to develop a

*Work done while at VMware Research

HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA

ACM ISBN 978-1-4503-5569-8/17/11…$15.00
https://doi.org/10.1145/3152434.3152450

position on making serverless architectures more generally applicable. Towards this end, we take the following steps:

- We benchmark Amazon's popular AWS Lambda service focusing on the requirements of network functions and identify the challenges more concretely.
- We develop strawman architectures and the associated cost implications for adapting existing serverless environments to be friendly to network functions.
- We present results with a few popular Click-based [17] network functions to support our hypothesis.

2 BACKGROUND AND CHALLENGES

In this section, we present a quick overview of the salient features of both serverless computing services and network functions. There is continuous and rapid evolution in both technical areas, and a current snapshot is provided with a view to aligning both efforts for significant benefits.

2.1 Serverless Computing Platforms

Serverless computing services enable users to package their applications as stateless functions that are housed in small short-lived compute units and dynamically scale functions based on user provided events. Most serverless offerings are built on top of light weight containers with the associated dynamic scaling mechanisms triggered by events that can be stored in a variety of other services. Typically, users register for events (e.g. http request, or when a file gets uploaded to the datastore) to trigger execution of their functions.

While there are a number of serverless computing platforms [1, 2, 5, 6, 8, 10] out there, for the purpose of this work, we choose Amazon's AWS Lambda for two main reasons. Firstly, it is the most developed and general environment today with support for chaining functions. Secondly, it has support for running native executables, which enables us to run existing network functions atop AWS Lambda. Each AWS lambda function can be provisioned with 128 MB - 1.5 GB of memory, can live for up to 5 minutes of computation, and can scale to a maximum of 1000 instances. The cost structure of using the lambda infrastructure has two components: a per-user request cost ($0.20 per Million requests) and a computation cost charged on a 100 ms granularity ($0.00001667 per GB-second). The cost parameters and provisioning of AWS Lambda and the other available services are sure to evolve, and there are many limitations of the overall design in how different applications can utilize the API and associated services. However, we view these design decisions as just a reflection of what service providers can provide today to presumably run a profitable service.

Thus our goal is to adapt the application to the service environment available while creating a wish list of features that might help these applications to attain high performance at a reasonable cost. A couple of trail-blazer research projects have already been proposed that adapt specific applications to use AWS Lambda with great benefits: a scalable video encoding application [12] and a distributed computing framework that can serve many common applications [15]. We consider a third set of packet processing applications which present even bigger challenges.

2.2 Network Functions

New network and application functionality has long been implemented in network/application middleboxes. Traditionally, these middleboxes have been proprietary with specialized hardware and customized software. Increasingly in both enterprise and telecommunications service provider environments, the hardware and software components of network functions are being decoupled, virtualized and deployed over commodity servers in private/public/telco clouds. The goal for this transformation is to corral the rapidly growing capital and operational expenditures of such environments by leveraging the favorable cost structures of cloud computing.

Many network packet processing functions require high throughput and low latency, creating technical challenges for virtualized networking stacks to process packets efficiently. Network functions can be chained together to form service chains implementing a composed complex service. A plethora of research papers have tackled these performance and chaining issues with kernel bypass, zero-copy mechanisms, for example in [3, 4, 14, 18]. In addition, several network functions may be stateful, and managing state in the face of scaling, migration and failures presents challenges that have been the subject of several papers, for example in [13, 16]. Verification and troubleshooting of stateful network paths and service chains is another topic that has been tackled in [19, 22]. The longer term vision for Network Function Virtualization (NFV) [9] is to further decompose each virtualized network function into smaller sub-functions and expose them as micro-services. Individual micro-services can be re-used across multiple service chains and independently scaled leading to further flexibility and efficient use of the infrastructure. This micro-service architecture for NFV and running a huge number of tiny NFs packaged in light weight containers on a single server has been proposed in a recent paper [23].

2.3 Technical Challenges

In many ways, the FaaS or serverless architectures are well aligned with the micro-service vision for NFs. However, there are several challenges in enabling packet processing network functions and service chains to leverage serverless computing. Here we list some of the mis-matches in goals that need to be overcome. As mentioned before, we use AWS Lambda as a concrete example, as it is one of the front-runners with the most advanced features at the time of our experiments.

