EdgeBalance: Model-Based Load Balancing for Network Edge Data Planes

Wei Zhang Abhigyan Sharma[*] Timothy Wood

The George Washington University [*]AT&T Labs Research

Abstract

Edge data centers are an appealing place for telecommunication providers to offer in-network processing such as VPN services, security monitoring, and 5G. Placing these network services closer to users can reduce latency and core network bandwidth, but the deployment of network functions at the edge poses several important challenges. Edge data centers have limited resource capacity, yet network functions are resource intensive with strict performance requirements. Replicating services at the edge is needed to meet demand, but balancing the load across multiple servers can be challenging due to diverse service costs, server and flow heterogeneity, and dynamic workload conditions. In this paper, we design and implement a model-based load balancer EdgeBalance for edge network data planes. EdgeBalance predicts the CPU demand of incoming traffic and adaptively distributes flows to servers to keep them evenly balanced. We overcome several challenges specific to network processing at the edge to improve throughput and latency over static load balancing and monitoring-based approaches.

1 Introduction

Edge datacenters running network services such as 5G, VPN, and broadband alongside 3rd party applications requiring low-latency [20] face many of the issues of a larger centralized datacenter such as multi-tenancy and workload dynamics. However, these challenges are worsened due to: (1) a network edge has fewer resources to begin with, and requires constant adjustment among tenants to meet their time-varying demands [15]; (2) a network edge runs computationally demanding applications and must meet strict throughput and latency requirements. Network functions (NFs) comprising a network service stretch performance of CPUs to their limits to support line rate performance [10], and are often deployed at the edge in part to reduce latency. Thus many applications running on the edge are expected to have real-time performance requirements, which requires resource allocation to provide stronger guarantees [23].

Existing work on resource management for network services has focused on two distinct parts of the problem. First, coarse-grained resource management is addressed using algorithms for placement of VMs and containers running network functions, e.g., E2 [18], and Stratos [8]. Second, fine-grained load management is achieved through load balancers that distribute work across service replicas. This can be achieved with the connection-level load balancers used for cloud servers, but as we show, such load balancers are not a good fit for the needs of NFs, especially at the edge.

Connection load balancers can use static or dynamic policies for dispatching connections. Many of today's cloud load balancers, Ananta [19], Maglev [7], adopt simple static policies. But, more sophisticated policies are needed such as treating “elephant" flows separately to avoid impacting performance for other “mice" flows at the edge. Further, connection load balancing at the network edge has several unique constraints compared to prior work on designing dynamic policies [2, 4, 5, 12, 14, 14]. (1) A connection's duration or its bandwidth cannot be estimated at the start. (2) A connection assigned to an NF cannot be migrated to (or restarted at) a different server. (3) The CPU use of NFs cannot be measured by the cloud infrastructure because high-performance NFs often use polling for network IO and report 100% CPU use on cores independent of the traffic they are processing [6]. (4) NFs (e.g., NATs) are often middleboxes that transform outgoing packets, yet affinity must be kept to ensure the return traffic also passes through the same function. These constraints necessitate a new load balancing approach.

Our primary contribution is the design, implementation and a preliminary evaluation of a *model-based* load balancing technique for edge network data planes. There are two components to this solution. First is a model to estimate the load of a network service on a core as a product of its per-packet processing cost and the rate of packets processed by the service on that core. Second is a local feedback-driven controller (specifically, a Proportional-Integral-Derivative, or a PID, controller [26]) that adapts load balancing weights if our estimated load on cores becomes unbalanced due to traffic changes, e.g., an arrival of an elephant flow or surge in traffic demand of a service. Importantly, neither the model nor the feedback controller requires load monitoring on NFs and uses mostly local information to create dynamic policies.

