Efficiently Offloading Intra-Server Orchestration to NICs

Paper #156, 12 pages

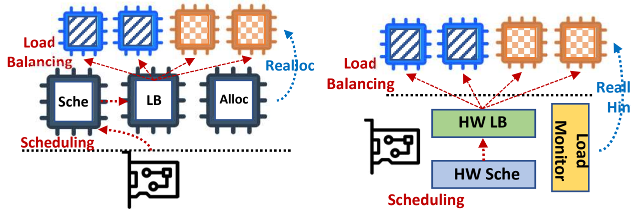
Abstract

Careful orchestration of requests at a datacenter server is crucial to meet tight tail latency requirements and ensure high throughput and optimal CPU utilization. Orchestration is multi-pronged and involves load balancing and scheduling requests belonging to different services across CPU resources, and adapting CPU allocation to request bursts. Centralized intra-server orchestration offers ideal load balancing performance, scheduling precision, and burst-tolerant CPU re-allocation. However, existing software-only approaches fail to achieve ideal orchestration because they have limited scalability and waste CPU resources. We argue for a new approach that offloads intra-server orchestration entirely to the NIC. We present RingLeader, a new programmable NIC with novel hardware units for software-informed request load balancing and programmable scheduling and a new light-weight OS-NIC interface that enables close NIC-CPU coordination and supports NIC-assisted CPU scheduling. Detailed experiments with a 100 Gbps FPGA-based prototype show that we obtain better scalability, efficiency, latency, and throughput than state-of- the-art software-only orchestrators including Shinjuku and Caladan.

1 Introduction

Modern cloud services generate thousands of RPCs in response to a single external request [35]. The services often need to provide microsecond-scale tail latencies for these RPCs to meet service level objectives (SLOs) [4]. What makes this challenging is that each server in a distributed system running multiple services receives many RPC requests of varying importance, and *intra-server orchestration*, which is necessary to provide low tail latencies and high CPU efficiency, itself incurs substantial latency and wastes CPU cycles.

Intra-server orchestration entails three aspects (Figure 1a): request scheduling, load balancing, and core assignment [6, 13, 14, 19, 24, 31, 33]. These tasks play an indispensable role in maintaining microsecond-scale tail latency, achieving high CPU efficiency and high throughput, and enforcing appropriate request prioritization. Request scheduling and load balancing determine, within and across services, in what **order** requests are processed and by **which** worker core [6, 14, 19, 33]. Load balancing reduces tail latencies by reducing worker queue lengths and improves CPU efficiency as fewer cores in the system are left idle when they could instead be processing requests. When requests or services have different SLOs or priorities, scheduling can eliminate head-of-line (HoL) blocking and guarantee tail latencies for critical workloads. Core re-allocation decides how cores process requests belonging to different services [13, 24, 31]. Fast re-allocation maintains low tail latency and improves CPU efficiency and throughput, as it can repurpose cores that are not needed by a latency-sensitive service toward batch services during periods of low load.



(a) **Intra-server orchestration** (b) **Ringleader**

Figure 1: Intra-server orchestration: today vs. Ringleader.

Coordinating orchestration tasks is a vision shared by other recent systems that have either on-loaded orchestration onto dedicated CPU cores [6, 14, 16, 19, 31, 33] or offloaded some aspects to SmartNICs [15]. Unfortunately, both sets of approaches have key limitations (Sections 2 and 8). On-loading has high latencies, poor scalability, and wastes CPU cycles. Using a dedicated centralized orchestrator core does not scale with increasing network line rates and worker core counts. Offloading orchestration to SmartNICs using on-NIC CPU cores has similar issues: the wimpy on-NIC cores have high latency overheads and scalability limitations.

We argue that effective orchestration requires a fundamentally different division of labor than onloading or SmartNIC-based approaches: Given recent advances in programmable network hardware, we start with an approach that offloads as many of the different aspects of orchestration as possible onto NIC hardware while systematically onloading onto host cores minimal functionality for precise scheduling and high performance. As NICs already process all incoming packets, offloading orchestration tasks can reduce request processing latency and save CPU cycles. We realize this division-of-labor in RingLeader, a system for offloading and executing intra-server orchestration on 100+Gbps NICs (Figure 1b).

In RingLeader, software running on CPUs uses a new OS-NIC interface to provide the NIC with per-core updates on request completions and relative priorities across arriving requests. Custom-built load balancing and scheduling units on the NIC interface with each other and leverage software-provided information to schedule precisely and enqueue requests within/across services at cores. By tracking NIC-local queues of requests waiting to be scheduled, the RingLeader NIC detects load changes and provides fine-grained reallocation hints to host cores via the same OS-NIC interface.

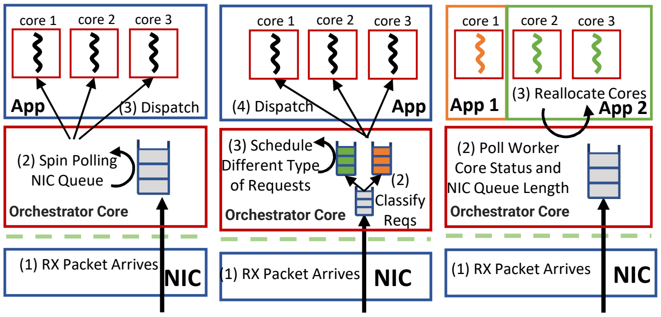
Several challenges arise in making this division-of-labor effective (Sec. 2.3): (1) carefully distributing packet buffering across the NIC and CPU cores to avoid core idling while tightly controlling request dispatch from the NIC to CPU cores; (2) coordinating request dispatching among per-core buffers and the on-NIC load balancing and scheduling engines to meet various load targets and scheduling policies; (3) developing hardware support to combine load balancing and scheduling decisions at line-rate; and (4) developing an efficient OS-NIC interface to enable low overhead coordination between the NIC and host cores. We make several innovations (Secs. 4 and 5) to overcome these challenges:

1. We leverage *shallow per-core request priority queues* alongside limited on-NIC buffering to overcome the challenges caused by PCIe latency and ensure requests dispatched by the NIC are processed quickly and with suitable prioritization.
2. We develop a novel load balancing algorithm, Join-Bounded-Shortest-Ranked-Queue (JBSRQ), which accounts for multi-service isolation/priorities *and* ensures good load balance across the per-core buffers. We build a new first-eligible-out (FEO) line-rate request scheduler that coordinates with the request load balancer.
3. We develop new NIC hardware that uses a reduction tree to calculate which core should process the current highest priority request at the line rate.
4. We introduce an OS-NIC interface with low CPU overheads and avoid generating extra PCIe messages. This provides an API for services to benefit from on-NIC orchestration, and achieves $\sim$ 50M messages-per-second for OS-NIC communication.
5. We develop simple NIC-assisted algorithms that support burst-sensitive core re-allocation across high/low priority requests by leveraging re-allocation hints provided by a new on-NIC load monitoring module.

We present a full evaluation of RingLeader's feasibility and effectiveness. From experiments performed on a 100 Gbps FPGA-based prototype, we find RingLeader is high-performance, scalable and CPU-efficient, and RingLeader provides better latency and throughput than existing state-of-the-art intra-server orchestrators, including Shinjuku [19] and Caladan [13]. For example, in an experiment with 30 worker hyperthreads, RingLeader was able to service 3× as many requests within a P99 SLO of 45μs as Shinjuku and RSS. We compare RingLeader's core allocation with Caladan running both a latency-sensitive service and a batch service, and RingLeader achieves up to 50% less latency for the latency-sensitive service and 1.3× throughput for the batch service.

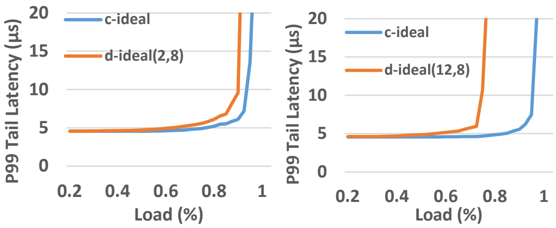
2 Background and Motivation

Online cloud services such as search, distributed model serving pipelines, and key-value caches are deployed today across thousands of physical machines. User requests to these services are composed of sequences of RPCs. Each RPC is processed using a two-layer scheduling framework: first, RPCs are assigned to servers, and then RPCs are dispatched to a service instance running on one of the server cores [43]. The latter, i.e., intra-server orchestration, which consists of load balancing requests and scheduling (ordering) them across service instances, and reallocating cores across services based on demand, play a crucial role in the ultimate performance experienced by requests.



(a) **Load Balancing** (b) **Scheduling** (c) **Core Allocation**

Figure 2: Illustrations of existing intra-server orchestration.



(a) **16 Worker Cores** (b) **96 Worker Cores**

Figure 3: Simulation results comparing centralized and decentralized scheduling. **C-ideal** uses the ideal centralized scheduling policy. **D-ideal(X,Y)** uses X decentralized orchestrator cores to schedule Y worker cores per orchestrator core.