- NF workloads may be compute intensive, long lived and stateful, while serverless functions are short-lived, stateless and run on lightweight containers with limited computing power. No persistent state can be maintained inside the lambda infrastructure and other cloud services may have to be utilized.

- Packaging packet processing workloads in event driven serverless APIs such as http requests may incur a high overhead and decrease performance.
- Most serverless frameworks do not yet have guaranteed quality of service (QoS), isolation and high levels of security. Yet, many NFs require these features when deployed in production.
- Users are not granted root access in most serverless environments, which is required to run some existing NFs, for example, to run raw sockets. In addition, in AWS Lambda, network connections to end points outside the lambda infrastructure can only be initiated from the lambda instances.
- There is limited support for chaining functions. Existing methods such as AWS Lambda's "step" function have high overheads.

In spite of the above challenges, we take the somewhat audacious position that packet processing applications can also benefit from serverless computing. In the next section, we develop strawman solutions to adapt packet processing applications to serverless environments.

3 SERVERLESS PACKET PROCESSING

To mitigate the above challenges with mis-matched requirements, we propose two strawman architectures. Our goal is to not flesh out each architecture in great detail but explore two designs to guide future directions. The first design choice is to decide the granularity of each request made to the serverless infrastructure. The second design choice is to decompose the NF processing, memory, and bandwidth requirements to fit each compute instance, as discussed next.

3.1 Granularizing NF processing

Given that serverless computing is aimed at parallel processing workloads, we first "granularize" the service function chains in two ways. Each network function to the extent possible is decomposed into its sub-functions with the expectation that each sub-function has different resource requirements and can scale independently. Additionally, the application flow-space is partitioned into sub-flows such that each sub-flow can traverse its own service chain and not exceed the bandwidth capacity of a compute instance. With this granularization, each compute instance is expected to run a whole network function or a sub-function of an NF. These granularizing decisions also need to be made according to the latency and throughput requirements of the service function chain, as overloading the serverless compute instances can reduce performance.

3.2 State Management

Since serverless computing instances are typically stateless, no persistent state can be maintained inside the serverless infrastructure. However, there are many stateful NFs that may need to maintain per-packet, per-flow or cross-flow state. Any state information that needs to be maintained for a packet or flow can be stored in one of the helper services of the serverless compute infrastructure (e.g., Redis key value store in AWS). Needless to say, state retrieval or updates need to incur low latency. State management is dependent on the serverless packet processing architecture and we deal with this in the following sections.

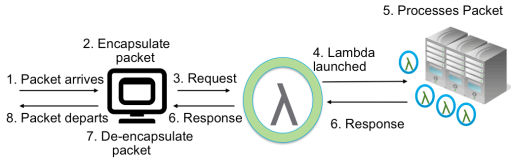

Figure 1: Per-Packet Architecture

3.3 Strawman Architectures

There are two extremes in the architectural design space of serverless frameworks to process granular NF packet processing workloads. One extreme is to use packets as the basic unit of request to the serverless compute service, while the second uses flows as the unit of the request. Neither framework is perfect and we discuss the pros and cons of each. Needless to say, there may be other request granularities in between, but we do not explore those here.

3.3.1 Packet-based Architecture. This architecture (Figure 1) is the most straightforward approach to directly leverage the serverless infrastructure with minimal additional components. Different NFs or microservices are uploaded to the serverless infrastructure. Each incoming packet (encapsulated into a http or any other request API) triggers the launching of compute instance. Essentially, packets are "sprayed" across the compute instances and each packet in a flow could be processed by different instances. The advantage is that we can natively leverage infrastructure features like chaining and dynamic auto-scaling up and down.