We implement our approach as a DPDK-based stateful load balancer named EdgeBalance. We evaluate EdgeBalance on the CloudLab testbed [22] for load balancing network services implemented in BESS [3]. Our results show that:

- EdgeBalance's network model can estimate CPU load with an absolute error of less than 5% for NFs with varying computation costs and at varying traffic loads.
- In a workload with elephant and mice flows, EdgeBalance achieves up to 50% higher throughput than the best static load balancing and achieves up to 1/3-rd the latency of a monitoring-based approach.
- For a time-varying workload with homogeneous flows, EdgeBalance achieves an equal load among servers in 1 sec which is several seconds faster than a static policy.

2 Why a new load balancer?

We explain why existing connection load balancers, in particular cloud load balancers [7, 16, 17, 19], are not sufficient to meet network edge load balancing needs.

Bidirectional affinity: A critical requirement for a network edge load balancer is to maintain affinity of a connection to an instance of a network service [19]. In fact, several cloud load balancers such as Ananta and Maglev provide affinity despite a changing pool of servers. However, they provide affinity only for an inbound connection to a server. Traffic sent by a server on the connection bypasses the load balancer for efficiency.

Thus while many cloud load balancers are able to perform optimizations such as bypassing the load balancer on the return path from a server, a network edge load balancer like EdgeBalance must ensure bi-directional flow affinity for correct behavior. Unfortunately, this eliminates the possibility of applying many existing load balancing frameworks in a network edge context.

Limitations of static load balancing: Many cloud load balancers support weighted load balancing across service instances. These weights are used to determine the fraction of new connections assigned to an instance. But, how these weights are to be set is often left unspecified [17, 19]. In practice, cloud operators can use simple static policies, e.g., weight of an instance is proportional to number of its cores. But, simplistic static policies can be highly sub-optimal, especially for heterogeneous flows, shown in evaluation section.

Challenges in dynamic load balancing: The above example shows that dynamic weight tuning is needed, but using dynamic weights has two main challenges. The first challenge is that of controller design. Dynamic control can be prone to oscillations and herd behavior if weights are tuned using stale traffic measurements or if the controller aggressively adjusts weights in response to the input [24]. Hence, designing and evaluating a stable controller for network services that balances controller responsiveness and stability is an important question. The second challenge is that of measuring load on network service instances. Weight adjustments depend on the current load of services, but high performance NFs are implemented using frameworks such as DPDK that perform polled network IO. Due to polling, they report 100% CPU utilization independent of the traffic they are processing. Thus inferring the load on NFs comprising a network service is the second important question.

3 EdgeBalance design

EdgeBalance is a dynamic, bidirectional load balancer for a network edge datacenter. To evenly balance the load across servers and cores, it adopts a model-based approach to estimate loads and applies a PID controller using local information to dispatch connections across network services.

Design goals: EdgeBalance aims to provide

- *Bidirectional affinity:* Provide affinity for inbound and outbound connections for NFs and other cloud applications, even when NFs modify packet headers.
- *Dynamic load balancing:* Dynamically equalize the load on all servers and cores to absorb any unexpected load on a server core.
- *Fast convergence:* Respond quickly to load imbalance due to flow skew, traffic fluctuations, etc..
- *High performance:* Achieve a high throughput in terms of packets and connections and add minimal latency.

Architecture overview: EdgeBalance comprises a data plane forwarder and control plane elements – policy, topology controller and monitor – shown in Figure 1. Its packet processing path goes through the forwarder, which can be replicated across multiple threads for scalability. The forwarder must efficiently redirect incoming packets to an edge server running the appropriate network service. A thread processes each packet in a "run to completion" manner by fetching the packet from a NIC RX queue, choosing a next-hop server, tracking the connection through a flow table, encapsulating the packet and delivering the packet into a NIC TX queue. Later, we will describe how the packet is encapsulated. To avoid contention between threads, each forwarder maintains its own statistics about the flows it processes. This data is then periodically aggregated by the monitoring component, which tracks statistics on a per-service basis. The topology controller tracks which services are active on which servers and cores and can start and stop additional replicas. Information from these components is fed to the policy component, which guides balancing decisions made by the forwarders. On every forwarder core, a garbage cleaner executes periodically to clean up the inactive

