IDIO: Orchestrating Inbound Network Data on Server Processors

Mohammad Alian, Jongmin Shin, Ki-Dong Kang, Ren Wang, Alexandros Daglis, Daehoon Kim, and Nam Sung Kim

Abstract—Network bandwidth demand in datacenters is doubling every 12 to 15 months. In response to this demand, high-bandwidth network interface cards, each capable of transferring 100s of Gigabits of data per second, are making inroads into the servers of next-generation datacenters. Such unprecedented data delivery rates on server endpoints raise new challenges, as inbound network traffic placement decisions within the memory hierarchy have a direct impact on end-to-end performance. Modern server-class Intel processors leverage DDIO technology to steer all inbound network data into the last-level cache (LLC), regardless of the network traffic's nature. This static data placement policy is suboptimal, both from a performance and an energy efficiency standpoint. In this work, we design IDIO, a framework that—unlike DDIO—dynamically decides where to place inbound network traffic within a server's multi-level memory hierarchy. IDIO dynamically monitors system behavior and distinguishes between different traffic classes to determine and periodically re-evaluate the best placement location for each flow: LLC, mid-level (L2) cache or DRAM. Our results show that IDIO increases a server's maximum sustainable load by up to ~33.3% across various network functions.

Index Terms—Cache, network, data direct I/O, datacenters

1 Introduction

The evolution of networking technology has led to network bandwidth in servers approaching memory bandwidth and proper handling of network traffic in memory hierarchy can noticeably affect overall performance. Data Direct I/O (DDIO) technology [1], [5]—introduced to address this challenge—allows the NIC to directly write data incoming from the network in LLC, reducing memory bandwidth utilization and drastically improving the performance of latency-sensitive applications such as Virtual Network Functions (VNFs). By default, DDIO uses two of the LLC's ways for all incoming network traffic, which we find generally works well, but still leaves significant performance gains on the table.

We identify a number of DDIO shortcomings in multi-tenant server environments, arising from the fact that all applications using the network share the same limited number of LLC ways. Even though the number of ways allocated for DDIO is configurable, they still have to be statically defined. First, there is no number of DDIO-allocated ways that is universally optimal [8]. Second, as DDIO ways are shared by all co-running applications, interference can significantly impact the performance of latency-sensitive applications, especially as the number of applications using DDIO approaches the LLC's limited associativity. The latest generation of Intel server processors, which reduces the LLC associativity from 20 to 11 ways, further exacerbates this challenge.

Based on these observations, we argue that there is a need for dynamic DDIO policies with differentiated network traffic placement decisions among applications. We propose **I**ntelligent **D**irect **I/O** (IDIO) technology, a next-generation DDIO mechanism that determines the placement of incoming data on per-application basis. Motivated by the latest evolution of Intel servers' cache hierarchy, which shrinks the LLC but quadruples each core's private MLC, IDIO extends network data placement capability from the LLC to each core's MLC.

2 Background and Motivation

2.1 DDIO Technology

As high-speed networking devices make inroads into the datacenter market, the classical DMA approach of writing network data into DRAM becomes a performance bottleneck for network-intensive applications. DCA [5] has been proposed to alleviate this problem and improve network performance, by bypassing DRAM and writing/reading data directly to/from the cache. Most of modern Intel's Xeon processors employ a DCA implementation called Intel Data Direct I/O (DDIO) technology [1]. DDIO directly uses the CPU's LLC for data communication between I/O devices and processors instead of detouring to DRAM, reducing both access latency and memory bandwidth utilization considerably.

Without DDIO, a packet transmission (i.e., TX) operation evicts the packet's cachelines after forwarding data to the I/O adaptor, under the legacy assumption that I/O operations are slow and infrequent. In contrast, DDIO does not evict the cachelines corresponding to data after TX, based on the insight that the same cachelines will be frequently reused with the high-speed I/O technologies. Consequently, DDIO reduces the memory access latency while serving much more read requests from the CPU and the I/O adaptors without DRAM accesses. For packet reception (i.e., RX), the I/O device directly writes data to the LLC without transferring data to DRAM. Depending on the presence of the target address in the LLC, DDIO either write-allocates or write-updates data delivered from the I/O devices in the LLC. As a result, the CPU can access the data arrived from I/O devices without detouring to DRAM, leading, again, to both reduced latency and memory bandwidth usage.