However, such an architecture would never be deployed due to the following reasons. Firstly, since most public serverless computing services have a per request cost, this approach would incur a per packet charge which would be prohibitively expensive. At AWS Lambda price structures, a million packets incur a cost of $0.20. To support a flow rate of 1 million packets per second and a flow duration of 1 minute would cost $12 per minute in addition to the actual computing costs. There are clearly other cost effective strategies. Secondly, this approach makes state management complex as each instance would need access to all pertinent state for all flows. For acceptable performance, one would want to cache state locally. However, this is not possible in the aforementioned approach. Lastly, this approach also experiences packet reordering due to the variable processing overheads of each request traversing the serverless infrastructure.

3.3.2 Flow-based Architecture. In the flow-based architecture (Figure 2), serverless computing instances also run NFs/sub-functions as in the per-packet approach. However, the big difference here is that each sub-flow is assigned to a specific computing instance with all packets of the sub-flow being processed by the same computing instance. This

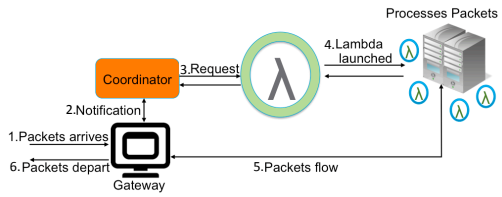

Figure 2: Per-Flow Architecture

removes the per-packet request overhead and hence the per packet cost, but this solution requires additional infrastructure to coordinate the flow-to-compute instance mapping as well as additional bootstrapping. Chaining also gets more complex as available mechanisms such as "step" functions cannot be used easily.

A core component of our architecture is a serverless computing instance coordinator/scheduler (similar to approaches in [12, 15]) which runs alongside the serverless infrastructure and does the above assignment process of service functions/modules and sub-flows to computing instances. As new sub-flows arrive and present their service function requirements, the coordinator (much like an SDN controller in spirit) creates requests to spawn new compute instances running the required network function. The requests generated have configuration information for the network function (e.g., firewall rules to process the sub-flow), service chain information (e.g., service chain IDs) as well as bootstrapping information for the compute instance to communicate with the sub-flow endpoint. Packets from the sub-flow can now directly traverse the service chain to receive the appropriate packet processing, without involving the coordinator.

The coordinator also keeps track of instance usage and makes resource decisions during the flow processing. For example, if a sub-flow duration is longer than the compute instance life time, the coordinator proactively spawns another instance that the sub-flow can be handed off to as needed. Further, the coordinator may keep track of flow state information and assist in the hand-off. Cross-flow state maintenance can also be aided by the central coordinator. The gateways (as shown in Figure 2, act as relays between the serverless infrastructure and the incoming traffic as each lambda initiates a connection with a gateway.

The main advantage of this approach is its potential relatively low cost. Using the same example above, 1 million packets per second belonging to only a small number of flows (say 50-100), would thus trigger just those many requests. One minute of computation also incurs negligible cost and so a back of the envelope calculation yields a total cost close to 1/1000-th of a dollar, without considering the cost of a coordinator, which would need to be hosted in a long lived VM/server cluster but amortized over many flows.

The main disadvantage of this approach is the additional complexity of the coordinator function and the inability to directly use the native auto-scaling and chaining mechanisms of serverless platforms. Auto-scaling triggers for network flows and chaining which is not tied to request/response functions are missing today. Finally, as mentioned above, since some flows will outlive the computing instance that it was assigned to, flow and state migration will also need to be handled.

4 EXPERIMENTAL VALIDATION

In order to validate our architectural hypotheses, we built a measurement framework and a basic prototype of the two strawman architectures on Amazon AWS Lambda. The basic prototype implementation is built for the sole purpose of conducting baseline experiments and does not have full functionality yet. We conducted experiments to micro-benchmark performance with three popular Click-based NFs [17], and included service chaining. Ideally, we would have liked to use production NFs or other widely used open-source NFs such as Snort [21] or Bro [20]. However, recall that we do not have root access and thus many of these NFs cannot be made to work on lambda instances and require source code changes. This rules out production closed source NFs. In addition, one of our goals is to decompose NFs and create efficient NF chains similar to OpenBox [11]. The Click framework allows for this and hence was chosen.