2.1 Intra-server Orchestration Today

State-of-the-art (SOTA) intra-server orchestration relies on a centralized software-based approach running in user-space [6, 10, 13, 19, 31, 33]. This approach addresses the unpredictable/high tail latency issues of conventional in-kernel approaches [4, 14, 18]. It also addresses both the load imbalance, poor tail latencies, and poor request scheduling issues of decentralized randomized RSS (receive-side steering) approaches such as IX and ZygOS [6, 33] and the imbalance and imprecision that results from the asynchrony of reactively programming aRFS (accelerated Receive Flow Steering) rules.

Figure 2 shows how existing orchestration mechanisms work: **(1) Request Load Balancing** (Figure 2a): For each service, one orchestrator core is dedicated to: 1) polling a centralized receive queue and 2) dispatching packets to worker cores according to their load. Packets are delivered from the NIC to the orchestrator core in a centralized, First Come First Serve manner. **(2) Request Scheduling** (Figure 2b): The orchestrator core identifies different request types and schedules competing requests, *e.g.*, by suitably prioritizing them. Request scheduling reduces HoL blocking and ensures RPCs with higher priority receive service first. **(3) CPU Allocation** (Figure 2c): When there are multiple services running on the host, the orchestrator core detects when services would benefit from more cores, and reallocates cores to ensure low latency and high CPU utilization under fast-changing load.

Multiple dedicated busy-polling orchestration cores may be needed to support demanding service workloads running across many cores, or when multiple services run on a server. **Limitations:** A core that is used as an orchestrator incurs overhead and is unable to perform service-specific processing; this is problematic because the CPU is the key bottleneck in today's network-intensive workloads. Further, a single orchestrator core's maximum throughput determines scalability w.r.t request processing rates. Shinjuku's "dispatcher" that performs request scheduling and load balancing only achieves 5M RPS (Requests-Per-Second) with a single core [15, 19]. With μs-scale requests, one orchestrator core can saturate ~ 5 worker cores. However, servers today may be equipped with 100s of cores and serve 100+Gbps demand.

The overheads of performing reallocations over a large pool of worker cores are not negligible either, limiting reallocation speed and precision. For example, with 16 worker cores (hyperthreads), Shenango's core allocator can only support packet rates of up to 6.5 Mpps, and this can only saturate a 10 Gbps NIC with 128B packets [31].

To achieve higher throughput, multiple orchestrator cores could be used. Each orchestrator core handles a set of worker cores, and the server relies on NIC RSS (Receive-Side Scaling) to spread requests across orchestrator cores. However, because orchestrator cores operate independently, it is not possible to simultaneously enforce request scheduling policies and ensure even load and high core utilization.[1]

We built a discrete-event simulator to quantify the impact of using multiple orchestrator cores on a given service. For simplicity, we focus here on comparing the load balancing performance between an ideal centralized approach (c-ideal), and an ideal decentralized approach (that ignores the costs of using many orchestrator cores). We generate requests with service times following an exponential distribution with a mean of 1 μs (Exp(1)).

Figure 3 shows the results for 16-core and 96-core systems. The saturation point of the d-ideal is much earlier than the c-ideal, especially when the worker core count is high. This is because the processing time for each request is unpredictable, and using RSS to partition requests between orchestrator cores leads to severe load imbalance. This imbalance causes CPU underutilization, unnecessary queuing, and increased latency.

Recent work improves on RSS by enabling work-stealing between cores to avoid load imbalance [24]. However, work-stealing incurs CPU overheads; it is hard to enforce request weights or priorities under work-stealing; and, as recent work has shown, centralized orchestration still significantly outperforms work-stealing (Fig. 3 in [24]).

2.2 A Case for NIC-Offloaded Orchestration

Using the NIC to perform orchestration has the potential to solve the key limitations associated with software-based approaches. Because all incoming requests necessarily pass through the NIC, the NIC could be an ideal location to perform *request scheduling* and *load balancing*; the NIC can buffer incoming requests and, in theory, make centralized scheduling and load balancing decisions without added latency. In contrast with software-only approaches that must sacrifice performance and efficiency to operate at scale, high-performance on-NIC accelerators can be designed to operate at the hyperscale required by today's line rates and core counts. Additionally, offloading orchestration tasks onto NIC hardware can further improve host CPU efficiency by freeing up host cores, removing inter-thread communication overheads, and improving the accuracy of scheduling and load balancing decisions. The NIC is also a good vantage point for fine-grained network load profiling and queuing delay monitoring, so the NIC can assist with CPU scheduling by providing hints regarding incipient load arriving over the network.

We further argue that once a decision has been made to offload load balancing to the NIC, it is necessary to *also* offload scheduling and load monitoring. To achieve the c-ideal line in Figure 3, it is necessary to perform centralized buffering and load balancing. However, once requests are buffered on the NIC, it is not possible to prevent a high priority request from being blocked inside the on-NIC buffer, and centralized on-NIC buffering hides information about buffered requests from a CPU-based scheduler, precluding informed scheduling.

2.3 On-NIC Orchestration Challenges

Achieveing on-NIC orchestration is challenging:

C1: To Buffer at cores or not: On-NIC orchestration requires tight coordination between the NIC and the host. A NIC-only approach where: (1) all incoming packets are buffered on the NIC, (2) the NIC computes which core and in what order to process incoming requests, and (3) cores pull "ready" requests from the NIC to process can, in theory, yield good load balance and adhere to scheduling policies perfectly, but can experience poor throughput and fallow cores due to PCIe latency. To improve throughput and utilization, we need to unload some amount of buffering onto the cores by allowing the NIC to send new packet descriptors to a core that is not yet finished processing its current request. But it is unclear how deep these per-core buffers should be and what queueing discipline they should implement. Deep FIFO buffers can improve utilization but impose HoL blocking with high-priority

[1] aRFS allows the orchestrator cores to program flow steering rules on the NIC [8], this cannot prevent load imbalance at short time scales because rules must be installed reactively and is imprecise because rules are installed asynchronously.

requests stuck behind low priority ones at a core.

C2: Coordination across cores, load balancing, and scheduling: Per-core buffering also needs to be coordinated with the load balancing and scheduling algorithms running at the NIC. For example, a NIC-based load balancer agnostic of the priorities of requests enqueued at per-core buffers - e.g., "enqueueing a request at the shortest queue" - can easily lead to HoL blocking. Likewise, a NIC-based scheduler that simply dequeues highest priority requests buffered at the NIC and tries to enqueue them at per-core buffers may inadvertently stall both high and low-priority requests and lead to non-work-conserving behavior when the buffers at the cores serving high-priority requests are all full (Section 4).

C3: Lack of existing hardware: Existing hardware architectures are insufficient for precise on-NIC load balancing and scheduling. For example, modern hardware priority queues, notably PIFO [39], can only be used to provide programmable packet scheduling; we cannot support both programmable scheduling and load balancing with just the PIFO abstraction.

C4: Host and NIC Communication Overheads: To efficiently offload request load balancing to the NIC, the CPU needs to provide load feedback to the NIC at a fine granularity (*e.g.*, per-packet). Furthermore, with 100+ Gbps NICs, PCIe throughput can become the performance bottleneck even in combination with optimized software stacks [29]. Thus, it is necessary to ensure that the CPU and PCIe overheads of CPU-NIC communication are low.

Overall, for effective orchestration, we need new NIC architectures for offloading load balancing and scheduling, coupled with new algorithms, and new OS-NIC interfaces.

3 RingLeader Overview

RingLeader is a new NIC architecture and OS-NIC interface that enables efficient and precise orchestration. In RingLeader, scheduling and load balancing are performed in tandem by an efficient and precise novel hardware offload on the NIC, and core allocation is performed by a host datapath OS with information from a new OS-NIC interface. We aim our discussion at servers equipped with a single NIC; we discuss multi-NIC support in Sections 9 and 11.3.

3.1 System Assumptions

In RingLeader, we assume that an application runs multiple services, where each service processes a specific type of request (e.g., latency-sensitive reads vs. throughput sensitive scans). A service can be replicated using multiple instances running across cores to handle load (e.g., deploying many read-oriented instances to serve heavy read traffic). We assume that distinct services can share a core; but our system also applies to cases where services need to be isolated across cores.

In RingLeader, the host uses a Demikernel-like single address space datapath OS [42]. This achieves 1) fast multiplexing between OS tasks (e.g., buffer management, I/O processing, core allocation, coroutine scheduling) and application-specific work, and 2) fast context switching between different services' computations. Figure 4 shows the system running multiple services. Each service launches multiple coroutines[2] on multiple cores; the coroutines are scheduled and managed by the datapath OS. We assume that each service is designed to run well on multiple cores.

The datapath OS uses cooperative scheduling [42]: a long-running coroutine will *yield* voluntarily after a few microseconds of running. The datapath OS schedules the highest priority runnable coroutine once a running coroutine yields. The policy for yielding and scheduling depends on cross-service priorities.