DDIO restricts the number of LLC ways it uses, preserving LLC capacity to avoid performance degradation of the running applications due to increased LLC pressure. The default number of LLC ways allocated for DDIO varies by platform, but typically accounts for 10–20 percent of the total LLC capacity. Furthermore, a system leveraging DDIO relies on constructive sharing of the LLC between the running applications and ongoing network traffic. The LLC size and associativity are therefore parameters that play a key role in DDIO's effectiveness.

Starting with the Skylake microarchitecture, Xeon processors, used predominantly in server platforms, introduced drastic changes to the cache hierarchy. Compared to its predecessor, Broadwell, the Skylake microarchitecture (1) reduced per-core LLC size from 2.5 to 1.375 MB; (2) changed LLC's inclusion policy from inclusive to non-inclusive; (3) reduced the LLC's associativity from 20- to 11-way; and (4) quadrupled the size of all private L2 caches to 1 MB, to compensate for the downscaled LLC. These changes, along with continuous bandwidth increase of network devices, have significant implications on the performance and effectiveness of DDIO technology. Reduced per-core LLC capacity means that DDIO technology has less cache space to store ever-increasing

- *Mohammad Alian is with the Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS 66045 USA. E-mail: alian@ku.edu.*
- *Jongmin Shin, Ki-Dong Kang, and Daehoon Kim are with the Department of Information and Communication Engineering, DGIST, Daegu 42988, South Korea. E-mail: {jshin, kd_kang, dkim}@dgist.ac.kr.*
- *Ren Wang is with Intel Labs, Aloha, OR 97078 USA. E-mail: ren.wang@intel.com.*
- *Alexandros Daglis is with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332 USA. E-mail: alexandros.daglis@cc.gatech.edu.*
- *Nam Sung Kim is with the Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Champaign, IL 61820 USA. E-mail: nskim@illinois.edu.*

Manuscript received 11 Nov. 2020; revised 2 Dec. 2020; accepted 9 Dec. 2020. Date of publication 15 Dec. 2020; date of current version 5 Feb. 2021.
(Corresponding authors: Mohammad Alian and Daehoon Kim.)
Digital Object Identifier no. 10.1109/LCA.2020.3044923

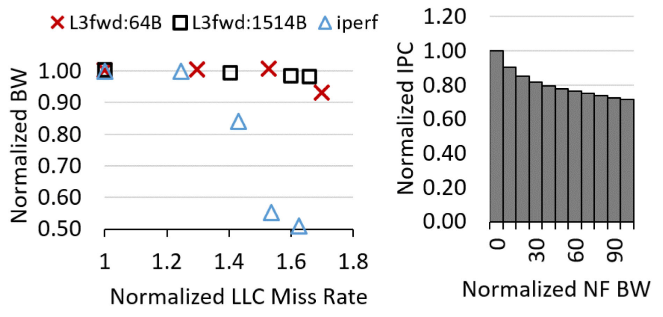

Fig. 1. Sensitivity analysis of network-intensive and memory-intensive applications to LLC interference.

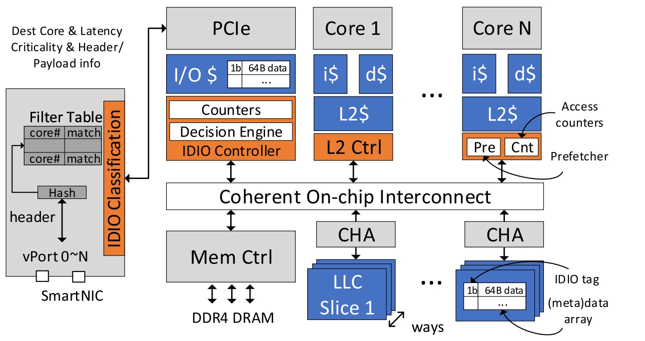

Fig. 2. IDIO overview.

inbound data from the network. This resource balance shift amplifies the DMA leakage problem—network packets written in the LLC by DDIO are evicted before their consumption by the application—observed even on Broadwell when co-running several network-intensive applications on the same server [8].