4.1 Experimental Framework

In our experimental framework, a set of VMs act as the sources and sinks of the packet streams. Requests for lambda services are launched from these VMs into the lambda infrastructure, which we treat as a blackbox. The traffic sources generate TCP traffic streams at various rates and maximum packet sizes. UDP traffic suffers significant loss as AWS Lambda currently only supports TCP/IP sockets for connections, and hence not used in our experiments. We compile three Click-based NFs - packet counter (PC), firewall (FW) and an intrusion detection system (IDS) to run as binary executables on the lambda instances, linked from the Python functions. Given that we do not have root/sudo permissions on lambda instances, we cannot use raw sockets. Instead we use the send/receive modules on Click with TCP sockets. In the per-packet approach, we use lambda "step" functions to chain various NFs. In the per-flow approach, one of the VMs is used to host the coordinator function. The chaining functionality in the coordinator is not implemented yet. The results presented below are obtained from running the experiments within the "us-east-2" AWS region. Similar trends were observed across the various AWS regions.

4.2 Experimental Results

First, we present micro-benchmark performance numbers on the AWS Lambda system, including "step" functions, followed by the lambda execution times of three NFs.

4.2.1 AWS Lambda Benchmarks. Prior papers [12, 15] have benchmarked the performance of AWS Lambda as well. We validated these numbers in our setup on AWS, and summarize them here. The lambda API gateway is a source of high latency with high variability, as reported in the AWS mailing lists as well. Using the lambda SDK to invoke lambda

instances yielded better results. Cold lambda startup times (includes the function loading overhead) are of the order of seconds and the median lambda request-response latencies are about 25 msec.

Given the nature of our application, we are interested in benchmarking network bandwidth in/out of the lambda infrastructure. Figure 3 shows one micro-benchmark of simultaneous bi-directional iperf [7] measurements in/out of a lambda instance at different memory sizes. The main take-aways are that on average 500 Mbps of bandwidth is available in both directions, with the inbound rate slightly lower than the outbound and with relatively little variation with memory sizes. We also conducted unidirectional bandwidth tests and found slightly higher throughput of about 580 Mbps.

From our measurements, we notice an interesting rate limiting behavior imposed by AWS. Typically, lambda functions are behind a NAT and undergo address translation to access VMs. Multiple lambdas may be behind the same "external" IP address but a different port number. The number of lambdas sharing the same IP address seems to vary but the rate limiting is done at the IP level leading to this behavior. Figure 4a shows that the scaling is not linear. However, we believe that this is an artificial limit imposed and is not fundamental to achieving higher throughputs. After all, [15] has reported over 40GB/sec throughputs to/from the S3 service.

Next we benchmark the performance of step functions since it helps with chaining NFs in the per-packet approach. Step functions are a special construct which enables one to build a complex distributed application deploying a state machine representation of an application easily. While step functions can be used for the purpose of service chaining of lambda functions, they have significant coordination overheads. Figure 4b depicts the latency overhead of step functions as the chain length is increased. In this experiment, we chain together N functions, each of which sleeps for 100 msec mimicking a computation of 100 msec. Even for a chain length of 1, where we package a lambda function in a single step function, the latency overhead is close to 46 msec (we do not include the time spent in the lambda function itself). Step functions are not ready to be used for NF chaining at this point given the high overheads. We also verified the independent scaling of individual functions in a chain. Three functions with sleep times of 10 ms, 100 ms and 400 ms respectively when chained together in a step function, instantiate 2, 3 and 5 lambda instances respectively. The number of lambda instances invoked for each function depends on its processing load, which is exactly how chained micro-service scaling should operate.

4.2.2 NF Benchmarking. One of the important considerations is to ensure that the NFs "fit" in each lambda instance so as to have acceptable per packet processing times. We packaged each NF in a lambda function and under a low packet load, measured the packet processing time (does not include the lambda infrastructure overheads) for two different packet sizes. Table 1 has these results. Each packet is encapsulated