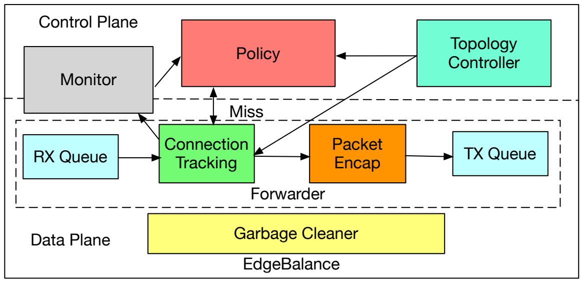


Figure 1: EdgeBalance architecture

flow entries from its flow table.

Bidirectional affinity: Figure 2 shows the sequence of nodes traversed by packets in both directions.

Processing at EdgeBalance: When a packet arrives (step 1), a datacenter router sends the incoming traffic from a client into EdgeBalance. Upon the first packet of a connection, EdgeBalance records the flow's 5-tuple of <src IP, dst IP, proto, src Port, dst Port> and its chosen network service <server ID, core ID, network service ID> in a flow table entry. Then, subsequent packets in this flow follow the same path. When a new flow in one direction arrives, EdgeBalance estimates the reverse flow information by swapping source IP, destination IP, source port and destination port. It then inserts a reverse flow entry with the same <server ID, core ID, network service ID> as the forward flow. For consistency, garbage cleaner will remove both of the flow entries from the flow table at the same time. Then, EdgeBalance encapsulates the chosen network service information including server ID, core ID, network service ID, and an update flag (set to 0) into a VXLAN header with the source mac address of the EdgeBalance server and destination mac address of the chosen server (step 2).

Processing at edge server: An edge server assists EdgeBalance in realizing bidirectional affinity. An edge server's data plane, called a *datapath node*, comprises a deparser module, a nexthop module and one or more network services implemented by NFs. The deparser module parses the outer VXLAN header and gets the core ID and service ID, and then delivers the packet to right core and network service. If an NF modifies the packet header, it will set the update flag in the packet meta data to 1. When nexthop module sees that the update flag is set, it resends it to EdgeBalance, which allows EdgeBalance to learn the network service mapping for the transformed packets (not shown in Figure 2). If no network service changes the packet header, the nexthop module will bypass EdgeBalance and send the packet towards the destination server (step 3).

Packets from server to client follow a similar procedure as shown in step 4-6. In particular, EdgeBalance uses the flow entry learned in steps 1-3 to send the packet to the correct instance of the network service (step 5) to achieve bidirectional affinity even if a network service modifies packet headers.

Model-based load estimation: The use of polling mode on datapath nodes makes it hard to measure the real CPU usage with traditional tools. Even if the datapath node can report its CPU usage to load balancer, it can easily send stale or noisy data, since in-network traffic changes rapidly. EdgeBalance takes another approach. If we know (1) the network services running on each core, (2) the per packet processing cost for a network service and (3) the number of packets for each network service, then we could predict the CPU usage at the load balancer instead of measuring it at datapath nodes as described next.

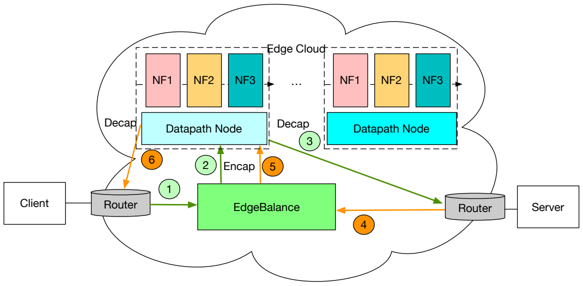


Figure 2: EdgeBalance packet path

For question (1), the topology controller has the knowledge of which cores a network service is running on. For question (2), the processing cost of a network service is affected by server heterogeneity. When a datapath node starts and network service services are deployed, datapath internally generates some UDP packets to go through each network service and then measures the processing cost and reports the tuple of <server ID, core id, service id, processing cost> to the monitor component on EdgeBalance. For question (3), from Figure 2, we can see that EdgeBalance handles every packet going through a network service. But, it needs to efficiently count the packet arrival rate for each service on a core. A naive way would be to aggregate the number of packets by stepping through the flow table. However, when the number of flows is huge, the aggregation time will dramatically increase and incur large prediction delay. Instead, since the number of cores and services are fixed, EdgeBalance maintains a three dimensional array of <server ID, core ID, service ID> to record the packet counts, which ensures that the CPU load can be predicted efficiently as follows:

