The NEBULA RPC-Optimized Architecture

Mark Sutherland
EcoCloud, EPFL
mark.sutherland@epfl.ch

Siddharth Gupta
EcoCloud, EPFL
siddharth.gupta@epfl.ch

Babak Falsafi
EcoCloud, EPFL
babak.falsafi@epfl.ch

Virendra Marathe
Oracle Labs
virendra.marathe@oracle.com

Dionisios Pnevmatikatos
National Technical University of Athens
pnevmati@cslab.ece.ntua.gr

Alexandros Daglis
Georgia Institute of Technology
alexandros.daglis@cc.gatech.edu

***Abstract*—Large-scale online services are commonly structured as a network of software tiers, which communicate over the datacenter network using RPCs. Ongoing trends towards software decomposition have led to the prevalence of tiers receiving and generating RPCs with runtimes of only a few microseconds. With such small software runtimes, even the smallest latency overheads in RPC handling have a significant relative performance impact. In particular, we find that growing network bandwidth introduces queuing effects within a server's memory hierarchy, considerably hurting the response latency of fine-grained RPCs. In this work we introduce NEBULA, an architecture optimized to accelerate the most challenging microsecond-scale RPCs, by leveraging two novel mechanisms to drastically improve server throughput under strict tail latency goals. First, NEBULA reduces detrimental queuing at the memory controllers via hardware support for efficient in-LLC network buffer management. Second, NEBULA's network interface steers incoming RPCs into the CPU cores' L1 caches, improving RPC startup latency. Our evaluation shows that NEBULA boosts the throughput of a state-of-the-art key-value store by 1.25–2.19× compared to existing proposals, while maintaining strict tail latency goals.**

***Index Terms*—Client/server and multitier systems, Network protocols, Queuing theory, Memory hierarchy**

I. INTRODUCTION

Modern large-scale online services deployed in datacenters are decomposed into multiple software tiers, which communicate over the datacenter network using Remote Procedure Calls (RPCs) [1]–[3]. The growing software trends of microservices and function-as-a-service have promoted this decomposition, raising system-level implications. RPCs to ubiquitous, performance-critical software tiers (e.g., data stores) perform very little computation per RPC and often exhibit µs-scale runtimes. Such software tiers have high communication-to-computation ratios, designating networking as the key performance determinant.

In response to these trends, datacenter networking technologies are evolving rapidly, delivering drastic bandwidth and latency improvements. Modern datacenter topologies offer ample path diversity and limit in-network queuing [4]–[6], enabling µs-scale roundtrip latencies. To accommodate growing demands for network bandwidth, commodity fabrics have been rapidly scaling their capacity, with 1.2Tbps InfiniBand and 1.6Tbps Ethernet on the roadmap [7], [8]. Although these improvements will help sustain the demands created by extreme software decomposition, they will also shift networking-associated bottlenecks to the servers themselves. The confluence of shrinking network latency and growing bandwidth with µs-scale RPCs has initiated a "hunt for the killer microseconds" [9] across all levels of the system stack, raising server design implications because historically negligible overheads (e.g., 15µs for TCP/IP [10]) have become bottlenecks.

In this work, we study two server-side sources of queuing that significantly affect the tail latency of µs-scale RPCs. The first is load imbalance, because the distribution of RPCs to a CPU's many cores directly affects tail latency. Load balancing for µs-scale RPCs has thus recently attracted considerable attention, with both software [11], [12] and hardware [13], [14] solutions to improve upon the static load distribution support modern NICs offer in the form of Receive Side Scaling (RSS) [15].

The second source of server-side queuing has received less attention. As network bandwidth gradually approaches memory bandwidth, memory accesses caused by incoming network traffic interfere with the application's accesses, creating queuing effects in the memory subsystem that noticeably degrade the tail latency of µs-scale RPCs. This bandwidth interference effect is especially noticeable in the context of latency-optimized networking technologies like DDIO, InfiniBand, and next-generation fully integrated architectures like Scale-Out NUMA [16]. We find that such interference can degrade a server's achievable throughput under strict tail latency goals by more than 2×. It is evident that avoiding such interference with specialized network traffic management in the memory hierarchy is necessary to leverage advancements in software architecture and networking hardware.

To tackle both sources of detrimental server-side queuing, we propose the **Ne**twork **Bu**ffer **La**cerator (NEBULA) architecture, optimized for fine-grained RPCs. While prior techniques exist to ameliorate each queuing source individually, no existing system tackles both. NEBULA extends a recently proposed hardware design for load balancing [13] with a novel NIC-driven buffer management mechanism, which alleviates detrimental interference of incoming network traffic with the application's memory accesses. Our key insight is that

This work was partially funded by Swiss National Science Foundation project no. 200021_165749, as well as Oracle Labs' *Accelerating Distributed Systems* grant.

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00027

network buffer provisioning should not be a function of the number of remote endpoints a server communicates with, but of an individual server's peak achievable RPC service rate. Consequently, network buffers that typically have footprints of many MBs, or even GBs, can be shrunk to 100s of KBs, delivering significant savings in precious DRAM resources, which drive the scale and cost of datacenters [17], [18]. More importantly, NEBULA leverages this observation to keep such small network buffers SRAM-resident via intelligent management, thus absorbing the adverse effects of bandwidth interference due to network traffic.

NEBULA's second new feature is NIC-to-core RPC steering. We introduce NIC extensions to monitor each core's queue of RPCs and directly steer the next queued RPC's payload into the correct core's L1 cache, just in time before the core picks it up for processing. Our design avoids L1 cache pollution—a key reason why network packet placement in DDIO-enabled NICs [19] has been restricted to the LLC—and accelerates RPC startup time, reducing overall response latency. Combined, NEBULA's two memory hierarchy management optimizations boost a key-value store's achieved throughput under tail latency constraints by $1.25 - 2.19\times$ compared to prior proposals.

In summary, our work makes the following contributions:

- We show that architects are currently faced with a dilemma: either building a system with memory bandwidth interference between the NIC and CPU cores, or with load imbalance between the cores. This dilemma is particularly pronounced when combining the immense bandwidth of future NICs with emerging µs-scale software layers.
- We address the aforementioned dilemma by proposing a co-design of the network protocol and NIC hardware that maximizes network buffer reuse in the server's LLC. We conduct a mathematical analysis that enables in-LLC buffer management by showing that strict application-level tail-latency goals can *only* be met by maintaining shallow queues of incoming RPCs, and that such queues are easily accommodated in the LLC.
- We advance the state-of-the-art in network packet placement by the NIC in the memory hierarchy. For the first time, we drive network packets all the way to L1 caches while avoiding cache pollution effects, improving the response latency of µs-scale RPCs.
- We present the first holistically optimized architecture for µs-scale RPCs, with integrated support for load balancing and network-aware memory hierarchy management.

The rest of the paper is structured as follows. §II highlights the impact of server-side queuing effects on the tail latency of µs-scale RPCs, arising from load imbalance and/or memory bandwidth interference due to incoming network traffic. §III quantifies these two effects with a queuing model. §IV and §V introduce NEBULA's design and implementation, respectively. We describe our methodology in §VI and evaluate NEBULA in §VII. We discuss NEBULA's broader applicability and prospects for datacenter adoption in §VIII, cover related work in §IX and conclude in §X.