The RingLeader NIC *buffers* received packets. This is reasonable because commercial NICs have a large amount of memory (tens of MBs of SRAM and 4–16 GBs of DRAM) [2, 7, 25–27]. If additional buffer capacity is needed, host DRAM can be used to buffer packet data with the RingLeader NIC only buffering packet descriptors.

RingLeader is designed to operate regardless of whether the transport layer is implemented in the NIC or on the CPU. On-NIC transport enables RingLeader to easily load balance and schedule at the RPC granularity, while on-CPU transport necessitates load balancing at the flow or flowlet granularity.

3.2 Key Ideas and Design Overview

RingLeader can schedule and load balance requests from different services in a given application; to this end, inter-service policies can be specified in RingLeader. We discuss how RingLeader can support policies *across* applications in Section 9. Furthermore, each service provides input to the RingLeader NIC to assist with scaling up/down the per-service allocated cores.

Ideas: RingLeader approximates an ideal centralized orchestration approach using the following ideas that address the challenges in Sec.2.3: (1) We employ *shallow priority queues* on each core (Sec. 4.2). The per-core coroutine scheduler prioritizes dequeuing certain requests from these queues to avoid HoL blocking inside the buffer. (2) The NIC uses a new Join-Bounded-Shortest-Ranked-Queue (JBSRQ) load balancing algorithm that utilizes the per-core priority queue behavior to inform load balancing decisions (Sec. 4.2). In addition, we develop a new priority-based on-NIC request scheduler called first-eligible-out (FEO) and a simple interface between the scheduler and the load balancer (Sec. 4.3); this helps coordinate the scheduler's dequeue actions with the load balancer by exposing available room at per-core buffers to the scheduler. (3) We present a novel NIC hardware architecture that uses a reduction tree to combine scheduling and load balancing decisions at line rate (Sec. 5). (4) We use memory-mapped IO and inlining metadata in packet descriptors to develop an efficient OS-NIC communication interface (Sec. 4.1).

[2] As defined in Demikernel, coroutines are light-weight user-level threads that encapsulate the OS or application computation

Example: Figure 4 illustrates how RingLeader operates when network packets are received. For simplicity, we only focus on load balancing and scheduling. Here, two services are run on a host. When a request packet enters RingLeader, it is processed by a programmable match+action (RMT) pipeline, which parses the packet's L3-L7 packet headers as necessary to identify the service that the packet belongs to and compute appropriate ranks. Then the packet is enqueued into the per-service packet buffer queue waiting to be scheduled.

The on-NIC request scheduler uses the FEO queue to schedule different services' requests according to a programmable policy. FEO schedules the highest priority service for which there is available room at a core where the service can run (this "eligibility" is provided via a mask). The on-NIC load balancer then steers this highest priority service's request to an eligible core that has the lowest rank (akin to queuing time) as computed by JBSRQ.

On each host core, the coroutine scheduler launches the runnable coroutine corresponding to the highest priority request. After the request finishes processing, the datapath OS provides load feedback to the NIC through the TX packet's descriptor or a separate MMIO register write.

We conclude the overview with a few more details.

Policies: RingLeader's load balancer, scheduler, and core allocator cooperate from top to bottom to enforce a given inter-service policy. Scheduling policies in RingLeader are expressed as a hierarchy of functions that compute the rank, rate, and/or transmission time for a packet [38–40]. Having a hierarchy of functions enables policies where multiple services can be grouped together, *e.g.*, two latency-sensitive services can be given equal priority over another service but different weights when competing with each other.

Core assignment: An on-NIC load monitor tracks the queuing condition for each service/request type. Each service can configure its trigger condition; when the scale-up/down threshold is met, the NIC sends a scaling hint to the least loaded core which runs this service/request type. The core allocator runs inside the datapath OS in a distributed fashion, *e.g.*, it can run on any core depending on which core receives the NIC hint.

4 RingLeader Design

We now discuss the design of the individual components of RingLeader. In Sec 4.1, we introduce the interface and mechanism that the NIC uses to communicate with the OS. In Sec 4.2, we introduce RingLeader's on-NIC request load balancer and our JBSRQ algorithm. In Sec 4.3, we describe the design of our FEO request scheduler and the non-blocking interface between the request scheduler and the load balancer. In Sec 4.4, we describe our NIC-assisted CPU re-allocation.

4.1 OS-NIC Interface

The OS-NIC interface in RingLeader (Table 1) is designed to minimize both HoL blocking latency and the CPU overheads of communicating orchestration metadata. We focus on the mechanisms here and outline the metadata exchanged over the interface at relevant places in later subsections.

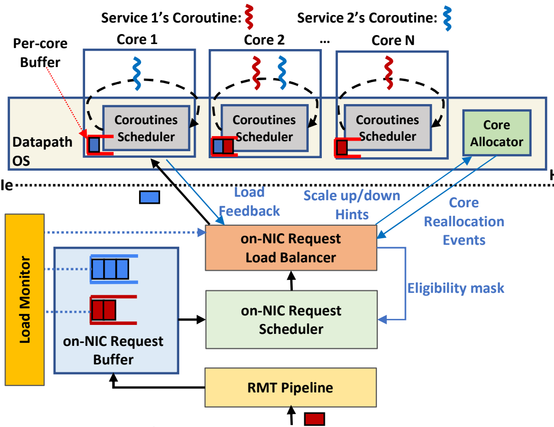


Figure 4: **RingLeader Design.**

CPU-to-NIC metadata: The datapath OS communicates with the NIC by writing to the NIC control registers via memory-mapped IO (MMIO) and via metadata in descriptors. Each core accesses a different set of cache-aligned NIC registers to increase MMIO write performance. Our microbenchmarks in Section 7.4 show the throughput for OS-to-NIC communication is roughly 50M messages per second.

NIC-to-CPU metadata: The NIC communicates with the OS through packet descriptors. The NIC-generated reallocation hint is inlined into the packet descriptor and sent to the per-core NIC queue. [3] The datapath OS polls the NIC queue and parses the NIC hint. To avoid HoL blocking, the NIC limits the number of outstanding unACKed hints per core.

Our interface allows the NIC to monitor and control the length of each per-core queue despite the inherent asynchrony caused by PCIe latency. This design also overcomes PCIe throughput limitations by avoiding generating new PCIe messages in the common case.

4.2 On-NIC Load Balancing with JBSRQ

RingLeader performs hardware-based request load balancing for each service using a Join-Bounded-Shortest-Rank-Queue (JBSRQ) algorithm to decide **when** and **where** to send a packet. JBSRQ is an extension of the Join-Bounded-Shortest-Queue (JBSQ) algorithm [21] that considers inter-service inference and priorities.

Per-core shallow priority queues: JBSRQ approximates centralized pull-based load balancing (which achieves ideal load distribution) using a combination of an on-NIC buffer and *shallow* bounded-size (*e.g.*, 4 requests) per-core queues.

When multiple services with different priorities co-exist in the same core, the per-core buffers (no matter how small) can cause undesirable HoL blocking. To avoid this HoL blocking, we implement the per-core buffers as *software priority queues*;

[3] If there is no active packet descriptor being sent from the NIC to the host, RingLeader will generate a new packet descriptor (for scaling down).

OS-to-NIC Interface	Description
RegisterService(s_id: X, ip: I, port: P, prio: O)	Register a new service X with the NIC.
EnableService(s_id: X, core_id : Y)	Notify the NIC that core Y is running service X.
DisableService(s_id: X, core_id : Y)	Notify the NIC that core Y is no longer running service X.
LoadFeedback(s_id: X, core_id : Y, count: C)	Notify the NIC that service X finishes C packets on core Y.
EnableLoadMonitor(s_id: X, trigger: T)	Enable load monitor for service X, with trigger condition T.
RearmLoadMonitor(s_id: X)	Notify the NIC that the host is ready to receive the next load hint for service X.
NIC-to-OS Interface	**Description**
LoadHint(s_id: X, hint: H)	Notify the host that service X's load has triggered the scale-up/down condition.

Table 1: **RingLeader OS-NIC Interface**

a core's coroutine scheduler uses the priority queue to enforce lightweight prioritized scheduling. The enqueue overhead of this priority queue is minimal given that the queue depth is ≤ 4, and priority calculation overhead is eliminated by the fact that the NIC scheduler computes priorities (Section 4.3) and simply carried along with packet descriptors. Further, the currently running lower priority request will yield to the highest priority request, which further reduces HoL blocking.

JBSRQ: Before describing our approach, we outline the sub-optimality of the classical join-bounded-shortest-queue (JBSQ) approach.

The main issue is that JBSQ does not consider the behavior of the host's priority queue.

