Dynamic Load Balancer in Intel Xeon Scalable Processor: Performance Analyses, Enhancements, and Guidelines

Jiaqi Lou
University of Illinois Urbana-Champaign
Urbana, IL, USA
jiaqil6@illinois.edu

Srikar Vanavasam
University of Illinois Urbana-Champaign
Urbana, IL, USA
srikarv2@illinois.edu

Yifan Yuan
Meta
Menlo Park, CA, USA
yifanyuan@meta.com

Ren Wang
Intel Labs
Portland, OR, USA
ren.wang@intel.com

Nam Sung Kim
University of Illinois Urbana-Champaign
Urbana, IL, USA
nskim@illinois.edu

Abstract

The rapid increase in inter-host networking speed has challenged host processing capabilities, as bursty traffic and uneven load distribution among host CPU cores give rise to excessive queuing delays and service latency variances. To cost-efficiently tackle this challenge, the latest Intel Xeon Scalable Processor has integrated an on-chip accelerator, named Dynamic Load Balancer (DLB). It consists of hardware queues, arbiters, and priority-based Quality of Service (QoS) features to maximize load-balancing performance while minimizing host CPU cycle consumption. In this work, we first compare the performance of DLB against popular software-based load balancers with a microbenchmark suite, demonstrating that DLB significantly outperforms them. Yet, DLB still consumes a significant number of host CPU cycles to prepare and enqueue work descriptors for received packets. Second, to eliminate the consumption of host CPU cycles in load balancing, we propose a system architecture/software co-design solution, AccDirect. Specifically, AccDirect leverages the Peer-to-Peer (P2P) communication capability of PCIe devices to enable a direct communication path between DLB and NIC, through which NIC directly enqueues the work descriptors. Our evaluation shows that AccDirect-based DLB offers practically the same performance as conventional DLB while reducing the system-wide power consumption of host CPU by 10%. Compared to the throughput of a commodity hardware-based load balancer for an end-to-end application, that of AccDirect-based DLB is 14–50% higher with comparable p99 latency. Lastly, we provide guidelines to make the best use of DLB, which is riddled with a vast configuration space and advanced features, after conducting a comprehensive evaluation.

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731026

CCS Concepts

• **Computer systems organization** → *Architectures*; • **Hardware** → **Networking hardware**; • **General and reference** → **Empirical studies**; **Measurement**; **Performance**; **Design**.

Keywords

Dynamic Load Balancer (DLB), On-chip Accelerator, Accelerator Chaining, P2P Communication

ACM Reference Format:
Jiaqi Lou, Srikar Vanavasam, Yifan Yuan, Ren Wang, Nam Sung Kim. 2025. Dynamic Load Balancer in Intel Xeon Scalable Processor: Performance Analyses, Enhancements, and Guidelines. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731026

1 Introduction

Modern network technologies are advancing rapidly, with cloud service providers and datacenters increasingly deploying high-speed NICs that deliver line rates ranging from 100Gbps to 400Gbps [34, 35]. This places a growing demand on more powerful CPUs to process all network packets. For example, a CPU connected to a 100Gbps NIC must process ~200 Million Packets Per Second (MPPS) when receiving 64B packets at the line rate, which exceeds the processing capability of any single CPU core. To tackle this challenge, recent work [12, 23, 30, 45] has explored software-based load balancers that efficiently distribute packets across multiple CPU cores. Although they offer flexible load balancing to meet various Quality of Service (QoS) requirements, they require at least one dedicated CPU core to monitor the load of individual worker cores and make load-balancing decisions. When a single CPU core is used for centralized load balancing, throughput becomes bottlenecked by the processing capability of that CPU core. For instance, the software-based DPDK packet distributor [9], relying on a single CPU core, can support a maximum load-balancing throughput of only 30MPPS. While multiple CPU cores can be utilized for distributed load balancing, the high overhead caused by synchronization and lock contention among these cores becomes a bottleneck [26].

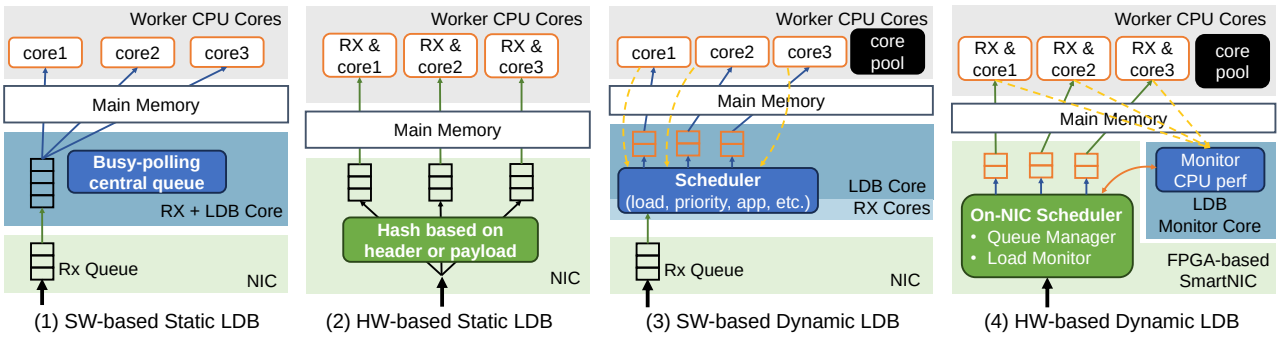


Figure 1: Software- and hardware-based intra-host load balancers.

Alternatively, hardware-based load balancers, including both static and dynamic ones implemented in NICs, can be employed to deliver significantly higher throughput and lower overhead for load balancing. The static load balancers, such as Receive Side Scaling (RSS) [32] and Intel Flow Director [21] have already been implemented in commodity NICs. They can eliminate the load-balancing overhead on CPUs without requiring modifications to applications or significant changes to NICs. However, by implementing only simple load balancing mechanisms, they are not effective, especially when network flow distributions are skewed and/or service times of packets are uneven [56]. In contrast, the dynamic load balancers do not suffer from these limitations because they implement more advanced load-balancing mechanisms in SmartNICs (SNICs) [16, 28, 50]. Yet, they must frequently collect performance statistics from individual host CPU cores and send them to the SNIC through the slow PCIe interface to determine the load on these cores. This incurs notable overhead on the host CPU, not only making the feedback loop less efficient and timely but also posing compatibility challenges with existing applications.

The latest Intel Xeon Scalable Processor integrates diverse on-chip accelerators within the chiplet [58]. These accelerators are designed to reduce the rapidly increasing datacenter tax [13, 25, 51], which refers to the consumption of CPU cycles for routine functions across hyperscale services in datacenters. Among these accelerators, Dynamic Load Balancer (DLB) is introduced to offload queue management and scheduling tasks from the CPU, offering the following capabilities. First, DLB can deliver unparalleled load-balancing throughput compared to the latest hardware-based dynamic load balancers developed to date. A high-end CPU typically integrates four DLB instances, each capable of handling up to 100 MPPS according to our extensive experiment. In other words, a dual-socket server based on such a CPU can load-balance 64B packets across CPU cores at the line rate of a 400Gbps NIC. Second, DLB also provides advanced QoS features, such as prioritized and flow-aware packet distribution for various traffic types (*i.e.*, ordered, unordered, atomic). Third, with its lock-free design, DLB significantly outperforms software-based shared queues in core-to-core communication. This makes DLB highly desirable for high-throughput, latency-sensitive applications that require multiple CPU cores to process packets and serve the requests encapsulated within these packets. Lastly, it facilitates the implementation of software-based packet processing pipelines, which typically span across multiple CPU cores.

This work is the first in the community to deep-dive into the architecture, functionality, and novel usage of DLB. Overall, we make the following three key contributions.

Contribution-1: Uncovering the benefits and limitations of DLB (§3). First, we begin by unveiling the hardware architecture and software stack of DLB more deeply than prior work [58], highlighting its unique and advanced hardware-based load-balancing capabilities. Second, we develop a microbenchmark suite, which will be open-sourced later, to comprehensively compare the performance and scalability of DLB against commodity software-based load balancers. Our evaluation shows that, thanks to its excellent scalability, a single instance of DLB can handle 100MPPS at low 99^{th}-percentile (p99) latency with five worker cores, while the software-based load balancers can deliver only 36–40MPPS even with eight worker cores, dropping most of the received packets. Lastly, we uncover that DLB still consumes a significant number of host CPU cycles at high packet rates, significantly diminishing one of its two key benefits—reducing the datacenter tax.

Contribution-2: Developing a framework for device-DLB direct communication (§4). First, we identify that the lack of a direct communication path between NIC and DLB requires the use of host CPU cycles to prepare work descriptors for received packets and enqueue them to DLB. Second, we propose AccDirect, which establishes a direct communication path between NIC and DLB, exploiting (1) the fact that each DLB instance is exposed as a PCIe device and (2) the PCIe Peer-to-Peer (P2P) communication mechanism [46]. This creates an unprecedented opportunity for the SNIC, which can now prepare the work descriptors and enqueue them to DLB on behalf of the host CPU cores. Lastly, we evaluate AccDirect-based DLB to show superior efficiency compared to conventional DLB. It offers practically the same performance as conventional DLB, while reducing the system-wide power consumption by 10%. When compared to a load balancer like hardware-based RSS, which is designed to work with an end-to-end RDMA-based KVS application, Masstree [50], AccDirect-based DLB provides 14–50% higher throughput with comparable p99 latency.

Contribution-3: Providing guidelines to unlock the optimal performance of DLB (§6). Naïvely using DLB out of the box yields suboptimal performance, as DLB offers both a vast configuration space and hidden advanced features for users. To provide guidelines for achieving the optimal performance of DLB, we first conduct a comprehensive evaluation of DLB with different configurations of queue depth, port types, wait modes, queue models, and queue and

packet priorities. Our evaluation shows that depending on the configuration, DLB presents as much as 1.2× and 1.3× higher maximum throughput and lower p99 latency, respectively. Second, we uncover a use case of an advanced feature—core affinity migration of atomic queues—to accomplish better load balancing for applications that process packets from specific flows.

2 Background

2.1 Design Space of Intra-host Load Balancer

Intra-host load balancing is crucial for maintaining high QoS by fairly distributing packets—each often requiring different service times—across CPU cores to prevent certain cores from becoming bottlenecks. Currently, software- and hardware-based load balancers have been developed to facilitate intra-host load balancing. These load balancers statically or dynamically distribute packets based on priorities, flows, or services. Notably, hardware-based load balancers have been implemented within (S)NICs. Figure 1 provides an overview of the four types of existing intra-host load balancers.