II. Background and Challenges

A. Online Services and Latency-Sensitive RPCs

Online services are deployed in datacenters to deliver high-quality responses to a plethora of concurrent users, with response times small enough to deliver a highly interactive experience. Responding to each query with such low latency often requires datasets to be replicated across servers, and software to be decomposed into multiple tiers which communicate over the network to synthesize a response. Inter-server communication typically occurs in the form of RPCs: a user query triggers a sequence of RPC fan-outs, forming a tree of sub-queries that spans hundreds or thousands of servers [20]. That fan-out requires strict limits on the *tail* response latency of each service tier [21], commonly referred to as a Service Level Objective (SLO).

We focus on services generating µs-scale RPCs, because of their unique challenges and growing presence. While already widespread in the form of in-memory data stores, services with this profile are becoming increasingly omnipresent because of the trend toward microservice software architectures [1]–[3]. Fine-grained RPCs are particularly vulnerable to latency degradation, as otherwise negligible overheads become comparable to the RPC's actual service time. As the frequency of µs-scale RPCs in the datacenter increases, so does the importance of handling them efficiently.

B. Architectures for Low-Latency Networking

As microservices imply a growth in communication-to-computation ratio [1], the first necessary step to handle them efficiently is the use of highly optimized network protocols and operating system components. Recent datacenter networking advancements to address this challenge include user-level network stacks (e.g., DPDK [22]), hardware-assisted solutions (e.g., dataplanes [23], [24], Microsoft Catapult [25]), and even at-scale InfiniBand/RDMA deployments [26], whose hardware-terminated protocol drastically shrinks the overhead of network stack processing. The demand for lower latency motivates even more radical proposals to approach the fundamental lower bound of networking latency—propagation delay—via on-server integration of network fabric controllers. Such designs already exist in both academia (e.g., Scale-Out NUMA [16], the FAME-1 RISC-V RocketChip SoC [27]) and industry (e.g., Intel Omni-Path [28], Gen-Z [29]). We expect similar latency-optimized solutions to find their way into datacenters soon, in response to the growing demands and latency-sensitive nature of online services.

With protocol and architectural optimizations to minimize the in-network component of each individual RPC, the next step is to address server-side inefficiencies. In this work, we identify two sources of queuing as primary obstacles for servers handling µs-scale RPCs. The first is the contention that rapidly growing network bandwidth can inflict on a server's memory channels. The second is load imbalance across a server's many cores. While there are existing techniques to address each challenge, no current solution addresses both.

C. Memory Bandwidth Interference or Load Imbalance?

The future of commodity network fabrics is one of tremendous bandwidth, with 1.2Tbps InfiniBand and 1.6Tbps Ethernet already on the roadmap [7], [8]. Such growth directly affects server design, as incoming network traffic becomes a non-trivial fraction of a server's available memory bandwidth. If the server is naively architected, network traffic will destructively interfere with the memory requests of the executing applications, causing queuing effects that noticeably degrade the tail latency of µs-scale RPCs. Therefore, it is imperative for future servers to prevent such interference by handling network traffic within their SRAM caches, which have the requisite bandwidth to keep pace.

High-performance user-level network stacks complicate the task of in-cache network traffic management. To provide the latency benefits of zero-copy and synchronization-free message reception, network receive buffers are provisioned on a per-endpoint basis (i.e., with dedicated buffers per connection), so that the server can receive a message from any communicating endpoint, anytime. Connection-oriented provisioning creates a fundamental scalability problem, wasting precious DRAM resources and raising performance implications for RPC libraries built on RDMA NICs [30]–[33]. Furthermore, multi-client interleaving of incoming requests results in unpredictable access patterns to network buffers, effectively eliminating the probability of finding these buffers in the server's LLC, thus causing increased DRAM bandwidth usage. We find that this second—often overlooked—effect can significantly hurt the latency of µs-scale RPCs.

Aiming to reduce the memory capacity waste of connection-oriented buffer provisioning, InfiniBand offers the Shared Receive Queue (SRQ) option to enable inter-endpoint buffer sharing [34]. SRQ's reduced buffer footprint can also implicitly ameliorate memory bandwidth interference due to increased LLC buffer residency. Another approach to prevent network buffers from overflowing into memory is to statically reduce the number of buffers available for allocation by the network stack, suggested in prior work such as ResQ [35]. Unfortunately, SRQ is vulnerable to load imbalance between the CPU cores, as it corresponds to a multi-queue system by design: clients must specify the queue pair (QP) each request is sent to, implying an a priori request-to-core mapping. The same multi-queue limitation is inherent to Receive-Side Scaling (RSS) [15] support, often used for inter-core load distribution [12], [23], [24]. Synchronization-free scaling of the ResQ approach for multi-core servers similarly results in a multi-queue configuration vulnerable to load imbalance.

Prior work has demonstrated that the distribution of incoming µs-scale RPCs to a server's cores crucially impacts tail latency. Due to the fundamentally better tail latency provided by single-queue systems, many proposals have advocated for a single-queue approach in network switches [14], operating systems [11], [12] and NIC hardware [13]. For µs-scale RPCs, the overheads of software-based load balancing, or any synchronization at all, can be comparable to the RPC service time itself; the corresponding throughput loss motivates using hardware mechanisms for load balancing. RPCValet [13], a recently proposed NIC-driven mechanism for synchronization-free, single-queue load balancing, improves throughput under SLO by up to 1.4× over a multi-queue system. However, RPCValet's network buffer provisioning is connection-oriented by design, and thus creates memory bandwidth interference. We now proceed to conduct a theoretical study of the effects of these two shortcomings, interference and imbalance, on RPC throughput under SLO.

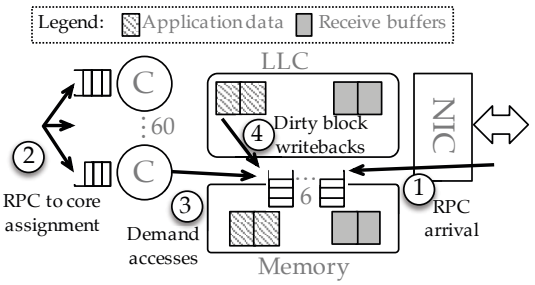


Fig. 1. Memory hierarchy associated with handling incoming RPCs, modeled by our queuing system.

III. Theoretical Study: Interference & Imbalance

To demonstrate the performance impacts of the previous quandary, we construct a first-order queuing model that captures the critical interactions between the network, cores, and memory. We assume a multi-core CPU and a NIC that places packets directly into the LLC, similar to DDIO [19]. Fig. 1 shows the modeled components of our queuing system and the interactions that occur during RPC handling: ① incoming traffic from the NIC placing RPCs into the memory hierarchy, ② assignment of RPCs to cores, ③ demand traffic generated by the cores while servicing RPCs, and ④ writebacks of dirty blocks from the LLC to memory.

Load-balancing implications pertain to step ②. We consider two hardware-assisted alternatives: multi-queue behavior like RSS/SRQ and single-queue behavior like RPCValet. In the former case, we optimistically assume uniform load distribution, which is the best-case performance for RSS/SRQ. We only consider the effect of the single- or multi-queue policy, without penalizing any system for implementation-specific overhead.

Memory bandwidth interference concerns relate to steps ①, ③ and ④. When network receive buffers exceed the LLC in size, writing new packets requires fetching the buffers from DRAM into the LLC first ①. The cores' memory requests also compete for memory bandwidth ③. Write accesses from ① and ③ create dirty blocks in the LLC, consuming additional memory bandwidth when they are evicted from the LLC ④.