$$Predict\_CPU_{ij} = \sum_{k=1}^{n} (Cost_{ijk} * Rate_{ijk})$$

$Predict\_CPU_{ij}$ is the predicted CPU for <server i, core j>. n is the total number of network services running on <server i, core j>. $Rate_{ijk}$ and $Cost_{ijk}$ are the packet rate and processing cost of network service k for <server i, core j>.

Dynamic load balancing via PID controller: EdgeBalance aims to evenly balance the load across servers and cores running network services. It does so using a PID controller to update the load balancing weight of each core at fixed intervals. Initally, the system sets an equal weight for all cores. The target for the controller is to set weights to ensure that the predicted CPU usage of a core $Predict\_CPU_{ij}$ remains equal to the average predicted CPU usage across across all cores. We apply the PID controller approach to compute the weight change diff_weight for each core at time intervals dt as follows:

```
err = kP * (predict_cpu - avg_predict_cpu) # P-term
err_sum += kI * err * dt # I-term
d_err = kD * (prev_err - err) * dt # D-term
prev_err = err
diff_weight = err + err_sum + d_err # PID output
```

We set the constants for proportional gain kP = 0.6, integral gain kI = 0.5 and derivative gain kD = 0.125 using experimental parameter tuning. In future, we plan to explore a reinforcement learning approach to set these parameters.

4 Evaluation

Setup: we use six servers on the NSF CloudLab testbed of "c220g5" type from the Wisconsin site. All of them have dual Intel(R) Xeon(R) Silver 4114 CPU @ 2.30GHz (2x10 cores), 64KB of L1 cache, 1024KB of L2 cache per core, and a shared 14080KB L3 cache, an Intel X710 10G Dual Port NIC and 200GB memory. Every server runs Ubuntu 14.04.1 with kernel 3.13.0-143-generic and uses Intel DPDK v18.02. We use both Pktgen-DPDK [21] and Cisco Trex [25] as traffic generators, and BESS [11] as our dataplane platform. We configure EdgeBalance to adjust weights every 30 ms in this experiment.

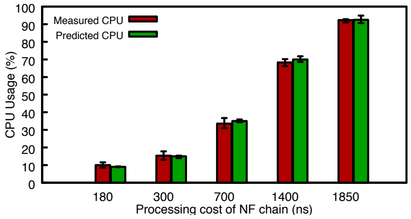


Figure 3: CPU usage prediction accuracy

CPU prediction accuracy: We first evaluate the accuracy of our CPU prediction model for NFs with different processing costs. We generate short flows (10 pkts per flow) with Trex and EdgeBalance sends all traffic to one core which runs a service chain of two NFs. The first NF rewrites a portion of each packet's body and has a fixed cost. The second NF has a variable amount of computation cost, which we adjust to observe its impact on overall CPU usage. Figure 3 shows that the average CPU calculated by the prediction model and the average measured CPU reported from datapath nodes have little difference, with an average absolute error of 1.8%. Next, we consider a more complex scenario with one lightweight and one heavyweight chain on the same core (not shown due to space constraints). As we increase the traffic rate while keeping the NF computation cost fixed, our model retains high accuracy, only experiencing greater than 5% error when the CPU usage passes 80% load; at this point, our model tends to overpredict the CPU needed, which is desirable since it results in a more conservative system. Next, we investigate if EdgeBalance is able to apply this model in improving end-user metrics such as throughput and latency.