2.2 DDIO Limitations

In this section, we study DDIO's memory and cache behavior to demonstrate its shortcomings and highlight improvement opportunities targeted by IDIO. We focus our study on VNFs, which are widely deployed in datacenter settings. Specifically, we use two representative micro-NFs for our demonstrations: L3 forwarding function implemented on top of DPDK (i.e., L3fwd:SIZE where "SIZE" is the size of packets sent by the load generator) [3] and iperf3 running on Linux kernel [2]. Since the L3fwd application, by default, only touches the header of received packets, we modified it to touch each packet's entire payload to have a more general NF memory access pattern. We run an NF and iperf on two cores and a single core, respectively, and an LLC antagonist on the server's remaining cores. The LLC antagonist pollutes the LLC by writing into a buffer and we control its intensity via adjusting the number of cores. The experimental methodology is detailed in Section 4.

Fig. 1(left) illustrates how memory interference at LLC, introduced by an application-antagonist, affects the performance of co-running NFs. The X-axis is the LLC miss rate normalized to that of running the NF without the LLC antagonist. The Y-axis plots the NF's maximum sustainable bandwidth normalized to that of running the NF without the LLC antagonist. Fig. 1(right) plots mcf's (a memory-intensive workload in SPECint benchmark suite) IPC when co-located with L3fwd:1518B. The NF here plays the role of the LLC antagonist and causes interference for mcf. The X-axis of Fig. 1(right) is the normalized bandwidth of the NF.

Prior work has extensively studied the sensitivity of CPU applications [6] and NFs [4] to LLC interference. The key takeaway points from our simple analysis are three. First, NFs with large packet sizes are less sensitive to LLC interference. As Fig. 1(left) shows, the bandwidth of L3fwd:1518B is insensitive to LLC interference, while L3fwd:64B's bandwidth drops by $\sim$7% at 69 percent normalized LLC miss rate. Second, different NFs have different sensitivity to LLC contention. For instance, in our setup, iperf is more sensitive than DPDK's L3fwd and its bandwidth reduces by $\sim$48% at 1.62$\times$ normalized LLC miss rate. Lastly, interference between NFs and CPU applications goes both ways: a co-running CPU application can be detrimental to an NF's performance and vice versa (see Fig. 1(right)).

In this work, we propose IDIO that leverages MLCs for better inter-thread network traffic isolation and IO cache capacity boost. Recent work proposed selective network data steering to private L1 data caches [7]; we expect MLC steering to capture similar latency and traffic isolation benefits while mitigating cache thrashing concerns. The rest of this paper introduces IDIO, offering solutions to the shortcomings of the DDIO.

3 IDIO Architecture

IDIO makes dynamic decisions about each packet's placement in the memory hierarchy, improving DDIO's flexibility in two dimensions. First, instead of applying a fixed policy for all incoming packets, IDIO applies differential treatment based on the per-packet quality of service targets. Second, IDIO considers three potential placements in the memory hierarchy: DRAM, LLC, and MLC. IDIO continuously monitors network data movement in the memory hierarchy and periodically adapts its placement decisions per traffic flow accordingly.

Fig. 2 shows IDIO's high-level architecture. IDIO introduces two new components: a packet classifier and the IDIO controller. In addition, we enhance tag arrays in the cache hierarchy to enable network data tracking, essential for adaptive placement decisions, and extend the MLC controllers to support MLC data injection. The packet classifier resides in the NIC and determines each incoming packet's priority class, as well as destination core. The on-chip IDIO controller constantly collects and monitors on-chip statistics to adaptively determine the best placement for each traffic flow. In the rest of this section, we detail the role and functionality of each component, as well as the dynamic policy governing the IDIO controller's decisions.

3.1 IDIO Packet Classifier

IDIO implements a packet classification logic inside the NIC to identify each incoming packet's latency criticality by monitoring their DSCP field. The DSCP (also known as ToS—Type of Service) field can be set by the `setsockopt` function for each socket connection and updated on the fly. DSCP can be used to distinguish among requests with different latency criticality. IDIO distinguishes between two packet priority classes, batch and latency critical, to dynamically identify the best data placement policy for each packet.

In addition, the packet classifier also identifies which core each packet is destined to, which is necessary information to enable effective MLC steering. IDIO's packet classifier builds on existing NIC support to determine each packet's destination core. We leverage SR-IOV and Ethernet Flow Director to create several virtual NIC ports (vPort) and pin them to network sockets created on each core using Application Device Queue (ADQ) [9]. To transfer the metadata extracted by the packet classifier on the NIC to the on-chip IDIO controller, we embed them within each DMA request by leveraging the reserved bits inside the PCIe's Transaction Layer Packet (TLP) headers.

