2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071135

Turbo: SmartNIC-enabled Dynamic Load Balancing of µs-scale RPCs

Hamed Seyedroudbari
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: hamed@gatech.edu

Srikar Vanavasam
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: srikar@gatech.edu

Alexandros Daglis
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: alexandros.daglis@gatech.edu

***Abstract*—Online services are decomposed into fine-grained software components that communicate over the network using fine-grained Remote Procedure Calls (RPCs). Inter-server communication often exhibits patterns of wide RPC fan-outs between software tiers, raising the well-known tail at scale effect and necessitating mechanisms that curb long response tail latencies. When handling µs-scale RPCs, request distribution across the cores of multicore servers is a major determinant of the resulting tail latency. Software approaches for inter-core RPC balancing introduce considerable overheads, throttling a server's peak throughput. On the other hand, existing NIC-based hardware mechanisms ameliorate software and inter-core synchronization overheads, but result in inter-core load imbalance that leaves significant performance improvement headroom.**

We introduce Turbo, a hardware on-NIC load-balancing mechanism that achieves near-optimal inter-core load distribution for the most fine-grained, light-tailed RPCs with service times of only a couple of µs. We implement Turbo on a programmable NIC and evaluate it on a range of different service time distributions and with a high-performance Key-Value store. Compared to hardware NIC-based mechanisms that statically spread load across cores, Turbo boosts throughput under a 99% latency Service Level Objective (SLO) of 30× the service time by up to 5×, and by up to 95× for a more aggressive 10× SLO target.

I. Introduction

Online services are the backbone of modern digital economies. Such services are typically deployed on large datacenters and are decomposed into multiple software tiers that communicate over the datacenter network via Remote Procedure Calls (RPCs). Because of the significance of controlling each service component's *tail* latency [13], service providers do not solely focus on maximizing raw throughput, but must optimize for attainable throughput under a Service Level Objective (SLO). The key metric of *throughput under SLO* often entails a response tail latency target, such as the 99th percentile or higher.

Due to the ongoing trend of extreme software decomposition, a plethora of online services feature performance-critical components that interact with fine-grained, µs-scale RPCs. Controlling the tail latency of such fine-grained RPCs is particularly challenging due to the well-known inefficiency of modern systems in handling µs-scale events [7]. A key tail latency determinant when handling µs-scale RPCs on manycore server CPUs at millions of requests per second is RPC distribution across cores: any load imbalance results in excessive queuing, which in turn defines each request's response latency. This work focuses on leveraging the programmability capabilities of modern smartNICs to improve tail latency by mitigating the impact of inter-core load imbalance when handling fine-grained RPCs.

Prior work in the space of µs-scale RPC load balancing broadly falls into two categories: software- and hardware-based. The flexibility of advanced software mechanisms [14], [22], [36], [38] is necessary to approach load-balancing optimality in the face of wide service time variability. However, software mechanisms entail overheads that become considerable in the context of the most fine-grained RPCs. To demonstrate, ZygOS [38] achieves near-ideal throughput under SLO for RPCs with a 10+µs average service time, but its efficiency drops precipitously for shorter service times. Maximizing throughput under SLO for fine-grained RPCs requires hardware-level solutions without any software intervention on the data path.

On the hardware front, prior solutions either build rebalancing mechanisms on top of existing Receive-Side Scaling (RSS) support in NICs [6], [40], or adopt exotic integrated NIC architectures [12], [20]. Systems in the former category eschew the overheads of software-based solutions, but still rely on coarse-grained rebalancing decisions at the network flow rather than RPC granularity, leaving significant performance improvement headroom. Load-balancing approaches in the latter category are not directly applicable to discrete NICs that are predominant in modern systems, due to the considerable NIC-CPU latency introduced by the I/O (typically PCIe) interface. In this work, we focus on developing a hardware load-balancing mechanism specifically for fine-grained RPCs that is readily applicable to modern platforms with discrete programmable NICs.

We propose Turbo, a NIC-based load-balancing mechanism for modern programmable NICs (aka, smartNICs), designed to maximize a server's throughput under stringent tail-latency SLOs when handling fine-grained RPCs. Turbo's simple but effective policies allow making load-assignment decisions at a per-RPC granularity, without throttling the NIC's peak line rate. Turbo's first policy, Join Shortest Queue (JSQ), assigns each newly arriving RPC to the core with the shallowest queue of pending requests. Turbo's second policy, Join Lightest

978-1-6654-7652-2/23/$31.00 ©2023 IEEE

Queue (JLQ), improves over JSQ by inferring each RPC's service time to maintain the total work assignments across cores balanced. JLQ's applicability is narrower, as it requires applications with a causal relationship between request type and resulting average service time. We demonstrate such a concrete use case using a high-performance Key-Value Store. We make the following contributions:

- We demonstrate that the ideal—for fine-grained RPCs—centralized First Come First Serve (FCFS) policy, implemented by the state-of-the-art NIC-driven load-balancing mechanism [12], is not realizable on modern discrete NICs, due to the high NIC-CPU latency introduced by the PCIe interface.
- We show that simple alternative load-balancing policies that can be implemented using minimal hardware resources on current discrete, programmable NICs approach the performance of the ideal centralized FCFS load-balancing policy.
- We present Turbo, an implementation of these simple policies on an FPGA-based programmable smartNIC and evaluate it on a range of microbenchmarks with synthetic service time distributions and on the high-performance Masstree Key-Value Store [30]. Turbo boosts throughput under a stringent 99% tail latency SLO of $30\times/10\times$ the request's service time by up to $5\times/95\times$, respectively, compared to static load distribution achieved by existing RSS NIC capabilities.
- We illustrate that Turbo's load-balancing at the individual RPC granularity is superior to advanced RSS-based mechanisms proposed in the literature [6], [40] that periodically reconfigure the NIC's load assignment functions, delivering $2.7-3.4\times$ lower tail latency, and less jitter at the same load.

Paper outline: §II provides background on the impact of load balancing on the tail latency of µs-scale RPCs and prior art in this space. §III highlights the challenges and opportunities of NIC-driven RPC load-balancing approaches via queuing simulations. We present Turbo's design and implementation in §IV and §V, detail our methodology in §VI, and evaluate Turbo in §VII. Finally, §VIII discusses related work and §IX concludes.

II. Background

A. µs-Scale RPCs and Tail Latency

The multiple tiers of online services typically communicate via RPCs. A server of one tier often communicates with several servers of the next tier, resulting in communication patterns of wide RPC fan-outs. Because of such fan-outs, tail latency becomes a critical figure of merit, as a tier's response latency is dictated by the next tier's last response [13]. Furthermore, fine-grained software decomposition often results in performance-critical tiers that handle short-lived requests, with service time in the order of a few µs. Given that modern systems are notoriously inefficient at handling µs-scale events [7], building effective mechanisms that optimize for the tail latency of µs-scale RPCs is an open challenge and an area of intensive research activity.

Prior work has identified queuing as a prime contributor to high RPC response tail latency [12], [22], [26], [38]. Poor load balancing across a manycore server CPU's cores exacerbates tail latency, because of few RPCs experiencing disproportionately long queuing time. Effective load balancing is particularly challenging for short-lived RPCs, because even sub-µs overheads introduced to improve load-balancing decisions account for a considerable fraction of service time, thus throttling peak sustainable service rate.

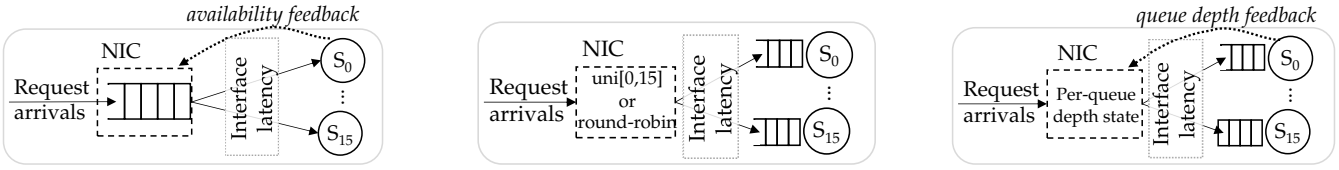
This work does not target general RPCs, but specifically focuses on the *finest-grained RPCs exhibiting single-digit µs service times with low variability*, which is the typical profile of ubiquitous software tiers such as data stores, state machine replication, network functions, and emerging "nanoservices" [20], [21]. We henceforth refer this class of RPCs as **FLT RPCs** (**F**ine-grained **L**ight-**T**ailed RPCs). Next, we summarize the most relevant areas of prior work and argue that handling FLT RPCs effectively requires making decisions in hardware at the granularity of individual RPCs. We defer a longer discussion of related work to §VIII.