Figures 5 (a), (b) show this limitation for two types of JBSQ algorithms: global-JBSQ and per-service-JBSQ. In global-JBSQ, which is used in RackSched [43], the NIC tracks per-core NIC queue lengths and always steers new requests to the core with smallest queue length. In per-service-JBSQ, which is used in nanoPU [17], the NIC tracks per-service queue length on each core and implements a JBSQ per service.

Given two services running on core 1 and core 2 where service A's priority is higher than service B, the example in Figure 5 (a) shows how global-JBSQ prevents new arriving high priority requests from preempting the on-host low priority requests. Since core 1's queue length is larger than core 2, global-JBSQ would dispatch the newly arrived service A's request to core 2. However, the optimal decision is to steer the request to core 1 as A's request would be served before B's request; the low priority request's queue length has little impact on the high priority request's completion time.

Similarly, Figure 5 (b) shows that per-service-JBSQ leads to sub-optimal performance for low priority requests.

To overcome JBSQ's limitations, we introduce JBSRQ. For simplicity, we assume each service has a single request type and that we are given priorities across services. In JBSRQ, the NIC tracks same-core services' queue lengths and service priorities. Then, for each service's request, the NIC selects the core that has the minimal rank, where rank is calculated as follows for a request for service A:

$$R[A].c = \sum_{P_x \geq P_A} Q[x].c + \lambda * \sum_{P_x < P_A} Q[x].c$$

Here, $R[A].c$ represents service A's rank on core c. P_A represents service A's priority, $Q[x].c$ represents service x's queue length on core c. λ is a constant factor between 0 and 1.

The underlying idea in JBSRQ is: when dispatching one service A's packet, the load balancer should consider the amount of the queue on a core that is contributed by requests of at least the same priority as A (because A's request cannot be scheduled ahead of such requests); the first term captures this. The rank calculation ignores the queue length contribution from all lower-priority requests. The factor λ and the summation in the second term captures the cost of waiting for a lower-priority request to yield before the higher-priority request is scheduled.

We now exemplify the benefit of using the JBSRQ policy. Figure 5 (c) shows that, when dispatching A's request, we could mostly ignore B's queue length. The calculated core 1's rank is smaller than core 2. Thus the newly arriving A's request is steered to core 1. Figure 5 (d) shows that, at low load, B's queue length can influence the load balancing policy for A's request; this is because we have added a small constant factor for the low priority request's queue length, which allows B to obtain fair service at low load. In this example, selecting idle core 2 is the optimal decision. This is because the overhead of scheduling or preempting B's request is non-negligible; thus choosing core 1 leads to a sub-optimal decision. Figure 5 (e) shows that when dispatching B's request, we should consider A's queue length, as a high priority request will always be served before a low priority request.

4.3 Non-blocking On-NIC Request Scheduler

We develop an FEO (First-Eligible-Out) priority scheduler that provides programmable per-cycle scheduling while supporting a non-blocking interface with the load balancer.

To understand why FEO is needed, consider PIFO [39], which assumes that, at any given time, all elements are eligible for scheduling. PIFO always schedules the smallest ranked element in the entire list of enqueued requests. However, given that we use shallow per-core buffers, we require that, for a given service, if a request's rank at all worker cores exceeds the queue-length bound, the load balancer must *hold* the request to avoid it getting dropped at a core. In this situation, the scheduler would block the rest of the lower-priority requests, which the load balancer could have dispatched to other potentially-idle cores.

FEO extends PIFO to avoid this problem by interfacing with the load balancer. As Figure 6 shows, when dequeuing elements, we first filter the set of elements eligible for dispatch and then schedule the smallest ranked element from that set. To enable this, the load balancer provides a bitmask that records the dispatching eligibility of each service.

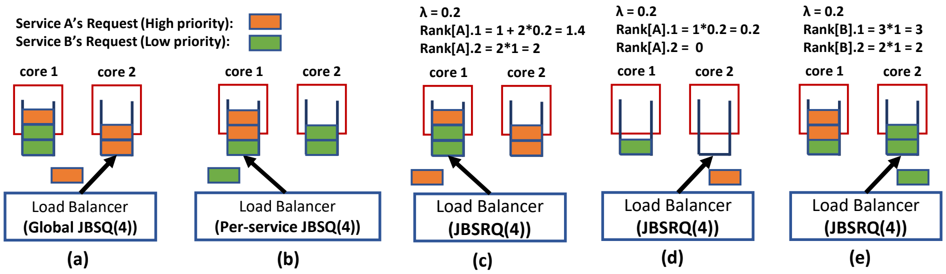


Figure 5: **Comparison between JBSQ and JBSRQ. In (b): Since core 1's low priority request queue length is smaller than core'2, per-service-JBSQ would dispatch the new arrived service B's request to core 1. However, the optimal decision is to steer the request to core 2. This is because a high-priority request's queue length would impact the low-priority request's completion time.**

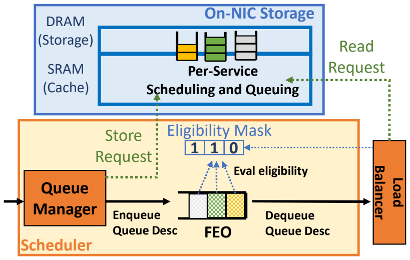


Figure 6: **On-NIC Request Scheduler**

The main difference between PIFO and FEO is the dequeue operation, which proceeds in two steps. First, we evaluate each element's eligibility in parallel by looking up its bit in the bitmask. Second, FEO uses priority encoding to select the front-most element whose eligibility is true, pops out the selected entry, and shifts the array. This design achieves a fast and parallel evaluation of elements upon dequeue. Additionally, FEO also provides buffer isolation by ensuring that the lowest priority request is dropped when buffers overflow.

4.4 NIC-assisted CPU Assignment

In RingLeader, each service could enable its own on-NIC load monitor through the interface defined in Table 1. RingLeader's load monitor supports rich triggers based on services' performance goals (e.g., latency or throughput) or scheduling policies. By default, scale-up trigger uses the congestion detection policy in Caladan [13]: if any service's request is found to be present in the on-NIC queue for two consecutive intervals, the NIC generates a scale-up hint. The scale-down policy is more conservative: within a time interval, if the maximum on-NIC queue length for a service never exceeds a threshold, the NIC generates a scale-down hint.

When a threshold is reached for a service, the load monitor generates a scale-up/down hint, inlines into the packet descriptor, and sends it to the buffer of the service's least-loaded core; it then *disarms* hint generation for this service. If no active packets are sent from the NIC to the cores, RingLeader will generate a new packet descriptor (to aid scale down).

The datapath OS polls the per-core queue and receives NIC hints. Then, the OS calls the core allocation function to decide whether/where to scale up/down this service.

Assignment strategies: In an ideal system with a perfect load balancing policy and no multiplexing overhead, the best core allocation policy would be **complete-share**: similar to Shinjuku [19], all services run on all the cores, and a service is immediately granted CPU when its request is dispatched. This policy can tolerate bursts well and ensure good CPU efficiency. In practice, though, multiplexing overheads (including preemption and yielding) are non-negligible. For example, even with state-of-the-art low-overhead interrupt mechanisms, multiplexing tasks in a core could incur at least 24% overhead [15, 19]. Therefore, frequent switching between services waste considerable CPU resources under complete-share.

Thus, RingLeader supports two additional core assignment strategies to balance the trade-off between burst tolerance and wasted CPU. In the **no-sharing dedicated** model (similar to Shenango [31], Caladan [13]), each service has its own dedicated core set. The core allocator reallocates cores between services at fine-granularity (e.g., 5 μs per reallocation). A dedicated core improves cache locality and avoids multiplexing overhead. But such a system will have worse burst tolerance since even a 5 μs reallocation interval cannot react to transient micro bursts [24].

In the **allow-sharing hybrid** model, each service has some dedicated cores, as well as cores shared with other services. The dedicated core is used to handle the long-term constant load, and the shared core is used when a burst of requests arrives. This balances multiplexing overhead and burst tolerance.

After the datapath OS successfully scales up/down a service, it calls the rearm function (Table 1) to rearm the load monitor for the service. Before being rearmed, the load monitor will not generate further hints for the service; this ensures only one in-flight hint per service and reduces the synchronization overhead inside the host's core allocator (e.g., only one core will receive the hint for a service at a time).

5 Hardware Design

We describe the hardware design of RingLeader's programmable load balancer and provide details on how it interfaces with FEO and with software priority queues. We end with an example to show how requests flow through the

RingLeader hardware. We provide benchmarks in Sec. 7.5.

5.1 Load Balancer Hardware

The load balancer unit uses two fundamental building blocks:
Per-core rank register array: JBSRQ needs to sum up the queue lengths of all services/request types on a core at or above a certain priority (Sec 4.2). Instead of spending cycles scanning and summing up different queue lengths, we maintain a pre-calculated rank register array for each priority level in the on-NIC SRAM. This register array stores each priority level's current rank on each core. When a request arrives, RingLeader directly reads out its rank according to the priority. The rank register array is updated asynchronously either when a request is dispatched or when load feedback is sent back from the worker core.
Reduction-tree-based "choose min": We use a hierarchical tree-based circuit for computing the "choose min" operation in JBSRQ to select a core. Although PIFOs are typically used for "choose min" operations in scheduling, using the same design in our load balancer is not feasible. This is because ranks are frequently updated as requests are dispatched and completed, and it is not possible to update the ranks of entries in a PIFO. Using a new reduction-tree-based design in RingLeader overcomes this limitation and allows for ranks to be updated frequently and in parallel.