We select parameters for Fig. 1's queuing system by reproducing the current best-performing server architecture for key-value serving: 60 CPU cores, 45MB of LLC, six DDR4 memory channels, and a 300Gbps NIC [36]. These parameters are also representative of modern servers such as Intel's Xeon Gold [37] and Qualcomm's Centriq [38]. We model a Poisson RPC arrival process, and assume RPCs

modeled after MICA SETs [39], which perform a hash-index lookup (64B read) followed by a 512B value write and have an average service time $\bar{S} = 630ns$. Additional queuing model details are available in Appendix A.

We evaluate the following four queuing system configurations using discrete-event simulation:

1) *RSS.* A multi-queue system with uniform assignment of incoming RPCs to cores and 136MB of receive buffers (see §VI for sizing details). This configuration suffers from load imbalance and memory bandwidth interference.
2) *RPCValet.* Single-queue load balancing of RPCs to cores with equal receive buffer provisioning to RSS. Although RPCValet solves load imbalance, it still suffers from bandwidth interference.
3) *SRQ.* Like RSS, a multi-queue system with uniform distribution of incoming RPCs to cores, but assumes *ideal* buffer management, where all network buffers are reused in the LLC, eliminating bandwidth interference.
4) NEBULA. Combines the best traits of *RPCValet* and *SRQ*: single-queue load balancing and ideal buffer management.

Note that both SRQ and NEBULA are hypothetical configurations we employ to demonstrate upper performance bounds of an idealized zero-overhead buffer management mechanism.

Fig. 2 shows the 99th% latency of the RPCs and the total memory bandwidth utilization of the modeled system, assuming an SLO of 6.5µs ($\simeq 10 \times \bar{S}$). The systems with connection-oriented buffer bloat (RSS and RPCValet) saturate early at a load of 0.61, because they exhaust the server's memory bandwidth by generating $127GB/s$ of traffic. Although RPCValet attains sub-µs 99th% latency until saturation—an order of magnitude lower than RSS at a load of 0.56—thanks to improved load balancing, it only supports 61% of the maximum possible load because of its memory bandwidth bottleneck.

SRQ scales beyond RSS and RPCValet, meeting the SLO up to a load of 0.76. SRQ outperforms RPCValet despite load imbalance because its network buffers are always reused in the LLC, thus eliminating all traffic from step ① and writebacks of dirty network buffer blocks ④. Effectively, network contention for memory bandwidth is removed from ③'s path.

Despite SRQ's improved performance, it still leaves 24% of the server's maximum throughput unprocured, due to its inability to balance load across cores. SRQ's lost throughput would increase proportionally to RPC service time variance; our model only considers the narrow distributions observed in object stores. NEBULA combines the best of RPCValet and SRQ, attaining near-optimal 99th% latency up to a load of 0.97, only saturating when it becomes CPU bound. Even at maximum load, NEBULA only consumes $\sim 50\%$ of the server's maximum memory bandwidth.

In conclusion, our model shows that RPC-optimized architectures must address both load balancing and memory bandwidth interference. Although load balancing has been extensively addressed in prior work from both architectural [13] and operating system [12], [14] perspectives, our models demonstrate that memory bandwidth contention is also a primary performance determinant. Next, we present the principles guiding our design to address memory bandwidth interference and attain the performance of Fig. 2's NEBULA configuration.

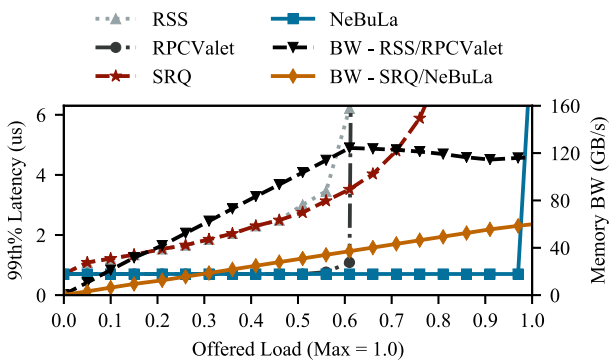


Fig. 2. Discrete-event simulation results, showing the impact of load imbalance and memory bandwidth interference on the tail latency of µs-scale RPCs.

IV. NEBULA DESIGN

In this section, we describe the insights guiding our design for latency-critical software tiers using RPCs, that addresses the problem of memory bandwidth interference. We begin by describing the salient characteristics of our network protocol and server system baseline, proceed to set out the additions we propose for the NEBULA architecture, and finally demonstrate the underpinning mathematical insights.

A. Baseline Architecture and Network Stack

Our baseline network stack features a hardware-terminated, user-level network protocol, with an on-chip integrated NIC [16], [40]. We choose this baseline because of its suitability for latency-critical software systems, and because systems featuring these characteristics are emerging in production datacenters (see §II-B). NEBULA's prerequisites from the underlying system are the following.

Prerequisite 1: RPC-oriented transport. Recent work advocates for RPC-oriented transports as a better fit than conventional bytestream transports in datacenters, as they enable improved flow control [41], latency-aware routing [6], and inter-server load balancing [14]. NEBULA requires an RPC transport because exposing the abstraction of an RPC in the transport layer enables the NIC to make decisions pertinent to the application-level unit that an RPC represents, rather than the limited view of packets as mere chunks of bytes.

Prerequisite 2: NIC-driven load balancing. NEBULA relies on a synchronization-free mechanism to load-balance RPCs between a server's many cores, for improved tail latency under SLO compared to statically partitioned RPC queues. In the presence of load imbalance, a server could begin violating the application's SLO due to increased application-layer queuing, hitting an earlier bottleneck than bandwidth interference, as shown by our analysis in §III. For the remainder of this paper, we adopt RPCValet [13] as our baseline architecture, which features an integrated NIC that delivers single-queue load balancing without incurring any software overheads.

B. Key Design Features

In-LLC RPC buffer management. With RPC as the network's transport abstraction (Prerequisite 1), the NIC can expose and manage a single network endpoint used by all its *N* clients, instead of one endpoint per client, reducing the set of buffers for incoming RPC requests from *N* to one. Our analysis in §IV-C shows that for a stable latency-sensitive system, the active buffer size required can be conveniently accommodated in a modern server's LLC—this lies at the core of our contributions. NEBULA leverages this opportunity to implement an in-LLC buffer management mechanism at the NIC, eliminating the memory bandwidth implications demonstrated in §III.

SLO-aware protocol. NEBULA targets latency-sensitive online services with a strict tail-latency SLO; a response violating the SLO has limited value. We leverage this qualitative characteristic to relax hardware requirements. For example, under conditions of heavy server-side queuing (deep queues), newly arriving RPCs that would inevitably violate the SLO are eagerly NACKed, informing the client early about increased load conditions on the server. Bounding queue depths by taking SLO constraints into account is synergistic with the goal of maximizing the LLC residency of RPC buffers. NEBULA includes the necessary protocol extensions to support judicious NACKing to inform applications early of SLO violations. This policy is also synergistic with existing tail-tolerant software techniques, which eagerly retry [21] or reject [42] requests predicted to be delayed. NEBULA's fail-fast approach to SLO violation complements these techniques, which can replace predictions with timely feedback.

Efficient packet reassembly. A direct effect of moving from a connection- to an RPC-oriented transport (Prerequisite 1) is that packets belonging to various multi-packet RPCs are intermingled and must be properly demultiplexed into destination buffers in the correct order. Given our basic assumption of a hardware-terminated protocol, such demultiplexing and reassembly needs to be handled by the NIC hardware. Our baseline architecture featuring an on-chip integrated NIC exacerbates the RPC reassembly challenge in two ways. First, the on-chip resources that can be dedicated to the NIC's reassembly hardware are limited by tight power and area constraints, as well as by the use of existing interfaces to the server's memory system. Second, because architectures with integrated NICs often use small MTUs (e.g., 64B in Scale-Out NUMA [16]), the frequency of RPC fragmentation and reassembly is exceptionally high.