B. Optimizing Queuing at µs Scale

The growing importance of handling µs-scale network events efficiently has led to several recent runtime and operating system proposals optimizing for throughput under stringent µs-scale tail latency targets [14], [22], [36], [38]. Most of these runtimes focus on efficiently handling heavy-tailed service time distributions and optimizing CPU utilization under multi-tenant scenarios. The flexibility of these software solutions is unparalleled by hardware-only mechanisms, but they leave significant room for improvement in the context of FLT RPCs. For example, ZygOS effectively employs high-performance, fine-grained work-stealing to react to spurious inter-core load imbalance, thus delivering performance within 15% of an ideal single-queue system for RPCs with sub-20µs service times. However, its efficiency drops sharply for shorter requests (50–60% of ideal with 5µs service times [38, §6.1]). Maximizing load-balancing efficiency for RPCs with sub-10µs service time requires *proactive hardware-based solutions.*

Modern NICs feature Receive Side Scaling (RSS) [32] to support increasing network line rates on multicore CPUs. RSS improves networking scalability, as it mitigates inter-core synchronization overheads by proactively distributing incoming *network flows* across multiple network queues, each assigned to one of the available cores. However, such distribution is not equivalent to load balancing, as it is performed by applying a static hash function on incoming packet headers and can result in considerable temporal inter-core load imbalance. RSS++ [6] and eRSS [40] extend RSS with auxiliary mechanisms built in software or in on-NIC programmable logic to ameliorate inter-core load imbalance over time by periodically reconfiguring RSS's flow redirection tables. Although better equipped than baseline RSS in minimizing overheads of µs-scale RPC handling, these solutions still have two main drawbacks.

First, periodic reconfiguration of RSS' control structures at 100µs–100ms timescales are too coarse, as they correspond



(a) Centralized FCFS (c-FCFS). (b) Distributed FCFS (d-FCFS). (c) Join Shortest Queue (JSQ).

Fig. 1: System configurations modeled in discrete event simulation.

to thousands of FLT RPCs with single-digit µs service times. We quantitatively demonstrate the drawback of such coarse-grained reconfiguration in §VII-E. Second, similar to baseline RSS, both approaches still rely on *network flow* as the unit of load to be balanced. Flows are coarser-grained than RPCs, which are ultimately the end-to-end unit of work for modern online services. There is growing consensus that RPC-oriented transports are a better fit than conventional bytestream transports in datacenter settings, because they enable improved flow control [33], [37], latency-aware routing [19], inter- and intra-server load balancing [12], [26], [44], and drastic datacenter workload acceleration opportunities [27], [31], [43], [47]. Therefore, it is timely and natural to transition to solutions that handle network traffic load balancing across cores *at the RPC rather than flow granularity*, enabling the finer-grained decisions needed to optimize for the tail latency of FLT RPCs.

C. RPC-Oriented NIC-Driven Load Balancing

Among prior work, RPCValet [12] is the solution that best caters to tail latency optimization for FLT RPCs. RPCValet balances RPCs to cores by implementing a—theoretically optimal for light-tailed service time distributions [39], [42], [46]—single-queue FCFS policy. To implement synchronization-free single-queue load balancing, RPCValet maintains a single queue of incoming RPCs at the NIC and only dispatches the RPC at the head of the queue as soon as a core becomes available, as sketched in Fig. 1a.

RPCValet's approach is only applicable to highly integrated SoC designs, where the interaction latency between the NIC and cores is negligible even compared to the most short-lived RPCs (i.e., 10s of ns). Servers in modern datacenters, however, predominantly feature *discrete* (rather than integrated) PCIe-attached NICs, where a single interface crossing incurs a latency overhead in the order of $500ns$ [34]. As a result, implementing RPCValet on a discrete NIC would result in prohibitive performance overheads. To illustrate, for a server running Memcached with an average service time of 2µs and a one-way interface-crossing latency of 500ns, RPCValet's greedy dispatch policy would incur a 1µs execution bubble per request, throttling peak sustainable throughput by 30%. This hefty overhead grows higher for more optimized software stacks that achieve even lower per-RPC service time, or for platforms with higher effective interface latency.

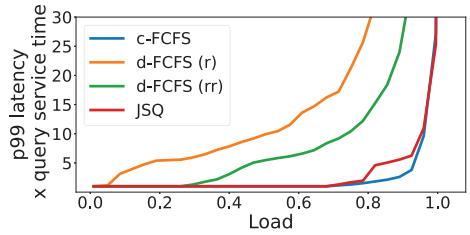
In summary, we investigate load-balancing mechanisms to optimize the tail latency of *FLT RPCs* implemented on modern *discrete programmable NICs*. Given the deployment of programmable NICs in datacenters [4], [9], [10], [16], our goal is to provide a practical proof-of-concept toolbox of policies for FLT RPC tail latency optimization in such platforms.

III. Load-Balancing Effect on µs-Scale RPCs

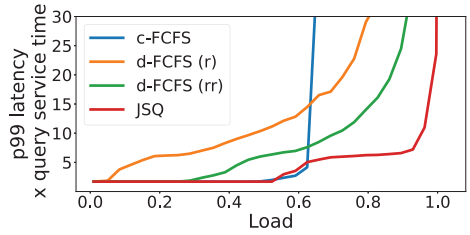
We perform a theoretical queuing analysis to demonstrate why the—theoretically optimal for light-tailed service time distributions—single-queue FCFS load-balancing policy is sub-optimal for discrete-NIC platforms with considerable NIC-CPU interface latency. We employ discrete event simulation to model a system with 16 serving units, Poisson request arrivals, and four load-balancing policies, shown in Fig. 1:

- *Centralized FCFS (c-FCFS):* The theoretically optimal single-queue load-balancing policy. Incoming requests wait in a single centralized queue and the request at the head of the queue is only dispatched to a serving unit as soon as the latter indicates its availability (Fig. 1a).
- *Distributed FCFS–random (d-FCFS(r)):* Incoming requests are assigned to serving units uniformly at random upon arrival by applying a hash function to each packet's header (Fig. 1b). d-FCFS(r)'s static nature resembles RSS, although RSS makes coarser-grained decisions than what we model here, as it distributes entire network flows rather than individual requests. d-FCFS(r) is more akin to Sprayer [41], which distributes load at a packet granularity.
- *d-FCFS–round-robin (d-FCFS(rr)):* Round-robin assignment of incoming requests to serving units upon their arrival (Fig. 1b). d-FCFS(rr) improves load balance for light-tailed distributions by replacing stateless hashing with minimal state in the request dispatching logic.
- *Join Shortest Queue (JSQ):* Each incoming request is assigned to the serving unit with the shallowest queue of pending requests (Fig. 1c). We consider JSQ because it can be practically implemented on a programmable NIC while maintaining minimal on-NIC state.

The interface latency between the on-NIC dispatch logic and the serving units affects the performance of feedback-based policies like c-FCFS. To illustrate, we model two different interface latencies as a fraction of average service time latency: 0% and 25%. 0% latency implies a negligible interface latency, as would be the case in an architecture with an integrated NIC like Scale-Out NUMA [35]. 25% means that the (one-way) latency to traverse the NIC-CPU interface is equal to $\frac{1}{4}$ of the average service time. For example, for a system handling 2µs RPCs and a PCIe-attached NIC incurring a NIC-to-CPU latency of $\sim 500ns$ [34], the interface latency is 25%. The same system's relative interface overhead would be lower for



(a) 0% NIC-CPU interface latency.



(b) 25% NIC-CPU interface latency.

Fig. 2: Tail latency on a system with 16 serving units as a function of load-balancing policy for a bimodal service time distribution (10% of queries are 5× longer). X-axis shows load normalized to the system's theoretical peak.

workloads with higher service times, and higher for workloads with sub-µs service times such as the MICA/HERD Key-Value Store [23], [28] or emerging "nanoservices" [20], [21].

For each of the four aforementioned system configurations, we model three service time distributions—fixed, exponential, and bimodal where 10% of requests have 5× longer service time than the short ones—and evaluate performance as peak sustainable load under a 99% SLO of 30× the query service time. Fig. 2 only shows results for the bimodal distribution for brevity, but our qualitative observations similarly apply to the other distributions.

Fig. 2a shows the behavior of a system with negligible NIC-CPU interface latency, as assumed by RPCValet [12]. c-FCFS is the best-performing policy, as theoretically expected. JSQ closely follows c-FCFS, as it ensures that the number of assigned requests per serving unit is always balanced. d-FCFS(r) is significantly outperformed by every other policy, and its gap from c-FCFS grows with the service time distribution's variability (not shown). Finally, although d-FCFS(rr) matches c-FCFS and JSQ for a fixed service time, it results in significant throughput loss for service time distributions with non-zero variability, like the shown bimodal distribution.