Software-based static load balancers (Figure 1(a)) require a CPU core to serve as the central point for receiving and queuing packets, while worker cores poll the central queue to dequeue and process packets. However, the throughput of such load balancers is bottlenecked by the limited processing capability of the load-balancing core and the lock contention among worker cores. Furthermore, continuous busy-polling by worker cores has been proven to be energy-inefficient for handling variable network packet rates [45].

Hardware-based static load balancers (Figure 1(b)) are widely supported by commodity NICs, notably through technologies like Receive Side Scaling (RSS) [32] applying a predefined hash function to packet header fields to distribute network packets across multiple hardware-based receive queues. While RSS is simple to implement in NICs, its load balancing efficiency heavily depends on the predefined hash functions, the packet field to be hashed, and the flow distribution [49].

Software-based dynamic load balancers (Figure 1(c)) employ one or more CPU cores to process received packets and route them to the load-balancing core, which then distributes these packets to different worker cores based on factors such as packet priorities and the load of individual worker cores. However, none of these load balancers can deliver 100 MPPS of throughput, which is required to load-balance 256B packets received at the line rate of a 200Gbps NIC which have started to be widely deployed, for the same reason as the software-based static load balancers.

Hardware-based dynamic load balancers (Figure 1(d)) offloads complex and demanding load-balancing tasks, such as queue and QoS management, to (FPGA-based) SNICs. It also requires the SNICs to coordinate with host CPU cores to monitor the load of worker cores, to dynamically adjust the distribution of packets. Recent work has demonstrated high-performance hardware-based load balancers [28, 50], but they support only certain packet sizes and/or limited use cases, with a maximum throughput of only 50MPPS.

2.2 PCIe Peer-to-Peer (P2P) Communication

Traditional PCIe communication between devices uses host memory as a temporary buffer, with the host CPU coordinating the communication. When devices were slow in the past, PCIe communication did not consume a notable amount of host resources (*i.e.*, CPU cycles and on-chip interconnect bandwidth, as well as memory space). However, with the significant improvement in the speed of not only I/O devices (*e.g.*, 400Gbps NIC and NVMe SSD) but also compute devices (*e.g.*, GPU and ML accelerator), such PCIe communication has started to consume a substantial amount of host resources [27].

To address this issue, PCIe P2P communication between devices, bypassing the host, has emerged as an attractive alternative. PCIe P2P facilitates communication between devices through their exposed PCIe memory spaces, such as NVMe Controller Memory Buffer (CMB) [4] and PCIe Base Address Register (BAR), effectively eliminating the consumption of host resources. For P2P write transactions, the initiator (or source) device writes data directly to the destination device's exposed memory over the PCIe bus, which links these two devices through the PCIe switch or Root Complex (RC), depending on the configuration of the PCIe hierarchy. P2P read transactions also bypass the host CPU and memory, following a procedure similar to P2P write transactions.

3 Understanding Intel Dynamic Load Balancer

Inspired by recent research [54], Intel has integrated DLB into its chipsets to offload queue management and scheduling tasks from the CPU, starting with the 4th-generation Intel Xeon Scalable Processor. DLB is located within the Data Accelerator Complex (DAC) on each CPU chiplet [58]. It is exposed to the system as a PCIe root-complex integrated endpoint device but communicates with CPU cores through a Coherent Mesh Interface (CMI), which offers a bandwidth of 50GB/s, comparable to that of PCIe 5.0. In this section, we begin by describing the DLB hardware architecture and its software stacks. We then compare the performance of DLB against other software-based balancers based on the DPDK Packet Distributor and `eventdev` libraries

3.1 Hardware Architecture and Software Stack

Hardware architecture. Figure 2 (left) illustrates the hardware architecture of DLB. It supports a producer-consumer model for various queue types (*i.e.*, direct, unordered, ordered, and atomic), where producer cores enqueue Queuing Elements (QEs) to DLB while consumer cores dequeue QEs from it. A QE refers to a 16B work descriptor containing metadata, such as event data (*e.g.*, a pointer to a memory address of a packet or request to be processed), scheduling (or queue) type, priority, queue ID, and flow ID, used for making scheduling decisions. DLB mainly consists of six components: ❶ producer ports, ❷ enqueue (or admission) logic, ❸ reorder buffer, ❹ queues, ❺ two-level arbiters, and ❻ scheduling and dequeue logic.

The producer ports are exposed as a PCIe BAR region of DLB, to which producer cores MMIO-write QEs to enqueue. The admission logic checks the validity of QEs received from the producer ports for DLB operations. Upon receiving QEs from the admission logic, the reorder buffer orders them based on their flow IDs and received orders, particularly for the ordered queue type, and then enqueues them to the queues based on queue IDs. The queues are implemented with on-chip memory in the DLB where QEs are buffered

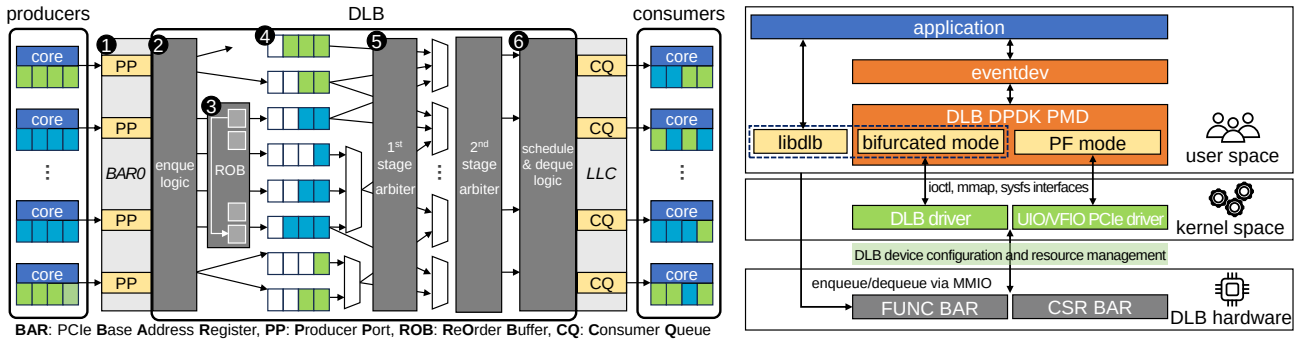


Figure 2: DLB hardware architecture (left) and software stack (right).

until they are scheduled. The two-level arbiters and scheduling logic dequeue QEs, schedule them based on their QoS requirements, priorities, flow IDs, and then push them to the associated consumer queues in a host memory region cached in the LLC.

At configuration time, producers set up producer ports and consumer queues, and specify the number, connection, and type of producer-consumer queues. Producers enqueue QEs to DLB through MMIO writes to the producer port address. One can use a 16B move instruction (`MOVNTDQ`) to enqueue one QE at a time. To improve enqueuing efficiency, we use `MOVDIR64B`, one of Intel's new x86-64 Accelerator Interfacing Architecture (AiA) instructions [58], to group and enqueue four QEs at a time. This instruction also allows us to send multiple groups of QEs to DLB simultaneously in a pipelined manner, without requiring fences around each MMIO write, which bypasses the cache hierarchy, and thereby enhances the enqueuing efficiency.

Software stack. Figure 2 (right) depicts the software stack and control flows of DLB. DLB can be used in a generic Linux environment with a DLB driver and `libdlb`, which provides APIs to simplify the adoption of DLB for programmers. DLB is also well-supported for DPDK, instantiated as an event scheduling device, called `eventdev` [8], with the DPDK Event Device library. DPDK provides a comprehensive set of user-space software libraries that parallelize network packet processing across multiple CPU cores for high throughput. Nonetheless, with hardware-based queue management and scheduling, DLB-based DPDK can provide higher throughput and lower overhead than software-based DPDK.

There are two modes of operation for DLB serving as a DPDK `eventdev` (orange boxes in Figure 2 (right)). One is the bifurcated Poll Mode Driver (PMD) mode in which part of the DLB functionality is managed by the kernel driver. The other is the Physical Function (PF) PMD mode, which manages and controls the entire DLB functionality from user space, providing direct access to the DLB hardware. While the workflow and programming interface of `libdlb` and DLB instantiated as `eventdev` in DPDK are largely similar, the different backend implementations lead to variations in performance and application.

3.2 Performance and Scalability

In this section, we compare the maximum performance and scalability of DLB with those of popular software- and hardware-based load balancers, focusing on intra-host load balancing.

Table 1: System setup.

	Server	Client
OS	Ubuntu 22.04.5 LTS (Linux 6.5.0)	Ubuntu 22.04.1 LTS (Linux 5.15.0)
Processor Model	Intel Xeon (Sapphire Rapids) Gold 6438Y+	Intel Xeon (Broadwell) E5-2660 v4
# Cores	32 (1 socket)	14 (1 socket)
System Memory	512 GB DDR5 16 DIMMs, 8 channels	64 GB DDR4-2400 2 DIMMs, 2 channels
NIC	BlueField-3	ConnectX-6 Dx

System setup. For performance evaluations, we set up a system consisting of a server and a client (Table 1). The server is equipped with a 32-core Intel 4th-generation Xeon Scalable Processor [17], where we use one of four DLB 2.0 instances in this work. It is also equipped with a 200Gbps NVIDIA BlueField-3 SNIC [39], which supports PCIe 5.0 ×16 and integrates a 16-core Arm Cortex-A78AE CPU. We use a DLB software package (version 8.9.0), which provides the DLB driver and `libdlb`, and apply its DPDK patch to a DPDK package (version 22.11.2). The client is armed with a 14-core Intel Xeon E5-2660 v4 CPU and a 200Gbps NVIDIA ConnectX-6 NIC.

Microbenchmark. We implement four microbenchmarks using load-balancing APIs provided by `libdlb` and DPDK libraries. They are designed to make worker cores perform simple touch-and-drop operations while eight producer cores generate packets at 100MPPS. We use such simple operations to evaluate the maximum load balancing capability of different load balancers. `dpdk-pd` is based on DPDK Packet Distributor [9], a library that provides simple, light-weight load balancing optimized for latency while guaranteeing packet ordering. In contrast, `dpdk-ed` is built with DPDK `eventdev`, a more feature-rich scheduling library that offers not only load balancing for multiple Rx CPU cores but also more advanced event scheduling and pipelining. Both utilize a dedicated CPU core for load balancing. `dlb-ed` and `dlb-lib` are constructed with DPDK `eventdev` and `libdlb` to use DLB for load balancing, respectively (§3.1). We use only the PF PMD mode for `dlb-ed` since both PF and bifurcated PMD modes present similar performance.