Because the "choose min" operation can also be costly when the core count is high, RingLeader uses a hierarchical tree-based circuit to compute this minimum value. This circuit lends itself to pipelining, and it can calculate a minimal ranked core at every cycle. In RingLeader, we found that with 64 cores, a 3 staged reduction tree pipeline can fit on a middle-end FPGA without any utilization or timing issues.

5.2 End-to-End Example

We now present a simple example that puts the hardware components of RingLeader all together. We have two active services running on a host with two cores; the services are prioritized as shown in the top right; all requests in service are the same priority. Figure 7 (A) shows each service's priority, as well as how five existing requests are queued on the host. We assume the λ in JBSRQ is 0.2, and the depth of each per-core queue is 3; the scheduling policy is a strict priority.

Our example is shown via the numbered steps in Figure 7: To start with, in the PIFO unit, both services' eligibility bit in the mask is true. Then, (1) PIFO schedules service 1's queue descriptor, with the highest priority (see top right). (2) The request scheduler reads service 1's request F from the request buffer. (3) Request F is sent to the load balancer, which then looks up the priority register and directly reads ranks from the register array. Priority 2's rank on core 1 is 0.4 and on core 2, rank is 1.4. (4) The load balancer checks the core bitmask. If a given service is not running on a core, the rank for this core is set to infinite. However, in this example, both cores run service 1, so the rank is not reset or modified.

(5) Per-core ranks are then sent to the hierarchical reduction tree to identify the destination core with the minimal rank, core 1. Then, request F is dispatched to core 1. (6) We update core 1's priority registers according to the JBSRQ algorithm (Sec 4.2). As shown in Figure 7's (B) table and looking at core 1 queue occupancy before enqueueing F at the top right, we add one to the ranks of both priority 1 and priority 2, and we add λ to priority 3's rank. [4]

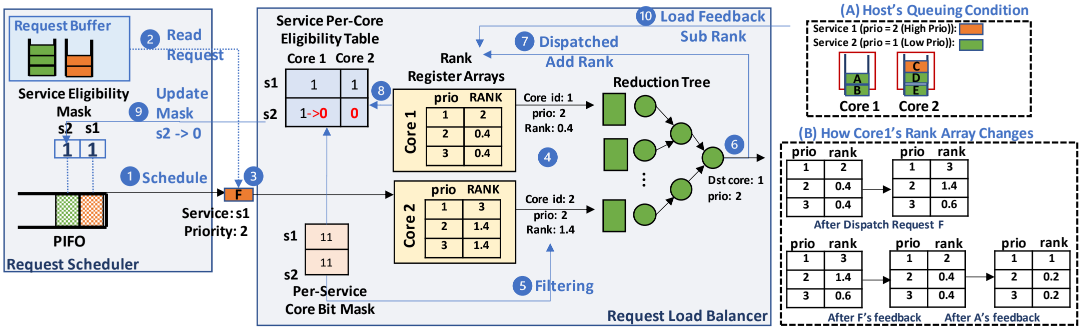
(7) After the update, priority 1's rank on core 1 reaches the rank boundary (which is 3). We then update the per-core eligibility table - where we log which service on which core has reached the rank bound, meaning that buffers are exhausted at the core and requests from the service can no longer be scheduled on the core. Given service 2 is mapped to priority 1 on core 1, we change service 2's eligibility on core 1 into 0. (8) Since service 2 is now ineligible to be dispatched on all the cores (core 2's queue was already full – see top right), in the request scheduler, we set service 2's eligibility bit into 0. Therefore, PIFO will no longer schedule service 2's request. (9) Finally, Figure 7's (B) table further shows how the rank array is updated when receiving feedback from core 1 after processing requests F and A. [5]

6 Implementation

Our implementation consists of an FPGA prototype for the RingLeader NIC, a user space NIC driver, and a library operating system built over Demikernel.
FPGA-based Prototype: The FPGA prototype is implemented in 4K lines of Verilog code, and uses the DMA Engine, Ethernet MAC and PHY provided in Corundum [12] run at a 250 MHz frequency with a data width of 512 bits.
RMT pipeline: We implemented a single-stage RMT pipeline in our FPGA prototype. The datapath OS preinstalls the appropriate rules in the pipeline through the NIC-OS interface.
On-chip Request Buffer: The request buffer is implemented using high-speed BRAM, which supports concurrent reads and writes at 128 Gbps. The size of the on-chip BRAM buffer is set to 800 KB. This buffer size can be increased by utilizing on-NIC DRAM in the future.
FEO scheduler and reduction tree: In our implementation, the FEO block runs at a 125 MHz frequency with a queue size of 64. The reduction tree supports 64 worker cores with a three-stage pipeline in the dispatcher. In the rank register array, each core has 8 physical priorities.
User space NIC driver: The user space poll mode driver for the RingLeader NIC is implemented in 1.5K lines of C code and provides DPDK-like kernel-bypass access to the NIC for standard NIC functions, in addition to providing all of the functions in Table 1.

[4] F belonged to a service with priority 2; after enqueueing it, priority 1's rank will see all 3 entries in the queue; priority 2's rank will see priority 2 requests (1) + λ (=0.2) times priority 1 requests (2); priority 3's rank will see priority 3 requests (0) + λ (=0.2) times priority 1 and priority 2 requests (3).

[5] F's feedback comes before A because of the software priority scheduler.



Core 1:

prio	RANK
1	2
2	0.4
3	0.4

Core 2:

prio	RANK
1	3
2	1.4
3	1.4

prio	rank	prio	rank
1	2	1	3
2	0.4	2	1.4
3	0.4	3	0.6

prio	rank	prio	rank	prio	rank
1	3	1	2	1	1
2	1.4	2	0.4	2	0.2
3	0.6	3	0.4	3	0.2

Figure 7: **End-to-End Example in RingLeader NIC**

The Datapath OS: We integrated RingLeader with Demikernel's catnip libOS using 800 lines of Rust. We made the following modifications to Demikernel: (1) We extended the catnip libOS to add support for RingLeader's user space driver. (2) We added multi-core support to Demikernel, which previously only ran on a single core. (3) We extended Demikernel's coroutine scheduler to enforce prioritized scheduling between different services' coroutines. (4) RPC requests yield to the coroutine after a fixed amount of work instead of always running to completion (Section 3).

7 Evaluation

Our evaluation answers the following questions:

(1) Does RingLeader achieve high performance for load balancing and request scheduling? How does RingLeader's tail latency and scalability compare to the state-of-the-art software-only approaches across different workloads and service time distributions? (Sections 7.2 and 7.3)

(2) How much do the individual components of RingLeader contribute to overall improvements? (Section 7.4)

(3) How does our NIC-assisted core assignment's resulting CPU efficiency and burst tolerance compare to the state-of-the-art software-only approaches? (Section 7.5)

(4) What is the scalability and hardware resource usage of the RingLeader NIC? (Sections 11.2)

7.1 Methodology

Testbed: We evaluate our system on a server with two Intel Xeon Gold 6326 16-core (32-thread) CPUs and 128 GB of RAM. This server runs Ubuntu LTS 20.0.4 with the 5.4.0 Linux kernel. In addition, the server has a 100G Alveo U280 Data Center Accelerator Card [1] atop which we implemented our 100G FPGA prototype. The server also has a Mellanox ConnectX-5 Ex 100 Gb NIC, which we use to run the Caladan baseline (described below). We use another client machine with a Mellanox ConnectX-5 Ex 100Gb NIC to generate load using DPDK. The client has the same CPU and OS version as the server. Our experiments (Ringleader and baseline experiments) don't consider NUMA, and direct all interrupts, memory allocations, and threads to the NIC-local socket.

RingLeader configuration: Software priority queue depth is set to 4, and λ (Sec. 4.2) is set to 0.2. (We study sensitivity to λ in the appendix (11.1), and find $\lambda = 0.2$ to be a good setting). The yielding interval is set to 5 μs.

Baselines: We compare RingLeader to three baselines:

Shinjuku [19]: Shinjuku uses centralized preemptive scheduling to achieve high-performance request load balancing and scheduling. Shinjuku only supports Intel 10G NICs and Linux kernel version 4.4.x, and it can only run on Intel cores because its fast preemption mechanism requires VT-x support. We use two Cloudlab [34] c6420 nodes (one client and one server, connected through a ToR switch) to run Shinjuku with kernel v4.4.0; each node is equipped with two 16-core (32-hyperthread) Intel Xeon Gold 6142 CPUs, and an Intel X710 10 Gigabit NIC. By default, Shinjuku uses two hyperthreads for orchestration – one for the network and another for the load balancer – collocated on the same physical core. The preemption interval is set to 5 μs. To ensure a fair comparison, we always assign one more physical core (two hyperthreads) to Shinjuku than RingLeader for running orchestration tasks.