Fig. 2b shows the same results when the NIC-CPU interface latency equals 25% of the average service time, which drastically changes the landscape of relative load-balancing policy performance. c-FCFS is now the *worst* policy because of significant execution bubbles (c.f. §II-C). The relative trends between the remaining three policies remain similar to Fig. 2a. The relative performance gap between c-FCFS and other policies grows as a function of interface latency, because c-FCFS is directly affected by it, while the other policies are largely insensitive. The relative performance of JSQ and the two d-FCFS variants is minimally affected by the change in interface latency. JSQ outperforms c-FCFS, d-FCFS(r), and d-FCFS(rr) by 1.5×, 1.2×, and 1.1×, respectively. The performance gap further grows to 1.5×, 2.0×, and 1.4× at a more stringent SLO target of 10× the query service time.

In summary, while proactive NIC-driven load-balancing decisions are necessary to maximize the handling efficiency of FLT RPCs, the c-FCFS policy implemented by the RPCValet state-of-the-art solution is not viable for discrete NICs, due to considerable NIC-CPU interface latency. This work revisits NIC-driven load-balancing policies for platforms with discrete (as opposed to integrated) programmable NICs and proceeds to implement and evaluate their efficacy using capabilities of such commercially available platforms.

IV. Turbo Design

This section delves into the insights and assumptions guiding Turbo's design. Turbo aims to remove any software involvement and inter-core synchronization overheads from the data plane of load-balancing decisions. We first discuss Turbo's architecture, followed by proactive load-balancing policies that are a good fit for practical hardware implementation on current discrete NICs.

A. High-level Architecture

Turbo is designed to implement variants of the JSQ load-balancing policy, depicted in Fig. 1c. Turbo's architecture comprises four main components:

i. *State bookkeeping:* NIC-side tracking of work that has been assigned to each core and is still pending. The state maintained depends on the load-balancing policy (§IV-B).
ii. *RPC assignment:* per-RPC work assignment decisions. Turbo's logic on the NIC inspects each incoming RPC and consults the current per-core state to select the core to assign the RPC to.
iii. *Ingress state update:* update of bookkeeping state upon RPC assignment.
iv. *Egress state update:* periodic feedback from the server's cores to update the on-NIC bookkeeping state. This feedback notifies the NIC what units of previously assigned work have been completed to reclaim resource allocations. Similar to the R2P2 protocol [26], we rely on the inherent request-response nature of RPCs to avoid introducing a new out-of-band communication mechanism between the CPU and the NIC. Thus, every outgoing RPC response carries metadata that the NIC consumes to update its state.

A critical requirement in Turbo's design is performing the above functionality at line rate without adding considerable latency on each RPC's critical path.

B. Load-Balancing Policies

Turbo leverages the architecture described in §IV-A to implement different load-balancing policies. Given the characteristics of FLT RPCs, the policies we consider are (i) *proactive*, as opposed to reactive; (ii) *dynamic*, as opposed to static; (iii) *work-conserving* for maximized resource utilization, and (iv) *fully hardware offloaded*, to avoid any software overhead. We

implement two policies: Join Shortest Queue (JSQ) and its variant Join Lightest Queue (JLQ).

1) Join Shortest Queue (JSQ): As introduced in §III, JSQ is an approximation of c-FCFS that is more fitting for discrete NICs. To implement JSQ, Turbo's four main components are configured as follows:

i. *State tracking:* A single value $Counter_i$ per core, keeping track of how many RPCs that have been assigned to each of the N cores used by the application are still pending.
ii. *RPC assignment:* Assign the newly arrived RPC to the core i with the currently smallest $Counter_i$ value.
iii. *Ingress state update:* For the selected core i, increment $Counter_i$.
iv. *Egress state update:* Every RPC response from core i received on the egress path decrements $Counter_i$.

2) Join Lightest Queue (JLQ): JLQ is a variant of JSQ that refines JSQ's per-core pending work estimation granularity by taking into account the expected service time of each incoming RPC and using it as a weight for state updates. While *not generally applicable*, estimating the expected service time at the time of RPC reception is possible for certain applications. The expected service time can either be directly indicated by the client in a dedicated field of the RPC header, or inferred by the NIC. Any such inference must be lightweight and should not compromise line rate operation or increase the RPC's end-to-end latency. For instance, for a KVS supporting reads, writes, and range queries, it is simple to install expected average service time per request type in a small lookup table on the NIC, and implement shallow RPC inspection at arrival time that checks the request type to infer that RPC's expected service time. To implement JLQ, Turbo's four main components are configured as follows:

i. *State tracking:* A single value $Weight_i$ per core, keeping track of the expected aggregate service time of all currently pending RPCs on core i.
ii. *RPC assignment:* Assign the newly arrived RPC to the core i with the currently smallest $Weight_i$ value.
iii. *Ingress state update:* Infer the expected service time ST for the arrived RPC and increment the selected core i's $Weight_i$ by ST.
iv. *Egress state update:* Whenever an RPC response is received on the egress path from core i, decrement $Weight_i$ by the originally inferred service time ST.

The simplicity of the load-balancing policies considered affords a straightforward hardware design featuring minimal state and simple logic. Simplicity caters to our underlying requirements of preserving the NIC's line rate and avoiding any noticeable increase of RPC end-to-end latency.

V. Turbo SmartNIC Implementation

This section describes Turbo's implementation as an inline ("bump-in-the-wire") accelerator on a reconfigurable FPGA-based SmartNIC.

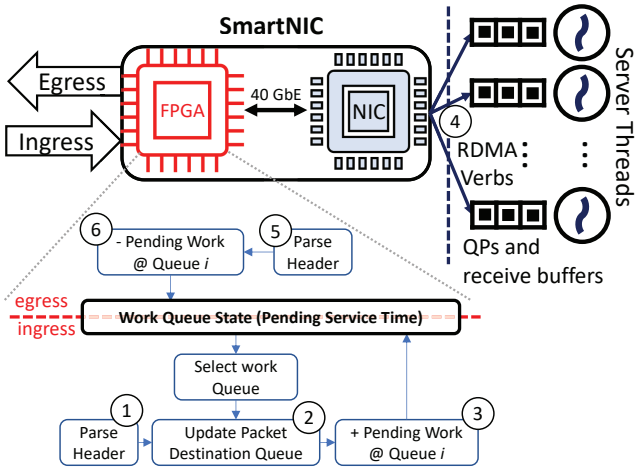


Fig. 3: Turbo implementation on a smartNIC.

A. Turbo RPC Load-Balancing Accelerator

We base our implementation on the Mellanox Innova Flex 4 Lx EN [1], which combines an RDMA-capable 40Gbps ConnectX-4 network adapter with a Xilinx Kintex Ultrascale FPGA. We build Turbo on top of the RoCEv2 RDMA UD transport. Each core registers its own Queue Pair (QP) and registers it with the NIC. Load-balancing decisions involve selecting one of the registered QPs to enqueue each arrived packet in. Our prototype implementation assumes a one-to-one packet to RPC request correspondence, hence no packet reassembly is involved; thus, making a load-balancing decision for an incoming packet is equivalent to making a decision for an incoming RPC. While this assumption facilitates our proof-of-concept implementation, the practical limitation it introduces is minimal, as we discuss in §V-D.

Fig. 3 shows a block diagram of our implementation. The FPGA is positioned between the network and the NIC, allowing it to inspect all network traffic and make load-balancing decisions at the NIC's 40Gbps line rate. Turbo features logic on both the ingress and egress path, sharing state that keeps track of the pending work per registered QP. The ingress path logic is responsible for inspecting every incoming packet and consulting the *Work Queue State (WQS)* maintained on the FPGA to determine which QP to assign the packet to (steps ① to ③ in Fig. 3). The FPGA then modifies the destination QP field of the packet's header, and the NIC DMA's the packet into the corresponding QP's receive buffer in the server's memory (step ④). Cores are continuously polling on their QPs for request arrivals and, after pulling a request and servicing it, they respond by posting a `send`. Finally, on the egress path, the FPGA inspects outgoing packets and uses extracted information to update the WQS (steps ⑤ and ⑥).

Fig. 4 demonstrates the microarchitecture of Turbo's ingress and egress path logic. The efficacy of any Turbo implementation lies in identifying the target QP, according to the implemented load-balancing algorithm, in a handful of clock cycles, while sustaining line rate operation. The first stage of both the ingress and egress datapath identifies the packets of

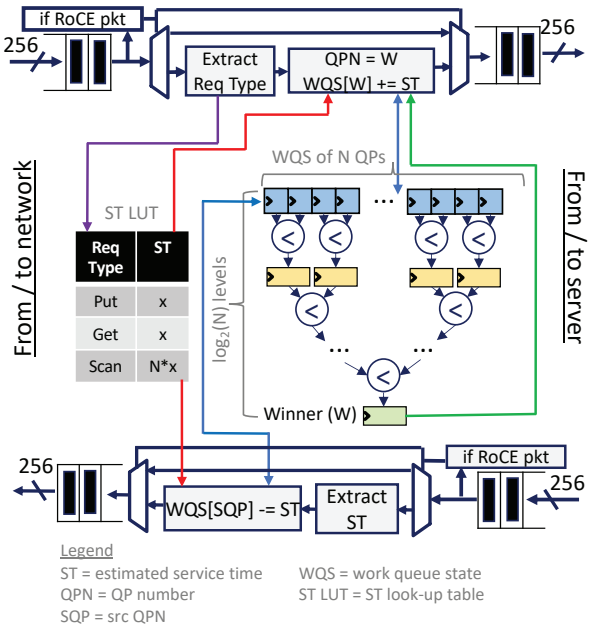


Fig. 4: Turbo ingress/egress logic.

interest using the destination port number in the UDP header encapsulated in the RoCE packets. Packets that do not match that destination port bypass the remainder of the ingress/egress datapaths. Fig. 5 shows the typical structure of a RoCEv2 packet used in our implementation.