Throughput and latency of intra-host load balancing. Figure 3 plots the throughput, average latency, and p99 latency of the four microbenchmarks that perform intra-host load balancing, where we do not show the average and p99 latency values when

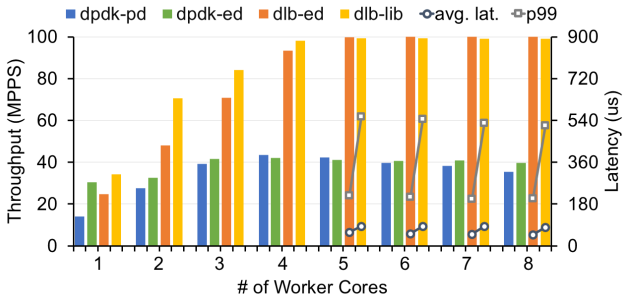


Figure 3: Throughput and latency of intra-host load balancers for varying numbers of worker cores.

packet drop occurs. Overall, this shows that DLB-based load balancers provide higher throughput and lower latency than software-based ones. Besides, as the number of worker cores ($N_{workers}$) increases over four, the throughput of DLB-based load balancers approaches 100MPPS. Meanwhile, the throughput of `dpdk-ed` decreases, and the latency increases because the scheduling overhead increases, while the processing capability of a single load-balancing core remains limited. As such, the throughput gap between DLB- and software-based load balancers increases. For example, when $N_{workers}$ is increased from two to eight, the throughput differences between `dlb-ed` and `dlb-lib` against `dpdk-ed` increase from 18% and 13% to 152% and 149%, respectively. Since `dpdk-pd` and `dpdk-ed` always drop packets regardless of $N_{workers}$, we do not provide any latency values for them. However, even considering only processed packets, they present significantly higher latency than `dlb-ed` and `dlb-lib` across the range of $N_{workers}$. Lastly, `dlb-lib` provides higher throughput and lower latency than `dlb-ed` when $N_{workers}$ is below five because of the software overhead of using the DPDK framework.

Performance of intra-host load balancers in an end-to-end setup. To understand the performance of DLB for intra-host load balancing in a more practical networking case, we set up a client and a server as follows. The client is set up to constantly send packets, each with a 64B payload. Note that this payload size is commonly used to evaluate networking performance as one of the two most common payload sizes in datacenter intra-host network traffic [5]. We evaluate 64B payload since the other size, 1KB, easily saturates the network bandwidth even at low MPPS. The client generates packets as follows. First, it generates the port number of packets, following a uniform distribution. This is to make the server NIC, which applies the port number to its RSS hash function, uniformly

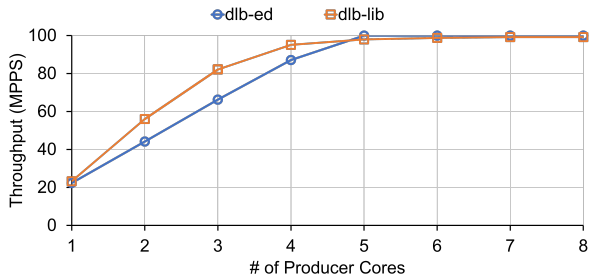


Figure 4: Throughput versus the number of CPU cores that prepare and enqueue QEs.

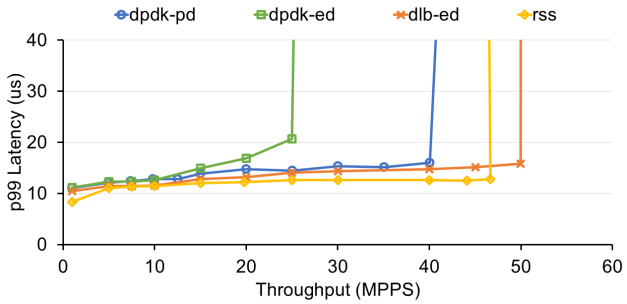


Figure 5: Impact of intra-host load balancers on end-to-end throughput and latency at different network packet rates.

distribute these packets across worker cores, representing the best-case scenario for `rss`. Second, it generates a processing time, which follows an exponential distribution with a mean service time of 100ns [30, 33], and then embeds it into the payload of each packet. When receiving a packet, a worker core obtains the processing time from the payload and starts to process the next packet after the specified amount of time has passed. The server uses 8 worker cores and 5 Rx cores (see Figure 4 and 'Limitations' below) for `dpdk-ed`, `dpdk-pd`, and `dlb-ed`. Unlike these microbenchmarks, `rss` performs both Rx and work on the same CPU core. Therefore, we allocate 13 CPU cores for `rss` for fair comparison between `rss` and the other microbenchmarks.

Figure 5 plots the p99 latency across different packet rates. This shows that, for a mean service time of 100ns with exponential distribution, `dlb-ed` provides 1.2×~2.0× higher throughput than `rss`, `dpdk-pd`, and `dpdk-ed` under a reasonable p99 latency. `dlb-ed` also offers 6.4%~46.9% lower p99 latency than `dpdk-ed`, and 2.7%~10.8% lower p99 latency than `dpdk-pd`. The results indicate that offloading queue management and scheduling to `dlb-ed` reduces software overhead and provides better scalability than both `dpdk-ed` and `dpdk-pd`. It is worth noting that, although both `dlb-ed` and `rss` perform load balancing with dedicated hardware, `dlb-ed` still requires some CPU cores to enqueue packets to DLB and results in 1%~21% higher p99 latency than `rss`. Nonetheless, `dlb-ed` outperforms `rss` on throughput because `rss` uniformly distributes packets without knowing the load of each worker core, which varies depending on the service time of each packet. This gives rise to unbalanced loads across worker cores, with the more heavily loaded cores contributing to higher p99 latency at high packet rate.

Limitations. In this section, we have demonstrated that DLB-based load balancers offer considerably higher performance and better scalability than software-based ones. Nonetheless, achieving a throughput of 100MPPS for intra-host load balancing requires five CPU cores solely to prepare and enqueue QEs to DLB (Figure 4). This significantly diminishes one of two key benefits of DLB—reducing the datacenter tax.

4 AccDirect

We have shown that even DLB consumes a significant number of host CPU cycles to deliver high load-balancing throughput as host CPU cores need to prepare and enqueue QEs (work descriptors) at high rates. In this section, we first propose AccDirect, a load-balancing solution designed to completely eliminate dependence on host CPU cores for using DLB. We then compare the performance

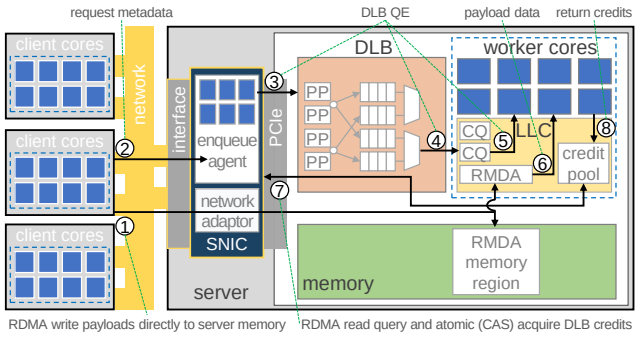


Figure 6: AccDirect with SNIC.

and system-wide power consumption of AccDirect with those of conventional DLB for intra-host load balancing, using synthetic workloads and an end-to-end KVS application, Masstree [10, 29, 50].

4.1 Overview

AccDirect provides enhanced DLB and NIC drivers to set up a PCIe P2P communication path between DLB and NIC, exploiting (1) the fact that DLB is exposed as a PCIe device to the system and (2) the PCIe P2P communication capability. Through this path, enhanced NIC, SNIC, or even clients can prepare QEs and directly enqueue these QEs to DLB, obviating the consumption of host CPU cycles (or resources). We briefly discuss trade-offs among these three choices below (§5 for in-depth discussion).

AccDirect with enhanced NIC is the most efficient solution. However, it requires dedicated hardware within the NIC to capture packets and prepare QEs, and then enqueue them to DLB after transferring these packets to buffers in a host memory region. Alternatively, AccDirect with clients can also prepare QEs and enqueue them to DLB in the server through established RDMA connections between the server and the clients, using commodity NICs. Although it does not need to make changes to NIC, it not only consumes client CPU cycles and network bandwidth but also demands changes to client application code. Lastly, AccDirect with SNIC is the most balanced solution among the three, which emulates the dedicated hardware using SNIC CPU cores. It captures most of the benefits of AccDirect with enhanced NIC while consuming negligible network bandwidth and client CPU cycles. Figure 6 depicts AccDirect with SNIC, which we will focus on in this work, where an agent running on SNIC CPU cores emulates AccDirect with enhanced NIC.

4.2 Direct Enqueue to DLB through P2P

Direct control of DLB with P2P. PCIe P2P facilitates a PCIe device to directly access a PCIe BAR region of another peer PCIe device, bypassing the host CPU and memory (§2.2). Beyond facilitating efficient data transfers between devices, it also allows devices to control each other's operations, creating opportunities for these devices to work together without host CPU involvement, which we refer to as 'chaining a device with another device' in this work. Meanwhile, Intel on-chip PCIe accelerators, including DLB, are exposed to the system as PCIe devices, with their operations controlled by MMIO accesses to BAR regions, similar to conventional off-chip PCIe accelerators. Given this, we hypothesize that other off-chip PCIe devices, such as (S)NICs, can also directly control

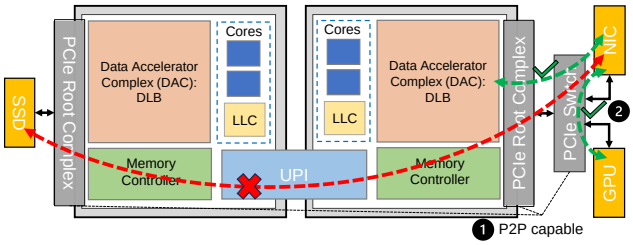


Figure 7: PCIe P2P-capable connections.

these on-chip PCIe accelerators through PCIe P2P, eliminating the consumption of host CPU cycles. While this appears theoretically feasible, we find little published research work and development efforts on PCIe P2P, which has made it challenging for the research community to fully understand and explore PCIe P2P.