Although most RPCs in the datacenter are small, many are still larger than 64B [41], [43]. Prior work circumvented these reassembly challenges by allocating dedicated buffers for messaging per node pair [13], leading to the buffer bloat implications detailed in §II-C. As NEBULA drastically reduces buffering requirements and shares buffers between endpoints, we employ a protocol-hardware co-design to support efficient packet reassembly, even at futuristic network bandwidths.

NIC-to-core RPC steering. When handling µs-scale RPCs, small latency components, such as on-chip interactions involved in the delivery of an RPC to a core, matter. Directly steering an incoming RPC from the network to a core's L1 cache, rather than having the core read it from memory or the LLC, can noticeably accelerate the RPC's startup. However, such an action has to be *timely* and *accurate* to avoid adverse effects.

A key challenge of NIC-to-core RPC steering is that the NIC generally does not know a priori which core will handle a given incoming RPC, and inaccurate steering would be detrimental to performance. A second peril is potentially over-steering RPCs, as directly storing several incoming RPCs into a core's L1 cache could thrash it. DDIO avoids these complications by conservatively limiting network packet steering to a small fraction of the LLC [19] and not further up the cache hierarchy, leaving available opportunities for performance improvement.

Successful NIC-to-core RPC steering requires breaking RPC handling into two distinct steps: arrival and dispatch. The goal of the arrival step is payload placement in an LLC-resident queue, to mitigate memory bandwidth interference. The goal of the dispatch step is to transfer an RPC from the in-LLC queue to a core's L1 cache, right before that core starts processing the RPC. The dispatch decision is also an integral part of the load-balancing mechanism (Prerequisite 2); in the case of steering, the focus shifts from *which* core to dispatch to, to *when*. Therefore, we extend RPCValet's basic load balancing mechanism from simple RPC-to-core assignment to complete payload steering. We defer implementation details to §V.

C. Bounded Server-Side Queuing

Our key insight motivating shallow SRAM-resident queues for incoming RPCs is that *SLO-violating RPCs typically spend most of their on-server time waiting in a deep queue*. Conversely, the on-server time of SLO-abiding RPCs is primarily spent on-core rather than waiting in a queue. Therefore, it is sufficient to allocate memory space for only enough RPCs whose queuing latencies will not correspond to SLO violation. Keeping these buffers SRAM-resident not only reduces the memory footprint, but also boosts performance in two ways. First, by lowering memory bandwidth demand, thus reducing queuing in the memory subsystem for application accesses. Second, by providing cores with faster access to the arrived RPC, thus reducing RPC startup time.

Setting a hard constraint on tail latency implies that the maximum amount of time an RPC can be queued at a server must also be bounded. A server's RPC response time is $t_r = t_q + t_s$, where t_q and t_s represent the average RPC queuing and service time, respectively. Assuming an SLO of $10 \times t_s$ as is common in the literature [12], [13] constrains queuing time to $t_q \leq 9t_s$. Simply put, to respect the SLO, a hypothetical server with a single processing unit must be operating at a load point where the 99th% of the distribution of the number of waiting RPCs is ≤ 9.

We conduct a queuing analysis to generalize this observation and estimate the queuing capacity required for a server that is operating under an SLO-abiding load. We model a k-core server

TABLE I
MEASURED LOAD AND QUEUE DEPTHS AT SLO, USING SYNTHETIC SERVICE TIME DISTRIBUTIONS.

Distribution	Max Load @ SLO	99th% Q. Depth @ SLO
Deterministic	0.999	54
Exponential	0.995	319
Bimodal	0.940	410

after an `M/G/k` queuing system, assuming Poisson arrivals and a general service time distribution. We are interested in the distribution $\hat{N}_q$ that represents the number of queued RPCs under a system load $A = \frac{\lambda}{\mu}$, where λ is the arrival rate of new requests, and μ is the per-core service rate (stability condition: $\lambda < k \times \mu$). The mean of the distribution, $E[\hat{N}_q]$ is given by Eqn. 1 [44], where $C_k(A)$ is the Erlang-C formula:

$$E[\hat{N}_q] = C_k(A)\frac{A}{k-A} \quad (1)$$

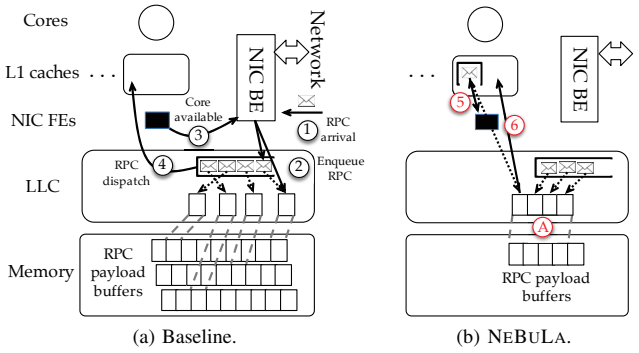
Although $E[\hat{N}_q] \xrightarrow{A \to k} \infty$, solving the equation for high system load results in $E[\hat{N}_q]$ values closer to k. For example, for a 64-core CPU at an extreme load of $A = 63$, $E[N_q] = 54$. Ideally, we would be able to analytically solve for the 99th% of the $\hat{N}_q$ distribution. However, closed-form expressions for the various percentiles of $\hat{N}_q$ are not known. We therefore use §III's queuing simulator to collect statistics for $\hat{N}_q$ using three service time distributions from ZygOS [12]:

- Deterministic: $P[X = \bar{S}] = 1$
- Exponential: $P[X] = \lambda e^{-\lambda x}$ $(\bar{S} = \frac{1}{\lambda})$
- Bimodal: $P[X = \frac{\bar{S}}{2}] = 0.9$, $P[X = 5.5 \times \bar{S}] = 0.1$

Table I shows the maximum load meeting an SLO of $10 \times \bar{S}$ and the 99th% of $\hat{N}_q$ at that load. The results corroborate the intuition that as the service time dispersion grows (e.g., for bimodal), the peak load under SLO drops and the 99th% queue depth increases. Additionally, the deterministic distribution's 99th% is equal to $E[\hat{N}_q]$, because there is no variability in the rate at which cores drain requests from the queue. This analysis shows that even at extremely high loads, the number of RPCs waiting in the queue is small enough to easily fit inside a server's existing SRAM resources. Provisioning for deeper queuing is effectively useless, because RPCs landing in queues deeper than the upper bound demonstrated by our analysis will violate the SLO anyway.

Provisioning a receive buffer of limited size on the server requires the transport protocol to signal a "failure to deliver" (NACK) if the request is dropped because of a full queue. It is up to the client to react to a NACK reception; for example, the request could be retried or sent to a different server, as proposed by Kogias et al. [45]. Exposing delivery failures to the client follows the end-to-end principle in systems design [46]: the client application is best equipped to handle such violations and should be informed immediately. Note that the client has to perform proactive load monitoring for SLO violations even if the transport protocol never rejects requests.

In summary, we show that meeting strict SLOs requires shallow server-side RPC queues. Thus, system designers can leverage this observation to provision small enough amounts of buffering that comfortably fit inside the on-chip caches of a modern server, eliminating the buffer memory footprint problem, as well as the latency implications because of memory bandwidth interference.



(a) Baseline. (b) NEBULA.