Ingress path: The ingress path completes two functions for every incoming packet of interest: (i) it identifies the QP the packet must be assigned to, according to the load-balancing policy, and (ii) it updates the WQS tracking the pending work on the selected QP. The former function relies on a pipelined binary tree structure to continuously—independently of packet arrivals—identify the lightest-loaded QP (the "QP winner") and return its updated search result to the datapath every clock cycle, allowing Turbo to make load-balancing decisions on a per-packet basis. The latter function's performed action depends on the implemented load-balancing policy. For JSQ, Turbo simply increments the state of the QP winner, indicating there is one more unit of work pending in that QP. For JLQ, Turbo must first infer the expected service time for the request. We implement this as a lookup table (ST LUT), with Fig. 4 demonstrating an example for a KVS application using `Put/Get/Scan` requests. The ingress path's logic extracts the request type from the packet's RPC header and uses it to index ST LUT and retrieve the expected service time, which is then used to update the QP winner's corresponding WQS entry. Finally, the ingress path logic modifies the destination QP number (QPN) in the packet's InfiniBand header, to control the QP the NIC will DMA the packet to. We disable the NIC's header checksum and invariant CRC (iCRC) validation to avoid packet drops due to intentional QPN modification.

Egress path: The egress path shares similarities with the ingress path, but is simpler, as it only has to update QP state for each outgoing packet corresponding to an RPC response,

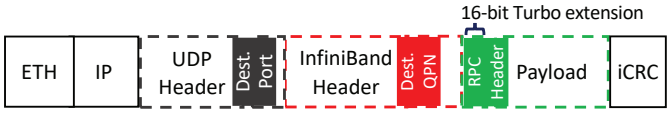


Fig. 5: RoCEv2 packet structure.

but doesn't have to make any load-balancing decisions. For each outgoing RPC response from QP_i, the egress path adjusts the QP's corresponding pending work state. For JSQ, this adjustment simply requires decrementing QP_i's WQS by one. For JLQ, Turbo must decrement QP_i's WQS by a value equal to the expected service time assumed by the ingress path at the time of the RPC's arrival. This could be done by inferring again what the RPC's service time was by looking up the service time table, as in the ingress path. We opt for a simpler alternative: when an incoming RPC's service time is inferred on the ingress path, that value is embedded as part of the RPC payload. The response packet also carries this value, which the egress path extracts and uses to decrement QP_i's state. Doing that requires a new field in the RPC header, shown as "16-bit Turbo extension" in Fig. 5 and further discussed in §V-B.

B. API and Software Extensions

We design Turbo for the low-latency RDMA transport to achieve kernel bypass and zero-copy data movement. Each thread servicing the target application registers its own QP and `send/recv` buffers using the *ibv_create_qp()* and *ibv_reg_mr()* APIs. Threads use `send/recv` RDMA Verbs [3] to exchange RPC requests and responses. Our implementation uses the RoCE network protocol, which encapsulates InfiniBand transport packets over Ethernet.

To enable the JLQ policy, we extend the RPC layer with a 16-bit *Request Type* (RT) field, which encodes each RPC's type (Fig. 5). Turbo extracts the RT from the RPC header and uses it to retrieve the expected Service Time (ST) from ST LUT, as described in §V-A. ST LUT's contents are pre-populated by the deployed application at initialization time. Alternatively, the RT field can be directly populated in each RPC by the application's client to indicate the expected service time for the outgoing request. In the latter mode of operation, ST LUT's contents implement the identity function. The absolute values RT is set to are unimportant; the effectiveness of JLQ depends on the accuracy of the *relative* service time values used for requests of different types with considerably different service times. The RPC library on the server side ensures the RT header value of each outgoing RPC response is carried over from the corresponding RPC request, so that Turbo's egress path updates the WQS correctly.

C. FPGA Platform Details

We use Vivado HLS to describe §V-A's datapaths in C++. Packets are streamed to the FPGA every clock cycle over a 256-bit AXI-Stream interface. The ingress and egress path logic operate on packets at 256-bit flit granularity—thus, the smallest RoCE packet consists of 3 flits. Once exported as an IP, our design is synthesized along with the NIC's wrapper

LUTs	FFs	Power (mW)	Clock (MHz)
2116 (0.6%)	2138 (0.3%)	26 mW	217

TABLE I: FPGA resource utilization for Turbo.

logic, placed and routed on the FPGA, and loaded onto the hardware in the form of a bitstream. All WQS memory elements are synthesized as flip-flops. Writes to the queue states from the ingress and egress path logic are scheduled on different clock cycles to always ensure a valid state of pending service time in each QP's corresponding WQS entry.

The smartNIC is attached to a 12-core server, so we base our evaluation on a system that exposes 12 QPs (16 in §VII-B's experiments employing 8 cores with 2-way SMT), resulting in a QP winner selection binary tree of four comparator levels (Fig. 4). More QPs can be trivially supported by increasing the tree's depth. Each of Turbo's ingress and egress datapaths add a single FPGA cycle to each packet's critical path. The ingress path's comparator tree is pipelined into four stages for our design's 4-level comparator tree, but is not on each packet's critical path, as the QP winner at any given time is continuously determined in the background, regardless of new packet arrivals.

The achieved clock period is 4.6ns, which results in a peak sustainable arrival rate of $\sim$ 56Gbps, exceeding the NIC's 40Gbps line rate. SmartNICs supporting higher line rates typically provision a wider interface to the FPGA (than our platform's 256 bits), allowing for a straightforward adaptation of Turbo's pipelines to meet those higher line rates. In addition, Turbo's ingress/egress path logic, which currently adds a single cycle per direction on the critical path, can be pipelined to achieve a higher clock rate if needed.

Table I summarizes Turbo's FPGA resource utilization, along with its resulting peak power consumption and clock frequency, as reported by Vivado post place and route. Turbo occupies a negligible fraction of the FPGA's resources, leaving ample room for more sophisticated policies or other accelerators to co-exist on the same smartNIC.

D. Turbo's Scope and Limitations

Turbo is a *specialized* NIC-driven load-balancing solution that specifically targets workloads relying on FLT RPCs. Thus, it consciously trades off flexibility and generality for high performance on such challenging applications, and is not intended as a replacement for more general software-based load-balancing mechanisms [14], [22], [36], [38]. Turbo can simply be disabled for applications that do not fit that profile.

Turbo currently has two limitations stemming from its implementation rather than its design. First, it's built on top of the RoCE UD transport to facilitate implementation. Although the same techniques could be carried over to an alternative implementation over connection-oriented and reliable transports, several applications already use lightweight unreliable datagrams (UDP or RoCE/IB UD). Second, a side-effect of using UD is the need to fit an RPC request (but not its response) in a single packet. Although Ethernet's typical MTU is 1.5KB, datacenter networks commonly support jumbo

Server	2× 12-core Intel Xeon Silver 4214 CPU Mellanox Innova Flex 4 Lx EN SmartNIC [1], 40GbE, Xilinx Kintex Ultrascale XCKU060 FPGA RHEL 7.2 - Linux 3.10.0-327.el7.x86_64 Xilinx Vivado (HLS) 2016.2 MLNX_OFED_LINUX-3.3-1.0.4.0.3
Client	2× 12-core Intel Xeon Silver 4214 CPU Mellanox ConnectX-6 Dx EN, 100GbE MLNX_OFED_LINUX-4.9-0.1.7.0

TABLE II: Experimental testbed specs and configuration.

frames (up to 9KB) [5], [31]. Given that the vast majority of messages in production datacenters are $< 1KB$ [33], we do not consider Turbo's current MTU limitation as a major concern. Furthermore, the MTU limitation does not preclude Turbo's use by applications requiring an occasional larger-than-MTU RPC. Prior work with similar limitations describes performance-neutral alternatives to handle infrequent large RPCs [24]. Finally, given Turbo's focus on FLT RPCs, we see little value in providing a "fast path" for larger-than-MTU RPCs, as serialization latency alone of large packets is comparable to the service time ranges we focus on (e.g., serializing a 16KB frame over 40Gbps exceeds 3µs).