Load balancing comparison – heterogeneous flows: For this experiment, we consider a workload that remains constant over time but exhibits heterogeneity in flow sizes. Specifically, a long-running elephant flow is generated by one traffic generator (Pktgen-DPDK), while a second traffic generator (Trex) generates mice flows with 10 packets per flow at 1 ms intervals. This workload is served by two datapath nodes of equal capacity that are running network service chains with the same processing cost of 700ns per packet.

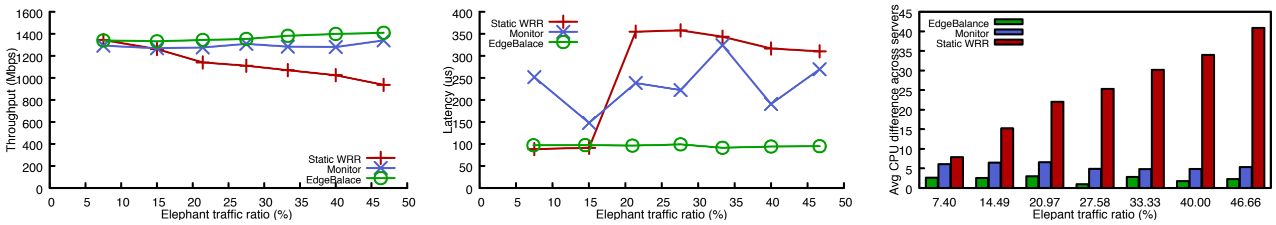
We compare EdgeBalance against **Static WRR**, which uses equal weights for both servers in this experiment. We note that equal weights are the best static weight setting for this experiment since CPU capacity and per-packet NF processing costs are identical for both servers. We also compare against **Monitor** – a variant of EdgeBalance that instruments NF code to monitor cycles spent by an NF in processing packets (vs. cycles spent in polling an empty NIC queue). It periodically (every 30 ms) reports CPU utilization as the fraction of cycles spent in processing packets, based on which our PID controller computes load balancing weights.

Our experiment evaluates the sensitivity of our findings to the percentage of elephant flow in the workload. We vary the fraction of traffic from the elephant flow from 7% to 47% keeping the total input traffic (in Mbps) constant. Figure 4a and Figure 4b respectively show the throughput and the latency achieved by all schemes. To explain the results in these two figures, Figure 4c shows the average absolute difference in CPU usage among the two servers for all schemes.

EdgeBalance achieves a nearly constant throughput and latency, which is independent of the fraction of traffic from the elephant flow. In Figure 4c, we find that, EdgeBalance sets the load balancing weights so that the difference in CPU usage among servers stays below 3%. This finding shows that EdgeBalance is able to mitigate the effect of the elephant flow on a server by assigning fewer new flows to that server.

Static WRR sees a reduction in throughput and an increase in latency upon increasing the fraction of traffic from the elephant flow. A higher fraction of traffic from the elephant flow hurts Static WRR's performance because it increases the imbalance in load among the two servers. The load difference increases to 41% for the heaviest elephant flow, due to which Static WRR has a 33% lower throughput than EdgeBalance, and a latency of more than 300 us. This experiment shows that simplistic statics policies do not perform well for heterogeneous workloads.

Monitor's throughput is close to EdgeBalance but its latency up to 3 times higher in this experiment. This scheme achieves a high throughput because on average, the two servers have a CPU usage difference of at most 7%. This



(a) Throughput

(b) Latency

(c) Average CPU usage difference

Figure 4: Varying fraction of elephant flow traffic

load difference is higher than that of EdgeBalance but is significantly better than that of Static WRR. But, Monitor has a high latency because it is prone to oscillation of load balancing weights it assigns to the two servers, partly due to the noise in measured CPU usage. These oscillations result in periods where packet queue lengths on NF nodes are higher than normal, which increases the latency for this scheme. In comparison, we find that the model-based approach taken by EdgeBalance results in more stable load balancing weights, which improves latency. While the oscillations for a monitor-based approach may be reduced if we increase the monitoring and weight adjustment interval significantly (say 1 sec), but it could hurt its responsiveness for dynamic workloads, a scenario we consider next.