3.2 IDIO Controller

We have one IDIO controller per PCIe root complex and each IDIO controller controls the data flowing from the vPorts connected to its PCIe bus. The role of the IDIO controller is to dynamically decide where to place the inbound network traffic: DRAM, LLC, or a core's MLC. The *Decision Engine* in Fig. 2 is the controller's entity making these placement decisions. The IDIO controller maintains a set of counters per physical core *i*: *dataRXCounter[i]* counts the amount of network data (cachelines) destined for core *i*; *prefetchedMLCCounter[i]* counts the amount of network data that is prefetched to core *i*'s MLC; *DRAMWritebacks [i]* tracks the amount of network data destined for core *i* that is directly written back to DRAM; *IOAccessCounter[i]* counts the amount of network data read through DMA and hits inside the LLC. Network data residing in the LLC are distinguished by an added IDIO bit.

The Decision Engine determines data placement after taking several inputs into account: the aforementioned per-core counters, additional counters inside each MLC controller (Section 3.3), as well as criticality and header/payload information embedded inside each DMA request. We detail the IDIO controller's data and control plane in Section 3.5, after introducing all the remaining supporting hardware components.

3.3 MLC Controller

The MLC (L2 cache) controller implements a prefetcher logic that takes prefetch hints from the Decision Engine and sends prefetch requests to the LLC. These prefetch hints are the mechanism IDIO employs to steer incoming network data to MLCs, when deemed appropriate. Each MLC controller maintains two counters to keep track of the number of missed accesses to packet payloads (*coreAccessCounter*), and number of hits to prefetched packet payloads (*hitMLCCounter*).

3.4 LLC and MLC Extensions

IDIO extends the LLC and MLC tag arrays with an *IDIOtag* bit per cacheline that indicates whether the corresponding cacheline stores a network packet's payload. All packet payloads DMA'ed over PCIe and through the IDIO controller into the cache hierarchy have the *IDIObit* of their corresponding cacheline set. The IDIO tag's value is set in the LLC when the cacheline is first written by the IDIO controller, and is carried into the MLC if the cacheline is moved from the LLC to the MLC by an MLC controller's prefetch request. The MLC controller uses the IDIO tag to distinguish cachelines brought to MLC by its own prefetch decisions as opposed to demand accesses from the CPU.

3.5 Dynamic Data Placement Policy

The IDIO controller determines the destination of each DMA request by taking three inputs into account: (i) data criticality (DMA write request classified as either "batch" or "latency critical"); (ii) the amount of untouched network data inside the cache hierarchy; and (iii) whether the DMA request contains a packet header or payload.

MLC data placement can be attractive for a couple of reasons, especially for the latest Intel Xeon Scalable processors, which scaled down the LLC size and quadrupled the MLC size, resulting in comparable LLC and MLC sizes per core. Because the LLC is non-inclusive of the MLC, injecting network data directly to MLC as well can approximately double the space that can be used for network data. In addition, MLC's lower ($\sim$4 ns versus 20 ns) and more predictable (private access versus shared NUCA) access latency can improve the performance of network-intensive latency-sensitive applications.

MLC data injection is only constructive if the core consumes the injected data before it is evicted from the MLC. The size of a network packet header is less than 64 Bytes and thus fits in one cacheline of typical size. Considering a network ring buffer size of 1,024, the maximum capacity that is occupied by injecting packet headers to MLC is 64 KB, which is $\sim$6% of a 1 MB MLC. Due to the moderate capacity requirement and latency criticality of headers, IDIO always injects the header of received packets to MLC. This decision also resonates well with demanding NF applications that most benefit from DDIO, as most NFs only operate on the header of received packets without touching the payload.

Each DMA request carries key metadata set by the on-NIC IDIO packet classifier (Section 3.1): *isLatencyCritical*, *isHeader*, and *destCore*, which indicate if the DMA request belongs to a latency critical application, if it contains a packet header, and the target physical core for the received packet, respectively. The IDIO controller keeps a two-bit *status* register for each physical core that indicates the memory-level destination (MLC, LLC, or DRAM) for incoming DMA requests carrying packets destined for that core. If *status[destCore]* is MLC, then IDIO sends prefetch hints to *destCore*'s MLC. If *status[destCore]* is LLC or the DMA request is latency critical, IDIO write-allocates the data inside the LLC. Finally, if *status[destCore]* is DRAM, data is directly DMA'ed into DRAM, bypassing the LLC.