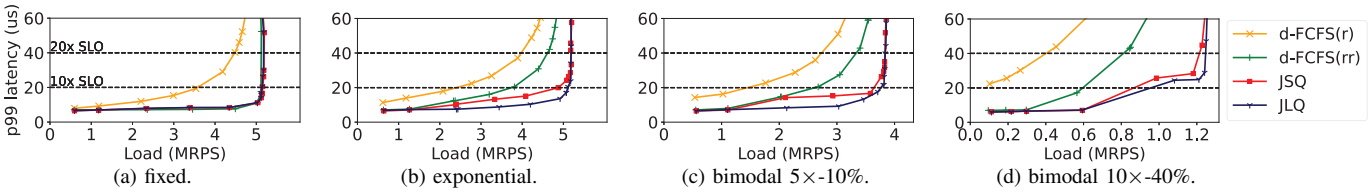
VI. Methodology

We developed two platforms to evaluate Turbo's load-balancing strategies: a discrete-event queuing simulator for fast first-order evaluation of different queuing systems and balancing policies, and a real-system experimental platform to evaluate our FPGA-based implementation.

a) Discrete-Event Simulator: We employ a C++ discrete-event queuing simulator for rapid design space exploration. The simulator was used to produce §III's motivational results, and in §VII to sanity-check results obtained from the real platform, extrapolate performance numbers to larger-scale systems, and estimate the behavior of prior relevant designs. The simulator models Poisson arrivals and offers a range of configurable parameters: arrival rate, number of serving units, service time distribution, load-balancing policy, and interface latency between the load-balancing logic and the serving units.

b) Experimental Platform: Our experiments were conducted on a pair of directly connected machines, with hardware specs detailed in Table II. One machine features a conventional 100G RDMA-capable NIC and is configured to be the client generating requests to load the server. The second machine is the server and features an FPGA-based programmable NIC where we deploy Turbo. On the server machine, we only use the 12 cores of the socket directly attached to the NIC, to isolate the impact of NUMA effects.

The client machine's role is two-fold: to put configurable load on the server and to accurately measure end-to-end request latency. To meet these two roles, we employ multiple loading threads and a single measuring thread—a separation necessary to avoid measuring bias due to client-side queuing effects [48]. The number of loading threads and outstanding requests per thread are the main knobs for controlling the generated load in each experiment. The measuring thread always has a single outstanding request. All latencies reported in the



(a) fixed. (b) exponential. (c) bimodal 5×-10%. (d) bimodal 10×-40%.

Fig. 6: Microbenchmark load-latency graphs for different service time distributions and load-balancing policies implemented on Turbo.

evaluation refer to the measuring client's end-to-end latency measurements and are averaged over three trials of one million RPCs issued by the measuring client. Our main evaluation metric is throughput under SLO in terms of achieved Million Requests Per Second (MRPS). Throughout our evaluation's load-latency graphs, we assume a default upper-bound p99 SLO of 30× the average service time of the measuring client's requests, and therefore, bound the y-axis to that value, but also mark the 10× and 20× thresholds with horizontal lines to investigate system behavior at tighter SLO bounds.

c) Workloads: We employ a microbenchmark to explicitly control the service time distribution and validate Turbo's high-level behavior with the queuing model. The clients embed a processing time value in the RT header field of each request they generate. The cores on the server side emulate the processing time indicated in the request, by invoking a `usleep` for the specified value. For our microbenchmark-based evaluations, the RT value in each request is also used by Turbo's JLQ policy to infer each request's expected service time (c.f. §V-B, second mode of ST LUT operation).

We also use the Masstree Key-Value Store (KVS) [30] to evaluate Turbo's load-balancing policies on a software component that is a cornerstone of virtually every online service. We replace stock Masstree's [2] UDP transport with `send/recv` RoCE UD verbs. We select Masstree because it is a high-performance KVS with µs-scale service times that also supports SCAN operations in addition to basic GET/PUT operations that many KVS are limited to. The mix of supported operations enables workloads with more service time distribution variety. For Masstree, Turbo implements JLQ by pre-populating its ST LUT with the average service time we measured for each query type via application profiling.

VII. Evaluation

In this section, we first employ our real-system Turbo implementation to answer the following three questions:

- Does Turbo deliver performance improvements for FLT RPCs that are in line with the expectations set by §III's theoretical queuing model (§VII-A)?
- How does Turbo compare to specialized software solutions for µs-scale RPCs (§VII-B)?
- What are Turbo's performance gains on a production-grade application with µs-scale service times (§VII-C)?

Next, we validate our discrete-event simulator's queuing model with the real-system results and use the validated model to answer three additional questions:

- How do Turbo's performance gains scale with higher CPU core counts (§VII-D)?
- How important is it to balance load at the individual RPC granularity (§VII-E)?
- What range of service time distributions is Turbo's proactive load balancing good for (§VII-F)?

A. Gains on Throughput under SLO

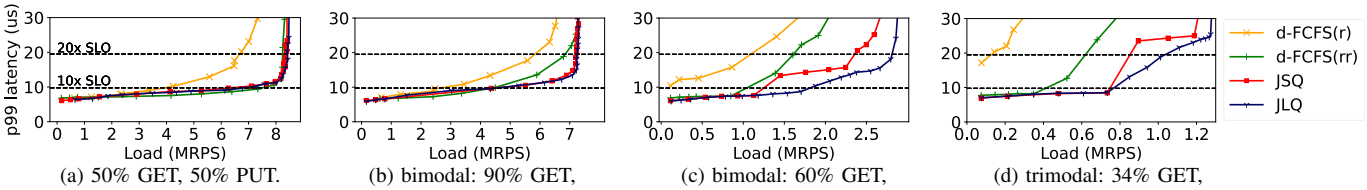
We first use the microbenchmark, instrumenting the loading clients to send requests following (1) a fixed processing time of 2µs; (2) an exponential processing time distribution with an average processing time of 2µs; and two bimodal processing time distributions where short requests have a processing time of 2µs and (3a) 10% of the requests are 5× longer, (3b) 40% of the requests are 10× longer. The measuring client always sends requests with a fixed 2µs processing time.

Fig. 6 shows the results. For fixed processing time (Fig. 6a), JSQ and JLQ are practically identical policies and achieve a peak throughput of 5.1MRPS. Turbo's JSQ/JLQ barely outperform d-FCFS(rr), as the lack of variability in service time results in no load imbalance. d-FCFS(r) achieves a 10% lower throughput than JSQ/JLQ because of the load imbalance caused by static load-oblivious inter-core RPC distribution.

Performance trends are more interesting for processing time distributions with non-zero variability (Figs. 6b to 6d). Turbo's load-balancing policies boost throughput under SLO by $1.17-1.85\times$ and $1.08-1.34\times$ compared to d-FCFS(r) and d-FCFS(rr), respectively. The merits of JLQ's ability to infer the service time of every RPC and balance load accordingly are more pronounced with larger service time variability. Although JSQ and JLQ achieve the same peak throughput, JLQ yields up to 40% lower tail latency at medium loads. In addition, throughput gaps between different load-balancing policies become more pronounced for more stringent SLOs (with the exception of the zero-variability fixed processing time distribution). For instance, for a 10× SLO, JSQ outperforms d-FCFS(r) and d-FCFS(rr) by $1.45-95\times$ and $1.0-1.7\times$, respectively, with JLQ achieving an additional throughput boost of up to 10% over JSQ.

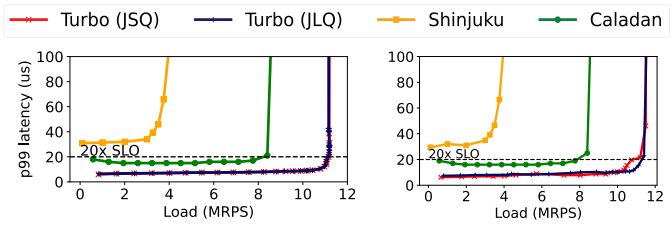
B. Comparison to Specialized Software Solutions

Turbo is a solution specialized for FLT RPCs with $< 10\mu s$ service times. Therefore, as previously mentioned in § II-B and V-D, it is not directly comparable with advanced software-based solutions for tail-latency optimization, which are much more flexible, offering robust performance for a wide range of



(a) 50% GET, 50% PUT. (b) bimodal: 90% GET, 10% $SCAN_{18}$. (c) bimodal: 60% GET, 40% $SCAN_{46}$. (d) trimodal: 34% GET, 33% $SCAN_{46}$, 33% $SCAN_{105}$.

Fig. 7: Effect of load-balancing policy on different query mixes on Masstree. The average service time for each of the used query types is: GET = 975ns, PUT = 1.35µs, $SCAN_{18}$ = 4.7µs, $SCAN_{46}$ = 9.18µs, and $SCAN_{105}$ = 17.7µs. The X subscript in $SCAN_X$ denotes the number of KV pairs retrieved by the query.