In this work, we conduct extensive experiments with various systems and devices to uncover hidden constraints for a host to support and enable PCIe P2P communication, especially for chaining an on-chip PCIe device with an off-chip PCIe device, which is summarized below and illustrated in Figure 7. ❶ PCIe devices must be connected through a P2P-capable PCIe switch or Root Complex. ❷ PCIe devices should be connected using specific PCIe topologies (*e.g.* behind the same P2P-capable PCIe switch) for the best performance. Besides, they must be connected to the same socket to bypass host CPU, memory, and inter-socket connection (*e.g.*, UPI link). This path is shown as the red dashed line in Figure 7. ❸ The BIOS such as 64-bit I/O must be configured properly. Lastly, it is important to note that meeting these constraints provides necessary but not sufficient conditions to enable PCIe P2P in most cases, which are often dependent on the specific hardware in use.

P2P-driven direct enqueue to DLB PCIe P2P communication can be performed using various methods, spanning from hardware implementations (*i.e.*, ASIC and FPGAs) to software implementations, such as low-level DMA programming (*e.g.*, DOCA-DMA on BlueField-3) and `verb` library for RDMA programming. In this paper, RDMA refers to RDMA technology including both one-sided and two-sided `verbs` and we use `verb` and RDMA interchangeably. Developing a general framework that enables direct enqueue from NIC to DLB requires a PCIe P2P implementation characterized by flexibility, compatibility, and user-friendliness. Among these options, we find one-sided RDMA `verbs` (*e.g.*, RDMA `READ` and `WRITE`) to be the sweet spot to implement such direct control of DLB from NIC for the following reasons. The one-sided `verbs` satisfy the minimum requirements for DLB controlling semantics while providing simple programming interfaces. Moreover, RDMA technology has been supported by (S)NICs widely used for intra-datacenter, high-performance networking [3], thereby eliminating the need for additional hardware. On the contrary, using low-level (S)NIC's DMA for P2P communication (*i.e.*, a DMA–based AccDirect) is dependent on specific hardware, driver, and/or programming interface, *e.g.*, BlueField-3 (BF-3) using DOCA-DMA [37] to program its DMA engine. We implemented and evaluated DOCA-DMA-based AccDirect in addition to RDMA-based AccDirect. DOCA-DMA-based AccDirect does provide ~7.5% lower latency and ~9.4% higher throughput on BF-3. However, DMA does not always perform better than RDMA `verbs` within the host, for example, DOCA-DMA on BF-2 shows 2.3× lower throughput for 64B write operation from SNIC

to the host [55]. On the other hand, both DMA and RDMA `verb` programming requires explicit memory space registration to grant SNIC hardware permission and visibility to access addresses in the registered memory space.

Given these, AccDirect implements a peer memory client kernel module for DLB which is complementary to existing `verb` library and kernel modules (RDMA-Core) [48]. Its exposed functions are defined in the `peer_memory_client` structure [31]. As such, the BAR space of DLB can be registered to RDMA programs and accessed with one-sided RDMA operations from off-chip PCIe devices or even remote clients with one-sided RDMA operations. We define the peer PCIe device, which exposes its memory space to the RDMA stack, as the peer memory client. Note that RDMA for PCIe P2P communication is incompatible with IOMMU [36] if the device driver does not register and manage I/O virtual address. This is because, in this case, PCIe P2P uses physical addresses for PCIe transactions while IOMMU considers them as I/O virtual addresses and then attempts to translate them to physical addresses. In addition, enabling IOMMU can result in downgraded performance for PCIe P2P communication [42]. We therefore leverage the IOMMU `passthrough` mode to achieve better performance and ensure that RDMA for PCIe P2P accesses the PCIe bus with physical addresses.

Host bypassing verification. We verify whether the peer memory client module enables DLB enqueue operations to effectively bypass the host by monitoring LLC miss rate and memory bandwidth consumption. Our extensive experiments show that directly enqueuing QEs to DLB from NIC with RDMA `WRITE` operation does not incur any discernible difference in LLC miss rate and memory bandwidth consumption compared to an idle system.

4.3 SNIC Agent

Although we have enabled direct DLB enqueue capability through one-sided RDMA `verbs`, the DLB has an on-device memory cap that necessitates credit control from the NIC to limit the number of in-flight QEs. Otherwise, it could potentially exhaust DLB, thereby violating QoS compliance. Consequently, it is prudent for the server side to manage the enqueue behavior and enforce QoS. Given the simple and light-weight nature of such bookkeeping tasks, we employ the low-profile CPUs on the modern SNICs to offload them from the high-performance x86 CPUs [14]. This is to not only save precious host CPU resources but also achieve better energy efficiency. Such an SNIC agent is designed to handle two tasks: (1) pack and enqueue QEs to DLB and (2) manage DLB credits.

Pack and enqueue QEs. The SNIC agent embeds the payload address and size into DLB QEs so that the worker cores can readily access the payload by extracting that address. We refer to this embedded information as metadata. There are two methods to obtain the metadata of network packets, either directly from SNIC hardware or from the client. Since modifications to the SNIC hardware are not feasible, we have opted to have the client send a 10-byte metadata packet using RDMA `WRITE`. This metadata includes a 64-bit target server address of the payload and a 16-bit payload size. The metadata path for enqueuing DLB is shown as ② and ③ in Figure 6. For SNIC to enqueue to DLB, we configure and register the corresponding device memory of DLB producer ports with RDMA, using the aforementioned DLB peer memory client. Clients send one or multiple metadata following the payload to the SNIC agent at a time. Upon receiving the metadata, the SNIC agent assigns the server address in the pointer field of a DLB QE and encodes the size as the side information in the QE. We build a ring buffer to temporarily store the request metadata on the SNIC to buffer the in-flight requests if the SNIC-DLB enqueue speed is not as fast as the client's request rate. Then, the SNIC agent directly enqueues DLB hardware after properly packing DLB QEs through reliable connections between the SNIC agent and each DLB producer port, without involving the host system.

DLB credit management. DLB relies on a software-based credit pool to make sure it is not overloaded, *i.e.*, DLB enqueue is paused when there is no credit for in-flight packets. Credit management is crucial for lossless data communication, application correctness, and avoiding frequent hardware warnings from DLB. Since we leverage SNIC as the enqueue agent for clients, SNIC needs to be able to query and acquire DLB credits to ensure the DLB is not overwhelmed. The software credit pool in the `libdlb` library is managed through CPU atomic operations (fetch-add and compare-and-swap). To maintain correctness, the SNIC must also access the credit pool using atomic operations, for which we employ RDMA atomics. One of the challenges is that RDMA atomic operations should maintain atomicity with CPU atomic operations as CPU cores also need to read and update the credit pool. However, RDMA atomic operations are by default implemented using PCIe read and write transactions, and only ensure atomicity among themselves [6, 24].

To resolve this mismatch, we configure the SNIC (*i.e.*, BlueField-3) to use PCIe atomic transactions. To be specific, we leverage `mlxconfig` [38] to configure the `PCI_ATOMIC_MODE` as `PCI_ATOMICS_ENABLED_EXT_ATOMICS_ENABLED`. We confirm that PCIe atomic is enabled for RDMA atomic operations by comparing PCM PCIe counters [19] running 8-byte RDMA atomic and write tests with `perftest` [47]. The PCIe Root Complex converts PCIe atomic operations to CPU-locked operations when accessing system memory [2]. This ensures mutual exclusion between PCIe and CPU accesses. As illustrated in Figure 6, the SNIC agent queries available DLB credits with RDMA `READ` operation and uses RDMA atomic `CAS` (compare-and-swap) operation to acquire a portion of credits (⑦) from the host. Note that RDMA `READ` and `CAS` are CPU-bypassing as well. Finally, when worker cores finish processing an RDMA request, it returns the credits back to the DLB credit pool as depicted by ⑧.

4.4 AccDirect Control and Data Planes

We implement the end-to-end AccDirect framework to demonstrate the performance and energy benefits of direct DLB enqueuing. The implementation of AccDirect can be divided into two main components: the control plane where the SNIC effectively coordinates with DLB, and the data plane which handles the application payload data path. Our implementation allows RDMA-based applications to be migrated to use DLB easily with minimal changes to applications.

Control plane. The control plane of AccDirect involves initializing DLB hardware and RDMA connections for applications, the runtime DLB credit management described in §4.3, and the SNIC agent which prepares and enqueues QEs from off-chip. Figure 8 presents

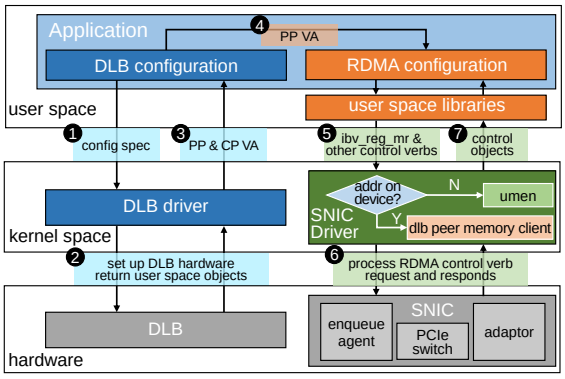


Figure 8: Enabling SNIC and DLB direct communication path.

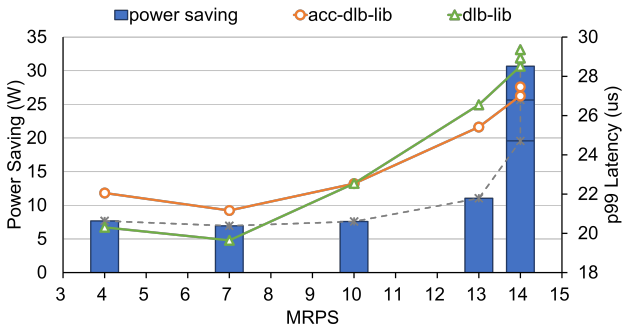


Figure 9: Absoluate power saving (2.8% to 10% system-wide power consumption) and p99 latency overhead of AccDirect comparing against host CPU-based baseline.

the initialization stage of DLB and RDMA. DLB configuration sets up DLB hardware including producer and consumer ports, DLB queues, and load-balancing queue type according to the user inputs (❶ and ❷). DLB driver processes the outputs from DLB hardware and returns user space objects to the application (❸). In particular, the virtual addresses of producer ports in the MMIO region are passed to the RDMA initialization (❹). This MMIO memory range, corresponding to the producer port, is then registered to RDMA with the DLB peer memory client as a peer memory region (❺). This memory region registration is required by RDMA to allow NICs to directly read or write target addresses without host CPU intervention. When an application attempts to register a memory region in DRAM, it uses `ibv_reg_mr` provided by the RDMA userspace library. The SNIC driver records the virtual to physical address mapping and pins the physical memory to avoid swap out [44]. The peer memory client enables such registration for a peer device memory. The mapping information is then stored in a memory translation table maintained both in host memory and on-NIC cache (❻).