Caladan [13]: Caladan reallocates cores between applications at a very fine granularity to increase CPU efficiency under changing workloads. We run Caladan on the same server and the same OS and kernel version as Ringleader.

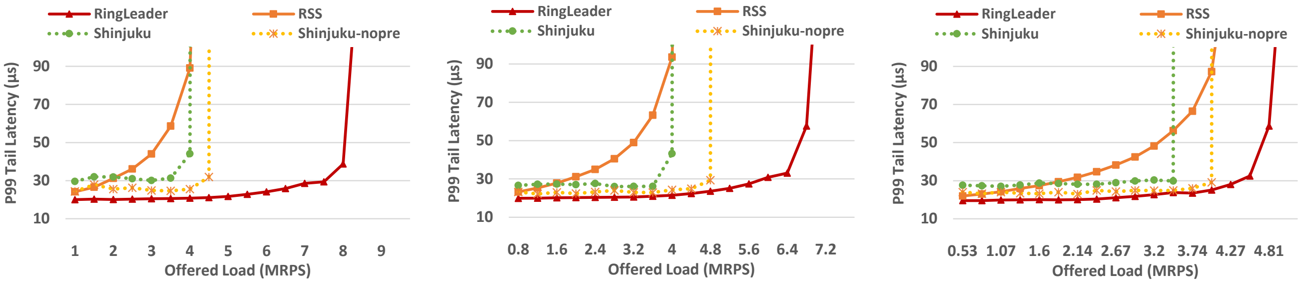
Caladan runs its IOKernel on a single dedicated core. Therefore, like Shinjuku, we always assign one more physical core to Caladan than RingLeader.

RSS: We also study a decentralized RSS-based system. In this baseline, worker hyperthreads are managed by the Demikernel datapath OS, and each worker polls its own large receiving NIC queue. Here, we use a bare metal 100G U280 FPGA NIC [1] that performs standard NIC functions.

Workloads: We employ both synthetic workloads and RocksDB.

Synthetic Workloads: (Table 2) Our synthetic workload is a server application where requests perform dummy work that we can control to emulate any target distribution of service times. This allows us to run microbenchmarks that systematically study how RingLeader and different baselines perform under different performance limits.

RocksDB Workloads: We also performed experiments with



(a) **30 workers** (b) **24 workers** (c) **16 workers**

Figure 8: Load balancing performance under Exp(3) workload.

Workloads	Description
Exp(3)	Single request type, service times follows exp distribution with mean 3 μsec.
Bimodal (95-5,5-100)	95% requests are high priority, take 5 μsec. 5% requests are low priority, take 100 μsec.
High Bimodal (99-3,1-100)	99% requests are high priority, take 3 μsec. 1% requests are low priority, take 100 μsec.

Table 2: **Synthetic Workloads**

RocksDB, a popular and widely deployed in-memory key-value store developed by Facebook [11]. We use RocksDB queries that are either GET/PUT requests or range SCANs.

To generate both the synthetic and RocksDB workloads, we developed an open-loop load generator similar to Shinjuku [19] that generates requests over user space UDP. It uses 12 threads to generate requests following a Poisson arrival process and specific service-time distributions and another 12 user space threads to receive server replies. Request latency is measured through timestamps carried inside packets. We ensure that the network speed and the load generator are not the bottlenecks in any experiment by checking for packet drops.

7.2 Load Balancing Performance

First, we evaluate the RingLeader load balancing unit using an Exp(3) workload that, for simplicity, only has a single type of request (with service times following an exponential distribution with a mean of 3 μsec). We compare RingLeader against three baselines: Shinjuku, Shinjuku-nopre, and RSS. In Shinjuku-nopre, we use Shinjuku without preemption.

Figure 8 shows the load balancing results when the server runs 30, 24, and 16 workers. Each worker is a hyperthread, and all workers run on the NIC-local socket. Across all levels of load, RingLeader provides the lowest tail latency. Also, RingLeader has the highest saturating throughput for all worker counts. This shows that RingLeader has the best scalability and load balancing precision. In contrast, the dedicated orchestrator core Shinjuku uses for networking and load balancing becomes a performance bottleneck when the offered load is $>$ 4.8 MRPS (Million-Request-Per-Second) and preemption is disabled and when the offered load is $>$ 4 MRPS and preemption is enabled. Figure 8 also shows that RingLeader consistently outperforms RSS, which distributes load unevenly across cores, hurting tail latency.

7.3 Scheduling Performance

Synthetic Workload: We now study RingLeader's ability to achieve high-performance scheduling across services/request types. We use the High Bimodal workload (Table 2) with two types of requests for one service: high priority requests that follow Exp(3) and low priority requests that follow Exp(100). We turn off core assignment in these experiments, so the two types of requests run on all worker hyperthreads.

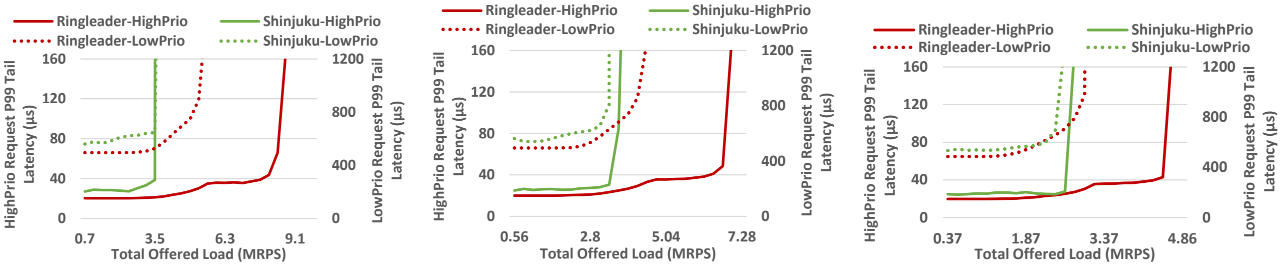
Figure 9 shows the results from this experiment. For all worker counts, RingLeader consistently outperforms Shinjuku. This is because Shinjuku's orchestrator cores become bottlenecked when the offered load is larger than 3.5 MRPS. In contrast, RingLeader has better scalability and lower latency.

Also, in RingLeader, high priority requests can still maintain low tail latency even when the low priority requests' load is saturated (*e.g.*, at load $>$ 3.5 in Fig 9a). This is because the on-NIC scheduler provides buffer isolation (Sec. 4.3) and ensures each request type is dropped separately.

RocksDB Workload: Next, we evaluate RingLeader's scheduling performance under the RocksDB workload. We use two request types: GET requests for a single key-value pair that execute within 5μsec; SCAN requests that scan 200 key-value pairs and require 60μsec. We also vary the yielding interval for SCAN across 40 items-per-yield (Y40), 20 items-per-yield (Y20), and 10 items-per-yield (Y10). Figure 10 shows that RingLeader's prioritized scheduler allows GET requests to avoid long queuing times due to SCAN requests. Aggressive yielding improves tail latency performance for short requests and adds a constant overhead to scan requests. RingLeader-assisted core re-allocation is a way to get around the constant yielding overhead.

7.4 Benefits of RingLeader Components

We now study how the individual components in RingLeader's load balancing and scheduling functionality contribute to overall performance; core assignment is turned off here (we study it later in Sec. 7.5). Figure 13 (in appendix) presents a comparison of RingLeader and reduced versions of RingLeader that remove/replace a single component . We use the bimodal workload shown in Table 2. **FEO:** Here, we turn off our scheduler eligibility bitmask



(a) **30 workers** (b) **24 workers** (c) **16 workers**

Figure 9: Load balancing and scheduling performance under High Bimodal workload.

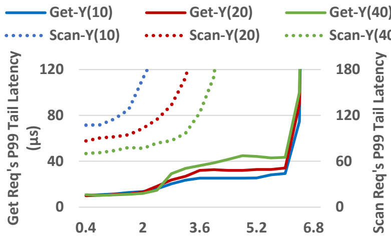
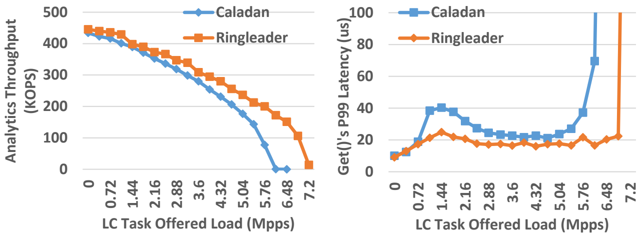


Figure 10: **RocksDB performance.**



(a) **Analytics throughput.** (b) **GET's P99 tail latency.**

Figure 11: Comparison with Caladan.