(a) Fixed service time. (b) Exponential service time.

Fig. 8: Comparison to Shinjuku and Caladan.

service time distributions, but are sub-optimal for workloads dominated by FLT RPCs. However, we attempt to provide a quantitative comparison point to highlight the fact that Turbo and software-based solutions target operational regions and timescales of qualitatively different ballparks.

We compare against Shinjuku [22] and Caladan [17] at an operational region where Turbo shines. To align our evaluation with experiments performed in the respective papers, we (i) use 16 hardware contexts on 8 cores (i.e., 2-way SMT)[1]; and (ii) deploy a microbenchmark that models synthetic RPC processing following a fixed and an exponential service time distribution, with a 1µs average service time. We reproduce results for Caladan by deploying the experiment on our experimental platform. We do not deploy Shinjuku, but directly use data presented in the respective paper that correspond to the same experimental setup [22, Figs. 5a & 5b].

Fig. 8 compares the results. Shinjuku cannot meet a 20× p99 SLO target even at the lowest load. At the same SLO target, Caladan delivers a peak throughput of 8.8MRPS, 23% lower than Turbo's achieved throughput of 11.5MRPS. At a relaxed SLO of 100× the average service time (i.e., 100µs), Turbo outperforms Shinjuku and Caladan by 2.8× and 1.3×, respectively. While Turbo delivers large performance gains over Shinjuku and Caladan, we note that these systems are designed to flexibly handle service time distributions with higher variance and, therefore, have a broader range of applicability. However, this comparison serves as a clear demonstration of the need for hardware support for FLT RPCs.

C. Masstree Key-Value Store

We continue our evaluation with the Masstree KVS. Fig. 7 shows results for four different query mixes generated by the loading clients. Fig. 7a corresponds to YCSB-A [11]; results for the YCSB-B and YCSB-C workloads are largely similar to YCSB-A, due to the comparable service times of GET and PUT queries, and are therefore omitted. Figs. 7b and 7c's bimodal distributions roughly match those evaluated in §VII-A using the synthetic service time microbenchmark, and Fig. 7d evaluates a trimodal service time distribution. The measuring client measures latency using GET requests.

Performance trends largely follow those observed in §VII-A's microbenchmark evaluation. d-FCFS(r) (corresponding to a static RSS-based mechanism like Sprayer [41]) is always the most inferior policy. YCSB-A (Fig. 7a) results in roughly fixed service times, hence JSQ and JLQ perform identically, while d-FCFS(rr) slightly lags behind due to naturally occurring small service time variability that JSQ/JLQ dynamically adapt to. JLQ noticeably outperforms JSQ only when the workload's service time variability grows considerably.

For the trimodal distribution (Fig. 7d), JSQ outperforms d-FCFS(r) and d-FCFS(rr) by 5× and 1.5× respectively. JLQ achieves an additional 6% boost. Turbo's delivered throughput under SLO nearly matches the peak theoretical throughput for the used service time distributions. For instance, for the trimodal distribution, JLQ's peak achieved throughput under SLO is within 4% of the theoretical maximum throughput (computed as number of cores divided by the workload's average service time of 9.2µs: $\frac{12}{9.2\mu s} = 1.3MRPS$). This minuscule gap is expected to grow for service times with higher variance, due to the limitations of proactive load balancing, which works very well for FLT RPCs, but its efficacy deteriorates for heavy-tailed service time distributions. We further explore the limits of proactive load balancing in §VII-F.

D. Scalability Analysis

This section studies Turbo's impact as a function of the number of cores. To overcome our experimental platform's core count limitations, we collect results up to 12 cores and extrapolate performance up to 64 cores using our discrete-event simulator's queuing model (§VI). We first confirm that the model accurately predicts the real platform's performance.

Queuing model validation. To get an accurate NIC-CPU interface latency estimate, we perform a zero-load experiment where the server immediately echoes every received message. The FPGA timestamps each packet on its ingress and on

[1]We note that this setup is an outlier, used solely in §VII-B's experiments. The rest of our evaluation uses 12 server cores with SMT disabled.

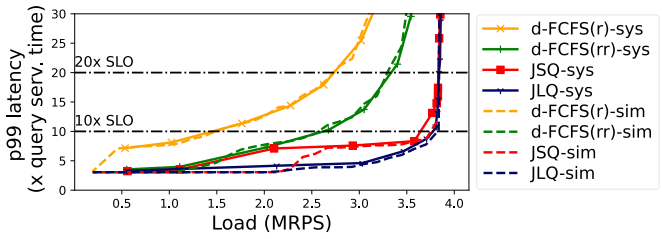


Fig. 9: Queuing model validation using Fig. 6c's 5×-10% bimodal distribution.

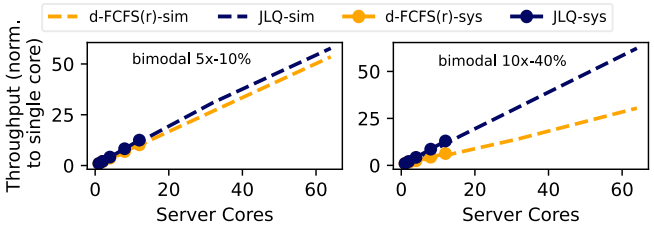


Fig. 10: Scalability analysis of attainable peak throughput under 30× 99% tail latency SLO.

its egress to derive $time_{on-server}$ and the CPU measures the time between each packet's reception and the subsequent `send` operation ($time_{on-cpu}$). We estimate the one-way NIC-CPU interface latency to be about 550ns, computed as $\frac{time_{on-server}-time_{on-cpu}}{2}$, and use this value to instrument the queuing model.

Fig. 9 overlays the *measured* load-latency curve for the bimodal 5×-10% service time distribution from Fig. 6c with the model's *predicted* results, omitting other service time distributions for brevity. We observe that the model predicts the real system implementation's performance with high accuracy. The maximum deviation in the model's predicted p99 latency is high (2.3×) for JSQ under medium load, but in every other case, the deviation is negligible. Most importantly, there is a near-perfect match between the predicted and actual peak throughput under our target 10× and 30× SLO. Hence, we next use the queuing model to accurately predict the peak achievable throughput under SLO with core counts exceeding our experimental platform's capabilities.

Scalability results. Fig. 10 shows achieved throughput under SLO with d-FCFS(r) and JLQ for Fig. 6's two bimodal distributions—5×-10% and 10×-40%—when scaling the core count from 2 to 64. Results in the 2–12 range are obtained from both our experimental platform and the validated queuing model, and results in the 12–64 range are extrapolated using the model. The complete overlap of the system and model lines in the 2–12 range further supports the model's accuracy. Turbo's improved load balancing allows near-linear throughput scaling under SLO, resulting in growing performance gap between JLQ and d-FCFS(r) with the number of cores, as inter-core load imbalance hampers the latter's performance.

E. Impact of per-RPC Load-Balancing Decisions

We now use our queuing model to demonstrate why Turbo's per-RPC load-balancing decisions are in principle superior to

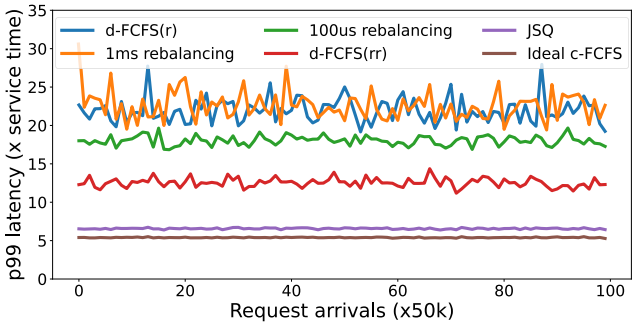
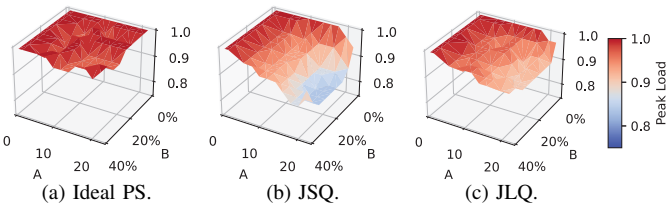


Fig. 11: Per-RPC load balancing vs. periodic rebalancing for exponential service time and 12 serving units at 80% load. Ideal c-FCFS assumes no interface latency between load-balancing decision-making and serving units.

adaptive RSS-based solutions like RSS++ [6] or eRSS [40] that periodically reassign connection hash buckets to cores. We model an optimistic instance of an adaptive RSS-based solution as follows: each incoming packet is hashed to one of 512 buckets, and then buckets are assigned to 12 cores ($\sim$ 43 buckets per core on average). We evaluate a best-case regrouping of buckets that minimizes inter-core load imbalance every 1ms and every 100µs, assuming *zero cost* for such rebalancing. The queuing model is instrumented as detailed in §VII-D and results are shown in Fig. 11.