Fig. 3. Overview of baseline and NEBULA architectures.

V. NEBULA IMPLEMENTATION

We now describe our NEBULA implementation, based on the design features outlined in §IV. We first briefly introduce the baseline architecture's critical features, then detail NEBULA's protocol and NIC hardware extensions.

A. Baseline Architecture

We use Scale-Out NUMA (soNUMA) [16] as our baseline architecture. soNUMA combines a lean user-level, hardware-terminated protocol with on-chip NIC integration. The NIC leverages integration into its local CPU's coherence domain for rapid interactions with the cores. Applications schedule soNUMA operations (e.g., `send`) using an RDMA-like memory-mapped Queue Pair (QP) interface.

RPCValet [13] is a NIC extension for RPC load balancing and a key enabler for NEBULA's RPC-to-core steering mechanism. RPCValet balances incoming RPCs across the cores of a multi-core server in a single-queue, synchronization-free fashion. Fig. 3a provides a high-level demonstration of its operation. RPCValet is based on soNUMA's NI_{split} architecture [40], where the NIC comprises two discrete entities, a frontend (FE) and a backend (BE). The former handles the control plane (i.e., QP interactions) and is collocated per core; the latter handles the network packets and data transfers between the network and memory hierarchy and is located at the chip's edge.

When a new RPC arrives ① the NIC BE writes its payload in the LLC, creates an RPC arrival notification ②—which contains a pointer to the RPC's payload—and stores it in a dedicated queue. As soon as a core becomes available to process a new RPC, its corresponding NIC FE notifies the NIC BE ③, which, in turn, dequeues the first entry from the arrival notification queue and writes it in the core's Completion Queue (CQ) ④. The core receives the RPC arrival notification by polling its CQ and follows the pointer in the notification message to read the RPC's payload from the LLC. This greedy load assignment policy corresponds to single-queue behavior.

```
while (true):
    payload_ptr = wait_for_RPC(msg_domain)
    //process the received RPC and build response...

    free_buffer(buffer_ptr, buffer_size)

    RPC_send(resp_buffer_ptr, buffer_size,
                  target_node, msg_domain)
```

Fig. 4. Pseudocode of an RPC-handling event loop.

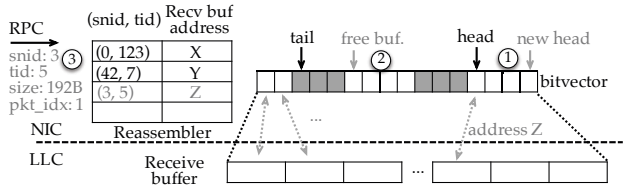


Fig. 5. NIC RPC reassembly and buffer management.

Fig. 3b highlights in red NEBULA's key extensions over Fig. 3a's baseline architecture. The first feature, marked as Ⓐ and detailed in §V-C and §V-D, is NEBULA's in-LLC network buffer management for reduced memory pressure. The second feature, NIC-to-core RPC steering, extends the baseline architecture's sequence of RPC arrival notification actions with payload dispatch, shown in steps ⑤-⑥ and detailed in §V-E.

B. RPC Protocol and Software Interface

We implement an RPC layer on top of soNUMA's `send` operation, maintaining RPCValet's messaging interface [13]: all nodes of the same service join the same messaging domain, which includes the buffers and data structures defining where incoming RPCs are placed in memory. Every node participating in a messaging domain allocates a receive buffer in its memory to hold the payloads of incoming RPCs. We change RPCValet's underlying connection-oriented buffer management to achieve NEBULA's key goal of handling all traffic within the server's caches. In NEBULA's case, after the receive buffers are allocated and initialized by software, they are managed by the NIC. We first focus on the software interface and detail hardware modifications later in this section.

Fig. 4 demonstrates the three functions the NEBULA RPC interface exposes to applications within a sample RPC-handling event loop. A server thread waits for incoming RPCs using the `wait_for_RPC` function, which can be blocking or non-blocking. The NIC sends RPC arrivals to this thread via the thread's CQ that is associated with the incoming RPC's messaging domain. After completing the RPC, the application invokes the `free_buffer` function to let the NIC reclaim the buffer. Finally, the application sends a response in the form of a new RPC, specifying the messaging domain, target node, and local memory location that contains the outgoing message. `RPC_send` has a return value indicating whether the outgoing RPC was accepted by the remote end or not, which only clients use to check whether their requests are NACKed by the server. In a well-designed system, the server does not use the return value, as clients should always provision sufficient buffering for responses to their own outstanding requests.

soNUMA acknowledges all messages at the transport layer. NEBULA extends this mechanism with negative acknowledgements (NACKs), which are responses to `send` operations if the receiving end cannot accommodate the incoming RPC. In response to a NACK reception, the application layer receives an error code. The most appropriate reaction to an error code is application-specific, and can range from simple retry of the same `send` at a later time, to arbitrarily sophisticated policies.

C. NIC Extension: Buffer Management

NEBULA's NIC manages a receive buffer per established messaging domain. After the software sets up and registers these buffers with the NIC, the NIC must dynamically allocate them to incoming RPCs, and place incoming packets appropriately. The buffer manager's role is to reserve enough space in the receive buffers for each RPC, ensuring each core has zero-copy access to the buffer until the application explicitly permits the NIC to reclaim it after the RPC is serviced. As the allocator must operate at line rate, simplicity is paramount. The NIC therefore uses a chunk allocation algorithm that manages each receive buffer as an array of consecutive slots in host memory. In allocators of this style, performing a new allocation is as simple as returning the next available slot(s).

The buffer manager uses a bitvector built into the NIC, containing a bit for every cache-block-sized (64B) slot in the receive buffer. Bits corresponding to allocated slots are set to `1` (shaded in Fig. 5). Upon a new RPC arrival, the NIC advances the receive buffer's head pointer by $\lceil RPC\_size/64B \rceil$ slots and sets the corresponding bits in the bitvector. In Fig. 5 step ①, the arrival of a new RPC with $128B < size \leq 192B$ would trigger the `head` pointer to advance three slots to `new head`. As this RPC has a size greater than one cache block and will thus arrive in more than one packet, it will also add an entry to the reassembler, described in §V-D.

Applications free slots by sending a `free_buffer` message specifying a buffer address and length to the NIC (see §V-B), which resets the bitvector's corresponding bits (step ②). After each `free_buffer` message, the buffer manager checks whether the tail pointer can advance, thus reclaiming receive buffer space. In Fig. 5's example, the tail cannot advance because the slot(s) immediately in front of `tail` are still allocated. If the head pointer reaches the tail pointer, the receive buffer is full and the NIC starts NACKing any new RPCs.

As receive slots are freed out of order by the applications, a naive implementation can suffer from internal fragmentation and thus excess NACKs in the case of a rare long-latency event (e.g., a core receives a hard IRQ while handling an RPC). Therefore, we implement a simple scrubbing policy to allow the buffer manager to scan through the bitvector and find the next free range; this operation is triggered when the head meets the tail pointer and is performed off the critical path of new allocation and freeing.

To size the receive buffer (and thus the bitvector), we rely on our analysis in §IV-C. We provision for $10 \times E[N_q]$ queued RPCs, which conveniently covers the 99% depth of even the

bimodal distribution considered in Table I. Factoring the RPC size as well, we size the receive buffer at $10 \times E[N_q] \times avg\_RPC\_size$. As per §IV-C's example, assuming a 64-core server at load $A = 63$ and an average RPC size of 1KB[1], our provisioning results in a 540KB buffer. This choice results in a 1KB bitvector, a modest SRAM cost for a large server chip.