The IDIO controller resets the value of *status* registers every 1 ms. Each core's *status* register is initialized to LLC. The IDIO controller calculates the amount of untouched prefetched data in each MLC by reading the *hitMLCCounter* value from each MLC controller and subtracting it from *prefetchedMLCCounter*. If this number exceeds a threshold, IDIO changes that core's *status* from MLC to LLC. We set the default value of that threshold to 10 percent of the MLC's capacity.

Our experimental results show that the average depth of an NF's circular buffer is less than 30 packets across different packet sizes. Based on this observation, we anticipate that in the common case, IDIO will always inject received network data to the MLC regardless of the application type since the descriptor ring depth in a stable state is shallow. Thus, IDIO's default policy is to keep all the on-the-fly packets inside MLC with moderate MLC capacity overhead. IDIO only sends received packets to LLC (or DRAM) in special situations where the CPU does not use packets' payload (e.g., an IP forwarding NF) or there is a temporal spike in the network load and/or core's service time (due to page faults, cache misses, interrupts, etc.).

To assess the benefit of caching packet payloads, IDIO calculates the amount of data that is sent to the processor (not DRAM) but not touched by the destination core or a network device. If *dataUsage* is low (less than 10 percent of the data received from the network), then it is not beneficial to place data inside LLC (neither MLC, obviously) and *status* is changed to DRAM. If *dataUsage* exceeds 90 percent, IDIO changes *status* to MLC again. High *dataUsage* suggests that injecting packet payloads to MLC can increase application performance. To ensure that IDIO doesn't get stuck into sending payloads to DRAM if the application transitions to a new phase where it benefits from placing payloads inside LLC or even MLC, IDIO opportunistically sets *status* to LLC after sending payloads to DRAM for 10 ms.

4 Evaluation

Methodology. For the sensitivity analysis in Fig. 1, we use two servers each equipped with an Intel Xeon Gold 6134 processor with 8 cores (16 threads), 24.75 MB LLC, and two DDR4-2,400 memory channels. The servers are connected to each other using four dual-port 10 GbE Intel NICs. We use SR-IOV to run multiple DPDK applications (NFs) on one server. On the second server, we deploy several instances of a DPDK packet generator (*pktgen*) application that send requests to the first server's NFs.

We implement IDIO in gem5 bare-metal full system simulator and deploy a minimal operating system that performs memory allocation, synchronization primitives, and networking. We simulate a scaled-down CPU, featuring two cores and a proportionally downsized LLC, provisioning 1.8 MB/core for a total of 3.6 MB and 16 ways associativity. This setup is sufficient to study the effects of in-bound network packet placement within the cache

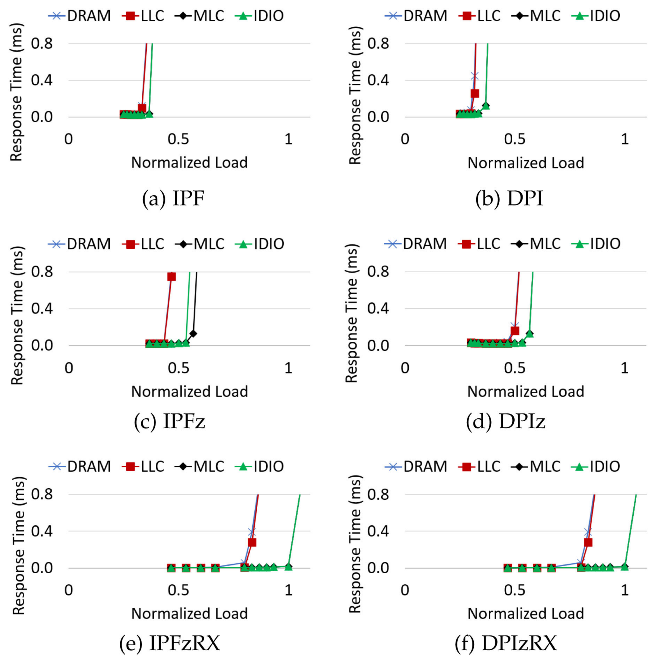

Fig. 3. 99th percentile response time of NFs using DDIO and IDIO. Load is normalized to MSL of the *zRX NF versions.