Evidently, higher reassignment frequency (every 100µs instead of every 1ms) places a tighter bound on tail latency. However, recomputing bucket groupings and reconfiguring the hardware on the NIC faces practical limitations, which led RSS++ [6] to a minimum reconfiguration window of 1ms. Rebalancing every 1ms fares similarly to the d-FCFS(r) policy that distributes load at a per-RPC (rather than per-flow) granularity using a static hash function without any periodic bucket rebalancing across cores. A 10× lower rebalancing interval (every 100µs) helps reduce average tail latency and jitter, but still results in almost 3× higher tail latency than JSQ. Even a simple round-robin policy at the individual RPC granularity (i.e., d-FCFS(rr)) fares better than the two adaptive RSS-like solutions, due to the short-tailed service time of the requests. Finally, Turbo's JSQ yields lower tail latency and jitter than all alternatives, approaching an ideal c-FCFS policy with zero latency between the load-balancer and the serving units (i.e., similar to RPCValet [12] that assumes an on-chip integrated NIC).

We reiterate that RSS-based solutions with periodic rebalancing strive to preserve network flow affinity, which Turbo's balancing at RPC granularity foregoes. However, as argued in §II-B, there is mounting evidence that RPC-oriented transports are a better fit than bytestream-oriented ones for online services in datacenters. Thus, for a significant class of applications, foregoing flow affinity for improved per-RPC tail latency is a favorable tradeoff.



(a) Ideal PS. (b) JSQ. (c) JLQ.

Fig. 12: Limits of proactive load balancing as function of service time variability. Peak sustainable load under 99% latency SLO of 30× average service time for bimodal distributions where $B\%$ of requests have $A\times$ longer service time.

F. Limits of Proactive Load Balancing

We conclude our evaluation by investigating the applicability range and limits of Turbo's proactive load-balancing approach beyond its main focus of fine-grained *light-tailed* RPCs, for which load-balancing policies in the FIFO family are optimal. We use our queuing model to evaluate the range of service time distributions where Turbo remains effective without requiring reactive software mechanisms for dynamic load rebalancing. We use bimodal distributions to cover a range of service time variability and compare JSQ/JLQ against *ideal* Processor Sharing (PS) in terms of peak throughput achieved under SLO. Ideal PS cycles through all pending requests at a 10ns granularity, with zero overhead.

Fig. 12 shows the results for a range of bimodal distributions where $B\%$ of requests have $A\times$ longer service time, with $A \in [0, 20]$ and $B \in [0\%, 40\%]$. PS remains insensitive to service time variability. In contrast, JLQ and JSQ become less effective as variability grows. However, their efficacy remains quite high for a significant range: JSQ's throughput under SLO virtually matches ideal PS when $A \leq 8$, and remains within 20% of ideal PS when Fig. 12's entire service time distribution range is considered. Compared to JSQ, JLQ improves throughput under SLO by up to 12%, especially for the more heavily skewed distributions, and performs within 11% of ideal PS across the board.

In conclusion, while Turbo's proactive load-balancing policies are most effective for FLT RPCs, they work quite well for service time distributions with considerable variability. Maximizing performance for a wide range of service time distributions requires software support for fast preemption or work-stealing investigated in prior work [22], [38]. Combining such techniques with Turbo can achieve the best of both worlds: peak performance for FLT RPCs, and high flexibility for applications with highly variable service time profiles.

VIII. Related Work

The growing significance of µs-scale online services and tail latency as a key figure of merit has given rise to a plethora of proposals, both software- and hardware-based, optimizing for advanced load-balancing techniques of µs-scale tasks to mitigate queuing effects on end-to-end latency. As discussed in §II-B, Turbo is a hardware-based solution for modern discrete NICs, drawing a high-level distinction between (i) software-based solutions, which are more flexible for wide service time distributions but sub-optimal for FLT RPCs, and (ii) hardware-based solutions that assume exotic NIC architectures [12] or make coarse-grained rebalancing decisions at the network flow granularity [6], [40]. We further elaborate on each category of related work next.

Specialized OSes. IX [8] leverages RSS to achieve scalable, high-performance userspace network stack processing, but does not consider load-balancing effects on queuing delays. ZygOS [38] and Shenango [36] leverage low-overhead work stealing to reactively counteract load imbalance arising due to the static load distribution nature of RSS. However, their efficiency compared to the—theoretically optimal for FLT RPCs—c-FCFS policy drops significantly below 10µs service times, where the relative overhead of work stealing becomes considerable. Shenango's follow-on work, Caladan [17], monitors contention on shared resources (i.e. cores, memory bandwidth, and caches) and dynamically re-allocates cores to tasks at a 20µs granularity, which remains too coarse-grained in the context of FLT RPCs. Shinjuku [22] employs lightweight preemption at a 5µs granularity to achieve near-optimal tail behavior for heavily tailed workloads, but is not designed for FLT RPCs with single-digit µs average service time. Perséphone [14] adopts a non-work-conserving approach by dedicating cores to requests with low service time, to avoid high relative queuing delays in the presence of heavily tailed service time distributions. In contrast to all OS-based approaches, Turbo targets the narrower scope of FLT RPCs, without introducing any software overheads and without requiring any OS modifications or an elaborate runtime system. Turbo could be combined with such specialized software systems for efficient handling of both FLT RPCs and heavy-tailed service time distributions.

RSS Variants. To mitigate load imbalance stemming from RSS's static nature of hash-based load distribution, recent hardware-focused proposals introduce mechanisms for periodic reconfiguration of the RSS indirection table that controls hash bucket to core mapping. RSS++ dynamically changes the RSS indirection table to redistribute load across cores at a 100ms–1s granularity [6]. We showed in §VII-E that such coarse-grained rebalancing leaves a large performance improvement opportunity on the table. eRSS proposes updating its on-NIC load distribution logic at a finer granularity of as low as 10µs [40]. eRSS aims to drive core allocation decisions in addition to inter-core load distribution, but, unlike Turbo, the proposed design is not implemented and demonstrated on a real system. Finally, although RSS variants preserve flow affinity, in contrast to Turbo's balancing at the per-RPC granularity, extensive prior work posits that new RPC-oriented transports are more fitting to datacenters than bytestreams [12], [19], [26], [33], [37], [44] (as previously discussed in §II-B).

Specialized Architectures. As elaborated in §II-C, the interface latency between on-NIC load-balancing logic and the on-server compute units affects the efficacy of the implemented

load-balancing policy. Specialized highly integrated architectures like RPCValet [12] and nanoPU [20] proposing NIC-CPU integration feature ns-scale interface latencies, making such designs apt for the single queue c-FCFS load distribution policy that is ideal for FLT RPCs. There have also been some commercial processors with an integrated NIC, like Tilera's TILE64 and Oracle's Sonoma [45], but such designs have not, so far, been mainstreamed in large enterprise system deployments. Turbo focuses on conventional architectures where the NIC is attached to the server over an IO interface—typically PCIe—which introduces considerable latency in NIC-CPU interactions.

SmartNICs, Switches, Middleboxes. A range of prior work employs switches, middleboxes, and programmable NICs for load balancing. Sprayer [41] identifies that the effectiveness of RSS in middleboxes directly depends on the number of flows, with too few flows yielding low CPU core utilization and too many flows causing imbalanced flow distribution across cores. Sprayer proposes finer-grained load balancing at the packet rather than flow granularity, but does not delve into adaptive load balancing and an in-depth study of its impact on tail latency. R2P2 [26] implements a JBSQ (Join *Bounded* Shortest Queue) policy on a switch to improve inter-server load-balancing—and by extension tail latency—at rack scale. Dagger [27] leverages a UPI memory interconnect between its smartNIC-offloaded RPC stack and the host CPU, but only considers a static distribution of requests across RX rings. FlexNIC [25] implements key-based steering to Key-Value access requests, which aims at improving throughput via improved data locality but may exacerbate inter-core load imbalance and amplify response tail latency. PANIC [29] hints at the feasibility of offloading packet scheduling algorithms on programmable NICs but does not investigate dynamic load-balancing policies on the NIC and their effect on queuing delay and tail latency on FLT RPCs. Frameworks like NICA [15] and alternative smartNIC architectures like FlexNIC, PANIC, or PsPIN [18] are orthogonal to Turbo's proposed approach to RPC load balancing and can be leveraged to produce instances of Turbo on different platforms.