D. NIC Extension: RPC Reassembly

Incoming RPCs exceeding the network MTU in size are fragmented at the transport layer; thus, they must be reassembled before being handed to the application. In a hardware-terminated protocol, such reassembly has to be performed in the NIC's hardware. The specific challenges for designing such reassembly hardware for NEBULA are the baseline architecture's small MTU (64B) and the NIC being an on-chip component. The former implies high reassembly throughput requirements; the latter implies tight area and power budgets.

In many emerging transport protocols for RPCs, all the packets of the same message carry a unique identifier (`tid`) assigned by the originating node [13], [14], [41]. Thus, an incoming packet's combination of `tid` and source node id (`snid`) uniquely identifies the message the packet belongs to. The reassembly operation can be described as performing an exact match between the (`snid, tid`) pair found in each incoming packet's header, and an SRAM-resident "database" of all RPCs that are currently being reassembled. Returning to Fig. 5's example, assume that the second packet of the RPC which previously arrived in step ① reaches the NIC in step ③. The packet's header contains the pair (`3,5`), which is looked up in the reassembler and receive buffer address `Z` is returned. Being the second packet of this RPC (`pkt_idx=1`), the NIC writes the payload to address `Z+64`.

The most common solution for exact matching at high throughput is to use CAMs [48], which are power-hungry due to the large number of wires that must be charged and discharged each cycle. Contrary to our initial expectation, deploying a CAM is a feasible solution. Just as NEBULA's design decision to bound the queue depth of incoming RPCs shrinks receive buffer provisioning requirements, it also sets an upper limit for the number of incoming RPCs that may be under reassembly. Consequently, NEBULA sets an upper size limit on the CAM required for RPC reassembly purposes. With §V-C's 64-core configuration as an example, we need a CAM with $10 \times E[N_q] = 540$ entries. To model the hardware cost of NEBULA's CAM, we use CACTI 6.5 [49] and configure it with the following parameters: built-in ITRS-HP device projections, a 22nm process, and dynamic power optimization with the constraint of meeting a 2GHz cycle time (targeting a futuristic 1Tbps network endpoint—i.e., a packet arrival every ~ 0.5ns). With a reported dynamic power of 45.3mW, such a CAM is reasonably accommodated on chip.

E. NIC-to-Core RPC Steering

RPCValet's mechanism to assign RPCs to cores involves metadata only: the NIC places a pointer to the buffer containing the next RPC's payload in the CQ of the thread that will service that RPC. NEBULA extends this mechanism to also trigger a dispatch of the RPC's payload to the target core's L1 cache. If this payload dispatch completes in a timely fashion, it reduces the RPC's execution time. The accuracy of such RPC payload prefetching is not probabilistic, as it is *based on prescience rather than prediction*: the hardware leverages the semantic information of an RPC arrival and assignment to a core to choreograph the cache hierarchy for faster RPC startup.

NEBULA's NIC-to-core steering mechanism is implemented as a sequence of additional steps after RPCValet's normal operation. First, we modify RPCValet's RPC arrival notifications (Fig. 3a, ②): in addition to the pointer to the RPC's payload, the notification also includes the payload's size. As soon as a new RPC is assigned to a core for processing (Fig. 3a, ④), its NIC FE reads the payload buffer's base address and size from the notification message and forwards them to the core's L1 cache controller as a prefetch hint (Fig. 3b, ⑤). The cache controller uses this information to prefetch the whole RPC payload before the core starts processing that RPC (Fig. 3b, ⑥). We discuss alternative mechanisms to prefetch hints for NIC-to-core RPC steering in §IX.

To guarantee timely RPC payload prefetches, we configure RPCValet to allow two RPCs instead of one to be queued at the NIC FE, thus giving the cache controller ample time to prefetch the second RPC's payload while the first one is being processed by the core. Such slight deviation from true single-queue RPC balancing corresponds to the JBSQ(2) policy proposed by Kogias et. al [14], and has been shown to preserve tail latency in an on-chip setting [13].

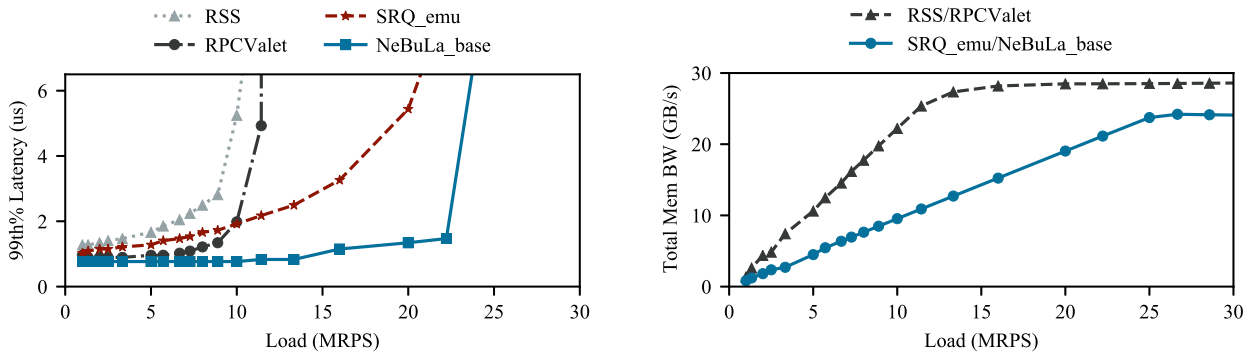
[1]90% of network packets within Facebook's datacenters are <1KB [47].

TABLE II
PARAMETERS USED FOR CYCLE-ACCURATE SIMULATION.

Cores	ARM Cortex-A57; 64-bit, 2GHz, OoO, TSO 3-wide dispatch/retirement, 128-entry ROB
L1 Caches	32KB 2-way L1d, 48KB 3-way L1i, 64B blocks 2 ports, 32 MSHRs, 3-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 16MB total 16-way, 1 bank/tile, 6-cycle latency
Coherence	Directory-based Non-Inclusive MESI
Memory	45ns latency, 2×15.5GBps DDR4
Interconnect	2D mesh, 16B links, 3 cycles/hop

VI. METHODOLOGY

System organization. We simulate one ARMv8 server running Ubuntu Linux in full-system, cycle-level detail, by enhancing QEMU [50] with the timing models of the Flexus simulator [51]. Table II summarizes the simulation parameters. The server implements the NI_{split} soNUMA architecture [40] with LLC, DRAM, and NIC parameters following the state-of-the-art system tuned for in-memory key-value serving [36], which provisions 118GB/s of DRAM bandwidth, a 300Gbps NIC, and a 60MB LLC for 60 CPU cores. To make simulation practical, we scale the system down to a 16-core CPU, maintaining the same LLC size and DRAM bandwidth per core; thus, we provision 31GB/s of DRAM bandwidth and a 16MB LLC. Commensurately scaling the NIC bandwidth indicates



(a) 99th percentile latency.

(b) Total memory bandwidth.

Fig. 6. Tail latency and bandwidth for all evaluated systems, using a 50/50 GET/SET query mixture.

provisioning an 80Gbps NIC. However, we found that under the most bandwidth-intensive workloads, NEBULA could saturate the full 80Gbps while still having ~15% idle CPU cycles; therefore, we increased NIC bandwidth to 120Gbps to permit NEBULA to reach the CPU cores' saturation point.