(Section 4.3), causing it to be degenerate to vanilla PIFO (Blocking_PIFO). Figure 13 shows that PIFO's performance is much worse than RingLeader when the load increases for high priority requests. This is because, in our two-request-type setting, low priority requests prevent high priority requests from entering the load balancer at high load.

Global-JBSQ: Next, we evaluate global-JBSQ(4), a load balancing algorithm similar to RackSched [43] where the NIC tracks per-core queue lengths and always steers new requests to the core with the smallest queue length. The queue length bound for each core is set to 4. Figure 13 shows that, even with preemption and the software priority queue enabled, the high priority request's tail latency is much worse than JBSRQ when the offered load is $>$ 2.24 MRPS. Because global-JBSQ does not consider the behavior of the software priority queue, a burst of long requests can occupy all per-core NIC queues, and new arriving high priority requests cannot be dispatched.

Per-service-JBSQ: We evaluate per-service-JBSQ(4), a load balancing policy that is similar to nanoPU [17] where the NIC implements JBSQ(4) per service. On each worker, the queue length limit for a service is 4. Figure 13b shows that low priority requests have worse performance than RingLeader because, when dispatching a low priority request, the NIC only looks into low priority requests' queue length, ignoring the influence of high priority requests on the queuing delay of low priority ones.

No Software Priority Queue: Figure 13a shows what happens when we disable per-core software priority queues (and cooperative yielding). High priority requests suffer a lot because a burst of low-priority requests can enqueue at presently idle cores and unduly delay later-arriving sensitive requests.

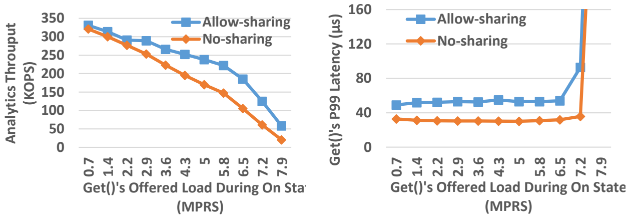
7.5 NIC-Assisted Core Assignment

We evaluate RingLeader's ability to detect load changes and aid in fast core reallocation. We experiment with two types of services running on the host. One serves high-priority latency-sensitive RocksDB GET requests. The other runs a best-effort analytics workload that continuously scans a range of the RocksDB database and performs data comparisons over the scanned results. We increase RocksDB GET's load gradually and measure offered load averaged over 10s intervals.

We compared RingLeader's core reallocation performance with Caladan. In this experiment, both Caladan and Ringleader have 16 worker hyperthreads, and the core assignment decision interval is 8μs. Furthermore, in this experiment, Ringleader uses the same CPU assignment strategy as Caladan, which is the no-sharing dedicated model.

We evaluate the analytics service's throughput and the GET request's tail latency. Figure 11 shows that RingLeader keeps the tail latencies of the GET request low while also allowing for spare CPU cycles to be shared with the best-effort analytics service. Furthermore, RingLeader yields both better GET requests tail latency and higher analytics workload throughput than Caladan because load-imbalance and work-stealing in Caladan increase latency and CPU load. For example, in Figure 11b, Caladan's latency goes up at load 1.44 Mpps since work-stealing happens most frequently at this point. In contrast, RingLeader consistently achieves near-optimal centralized scheduling and low overhead core assignment.

Next, we use the on-off traffic pattern to compare the performance of two types of CPU allocation policies: allow-sharing hybrid and no-sharing dedicated (Section 4.4). During the on state, the traffic source generates GET requests; during the off state, the traffic source stops sending. The switching



(a) **Analytics throughput.** (b) **GET's P99 tail latency.**

Figure 12: Core reallocation under different policies.

time between the on/off states is 0.8 ms. Under this pattern, core reallocation happens several times every 0.8 ms. Figure 12 shows that the allow-sharing policy has better analytics throughput as it allows the two services to coexist in the same core, accommodates small timescale bursts of arrivals, and minimizes CPU waste. However, the no-sharing has better tail latency for GET requests because using dedicated cores improves cache locality and avoids multiplexing overheads; nevertheless, allow-sharing's tail latency stays relatively low and flat for the most part. Given this information, an admin can configure RingLeader to pick a core assignment policy based on the relative importance of low tail latency for sensitive services versus not starving batch services.

8 Related Work

Software approaches: RingLeader addresses the key scalability and performance limitations of other orchestration systems like IX [6], ZygOS [33], Shenango [31], and Shinjuku [19]. ghOSt [16] and Syrup [20] use userspace CPU scheduling policies; they are complementary with RingLeader.

Hardware approaches: Shinjuku-on-SmartNIC [15] provides centralized preemptive request scheduling on an ARM-based SmartNIC, but scheduling requests on wimpy on-NIC cores has limited processing speed and introduces tens of microseconds of latency [22].

PIEO [38] extends PIFO to support efficient extraction for time-based scheduling algorithms, but it cannot simultaneously support scheduling and load balancing. This is because it only supports packet extraction as a function of time and hence cannot be used to support an eligibility mask.

Recent related work uses JBSQ to control HoL blocking when load balancing in systems with communication latency [9, 15, 17, 21, 43]. As shown in Section 7.4, our new JBSRQ algorithm can be used in these systems to improve performance under multi-service scenarios. Further, these other orchestration systems like nanoPU [17], RackSched, and RPC-Valet [9] do not perform centralized request scheduling, so high priority requests can suffer more HoL blocking.

Elastic RSS [36] uses a NIC to perform both load balancing and core allocation. However, it buffers packets at CPU cores and not on the NIC, leading to load-imbalance, and it does not schedule packets, leading to HoL blocking.

Transports: Improvements in transport protocols are complementary to RingLeader. Both new transport protocols like MTP [41] and EQDS [30] and projects that offload transport protocols to SmartNICs [3, 28, 32, 37] can enable message-level load balancing and scheduling in an orchestration system, so RingLeader would benefit from their adoption.

9 Discussion

Multi-NIC support: Although our design as presented so far assumes a single NIC per server, there are a few different ways RingLeader can be configured to support multiple NICs in a single server: 1) a master/slave configuration (described below), 2) hard-partitioning workers (Section 11.3), or 3) a cooperative configuration (Section 11.3). In a master-slave configuration, each NIC will transfer data to main memory independently but not perform dispatching. Instead, each slave NIC sends descriptors about pending requests to the master NIC, which is solely responsible for orchestration. With the master-slave configuration, NICs need to generate extra control messages and requests from slave NICs must wait for descriptors to be sent between NICs over the PCIe bus before dispatching, which incurs a small amount of extra latency.

Multiplexing Mechanism: RingLeader uses cooperative scheduling, requiring developers to insert yield statements for low priority services. However, RingLeader is also compatible with other multiplexing mechanisms, such as: 1) optimized APIC interrupts [15], and 2) compiler interrupts [5]. Optimized APIC interrupts are a low priority service that can set a timer that will deliver a low-overhead interrupt once the time slice expires. Compiler interrupts use compile-time instrumentation to allow programs to call an interrupt handler at a regular intervals with little performance impact.

Multi-process Support: RingLeader inherits a key assumption in Demikernel today [42], namely that the data path OS and services run in a single process. However, similar to recent work like Snap [23], we can extend RingLeader to support multiple processes by using Demikernel as a standalone process that multiplexes I/O across client processes through shared memory regions. Such an extension would naturally enable RingLeader to support policies across applications (as opposed to policies across services in an application).

10 Conclusions

Existing intra-server orchestration approaches have limited scalability, poor precision, and high overheads. We address these problems by introducing RingLeader, a new system that efficiently offloads orchestration in their entirety to a programmable NIC while minimally onloading limited functions to host cores. RingLeader introduces a novel OS/NIC interface, a new load balancing algorithm and scheduler, and a hardware element that combines the decisions of the two. Our experiments with a prototype on a 100 Gbps FPGA NIC show that RingLeader offers good tail latency, high throughput, good CPU utilization, and effective core reallocation.

References

[1] Xilinx alveo u280. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[2] Alpha Data. ADM-PCIE-9V3 - High-Performance Network Accelerator. https://www.alpha-data.com/pdfs/adm-pcie-9v3.pdf.

[3] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 93–109, 2020.

[4] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[5] N. Basu, C. Montanari, and J. Eriksson. Frequent background polling on a shared thread, using light-weight compiler interrupts. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1249–1263, 2021.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. {IX}: A protected dataplane operating system for high throughput and low latency. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 49–65, 2014.

[7] Broadcom. Stingray SmartNIC Adapters and IC. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[8] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal. Understanding host network stack overheads. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM. Association for Computing Machinery, 2021.

[9] A. Daglis, M. Sutherland, and B. Falsafi. Rpcvalet: Ni-driven tail-aware balancing of µs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[10] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 621–637, 2021.

[11] Facebook. RocksDB. http://rocksdb.org/.