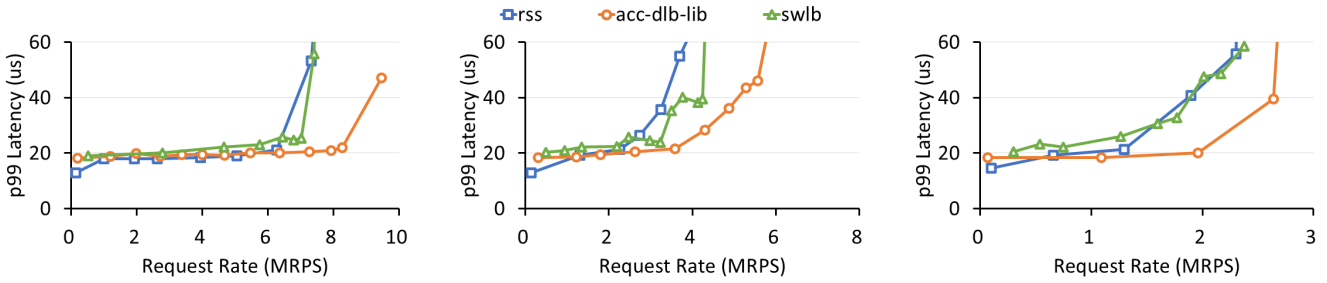
During the configuration phase, we set up RDMA reliable connections between the SNIC agent and the host CPU cores to remotely enqueue QEs and manage DLB credits from SNIC agent. Since we cannot modify existing SNIC hardware to obtain the host addresses requests are written into, we also set up reliable RDMA connections between SNIC agent and client cores, which are used for getting request metadata (*i.e.*, address and size) directly from the client as a workaround. To reduce the performance overhead from metadata transfer, we consolidate multiple requests into a single batch whenever possible. Once the SNIC agent receives the RDMA request metadata, it extracts the corresponding request metadata (*i.e.* address and size) and, if there are sufficient DLB credits, enqueues DLB with RDMA `WRITE`. We discuss the overhead and performance projection of hardware-assisted designs in §5.

Data plane. The data plane of AccDirect consists of both request payload and metadata transmission. Among RDMA's three transport modes [41], reliable/unreliable connection transport modes are not typically used in existing applications that require load-balancing. This is mainly because one connection is coupled with one thread, making no opportunity for load-balancing. However, AccDirect can support all three RDMA transport modes for payload transmission. It enables load balancing across multiple RDMA connections by using an SNIC agent to encode the request address into DLB QE, utilizing the pointer-passing nature of DLB. To be more specific, AccDirect establishes two sets of RDMA connections: one between the client and server for payloads, and another between the client and SNIC agent for request metadata (*i.e.*, address and size). For application-level data communication, the client directly sends data payload to the server memory (①) after which the client also sends RDMA request metadata to the SNIC agent (②), as shown in Figure 6. Then, the SNIC agent packs and enqueues QEs into DLB for load-balancing decisions. DLB performs load-balancing across all worker threads. Once the scheduling decision is made, the worker core dequeues QEs from DLB through its consumer queue (⑥). Since the request pointer and size are embedded in the QE, the worker core can directly read and process the request without interfering with other workers. In this way, AccDirect decouples the RDMA connection from a particular thread by leveraging DLB's pointer-passing capability to flexibly schedule requests to any worker threads without overhead due to lock contention and false sharing of cache lines. The payload data transfer path is shown as ① and ⑥ in Figure 6, maintaining the existing application datapath.

4.5 Performance and Power

In this section, we first use a microbenchmark to evaluate the performance and power consumption of AccDirect against the baseline which uses dedicated host CPU cores to enqueue DLB. We also evaluate an end-to-end multi-core key-value store application, Masstree [10, 29, 50], with various query mixes.

Microbenchmark. To compare the performance and power consumption of CPU-driven DLB with those of AccDirect-driven DLB, we use two microbenchmarks: `dlb-lib` (§3.2) and `acc-dlb-lib`, where the latter is AccDirect-driven DLB that uses SNIC CPU cores to enqueue received packets. We use 8 worker cores on the server side and sweep the number of client cores from 2 to 8. The worker touches the payload and performs dummy work with a constant service time of 500*ns*. As we sweep the request rate, the power difference between using SNIC and the host CPU becomes more significant. AccDirect is able to save 30.66 Watts (10%) at a request rate of 14.5 MRPS. AccDirect has an average of 1.6μs higher p99 latency than the CPU baseline implementation when the request rate is lower than 11 MRPS. This latency overhead mainly comes



(a) Bimodal: 50% GET, 50% $\texttt{SCAN}_{18}$. (b) Bimodal: 75% GET, 25% $\texttt{SCAN}_{75}$. (c) Trimodal: 34% GET, 33% $\texttt{SCAN}_{46}$, 33% $\texttt{SCAN}_{105}$.

Figure 10: Masstree request latency of the software baseline and DLB with different query mixes. $\texttt{SCAN}_X$ denotes a query that retrieves X key-value pairs.

from an extra RDMA `WRITE` operation used for enqueuing QEs from SNIC to DLB. The latency overhead is amortized when the request rate becomes higher since it can be hidden by the overlapping of computation and communication. Therefore, as the request rate increases, `acc-dlb-lib` can achieve 2.35μs better p99 than `dlb-lib` because of less host resource contention.

End-to-end application. We further evaluate AccDirect using the end-to-end Masstree Key-Value Store (KVS). The Masstree KVS, adapted from Turbo [50], replaces the original UDP transport with `send/recv` two-sided unreliable datagram (UD) `verbs` for request and respond transmissions. We use UD `verbs` for the purpose of emulating RSS-like mechanisms which allow the client to dispatch requests across server's receive queues randomly or following certain policies (*e.g.*, round-robin). The randomized approach models RSS's static hash function, which, in the best case, uniformly distributes request datagrams among queues. We integrate the Masstree KVS with AccDirect (`acc-dlb-lib`), enabling clients to send requests through the SNIC and DLB. Responses are sent back to the client using two-sided RDMA UD `verbs`, identical to the `rss`'s response mechanism. We also evaluate AccDirect against a software-based dynamic load balancer (`swlb`) which is identical to `dpdk-pd` discussed in §3.2. We use 16 worker cores for all Masstree evaluations while `swlb` uses another 2 CPU cores for receiving and distributing payload respectively. Figure 10 compares the 99th percentile latency among `rss`, `swlb`, and AccDirect (`acc-dlb-lib`) for different query mixes. For all mixes, AccDirect offers lower latency at high loads, with the p99 latency gap between AccDirect and both `rss` and `swlb` increasing as the load increases. Specifically, AccDirect sustains 31% and 20% higher load under a 60μs latency target on average than `rss` and `swlb`, respectively. This improved performance stems from AccDirect's enhanced load balancing capabilities, enabled by the DLB's centralized hardware-offloaded queue architecture. Unlike RSS, where requests are randomly distributed across worker queues, DLB's design allows worker cores to pull requests from a shared queue, reducing head-of-line blocking. Meanwhile, AccDirect outperforms `swlb` since it reduces software overhead and the payload pointer passing across receiving, distributor, and worker cores. In this way, AccDirect maintains lower latency at different packet rates.

5 Discussions

Optimization of AccDirect with SNIC. Similar to many load-balancing and queue management solutions, DLB embeds payload pointers in QEs and distributes them across worker cores for payload processing. Therefore, the payload address must be known before enqueuing a QE into DLB. For applications using RDMA, enhanced NIC can inspect every packet to determine the memory access address; see 'Supporting other inter-host network stacks with enhanced NIC' below for applications using other high-performance networking stacks such as DPDK. Since QE preparation and enqueue logic is simple and deterministic, it is feasible to implement an agent for enhanced NIC at a negligible cost.

In this work, however, as we evaluate AccDirect with a commodity SNIC, we make clients send 10B metadata packets along with packets encapsulating requests to the server. These metadata packets contain the server memory addresses that these requests will access, but they can consume non-negligible amounts of network bandwidth, especially for small packet sizes at high rates. To address this issue, we propose to batch multiple metadata packets into one metadata packet at high packet rates. We observe that this batching approach significantly reduce the consumption of network bandwidth while negligibly affecting throughput and latency.

Supporting other inter-host network stacks with enhanced NIC. Unlike one-sided RDMA operations, where both the client and the packet header explicitly specify the target address, other high-performance network stacks (*e.g.*, DPDK) determine the target address solely on the server side. Therefore, implementing an end-to-end application that leverages AccDirect requires network protocol library and/or driver-level co-design. Take DPDK as an example, the host data buffer descriptors are organized in one or multiple ring buffers. As such, the agent in enhanced NIC needs to track the head pointers of these buffers. The agent can determine these addresses through device-to-host (D2H) memory accesses over PCIe (through RDMA over PCIe) or CXL, where the latter provides very fast and efficient accesses [22, 52]. DPDK might necessitate temporary buffering of payload data on SNIC memory. However, modifying the NIC driver to initiate packet DMA directly from SNIC can allow for the direct transfer of payload to host memory, streamlining the data flow process.

AccDirect for generic cross-device load balancing and accelerator chaining. AccDirect introduces two key concepts that enhance system performance. Firstly, beyond balancing loads across CPU cores, AccDirect can be extended for cross-device load balancing by efficiently scheduling tasks among accelerators and the CPU. AccDirect has proved that the SNIC agent is capable of controlling other PCIe accelerators. Therefore, an extended version of SNIC agent can potentially arbitrate workloads and direct tasks to the DLB for CPU-centric operations or to PCIe accelerators such as FPGAs via P2P communication. This cross-device load balancing can further ensure optimal device utilization and improved overall system performance.

Secondly, AccDirect demonstrates the effectiveness of accelerator chaining with DLB and SNIC, which is essential for harnessing the strengths of diverse specialized hardware, enabling a seamless and decentralized processing pipeline that minimizes latency and maximizes throughput. AccDirect can be easily extended to work with other on-chip devices and off-chip accelerators such as FPGAs and GPUs. Furthermore, RDMA `verbs`-based AccDirect establishes a scalable abstraction and framework for chaining additional accelerators not only on the local server but also remote servers, even if the underlying fabrics can be different. We believe AccDirect lays the groundwork for a flexible and scalable generic accelerator chaining approach for efficiently utilizing on-chip, off-chip, and remote accelerators.