We also evaluate how each approach (EdgeBalance, static WRR, monitor-based approach) behaves when keeping the traffic ratio from the elephant flow to be constant, but gradually increase the workload intensity for mice and elephant flows, not shown because of the limited space.

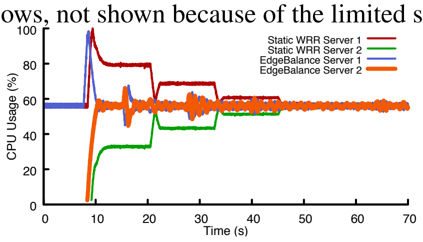


Figure 5: CPU usage for a dynamic workload

Load balancing comparison – dynamic workload: We consider a time-varying workload and demonstrate how quickly CPU usage converges upon a workload change. We begin with one datapath node (server 1) with NF chain processing cost 700ns, and send traffic (15 flows/sec) into this node through the load balancer. If the average CPU usage is over 95%, the load balancer will mark that datapath node as overloaded, which requires additional servers to handle the traffic. At 9 seconds, we double the traffic, causing a new datapath node to be added into the system. Since Static WRR dispatches the new flows in a round robin manner, it takes a much longer time to converge to an equal load, which is about 35 sec. On the other hand, EdgeBalance can predict and be aware of the load on server 1, it can assign more flows into the newly added server right after it is added, so the load on the two servers equalizes in close to a second as shown in Figure 5. In conclusion, EdgeBalance appears to achieve its design goals for dynamic workloads as well.

5 Related work

Cloud load balancers: A primary focus of cloud load balancers implemented in software is to support a scale-out design [7, 19]. Extending EdgeBalance to support a scale-out design is an important area of future work. To reduce the server CPU cores for load balancing, two kinds of approaches have been considered. First, offload stateful load balancing to switch ASICs as in SilkRoad [16]. However, these approaches have limited flow scalability due to limited flow table sizes on ASICs. Second, use a stateless load balancer but instead rely on connection state at servers to provide affinity [1, 17]. However, these approaches require extensive support in server applications.

Load balancing algorithms: Several papers [2, 4, 5] have used layer-7 information such as URLs or request sizes for load balancing. But, an edge network load balancer operates on layer-3/4 information and may not have layer-7 layer information. Dynamic load balancing based on server load has been explored by Network Dispatcher [14]. However, a server's CPU load is difficult to estimate for polled IO NFs. Further, their paper does not provide sufficient details to implement its dynamic algorithm. Some papers have proposed migrating a connection to another server [13] or restarting a connection on another server [12]. While restarting a flow for load balancing is not practical for network services, connection migration may be supported using techniques such as OpenNF [9]. we plan to explore how connection migration can enable better load balancing in the context of NFs.

6 Conclusions

We have shown that a stateful load balancer can meet the bidirectional affinity requirements that are specific to a network edge cloud. Further, we propose a model-based load balancing approach combines information already present at a stateful load balancer with a minimal number of parameters that model the processing cost of network services. Our initial experience suggests that this model has low prediction error, it improves performance when dealing with skewed flows and services with heterogeneous processing costs, and it responds quickly to changes in workload.

7 Discussion

This paper has shown promising preliminary results and the potential of using PID controller and CPU prediction models to dynamically distribute flows across network service chains in an edge environment. We will discuss key challenges for edge load balancing and look for feedback on our future work, including:

Edge load balancing: We have argued that the edge environment demands a new type of load balancer, particularly when the edge is being used for network functions. We seek feedback on what load balancing challenges attendees see as the most pressing at the edge, such as balancing the complexity of the load balancer's policies versus the overhead they incur.

Overhead and cost of state: Despite the trend towards stateless services, we argue that EdgeBalance needs to track flow state for flow affinity in case the backend server pool changes dynamically (addition or removal). We expect the discussion can focus in part on when stateful load balancers are required, and how stateful systems can be designed to achieve high scale.