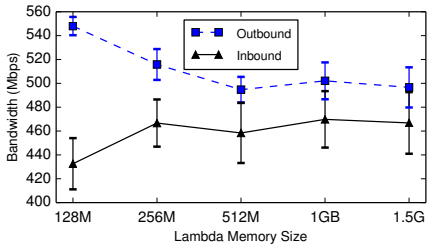

Figure 3: Bi-directional bandwidth in/out of a lambda

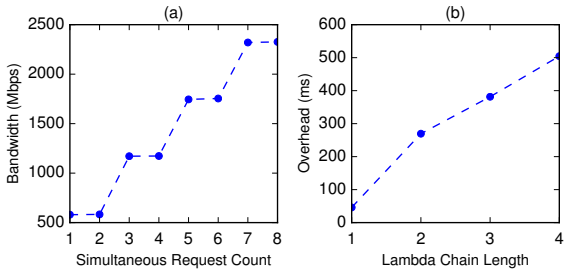

Figure 4: (a) Bandwidth scaling with multiple lambdas; (b) Step Function Overhead

Table 1: Mean Execution Time of NFs

Packet Size	64 Bytes	1500 Bytes
Packet Counter	0.34 ms	0.40 ms
Firewall	0.84 ms	0.73 ms
IDS	0.91 ms	0.92 ms

in a http request like it would be in the per-packet approach, which has additional overhead. We have not yet attempted to optimize the NF processing times but we believe this can be further reduced. As expected, the packet counter has lower processing overheads than the firewall. The difference in processing time with the increase in packet sizes is minimal and we believe that in both cases the cost involved to start the NF and transfer the packet from the lambda runtime to the click environment dwarfs the actual packet processing time.

These NFs are stateless and we assume that all associated state is available to the lambda instance running the NF. As mentioned earlier, state management may be done via a combination of external services associated with the serverless compute service and local caching. For example, prior work [15] has reported access latencies to the Redis in-memory key value store less than 1 msec for upto 1000 workers. As discussed in [16], the goal is to get about 100 μsec - 500 μsec, which we believe will be available as emerging technologies and services get rolled out.

4.2.3 Evaluation of NF processing architectures. Based on the benchmarking results above, we evaluated the per-packet and per-flow approaches from both performance - throughput and latency, and cost perspectives and present our preliminary findings. Serverless infrastructures and features are yet emerging and are under constant churn. While our evaluation is done with current offerings and price structures, we strongly believe that these will all evolve and impact our findings. Our hope is that studies like ours will affect this evolution.

Table 2: Per-Flow : Throughput and Latency

Network Function	Mean (Mbps)	Std-Dev (Mbps)	Mean (msec)	Std-Dev (msec)
Packet Counter	367.14	18.34	2.89	1.14
Firewall	355.90	18.72	3.39	1.41
IDS	339.03	21.85	3.40	1.79

In the per-packet architecture, we chained together the PC and FW NFs with a "step" function. Given that the NFs have different setup and processing times, the two lambda functions scaled independently to 70 and 113 instances respectively at a low load of 100 packets per sec for 10 seconds.

In Table 2, we provide micro-benchmarks for the per-flow approach. We report the mean throughput and round-trip latency from a source to a sink through a given NF. Each lambda is provisioned with 128 MB of memory. We report approximately 350 Mbps of throughput and under 4 msec of round-trip latency for a single flow (consisting of 1500 Bytes packets) and a single NF hosted in a lambda instance. While these are modest and unoptimized results, it gives us hope for what future serverless environments may provide. A naive and optimistic scaling to a say, 100 lambdas can provide up to 35 Gbps of NF processing throughput, assuming that rate limits imposed today can be lifted.

An interesting thought exercise and back-of-the envelope cost calculation is as follows. Assume that 500 Mbps of flow throughput is possible per lambda for a firewall function with acceptable latency. To construct a 40 Gbps firewall function would require 80 128-MB lambdas. If each flow duration is 5 minutes, to run this 40 Gbps firewall for an hour, would cost $0.60. Just for amusement, the cost using a per-packet approach for the same flow requirements is over $4000. A VM based solution where the cost is on a per hour basis, this would cost about $8.