[12] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46. IEEE, 2020.

[13] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[14] H. Golestani, A. Mirhosseini, and T. F. Wenisch. Software data planes: You can't always spin to win. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 337–350, 2019.

[15] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 60–68, 2019.

[16] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis. GhOSt: Fast and flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP. Association for Computing Machinery, 2021.

[17] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, N. McKeown, and C. Kim. The nanopu: Redesigning the cpu-network interface to minimize rpc tail latency. *arXiv preprint arXiv:2010.12114*, 2020.

[18] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 489–502, 2014.

[19] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for µsecond-scale tail latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 345–360, 2019.

[20] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP. Association for Computing Machinery, 2021.

[21] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. {R2P2}: Making {RPCs} first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.

[22] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333, 2019.

[23] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[24] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2022.

[25] Mellanox Technologies. Innova - 2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf.

[26] Mellanox Technologies. Mellanox BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[27] Mellanox Technologies. NVIDIA Mellanox BlueField-2 DPU. https://www.mellanox.com/products/bluefield2-overview.

[28] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2020.

[29] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[30] V. Olteanu, H. Eran, D. Dumitrescu, A. Popa, C. Baciu, M. Silberstein, G. Nikolaidis, M. Handley, and C. Raiciu. An edge-queued datagram service for all datacenter traffic. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2022.

[31] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 361–378, 2019.

[32] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the RPC tax in datacenters. In *Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO. Association for Computing Machinery, 2021.

[33] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[34] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014.

[35] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM. Association for Computing Machinery, 2015.

[36] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 71–77, 2019.

[37] R. Shashidhara, T. Stamler, A. Kaufmann, and S. Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2022.

[38] V. Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 367–379, 2019.

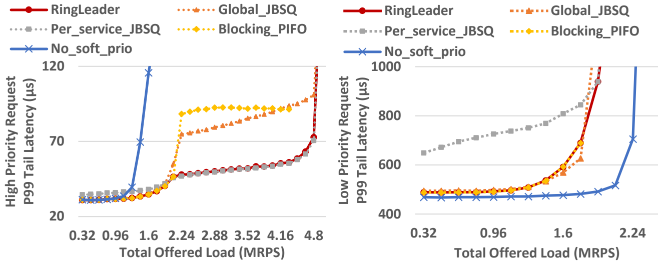
[39] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.

[40] B. Stephens, A. Akella, and M. Swift. Loom: Flexible and efficient {NIC} packet scheduling. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 33–46, 2019.

[41] B. E. Stephens, D. Grassi, H. Almasi, T. Ji, B. Vamanan, and A. Akella. Tcp is harmful to in-network computing: Designing a message transport protocol (MTP). In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets. Association for Computing Machinery, 2021.

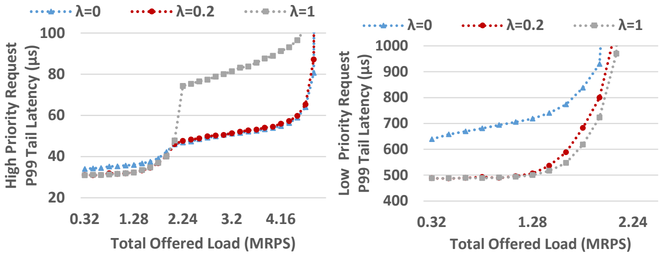
[42] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam. The Demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP. Association for Computing Machinery, 2021.

[43] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1225–1240, 2020.



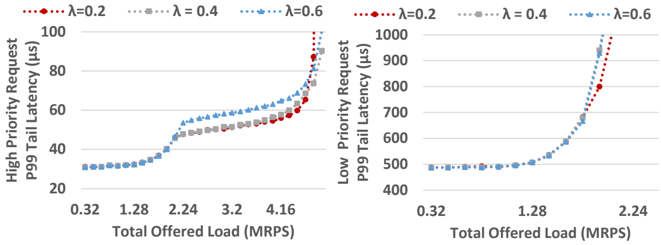
(a) **High Prio's P99 tail latency** (b) **Low Prio's P99 tail latency**

Figure 13: Comparison of RingLeader and reduced versions.



(a) **High Priority Request's P99 tail latency** (b) **Low Priority Request's P99 tail latency**

Figure 14: JBSRQ's performance under different λ.



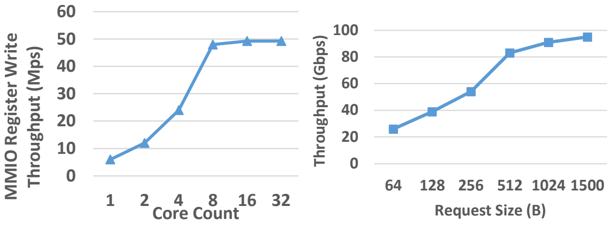
(a) **High Priority Request's P99 tail latency** (b) **Low Priority Request's P99 tail latency**

Figure 15: JBSRQ's performance under different λ (0< λ <1).

11 Appendix

11.1 JBSRQ Policy parameters

We evaluate how the constant factor λ in JBSRQ influences system throughput and tail latencies under the Bimodal workload. Figure 14 shows that, for both high priority and low priority requests, a constant λ between 0 and 1 brings significant performance gain over $\lambda = 0$ or $\lambda = 1$. When $\lambda = 0$, the rank calculation does not consider preemption overheads. Under low load (load < 2.24), $\lambda = 0$ leads to unnecessary preemption, increasing tail latency for both high priority and low priority requests. When $\lambda = 1$, JBSRQ equals global-JBSQ. This prevents new arriving high priority requests from preempting the on-host low priority requests when load > 2.24, increasing tail latencies for high priority requests. An intermediate λ is necessary to achieve good performance under both high and low load. Figures 15 shows how different λs between 0 and 1 influence JBSRQ's performance. There is not much difference between $\lambda = 0.2$ and $\lambda = 0.4$. When λ grows to 0.6, performance drops because of the same issue with $\lambda = 1$.



(a) **MMIO throughput.** (b) **RingLeader throughput.**

Figure 16: Scalability of RingLeader

Module	Setting	LUTs(%)	BRAM(%)
Load Balancer	(16 priorities * 64 cores)	2.82	0.10
	(16 priorities * 128 cores)	2.86	0.10
	(32 priorities * 64 cores)	6.07	0.00
Scheduler	FEO = 16	0.24	0.01
	FEO = 64	1.00	0.01
Packet Buffer	800 KB	0.16	6.54

Table 3: **FPGA resource usage for different components.**

11.2 Scalability and Resource Usage

We now study RingLeader's performance upper bound and it's hardware resource usage.

OS-NIC interface: Figure 16a shows the throughput for the OS-to-NIC interface. The communication throughput between a single worker hyperthread and the NIC is 6M register writes per second, and with 8 workers, the throughput can reach 50M. The result shows that RingLeader can achieve low-overhead, fast OS-to-NIC communication.

System Throughput and Latency Overhead: Figure 16b shows that RingLeader can achieve line-rate load balancing and scheduling. In this experiment, the host uses 8 worker hyperthreads, every request finishes immediately, and the rank bound is set to 16. The result shows that RingLeader can achieve 100G line rate scheduling with MTU-sized packets and 50Mpps for 64B packets.

We measure RingLeader's latency overhead by adding hardware timestamps. We find that a request can be scheduled and dispatched within 150 ns. The end-to-end host ping-pong latency is 6 μs, which is close to commercial NICs.

Hardware Resource Usage: Our U280 FPGA has 1300k LUTs in total. Table 3 shows different components' resource usage under different settings. The load balancer and scheduler occupy most of RingLeader's on-chip logic. When the load balancer is configured with 16 priorities and 64 worker counts, it consumes around 2.82% of the logic area. With 32 priorities, it consumes 6.07%. Furthermore, when FEO uses 16 entries, it consumes 0.24% of the logic area, and when the size is 64, it consumes 1.00%. Overall, we find that RingLeader can easily fit on an FPGA without utilization or timing issues.

11.3 Supporting Multiple NICs

While our design aimed at servers equipped with a single NIC, we believe that it can be extended to support multi-NIC settings as well. We discuss a few options below.

Hard-partitioning workers: The simplest approach is to partition workers across NICs. Here, for example, each NIC orchestrates its local NUMA node. This allows each NIC to perform ideal centralized scheduling independently. However, this requires all network traffic for a service to be sent to a specific NIC.

Cooperative multi-NIC orchestration: In this scenario, each worker can receive packets from multiple NICs. The control message in Table 1 is replicated across all NICs to achieve cooperative orchestration. When a request finishes, the worker sends load feedback to all NICs, so that every NIC knows up-to-date worker queue lengths. Also, when a NIC dispatches a request to a worker, the NIC notifies other NICs of this action. Replicating control messages is the primary trade-off introduced by this approach. However, this is unlikely to be a bottleneck given that control messages are small and MMIO throughput (16a) over PCIe is high.