TABLE 1
iperf and Memcached Performance Results

benchmark	DRAM	LLC	MLC	IDIO
iperf normalized bandwidth	0.68	0.66	1	0.96
memcached normalized QPS	0.87	0.95	1	0.99

hierarchy, while taking into account the effects of interference between co-located applications with different characteristics. We plug a load generator into the gem5's NIC model that generates requests following exponential inter-arrival time distribution and configurable burst size.

We evaluate IDIO using six NFs: (i) IP forwarding (IPF) receives a request from the network, copies the packet to a new buffer, performs murmur hash on the *header*, and transmits the packet over the network; (ii) IPF with zero copy (IPFz) does not copy the received packet into a new buffer; (iii) IPFz without packet transmission (IPFzRX) terminates the request's processing after consuming the received data inside the NF; (iv) Deep Packet Inspection (DPI) receives a request from the network, copies the packet to a new buffer, performs murmur hash on the *entire* packet, and transmits the packet over the network; (v) DPI with zero copy (DPIz); and (vi) DPIz without packet transmission (DPIzRX). These micro-NFs model the general behavior of the majority of NFs running in datacenters. Furthermore, we also evaluate IDIO using iperf and memcached.

Experimental Results. We compare the performance of IDIO against inbound network data placement inside DRAM, LLC, and MLC, which we refer to as *static* configurations. Fig. 3 shows the 99th percentile response time of NFs running on a server deploying three static configurations (DRAM, LLC, MLC) and IDIO. Although baseline DDIO (LLC configuration) technology is effective in significantly reducing memory bandwidth utilization, we observe that it has very little impact on the tail latency of NFs running on an isolated server. On the other hand, MLC and IDIO are very effective in reducing tail latency and improving the server's Maximum Sustainable Load (MSL). We define MSL as the load level at the knee of 99th percentile response time versus load graph. Across different NFs, IDIO and MLC both increase MSL by 20.0, 30.8, 28.0, 33.3, 14.3, and 27.9 percent for IPF, IPFz, IPFzRX, DPI, DPIz, and DPIzRX, respectively.

Table 1 compares iperf bandwidth and memcached queries per second (QPS), normalized to MLC, for different network data placement policies. IDIO clearly outperforms LLC and DRAM placement and is comparable to MLC placement, performing within 4 and 1 percent for iperf and memcached, respectively. Both Fig. 3 and Table 1 seem to indicate that MLC forwarding is always a good idea. However, depending on the NF's memory access pattern, we expect static MLC forwarding to result in MLC thrashing, on-chip interconnect interference, and increased power consumption. As future work, we aim to study more complex NF and CPU application scenarios under static DDIO and IDIO configurations.

Acknowledgment

This work was supported in part by an US National Science Foundation under grant CNS-1705047 and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) under grant NRF-2020R1C1C1013315. Mohammad Alian and Jongmin Shin are co-first authors.

References

[1] Intel data direct I/O technology (Intel DDIO): A primer. Technical report.
[2] iPerf: The ultimate speed test tool for TCP, UDP and SCTP.
[3] L3 forwarding with access control sample application. [Online]. Available: https://doc.dpdk.org/guides/sample_app_ug/l3_forward_access_ctrl.html
[4] R. Durner, C. Sieber, and W. Kellerer, "Towards reducing last-level-cache interference of co-located virtual network functions," in *Proc. 28th Int. Conf. Comput. Commun. Netw.*, 2019, pp. 1–9.
[5] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct cache access for high bandwidth network I/O," in *Proc. 32nd Int. Symp. Comput. Archit.*, 2005, pp. 50–59.
[6] D. Lo *et al.*, "Improving resource efficiency at scale with Heracles," *ACM Trans. Comput. Syst.*, vol. 34, no. 2, pp. 1–33, 2016.
[7] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis, "The NEBULA RPC-optimized architecture," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 199–212.
[8] A. Tootoonchian *et al.*, "ResQ: Enabling SLOs in network function virtualization," in *Proc. 15th USENIX Conf. Netw. Syst. Des. Implement.*, 2018, pp. 283–297.
[9] A. Vasudevan, "Intel ethernet 800 series with application device queue (ADQ)," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/network-connectivity/800-series-adq-technology.html