6 Make the Most out of DLB: Performance Analysis and Guidelines

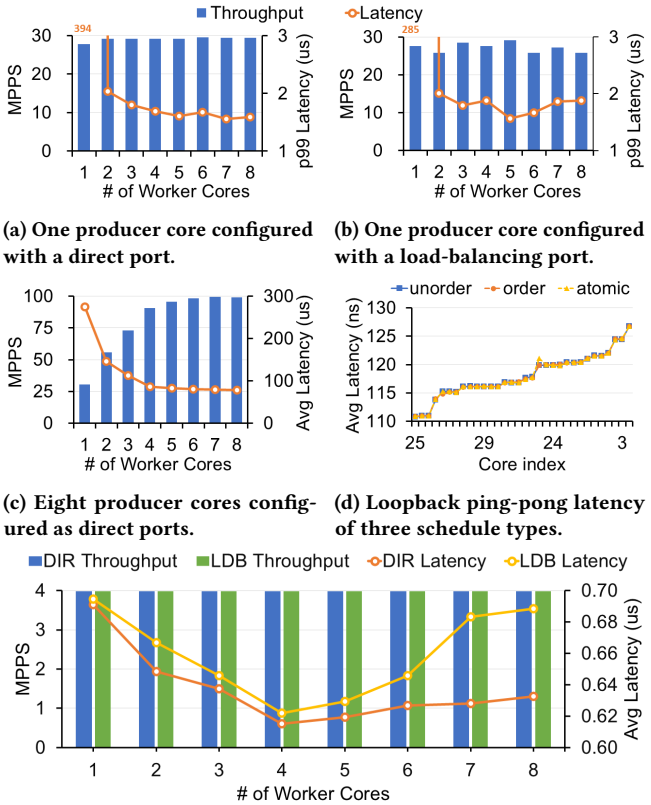
We detail the characterization of DLB through a series of microbenchmarks and outline various trade-offs and beneficial use cases that should be considered when employing DLB to accelerate applications. Since we have discussed the DLB performance within the DPDK framework in §3, we will mainly focus on the core-to-core performance results with our microbenchmarks built on top of `libdlb`. To our knowledge, this is the first publicly available characterization and guideline for using DLB. We discuss several important implications of configuration parameters that significantly affect DLB's throughput and performance determination. Since DLB is connected with the Coherent Mesh Interface, we comprehensively analyze DLB's throughput and latency within a core-to-core intra-package setup. Our microbenchmarks also characterize DLB's QoS features (packet and queue priority) and potentially employed implementation policies. Lastly, we evaluate the atomic queue type supported by DLB and propose a unique use case involving mini-flow to improve data locality.

6.1 Implications from DLB Configurations

Our experiences indicate that to unleash DLB's optimal performance, four DLB parameters, including enqueue depth, dequeue and Consumer Queue (CQ) depth, dequeue wait modes, and port type selection, must be carefully configured.

Implication 1: Performance benefits and load-balancing efficiency tradeoffs of batching. Batching is beneficial because it reduces the overhead associated with processing individual tasks, including DLB enqueue and dequeue tasks. Enqueue depth defines the maximum size of an input burst that can be sent consecutively to the DLB without requiring a store fence. By increasing the enqueue depth, we can amortize the overhead associated with the store fence, thereby enhancing overall processing efficiency. Two parameters control DLB's dequeue behavior: CQ depth and dequeue depth. The CQ depth configures the total number of QEs that can be maintained in one CQ, while the dequeue depth sets the dequeue burst size associated with the receive function call. During the dequeue process, DLB automatically arbitrates the QEs to Consumer Queues (CQs) located in the LLC. Subsequently, worker cores fetch QEs in bursts determined by the dequeue size. This implies that the dequeue depth should be set smaller than the CQ depth. Unlike enqueue depth, larger dequeue and CQ depth are not always better than smaller ones. Since DLB arbitrates QEs to workers based on current CQ fullness, the choice of CQ depth affects the load-balancing efficiency. Typically, large CQ depth may lead to the head-of-line blocking issue while shorter CQs allow better and fairer load balancing among all worker cores with the cost of more frequent QE refill from DLB.

Implication 2: Performance and energy-efficiency tradeoffs of different dequeue wait modes. There are three dequeue wait modes: (1) poll mode implements busy-polling the CQ until at least one event is received, (2) `umwait` puts the core to low-power mode until a change in the CQ by the coherence signal [58], and



(a) One producer core configured with a direct port.

(b) One producer core configured with a load-balancing port.

(c) Eight producer cores configured as direct ports.

(d) Loopback ping-pong latency of three schedule types.

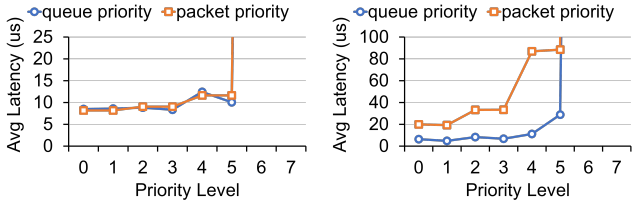
(e) One producer core configured as a direct (DIR) or load-balancing (LDB) port, with a fixed enqueue rate of 4 MPPS.

Figure 11: Throughput and packet latency of DLB using one or eight producer core(s) and eight worker cores.

(3) interrupt mode where the core sleeps until an event receive interrupt is raised. As we transition from poll mode to `umwait` mode and then to the interrupt mode, the latency tends to increase, while the energy efficiency improves. The dequeue wait mode can be configured individually for different consumer ports or dynamically at runtime. This flexibility allows for an adaptive approach, where the dequeue wait mode can be adjusted based on the current request rate or predefined to suit the specific characteristics of each core's computational task.

Implication 3: Choosing the right port type in DLB systems crucially enhances performance predictability and optimization. Aside from the enqueue, dequeue, and consumer queue depths, the selection of port type significantly impacts the performance determinism. DLB distinguishes between two main types of ports: the load-balancing port and the direct port. The direct port is further categorized into two sub-types: one dedicated to enqueuing and the other exclusively for dequeuing. Producers should always be configured to direct ports to ensure optimal performance. Comparing Figure 11a and Figure 11b, configuring the producer port as a direct port type gives up to 1.2× higher throughput and 0.7~1.3× lower p99 tail latency than configuring it as a load balancing port. The higher tail latency of direct port type when using one worker core is due to more significant queuing delays since it has a higher throughput that overloads a single worker core. Queues in the DLB can build up when the dequeue rate cannot keep pace with the enqueue rate, which lead to high queuing delay. In addition, DLB uses a credit-based mechanism to prevent overwhelming internal queue storage. The enqueue operation from producers can be blocked when running out of credit, leading to prolonged queuing delays. As more worker cores are employed, using direct port type for producers always offers higher throughput and lower p99 latency. To our knowledge, the load balancing port configuration for producers results in increased consumption of the internal resources of the DLB (*i.e.*, QE arbitration and dispatch logic) and thus less stable performance. To eliminate the impact of queuing and credit management, we set a fixed enqueue rate of 4 MPPS on the producer core. Figure 11e demonstrates that employing direct port can achieve 1% to 9% lower latency than using load balancing port because of less internal resource consumption, which remains non-negligible. For the case with 8 worker cores, employing direct port type for producer port also offers 1.2× and 1.3× higher throughput and lower p99 latency, respectively. This observation indicates that a load-balancing port consumes more DLB internal resources (*e.g.* hardware credits and arbitration logic) than using a direct port. Therefore, port types should be chosen carefully to get more deterministic and optimal performance, especially for producers.

Queue models. To understand the latency overhead introduced by DLB, we conduct the loopback latency test, where the Queuing Elements (QEs) are enqueued and dequeued from the same CPU core to DLB in a ping-pong style. Across three schedule types, unordered, ordered, and atomic, DLB yields an average latency of 118*ns*, roughly twice the memory access latency. As illustrated in Figure 11d, the latency overhead incurred by different CPU cores accessing the DLB exhibits a variation of approximately 10%. This is because the 4th-generation Intel Xeon Scalable Processor has four chiplets each containing one DLB, but our testbed only enabled one DLB instance. The distance between CPU cores and DLB varies from core to core and thus leads to a roughly step-like latency increment.



(a) Apply packet and queue priority individually. **(b) Apply packet and queue priority simultaneously.**

Figure 12: DLB's packet and queue priority performance.

We then perform a series of core-to-core experiments to evaluate the overall throughput and latency across all types. We will take the unordered schedule type as an example because it is most widely used in load-balancing scenarios. The producer core(s) is configured to be a direct port(s) for optimal performance. In this case, we chose the enqueue and dequeue depth to be 32 to allow batching. We first evaluate a single-producer case. As illustrated in Figure 11a, when there is no queuing (*i.e.* using more than two worker cores), observed latency stabilizes at approximately 1.56 μs. Then, we demonstrate the scalability of DLB by progressively increasing the number of worker cores used while keeping the number of producer cores constant. Figure 11c reveals that the throughput of DLB scales linearly until it reaches the limits imposed by the hardware capacity.

Queue and packet priorities. DLB provides fine-grained hierarchical packet and queue service priorities for QoS purposes. Each Consumer Port (CP) is able to connect with up to 8 queues with different priorities. The application assigns queue priorities at the configuration stage. On the contrary, the packet priorities are encoded in the QE field at the time of enqueuing.

Our microbenchmark experiments explore the behaviors of both queue and packet priorities to understand the efficacy of priority-based scheduling. DLB supports up to **eight** levels for packet and queue priorities as claimed, respectively. We leverage 8 producers and 8 workers in all experiments, where each producer randomly assigns packet priorities to QEs and each producer is connected to only one DLB queue while all workers are connected to all 8 queues. We add a fixed 100ns service time for each packet. However, our evaluations of DLB hardware in commercially available servers reveal that there are only **four** hardware priority groups for both packets and queues (Figure 12a). We believe the high-priority groups (*i.e.* 0-5) likely employ a weighted round-robin policy, whereas the lower priorities (*i.e.* 6 and 7) utilize a best-effort policy. This is evidenced by the step-up elevated latencies among priority groups 0-5 and very long average latencies for priorities 6 and 7. The two priority hierarchies can be applied simultaneously. Thus, we have also conducted experiments to assess the performance of cooperative queue and packet priorities. As shown in Figure 12b, queue priority has precedence over packet priority. The findings suggest that a QoS-based load-balancing design utilizing DLB must account for two critical factors: (1) the lowest priority group should not be used if a performance guarantee is required, and (2) queue priority scheduling happens before the packet scheduling.