IX. Conclusion

We presented Turbo, an on-NIC hardware-based load-balancing mechanism specialized for the light-tailed µs-scale RPCs prevalent in modern online services. Turbo is readily deployable on current programmable smartNICs, as demonstrated by our prototype on the FPGA-based Mellanox Innova Flex NIC. Turbo's load-balancing decisions are proactive and are made at the individual RPC granularity, without any software involvement, which introduces considerable overhead in the context of µs-scale RPCs. Turbo's fine-grained decisions lower tail latency and improve latency predictability, as compared to RSS-based alternatives with periodic reconfiguration. Using a set of microbenchmarks and the Masstree high-performance KVS, Turbo boosts peak throughput by up to $5\times/95\times$ under a stringent 99th percentile tail latency SLO of $30\times/10\times$ the request's service time, respectively, compared to static load distribution using existing RSS NIC capabilities.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and feedback. We are grateful to Haggai Eran, Rafael Oliveira, and Callie Hao for their invaluable help on technical problems encountered while working on the Innova Flex platform, to Xueyang Liu who helped develop the initial RDMA microbenchmark used in the evaluation, and to Joshua Fried who assisted with deploying Caladan on our servers. We also thank Marina Vemmou, Albert Cho, Anirudh Sarma, and Divya Kiran Kadiyala for their constructive feedback at early stages of the work and on drafts of the paper. This work was supported by the National Science Foundation (award NSF-CCF-2006602) and by a Google Faculty Research award.

Appendix

A. Artifact Abstract

This artifact contains the discrete-event queuing simulator used in Turbo's methodology to model a discrete NIC enqueuing *Poisson*-arrived requests with various service time distributions into queues of a multi-core system and simulates the static and dynamic load-balancing policies we assessed in § III and VII. This artifact provides the infrastructure to reproduce Figs. 2 and 10 to 12. The objective of this artifact is to allow users to reproduce the results presented in this paper and encourage the research community to use our simulator for other relevant research.

B. Artifact Checklist (*meta-information*)

- **Compilation:** Although other versions may work equally well, we recommend using gcc version 7.5.0 for compilation.
- **Runtime Environment:** Root access is not required to reproduce the results of this paper.
- **Metrics:** The unit of data reported is 99th percentile latency as a multiple of service time.
- **Outputs:** Once the experiments have completed, the measured latency for each load point is stored in CSV files, which are then converted to figures as PDF files after plotting the data.
- **Experiments:** Scripts are included to reproduce results generated with our evaluation methodology's queuing model, namely Figs. 2 and 10 to 12 of the paper.
- **Required Disk Space:** 1.5GB.
- **Time needed to complete experiments:** Approximately 2–3 hours, assuming a server processor with 24 cores. Runtime will largely depend on the number of cores used, as the executed simulations are highly parallelizable.
- **Publicly Available:** Yes.
- **Code licenses:** MIT.
- **Workflow framework used**: Bash script.
- **Archived**: https://doi.org/10.5281/zenodo.7410082.

C. Description

1) How to access: The artifact can be downloaded from Zenodo: https://doi.org/10.5281/zenodo.7410082.

2) Hardware dependencies: There are no specific requirements, however, we recommend running the experiments on a machine with $\geq$ 24 cores for completion within the aforementioned timeframe.

3) Software dependencies: The artifact requires a setup with gcc version 7.5.0 and Python 3.9.7—installed via Anaconda—with plotting library *matplotlib* and data analysis library *pandas* installed as well.

D. Installation

- **gcc:** Please ensure you have gcc version 7.5.0 installed on your system.
- **Python:** If you don't have Python 3.9.7 installed please visit https://docs.anaconda.com/anaconda/install/index.html for installation steps. Installing Python via Anaconda will also install *numpy*, *pandas*, and *matplotlib*, all of which are required for plotting.

E. Experiment Workflow

After installing the required software and downloading the artifact, the user should invoke the `run.sh` script. The script launches all experiments, generates data for the plots, and plots the figures.

F. Results

After the aforementioned script's execution completes, the `plots` folder will contain subfolders for each of the following plots in the paper:

- `interface_latency` reproduces Fig. 2.
- `scalability` reproduces the data corresponding to Fig. 10's discrete-event simulation results.
- `timeline` reproduces Fig. 11.
- `surface` reproduces Fig. 12.

G. Notes

- We have tested this artifact against gcc 7.5.0 and Python 3.9.7, and although other versions may work equally well, we recommend that these versions are used.
- The generated results are not deterministic across many runs, however, they can be directly compared against the prevalent trends present in our paper's figures.

H. Methodology

This artifact was prepared following the methodology recommended by the cTuning foundation: http://cTuning.org/ae/submission-20201122.html.

References

[1] Innova Flex 4 Lx EN Adapter Card. https://pdf4pro.com/amp/cdn/innova-flex-4-lx-en-adapter-card-compsource-com-4bb05c.pdf.

[2] Masstree. https://github.com/kohler/masstree-beta/commit/cef4cc4f68953bdc4d0aec736cb8e1ce0700a4ae.

[3] RDMA Aware Networks Programming User Manual. https://network.nvidia.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[4] Amazon Web Services. Amazon EC2 F1 Instances: Enable faster FPGA accelerator development and deployment in the cloud. https://aws.amazon.com/ec2/instance-types/f1/.

[5] Amazon Web Services. AWS FPGA developer kit now supports Jumbo frames in virtual ethernet frameworks for Amazon EC2 F1 instances. https://aws.amazon.com/about-aws/whats-new/2021/10/aws-fpga-jumbo-amazon-ec2-instances/.

[6] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostic. RSS++: load and state-aware receive side scaling. In *Proceedings of the 2019 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 318–333, 2019.

[7] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI)*, pages 49–65, 2014.

[9] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 7:1–7:13, 2016.

[10] Jian Chen, Xiaoyu Zhang, Tao Wang, Ying Zhang, Tao Chen, Jiajun Chen, Mingxu Xie, and Qiang Liu. Fidas: fortifying the cloud via comprehensive FPGA-based offloading for intrusion detection: industrial product. In *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*, pages 1029–1041, 2022.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, 2010.

[12] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, pages 35–48, 2019.

[13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[14] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 621–637, 2021.

[15] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 345–362, 2019.

[16] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–66, 2018.

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

[18] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beránek, Luca Benini, and Torsten Hoefler. A RISC-V in-network accelerator for flexible high-performance low-power packet processing. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, pages 958–971, 2021.

[19] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 29–42, 2017.

[20] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Proceedings of the 15th Symposium on Operating System Design and Implementation (OSDI)*, pages 239–256, 2021.

[21] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. The Case for a Network Fast Path to the CPU. In *Proceedings of The 18th ACM Workshop on Hot Topics in Networks (HotNets-XVIII)*, pages 52–59, 2019.

[22] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for µs-scale Tail Latency. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 345–360, 2019.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 295–306, 2014.

[24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 185–201, 2016.

[25] Antoine Kaufmann, Simon Peter, Thomas E. Anderson, and Arvind Krishnamurthy. FlexNIC: Rethinking Network DMA. In *Proceedings of The 15th Workshop on Hot Topics in Operating Systems (HotOS-XV)*, 2015.

[26] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[27] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, pages 36–51, 2021.

[28] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[29] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 243–259, 2020.

[30] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 2012 EuroSys Conference*, pages 183–196, 2012.

[31] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve D. Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena E. Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, 2019.

[32] Microsoft Corp. Receive Side Scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx.

[33] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 221–235, 2018.

[34] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 327–341, 2018.

[35] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 3–18, 2014.

[36] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, 2019.

[37] John K. Ousterhout. A Linux Kernel Implementation of the Homa Transport Protocol. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 99–115, 2021.

[38] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[39] Kavita Ramanan and Alexander L. Stolyar. Largest weighted delay first scheduling: large deviations and optimality. *Annals of Applied Probability*, 11:1–48, 2001.

[40] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking (APNet)*, pages 71–77, 2019.

[41] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. A Case for Spraying Packets in Software Middleboxes. In *Proceedings of The 17th ACM Workshop on Hot Topics in Networks (HotNets-XVII)*, pages 127–133, 2018.

[42] Alexander L. Stolyar. Control of end-to-end delay tails in a multiclass network: LWDF discipline optimality. *Annals of Applied Probability*, 13:1151–1206, 2003.

[43] Mark Sutherland, Babak Falsafi, and Alexandros Daglis. Cooperative Concurrency Control for Write-Intensive Key-Value Workloads. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*, 2023.

[44] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra J. Marathe, Dionisios N. Pnevmatikatos, and Alexandros Daglis. The NeBuLa RPC-Optimized Architecture. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, pages 199–212, 2020.

[45] Basant Vinaik and Rahoul Puri. Oracle's Sonoma processor: Advanced low-cost SPARC processor for enterprise workloads. In *Hot Chips Symposium*, pages 1–23, 2015.

[46] Adam Wierman and Bert Zwart. Is Tail-Optimal Scheduling Possible? *Oper. Res.*, 60(5):1249–1257, 2012.

[47] Arash Pourhabibi Zarandi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. Cerebros: Evading the RPC Tax in Datacenters. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 407–420, 2021.

[48] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 456–468, 2016.