Application software. We use the MICA in-memory key-value store [39] with the following modifications: (i) we ported its networking layer to soNUMA, (ii) for compatibility reasons with our simulator, we ported the x86-optimized MICA to ARMv8. We deploy a 16-thread MICA instance with a 819MB dataset, comprising 1.6M 16B/512B key/value pairs. We use the default MICA hash bucket count (2M) and circular log size (4GB). Larger datasets would further reduce the LLC hit ratio, exacerbating the memory bandwidth interference problem that NEBULA alleviates.

We build a load generator into our simulator, which generates client requests at configurable rates, using a Poisson arrival process, uniform data access popularity, and the following GET/SET query mixtures: 0/100, 50/50, 95/5. Unless explicitly mentioned, results are based on the 50/50 query mix.

Evaluated configurations. We evaluate five different designs to dissect NEBULA's benefits:

- **RPCValet:** We use RPCValet [13] as a baseline architecture, which features NIC-driven single-queue load balancing, optimized for tail latency. RPCValet provisions its packet buffers in a static connection-oriented fashion, resulting in buffer bloat with high connection counts. Assuming a cluster size of 1024 servers, a maximum per-message size of 512B, and 256 outstanding messages per node pair, RPCValet's buffers consume 136MB, significantly exceeding the server's LLC size. This provisioning represents any connection-based system that allocates buffer space per endpoint, such as RDMA-optimized [31] and UDP-based RPC systems [33].
- **RSS:** A representative of the Receive Side Scaling (RSS) [15] mechanism available in modern NICs. Our implementation optimistically spreads requests to cores uniformly. Like RPCValet, RSS suffers from buffer bloat, and also suffers from load imbalance being a multi-queue system.
- **NEBULA$_{base}$:** This configuration retains RPCValet's load-balancing capability and adds NEBULA's protocol and hardware extensions for space-efficient buffer management. Following our analysis in §IV-C, NEBULA$_{base}$ allocates 81KB of buffering for incoming RPCs, corresponding to $10 \times E[N_q] = 160$ slots of 512B each.
- **SRQ$_{emu}$:** A proxy for InfiniBand's Shared Receive Queue (SRQ) mechanism, enabling buffer sharing among end-points/cores to tackle the buffer bloat problem. We optimistically assume the same hardware-based buffer manager as NEBULA$_{base}$ without any software overheads normally involved when the threads post free buffers to the SRQ. Unlike NEBULA$_{base}$, existing SRQ implementations do not feature hardware support for load balancing. Hence, SRQ$_{emu}$ represents an RSS system without buffer bloat, or, equivalently, a NEBULA$_{base}$ system without hardware support for single-queue load balancing.
- **NEBULA:** The full set of our proposed features, namely NEBULA$_{base}$ plus NIC-to-core RPC steering.

Evaluation metrics. We evaluate NEBULA's benefits in terms of throughput under SLO. Our SLO is a 99th% latency target of $10\times$ the average RPC service time [12]. All of our measurements are server-side: each RPC's latency measurement begins as soon it is received by the NIC, and ends the moment its buffers are freed by a core after completing the request. As NEBULA can NACK incoming RPCs under high load, we conservatively count NACKs as ∞ latency measurements.

VII. EVALUATION

A. Impact of Load Imbalance and Bandwidth Interference

We start by evaluating §VI's first four designs to quantify the impacts of load imbalance and memory bandwidth interference. Fig. 6 shows 99th% latency and memory bandwidth as a function of load. Fig. 6b groups the series for RSS/RPCValet and SRQ$_{emu}$/NEBULA$_{base}$ together, as they are effectively identical. RSS performs the worst, as it suffers from both load imbalance and bandwidth contention. Although RPCValet delivers $2.6\times$ lower 99th% latency than RSS at 10MRPS

TABLE III
SENSITIVITY ANALYSIS FOR QUERY MIXTURE.

GET/SET Mix	RPCValet MRPS	RPCValet BW@SLO	NEBULA MRPS	NEBULA BW@SLO
0/100	11.4	24.5 GB/s	26.7	22.6 GB/s
50/50	11.4	25.3 GB/s	22.2	21.1 GB/s
95/5	11.4	24.8 GB/s	22.2	20.9 GB/s

due to superior load balancing, both systems saturate beyond 11.4MRPS.

Fig. 6b sheds light on the source of the performance gap between SRQ_{emu} and the two systems suffering from buffer bloat. RSS and RPCValet utilize about 25.3GB/s of memory bandwidth at 11.4MRPS, greater than 80% of the server's maximum of 31GB/s. In contrast, SRQ_{emu} consumes less than 75% of that bandwidth at 20MRPS and therefore delivers $1.75\times$ higher load than RSS/RPCValet, corroborating our claim that memory bandwidth contention can negatively impact latency. SRQ_{emu}'s performance, combined with the small difference between RPCValet and RSS, may seem to suggest that load balancing is unimportant. However, we demonstrate next that as soon as the bandwidth bottleneck is removed, load balancing has a major performance impact.

NEBULA_{base} is the only system of the four that avoids both destructive bandwidth contention and load imbalance, attaining a throughput under SLO of 22.2MRPS with a sub-2µs 99th% latency. We measured the mean zero-load service time of MICA RPCs using 512B payloads to be ∼630ns, which corresponds to a throughput bound of 25.3MRPS. Thus, NEBULA_{base} is within 12% of the theoretical maximum. Before saturation, NEBULA_{base}'s minimal 81KB of buffers are adequate and we observe no NACK generation, which concurs with our theoretical analysis suggesting that $10 \times E[N_q]$ receive buffer slots suffice for a server operating under strict SLO. Beyond ∼22MRPS, the system quickly becomes unstable, server-side queues rapidly grow, and the number of NACKs escalates. We consider this to be a non-issue, because operating a server beyond its saturation point is not a realistic deployment scenario. Overall, NEBULA_{base} outperforms SRQ_{emu} by $1.2\times$ in terms of throughput under SLO and by $2.2\times$ in 99th% latency at a load of 16MRPS, due to improved load balancing.

B. Sensitivity to Workload GET/SET Ratio and Item Size

We now study the sensitivity of NEBULA_{base} and RPCValet to different workload behaviors by varying the GET/SET query mix. Table III reports the maximum throughput under SLO and the memory bandwidth consumption at peak throughput for each query mix we experimented with. We find that both systems are largely insensitive to query mixture, as RPCValet reaches the same saturation point for all three workloads, remaining bottlenecked by memory bandwidth contention in all cases. As the fraction of GETs increases, the NIC-generated bandwidth drops because GET payloads only carry a key, as compared to SETs that carry a 512B value. Despite less NIC-generated bandwidth, the cores' aggregate bandwidth commensurately rises, because they must copy 512B values out of the data store into the message sent in response to the GET. Ultimately, memory bandwidth usage per query remains roughly constant.

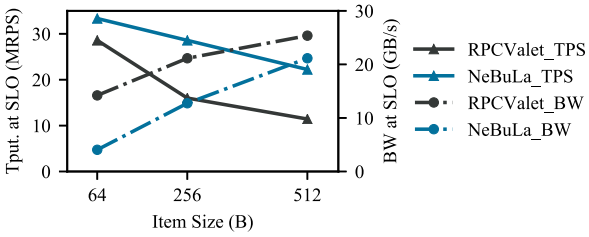


Fig. 7. Performance of RPCValet and NEBULA varying the MICA value size, using a 50/50 GET/SET query mix.