5 DISCUSSION

Serverless computing presents a new and exciting way to consume computing with a potentially attractive price structure and lower management overhead for the users. The first generation of serverless computing is aimed at serving a subset of applications which are stateless, event-driven, decomposable into microservices and eminently parallelizable. Our position is that the next generation of serverless computing should be generalizable to other applications while retaining the original benefits. We explore this position by considering an application class - network packet processing, that clearly does not fit the original design of serverless computing and propose strawman approaches to support this application class. We conducted basic measurements of the Amazon AWS Lambda service to better understand the constraints and produce the following list of requirements and future directions for the next version of serverless computing.

- While most serverless implementations have an impressive array of event sources and types, there are no native and obvious ways to trigger network processing events, for example, related to packets/flows/application messages. Efficient ways to invoke computing instances are needed - using http to invoke lambda functions is clearly limiting.
- Providing root access, would enable easier and wider deployment of this class of applications.
- Guarantees on resources, particularly bandwidth and latency for this application class is important. While the main service offering is computing cycles today, network and I/O provisioning needs to be explicit and made visible to users.
- Packet processing operations occur at the micro-second granularity, while the pricing structure is based on a larger granularity (e.g., 100 msec for AWS Lambda).
- Existing serverless offerings separate compute and storage. Designing a scalable remote storage service that meets the demands of this class of applications is needed.
- Most of the serverless implementations have a simple fault tolerance model - when a function fails, rerun it. While such a model may work fine for idempotent functions, it will not work for all classes of applications. Also, there is no support for local state fault tolerance on these platforms as they were designed for stateless functions. However, providing efficient function and state fault tolerance is a must for supporting this class of applications.
- The scaling power of serverless environments comes from stateless and infrastructure agnostic placement and scheduling of compute instances. This is particularly problematic for application units that have some ordering or coordination requirements. Chaining support is noteworthy in this context, but is not widely available or has too heavy an overhead currently. Understanding the trade-offs of some stateful placement and providing native support for these additional requirements is an important direction.

We realize that many of the "asks" above come with an increased price tag and may negate the cost benefits. However, we believe that many of the current design constraints on serverless computing are not fundamental limitations; rather, they can be carefully lifted to provide a cost effective as well as flexible solution. Resource provisioning and the associated costs of public and private clouds is different and examining serverless as a cloud native design pattern is a promising direction. Until more flexible serverless infrastructures are available, purpose built application-specific serverless environments may be needed to address the requirements.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. The first author is grateful for early helpful discussions with Remzi Arpaci-Dusseau, Shruthi Racha and Anshul Purohit. Arjun and Aditya were supported in part by NSF grants CNS-1563011, CNS-1636563 and CNS-1547613.

REFERENCES

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

[2] 2017. Azure Functions. https://functions.azure.com. (2017).

[3] 2017. Berkeley Extensible Software Switch (BESS). http://span.cs.berkeley.edu/bess.html. (2017).

[4] 2017. Data Plane Development Kit (DPDK). http://dpdk.org. (2017).

[5] 2017. Google Cloud Functions. https://cloud.google.com/functions. (2017).

[6] 2017. IBM Bluemix Openwhisk. https://www.ibm.com/cloud-computing/bluemix/openwhisk. (2017).

[7] 2017. Iperf. Documentation. http://software.es.net/iperf/. (2017).

[8] 2017. IronFunctions. https://github.com/iron-io/functions. (2017).

[9] 2017. Network Functions Virtualisation - Introductory White Paper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf. (2017).

[10] 2017. OpenLambda. https://open-lambda.org. (2017).

[11] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 511–524.

[12] 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)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[13] 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 (SIGCOMM '14)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/2619239.2626313

[14] 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)*. USENIX Association, Seattle, WA, 445–458. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

[15] 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 (SoCC '17)*. ACM, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[16] 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)*. USENIX Association, Boston, MA, 97–112. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

[17] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[18] 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)*. USENIX Association, GA, 203–216. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[19] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 699–718. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths

[20] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer networks* 31, 23, 2435–2463.

[21] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks.

[22] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Banerjee, Jeongkeun Lee, and Joon-Myung Kang. 2016. SFC-Checker: Checking the correct forwarding behavior of Service Function chaining. In *Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE Conference on*. IEEE, 134–140.

[23] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. 2016. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, New York, NY, USA, 3–17. https://doi.org/10.1145/2999572.2999602