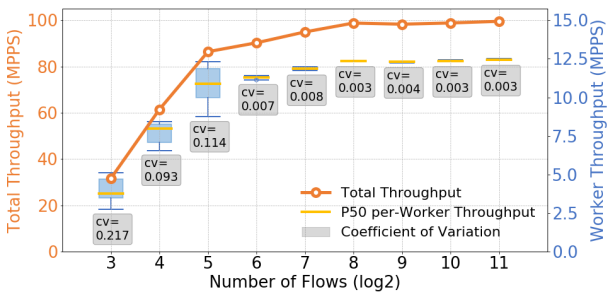


Figure 13: Throughput for different numbers of flows (in log_2 scale) with eight producers/Tx cores and eight worker cores.

6.2 Use of an Advanced Feature: Core Affinity Migration of Atomic Queues

Recent practices of software-defined networking (SDN) and network functions virtualization (NFV) segment a larger network flow into smaller, more manageable parts, called mini-flows [1], such that traffic can be handled more efficiently and with finer granularity. In this case, each mini-flow is usually processed by the same host CPU core to minimize synchronization overhead, and the core affinity is fixed throughout its lifetime. However, static core affinity can lead to load imbalance when a mini-flow becomes significantly busier than others, causing the assigned core to be overutilized while others remain underutilized. Such imbalance results in inefficient CPU resource usage. DLB atomic queue allows worker cores to operate on a per-flow basis without using locks, while also dynamically migrating core affinity when possible. DLB schedules the QEs with dynamic core affinity based on the flow identifier specified when enqueuing DLB.

We evaluate the dynamic core affinity migration with microbenchmarks to understand the core affinity migration of DLB's atomic queues. In all experiments, we use 8 producer cores and 8 worker cores. Figure 13 shows that the overall throughput improves when the number of flows increases mainly because the worker core affinity migration happens more frequently. In addition, we calculate the Coefficient of Variation (CV) based on the load on each worker core to determine how the load is distributed across all worker cores. Since the microbenchmark simply touches then drops the request, the ideal scenario for load balancing is achieving an equal distribution of requests across all worker cores which corresponds to a smaller CV. From Figure 13, we can observe that CV decreases as the number of flows increases, which indicates a better load balancing among all worker CPU cores. We also observe that DLB determines migration internally based on CPU utilization when there is no QE with the flow identifier being processed or scheduled to any worker core (*i.e.*, in any consumer queue). This observation implies that core affinity migration happens more frequently if the CQ depth is smaller. The above observations justify that it makes sense to use mini-flows with DLB atomic queue for better locality and performance of heavy flows by segmenting them with client- or application-provided enqueue flow identifiers.

7 Related Work

Software-based Load Balancer. Shenango [45] addresses the challenge of maintaining microsecond-level tail latency for data center applications and proposes a software scheduler that reallocates CPU cores at fine granularity (every 5 us), exploiting the maximum CPU efficiency for both latency critical and batch processing applications. It monitors application latency and system performance counters, and runs an IOKernel as a privileged centralized scheduler to distribute packets and dynamically orchestrate core reallocation. Caladan [12] employs a similar methodology and designs a new CPU scheduler to achieve performance isolation between tasks without resource partitioning. However, these works are limited to relatively low network traffic rates (*e.g.*, 10s of Gbps), and the centralized CPU core(s) for scheduling and performance monitoring is hard to scale and leads to inefficiency of CPU core resource usage.
NIC-based Load Balancer. Another research direction relies on NIC hardware to direct packets. In addition to the conventional fixed-function RSS, configurable techniques, such as Intel Flow Director [21] and NVIDIA/Mellanox's flexible parser [40], can be used to direct packets to the core running the application that will consume them to improve locality. This is done by matching hashes of each packet's headers against a filter table populated either by software or by sampling outgoing packets from each core. While these techniques can improve packet processing locality over RSS, they do not necessarily perform better load balancing between cores because of the inherent imbalances of network flows. Intel Application Device Queues (ADQ) [20] isolates hardware NIC queues between applications leading to better performance isolation and predictability but does not address load imbalance between cores. There are also more reactive and flexible NIC load balancing proposals with customized and programmable hardware. Turbo [50] is a SmartNIC-based inter-core load balancer for RPCs. It can improve tail latency by steering requests to cores with shorter queues, however, it only supports unreliable datagrams and requires a specialized FPGA-based SNIC. RingLeader [28] proposes offloading intra-server orchestration tasks to a SNIC. However, it still relies on software running on CPUs to perform monitoring and policy updates and the performance and scalability may be bottlenecked by the limited on-NIC resources as the network bandwidth grows.
SmartNIC to Device Communication. Lynx [53] is an accelerator-centric server architecture that offloads both the data and control planes of a server from the host CPU to a SmartNIC. By leveraging PCIe peer-to-peer (P2P) communication, Lynx allows accelerators to directly exchange data without involving the host CPU. To manage incoming network traffic, Lynx employs a dispatch policy that distributes messages across message queues located in accelerator memory. However, it does not account for load balancing between these queues, which can impact performance under certain workloads. Rambda [57] takes a different approach, enabling direct communication between cache-coherent accelerators and remote clients over RDMA. This is achieved by allowing accelerators to access RDMA work queues in the host memory via the cache-coherent interconnect. Unlike the PCIe P2P mechanism, Rambda's approach requires data to flow through the host CPU before reaching the accelerator. FlexDriver [11] implements accelerator-to-NIC communication over PCIe P2P by introducing a hardware-based NIC driver within the accelerator. This design allows the accelerator to control the NIC in a manner similar to how a CPU interacts with it. In contrast, AccDirect avoids the need for such hardware modifications. The DLB in AccDirect acts solely

as the target of RDMA `WRITE` operations, without initiating any communication itself, simplifying the design.

8 Conclusion

The advent of DLB, a commodity on-chip accelerator for queue management and load balancing, opens up new design and optimization space for intra-host load balancing mechanisms. This work paves the way for such explorations by a comprehensive characterization study on DLB's performance, a novel design enabling end-to-end load balancing with device-DLB direct communication named AccDirect, and a set of guidelines for practical and efficient DLB usage. Our evaluations show that AccDirect-based DLB is able to reduce system-wide power consumption by up to 10% and offer 14%–50% higher throughput for an end-to-end RDMA-based KVS application, Masstree, compared against the baseline. Furthermore, the AccDirect framework enables new opportunities for accelerator coordination, allowing the offloading of datacenter tax execution from the host CPU, thereby improving overall system efficiency.

Acknowledgments

This work was supported in part by a generous gift from Intel Corp; by a National Research Foundation of Korea (NRF) grant (No. RS-2024-00405857) funded by the Korea government (MSIT); and by the MSIT, Korea, under the Global Scholars Invitation Program (RS-2024-00456287) supervised by the IITP.

A Artifact Appendix

A.1 Abstract

Our artifact contains all source files for AccDirect and provides instructions on how to reproduce key experimental results shown in the paper. There are three key sets of experiments: (1) experiments that evaluate the throughput and p99 latency of existing intra-host load balancers with both core-to-core and end-to-end setups in (Figure 3, Figure 4, Figure 5); (2) experiments showing the benefits of using AccDirect in p99 latency, throughput, and power saving (Figure 9, Figure 10); (3) microbenchmark experiments that justify the guidelines to make the best use of DLB in §6.

A.2 Artifact check-list (meta-information)

- **Program:** Masstree [50], DPDK [7] and `libdlb` [18] benchmarks.
- **Compilation:** gcc 9.4+ and Python 3
- **Run-time environment:** Ubuntu 22.04 LST with Linux kernel 6.2+; DPDK 22.11.2 [7]; DLB Linux release 8.9.0 [18].
- **Hardware:** A server with Intel Xeon Gold 6438Y+ CPU with DLB enabled and NVIDIA BlueFiled-3 SmartNIC; a client with Intel Xeon E5-2660 v4 CPU and ConnectX-6 Dx NIC.
- **Metrics:** Throughput, p99 latency, average latency, and power consumption.
- **Output:** Figures and raw experimental results are stored under `scripts/fig*/`.
- **How much disk space required (approximately)?:** $\leq$ 20 GB.
- **How much time is needed to prepare workflow (approximately)?:** 1.5 hours.
- **How much time is needed to complete experiments (approximately)?:** $\sim$ 30 hours.
- **Publicly available?:** Yes, on https://github.com/ece-fast-lab/ISCA-2025-DLB.
- **Code licenses (if publicly available)?:** MIT license, BSD-3-Clause and GPL-2.0 license.

A.3 Description

A.3.1 How to access: The source code, scripts, and instructions on how to use AccDirect can be accessed on GitHub (https://github.com/ece-fast-lab/ISCA-2025-DLB).

A.3.2 Hardware dependencies: A server with an Intel 4^{th}-generation Xeon Scalable Processor which has at least one instance of on-chip accelerator, Dynamic Load Balancer (DLB). We have tested AccDirect on Intel Xeon Gold 6438Y+ CPU. We also use NVIDIA Bluefield-3 on the server which is a PCIe-attached Arm CPU-based SmartNIC. A client with 100/200 Gbps ConnectX-6 Dx NIC, we tested with Intel Xeon E5-2660 v4 (Broadwell).

A.3.3 Software dependencies: AccDirect relies on a modified version of DPDK and DLB driver. Meson, Ninja, and gcc 9.4+ are required to build benchmarks and applications. Python 3 is required to run experiments and generate figures.

- **Linux kernel:** 6.2+.
- **DLB software release:** 8.9.0 [18].
- **DPDK:** 22.11.2 [7] (modified to support using DLB and dynamic queue relink).
- **Masstree:** RDMA-based Masstree (modified from [50]).

A.4 Installation

Detailed up-to-date installation instructions can be found on GitHub Experiment Workflow. Here is a quick start. First, make sure the following kernel boot parameters are properly set in `/etc/default/grub`. Then, run `sudo update-grub` and reboot the server to apply these modifications.

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash intel_iommu=on iommu=pt
      no5lvl" #isca2025
```

Second, clone the artifact repository and install all dependencies, benchmarks, and applications with our provided script.