Prediction robustness: In our preliminary experiments, we evaluate the accuracy of CPU prediction model by varying the length of service chains and computation cost on Intel X86 platform. We plan to evaluate the robustness of the model by conducting experiments on other hardware platforms with other common network functions such as NATs, IDSes, and stateful firewalls.

Cache interference: To efficiently use resources, multiple NFs can be consolidated on the same server. In such a deployment, NFs contend for resources such as LLC (last level cache). For stateful NFs, the processing cost of a packet may vary based on cache pollution level. We plan to investigate and incorporate the cache interference in the CPU prediction model.

Scale-up and scale-out scalability: Scalability is a key factor for a load balancer. The load balancer itself should add minimum latency to the users' traffic. EdgeBalance leverages lockless and per core data structure to efficiently scale up across the cores. We are measuring the maximum throughput by varying the number of cores. In a large scale network deployment, multiple load balancers are required to avoid a bottleneck at the load balancer itself. We are investigating how to exchange minimum information across load balancers to efficiently balance the flows for a large scale deployment.

Stateful NFs and real traces: Multiple network functions are stateful, which means they need to manage and maintain per-flow state internally. In future, we will use advanced stateful network functions (e.g., a stateful IDS) in conjuction with real network edge traces to evaluate EdgeBalance.

Acknowledgements: This work was supported in part by NSF grants CNS-1253575 and CNS-1763548.

References

[1] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. Balancing on the edge: Transport affinity without network state. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 111–124, 2018.

[2] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2000.

[3] BESS. Berkeley Extensible Software Switch. https://github.com/NetSys/bess. Accessed: 2018-06-06.

[4] Emiliano Casalicchio and Michele Colajanni. A client-aware dispatching algorithm for web clusters providing multiple services. *WWW*, 1:535–544, 2001.

[5] Mark E. Crovella, Mor Harchol-Balter, and Cristina D. Murta. Task assignment in a distributed system: Improving performance by unbalancing load, 1997.

[6] DPDK. DPDK data plane development kit. https://www.dpdk.org. Accessed: 2018-06-06.

[7] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 523–535, 2016.

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

[9] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 163–174. ACM, 2014.

[10] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network functions virtualization: Challenges and opportunities for innovations, 2015.

[11] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[12] Mor Harchol-Balter. Task assignment with unknown duration. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 214–224. IEEE, 2000.

[13] Mor Harchol-Balter and Allen B Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems (TOCS)*, 15(3):253–285, 1997.

[14] Guerney DH Hunt, Germán S Goldszmidt, Richard P King, and Rajat Mukherjee. Network dispatcher: A connection router for scalable internet services. *Computer Networks and ISDN Systems*, 30(1-7):347–357, 1998.

[15] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 286–299. IEEE, 2018.

[16] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, 2017.

[17] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 125–139, 2018.

[18] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.

[19] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 207–218. ACM, 2013.

[20] Larry Peterson, Ali Al-Shabibi, Tom Anshutz, Scott Baker, Andy Bavier, Saurav Das, Jonathan Hart, Guru Palukar, and William Snow. Central office re-architected as a data center. *IEEE Communications Magazine*, 54(10):96–101, 2016.

[21] Pktgen-DPDK. Traffic generator: Pktgen-dpdk. https://git.dpdk.org/apps/pktgen-dpdk/.

[22] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[23] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[24] Abhigyan Sharma, Arun Venkataramani, and Antonio A Rocha. Pros & cons of model-based bandwidth control for client-assisted content delivery. In *2014 sixth international conference on communication systems and networks (COMSNETS)*, pages 1–8. IEEE, 2014.

[25] Trex. Cisco traffic generator: Trex. https://github.com/cisco-system-traffic-generator/trex-core.

[26] Wikipedia. PID Contoller. https://en.wikipedia.org/wiki/PID_controller. Accessed: 2019-11-11.