```
$ git clone https://github.com/ece-fast-lab/ISCA-2025-DLB
$ cd ISCA-2025-DLB
$ git submodule update --init --recursive
$ git submodule update --remote
$ cd scripts/common/ && ./build_all.sh {server|snic|client}
```

Before running the experiments, please update the `REVIEWER_ID` in `scripts/common/env_setup.sh` and source the file to apply changes. Finally, enter `scripts/fig*/` to run scripts and generate figures.

A.5 Experiment workflow

Follow Experiment Workflow in the GitHub README on how to use scripts and the recommended order of experiments.

A.6 Evaluation and expected results

As our experiments are conducted on real hardware over the network, occasionally with non-deterministic behaviors [15, 43], the results may slightly vary due to several influencing factors. We summarize the key observations and takeaways from each figure showing our experimental results.

Figure 3: DLB-based solutions (`dlb-ed` and `dlb-lib`) can scale with the number worker cores used. The highest throughput of software-based load balancers (`dpdk-pd` and `dpdk-ed`) is ∼40 MPPS.

Figure 4: Achieving the maximum throughput of 100MPPS requires five CPU cores solely to prepare and enqueue QEs to DLB.

Figure 5: Using DLB (`dlb-ed`) for intra-host load balancing outperforms software-based solutions (`dpdk-pd` and `dpdk-ed`) and `rss`.

Figure 9: AccDirect which directly enqueues to the DLB from SmartNIC (`acc-dlb-lib`) provides lower p99 latency and consume less power than the host CPU-based baseline (`dlb-lib`).

Figure 10: AccDirect (`acc-dlb-lib`) offers the higher request rate under 60μs p99 latency constraint. While the trends across Figures 10(a)–(c) are similar, reproducing all of Figures 10(a)–(c) takes hours to get a reliable p99 latency value. Therefore, We only reproduce Figure 10(a) as a representative result.

Figure 11: (a), (b), and (e) demonstrate one of the primary observations—Implication 3—made in Section 6: using LDB ports as producer ports incurs greater overhead than DIR ports, as evidenced by the higher average latency when the enqueue rate is fixed at a low level. Since (e) captures (a) and (b), we reproduce only Figure 11(e). Since (c) and (d) are to support only minor observations, we omit reproducing them.

Figure 12: (a) The average latency remains relatively stable up to priority level 5 but increases significantly beyond this point. (b) The queue priority supersedes the packet priority.

References

[1] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. 2018. Detecting heavy flows in the SDN match and action model. *Computer Networks* 136 (2018).

[2] Jasmin Ajanovic, Mahesh Wagh, Prashant Sethi, Debendra Das Sharma, David J. Harriman, Mark B. Rosenbluth, Ajay V. Bhatt, Peter Barry, Scott Dion Rodgers, Anil Vasudevan, Sridhar Muthrasanallur, James Akiyama, Robert G. Blankenship, Ohad Falik, Avi Mendelson, Ilan Pardo, Eran Tamari, Eliezer Weissmann, and Doron Shamia. 2017. Atomic operations in PCI express. https://patents.google.com/patent/US9535838B2/en?q=(Atomic+operations+in+PCI+express)&oq=+Atomic+operations+in+PCI+express+

[3] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. 2023. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*.

[4] Bates, Stephen and Duer, Oren. 2018. Enabling the NVMe™ CMB and PMR Ecosystem. https://nvmexpress.org/wp-content/uploads/Session-2-Enabling-the-NVMe-CMB-and-PMR-Ecosystem-Eideticom-and-Mell….pdf.

[5] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2010. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review* 40, 1 (2010).

[6] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. 2017. Fast in-memory transaction processing using RDMA and HTM. *ACM Transactions on Computer Systems (TOCS)* 35, 1 (2017), 1–37.

[7] DPDK. Accessed in 2024. DPDK All Releases. https://fast.dpdk.org/rel/.

[8] DPDK. Accessed in 2024. Event Device Library. https://doc.dpdk.org/guides-18.05/prog_guide/eventdev.html.

[9] DPDK. Accessed in 2024. Packet Distributor Library. https://doc.dpdk.org/guides/prog_guide/packet_distrib_lib.html.

[10] Eddie Kohler. Accessed in 2024. Masstree. https://github.com/kohler/masstree-beta.

[11] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. 2022. FlexDriver: a network driver for your accelerator. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 22)*.

[12] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[13] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. 2023. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA 23)*.

[14] Jinghan Huang, Jiaqi Lou, Yan Sun, Tianchen Wang, Eun Kyung Lee, and Nam Sung Kim. 2023. Making sense of using a smartnic to reduce datacenter tax from slo and tco perspectives. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 28–42.

[15] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D Gribble. 2013. DDOS: taming nondeterminism in distributed systems. *ACM SIGPLAN Notices* 48, 4 (2013), 499–508.

[16] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[17] Intel Corporation. Accessed in 2024. 4th Gen Intel Xeon Processor Scalable Family, sapphire rapids. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html.

[18] Intel Corporation. Accessed in 2024. Intel Dynamic Load Balancer. https://www.intel.com/content/www/us/en/download/686372/intel-dynamic-load-balancer.html.

[19] Intel Corporation. Accessed in 2024. intel/pcm: Intel® Performance Counter Monitor (Intel® PCM). https://github.com/intel/pcm.

[20] Intel Corporation. Accessed in 2024. Intel® Ethernet 800 Series - Application Device Queues (ADQ) in a Kubernetes Environment. https://networkbuilders.intel.com/docs/networkbuilders/intel-ethernet-800-series-application-device-queues-adq-in-a-kubernetes-environment-solution-brief-1664467340.pdf.

[21] Intel Corporation. Accessed in 2024. Introduction to Intel® Flow Ethernet Flow Director and Memcached Performance. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf.

[22] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, and Nam Sung Kim. 2024. Demystifying a CXL Type-2 Device: A Heterogeneous Cooperative Computing Perspective. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 24)*.

[23] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance RDMA systems. In *2016 USENIX annual technical conference (USENIX ATC 16)*. 437–450.

[25] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd annual International Symposium on computer architecture (ISCA 15)*.

[26] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. 2012. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC 12)*.

[27] Xinhao Kong, Jiaqi Lou, Wei Bai, Nam Sung Kim, and Danyang Zhuo. 2023. Towards a Manageable Intra-Host Network. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 206–213.

[28] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella. 2023. Ringleader: Efficiently Offloading Intra-Server Orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*.

[29] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the EuroSys 2012 Conference (EuroSys 12)*.

[30] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.

[31] Mellanox. Accessed in 2024. RDMA/core: Introduce peer memory interface. https://patchwork.ozlabs.org/project/ubuntu-kernel/patch/20210830225014.1613762-3-dann.frazier@canonical.com/#2744167.

[32] Microsoft. Accessed in 2024. Introduction to Receive Side Scaling. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[33] Seyedamirhossein Mirhosseininiri. 2021. *Datacenter Architectures for the Microservices Era*. Ph. D. Dissertation.

[34] Network World. Accessed in 2024. Speed race: Just as 400Gb Ethernet gear rolls out, an 800GbE spec is revealed. https://www.networkworld.com/article/3538789/speed-race-just-as-400gb-ethernet-gear-starts-rolling-out-800gb-ethernet-gets-standardized.html.

[35] NVIDIA. Accessed in 2024. ConnectX NICs 10/25/40/50/100/200 and 400G Ethernet Network Adapters. https://www.nvidia.com/en-us/networking/ethernet-adapters/.

[36] NVIDIA. Accessed in 2024. Developing a Linux Kernel Module using GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[37] NVIDIA. Accessed in 2024. DOCA Documentation v2.10.0. https://docs.nvidia.com/doca/sdk/doca+dma/index.html.

[38] NVIDIA. Accessed in 2024. mlxconfig – Changing Device Configuration Tool. https://docs.nvidia.com/networking/display/mftv4240/mlxconfig+%E2%80%93+changing+device+configuration+tool.

[39] NVIDIA. Accessed in 2024. NVIDIA BlueField Networking Platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[40] NVIDIA. Accessed in 2024. NVIDIA MLX5 Ethernet Driver. https://doc.dpdk.org/guides/nics/mlx5.html.

[41] NVIDIA. Accessed in 2024. RDMA Aware Networks Programming User Manual: Transport Modes. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/transport+modes.

[42] NVIDIA. Accessed in 2024. Understanding the iommu Linux grub File Configuration. https://enterprise-support.nvidia.com/s/article/understanding-the-iommu-linux-grub-file-configuration.

[43] Peter Okech, Nicholas Mc Guire, and William Okelo-Odongo. 2015. Inherent diversity in replicated architectures. *arXiv preprint arXiv:1510.02086* (2015).

[44] Li Ou, Xubin He, and Jizhong Han. 2009. An efficient design for fast memory registration in RDMA. *Journal of Network and Computer Applications* 32, 3 (2009).

[45] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[46] PCI-SIG. Accessed in 2024. PCI Express® Base Specification Revision 4.0. https://pcisig.com/specifications.

[47] Perftest. Accessed in 2024. OFED perftest. https://github.com/linux-rdma/perftest.

[48] RDMA-Core. Accessed in 2024. RDMA Core Userspace Libraries and Daemons . https://github.com/linux-rdma/rdma-core.

[49] Hugo Sadok, Miguel Elias M Campista, and Luís Henrique MK Costa. 2018. A case for spraying packets in software middleboxes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNet 18)*.

[50] Hamed Seyedroudbari, Srikar Vanavasam, and Alexandros Daglis. 2023. Turbo: Smartnic-enabled dynamic load balancing of μs-scale rpcs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA 23)*.

[51] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*.

[52] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. 2023. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 105–121.

[53] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*.

[54] Yipeng Wang, Ren Wang, Andrew Herdrich, James Tsai, and Yan Solihin. 2016. CAF: Core to Core Communication Acceleration Framework. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT 16)*.

[55] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*.

[56] Shinae Woo and KyoungSoo Park. 2012. Scalable TCP session monitoring with symmetric receive-side scaling. *KAIST, Daejeon, Korea, Tech. Rep* 144 (2012).

[57] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan R. K. Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. 2023. RAMBDA: RDMA-driven Acceleration Framework for Memory-intensive μs-scale Datacenter Applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA 23)*.

[58] Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Sanjay Kumar, Philip Lantz, Vivekananthan Sanjeepan, Jorge Cabrera, Atul Kwatra, Rajesh Sankaran, et al. 2024. Intel Accelerators Ecosystem: An SoC-Oriented Perspective: Industry Product. In *the 51st Annual International Symposium on Computer Architecture (ISCA 24)*.