In contrast, NEBULA_{base} experiences a 19% throughput increase for the 0/100 mix compared to 50/50, because the mean service time drops by 70ns. This improvement happens because of different data copy directions. For a SET, the core loads the incoming value from the NIC's small on-chip cache or the LLC (where the payload buffer resides after the NIC writes it), and then must write it to the DRAM-resident MICA log. As the payload buffer is already on-chip, the time that the core's LSQ is blocked on the payload's data is a single remote cache read. In contrast, GETs first must fetch the value from the MICA log, and write it to the response buffer; thus, GETs block the core for a longer period of time.

Next, we evaluate the impact of varying MICA's item size. Fig. 7 shows the maximum throughput under SLO, and memory bandwidth of RPCValet and NEBULA_{base} with 64B, 256B and 512B items. Items smaller than 64B (cache line size) result in the same memory bandwidth utilization as 64B items. As item size shrinks, RPCValet's throughput under SLO commensurately increases, reaching 16MRPS with 256B items and 26.7MRPS with 64B items: smaller items naturally result in less memory bandwidth generated from both the NIC and the CPU cores, alleviating memory bandwidth contention. For item sizes larger than 64B, RPCValet becomes bandwidth-bound, capping throughput under SLO at ∼ 21MRPS.

NEBULA_{base}'s performance also improves with smaller items. Cores are less burdened with copying data to/from the MICA log, reducing the mean RPC service time by 9% and 19% with 256B and 64B items, respectively. The shorter service time results in a saturation point of 33.3MRPS with 64B items. This is $1.16\times$ higher than RPCValet even when bandwidth contention is not a factor, because NEBULA_{base} eliminates the costly step of write-allocating payloads into the LLC before an RPC can be dispatched to a core. Finally, we emphasize that NEBULA_{base} attains equal throughput under SLO as RPCValet, handling items $4\times$ larger (256B vs. 64B).

C. NIC-to-Core Steering

Finally, we evaluate NEBULA's NIC-to-core steering mechanism. Fig. 8 compares NEBULA_{base} (no prefetching), NEBULA (NIC-to-core steering enabled) and NEBULA_{SWPref}, which is the same hardware configuration as NEBULA_{base}, but contains

a modification to the RPC handling loop (Fig. 4) to prefetch any future RPCs found waiting in the CQ. As the software prefetch overhead is on the critical path, we optimize the prefetch logic to scan a maximum of 8 slots (which fit in a single cache block) in the CQ to find an RPC, and set a prefetch degree of one. We measure the overhead of this code as 60ns.

Below 16MRPS, all configurations perform identically, because each core only has one RPC outstanding, and the software overhead to scan the CQ is small enough to complete before the next RPC arrival. Above 16MRPS, the software overhead begins to manifest itself, causing a 31% increase in 99th% latency compared to NEBULA$_{base}$ at 22MRPS. As 60ns represents $\sim 10\%$ of MICA's service time, we conclude that prefetching at such high loads requires hardware support to eliminate the overhead of determining prefetch addresses in software.

Our expectation is that NEBULA should trim ~ 50 cycles (25ns) from each RPC's runtime, hiding the latency of fetching the remotely cached RPC payload that is needed to access MICA's hash index. This improvement should only show at the tail, because in the average case, the cores only have one RPC in their CQ without a next RPC to prefetch. Between 16 and 22MRPS, NEBULA improves the 99th% latency by 64ns, i.e., a 10% reduction in RPC service time. We attribute the roughly $2\times$ difference with our expectation to the increased cache subsystem latencies in the loaded system. Therefore, the benefit of removing the longer wait for the RPC's payload from the critical path via timely prefetching increases. As a result, NEBULA outperforms NEBULA$_{base}$ by 3MRPS. At high load, the fraction of RPCs dispatched to a core with CQ depth $>= 1$ grows to 75%, making NEBULA's 10% service time reduction the common case.

We also repeated the same experiment with 64B payloads, which have reduced on-core service times of 510ns. NEBULA$_{SWPref}$'s overhead grows to $> 10\%$ of the service time, and therefore NEBULA delivers $3\times$ lower 99th% latency at 22.2MRPS. NEBULA delivers the same 3MRPS throughput benefit for both 64B and 512B payloads.

Employing all of its features, NEBULA improves throughput under SLO over current multi-queue (SRQ) and single-queue (RPCValet) systems by $1.25\times$ and $2.19\times$ respectively. With both 512B and 64B RPCs, NEBULA saturates at maximum CPU throughput, and does not leave significant performance improvement headroom.

VIII. DISCUSSION

Deployments benefitting from NEBULA. Datacenter services are increasingly adopting a microservices architecture, where the overall functionality is broken down into modular components and a single user request reaches across hundreds or thousands of servers [20]. As the number of layers comprising these services grows, so does the importance of minimizing the latency of each inter-layer RPC-based interaction—a typical case of the "tail at scale" effect [21]. The more pervasive such decomposition is across datacenter-deployed services, the broader NEBULA's applicability. We evaluate a Key-Value

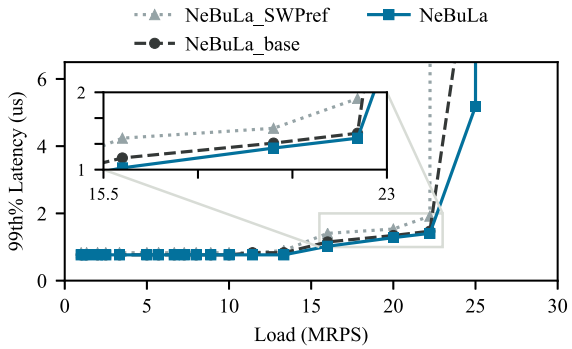


Fig. 8. Comparison of NEBULA prefetching policies. MICA uses 512B values and a 50/50 GET/SET query mix.

Store (KVS) as a representative of µs-scale software, because it is an extensively optimized application that is used by virtually every service and is well-known to exhibit µs-scale service times [12], [13], [33].

In general, for an application to benefit from NEBULA, it must have at least one of the following characteristics in addition to strict tail-latency constraints: (i) intensive network traffic, which, when combined with application memory bandwidth usage, approaches the system's total memory bandwidth; or (ii) µs-scale on-core service times. For the former, NEBULA removes bandwidth interference effects inflicted by system scale, by limiting the number of RPCs queued on the server and ensuring in-cache network packet residency. For the latter, NEBULA converts 80–100ns per-packet DRAM accesses into L1 cache hits, which our evaluation shows boosts throughput under SLO by more than 10% for sub-µs RPCs on a 16-core server. As examples, we identify two other applications that exhibit these characteristics and also form important building blocks of datacenter services.

State machine replication (SMR) provides fault-tolerance by electing a *leader* server to replicate incoming RPCs across N *followers*, which are often themselves KVS. SMR leaders meet both criteria for benefitting from NEBULA. With service times of ~1µs [33], they benefit from NEBULA's NIC-to-core steering. Furthermore, because they replicate incoming RPCs in a $1:N$ fashion, their network packet buffers may spill into DRAM, creating memory bandwidth interference, and would therefore benefit from NEBULA's in-cache buffer management.

Network function virtualization (NFV) workloads execute user-defined packet processing operations inside a software switch (e.g., OpenVSwitch). NFV operations exhibit 300ns–5µs service times and LLC-resident datasets [35], [52]; therefore, NFV would benefit from NEBULA's NIC-to-core steering.

Finally, NEBULA can benefit co-located application deployments. When a latency-critical application is co-located with one that is LLC- and/or memory bandwidth-intensive, NEBULA will constrain the LLC space taken up by the latency-critical application's packet buffers, improving its LLC hit ratio on network packets, while indirectly also benefitting the bandwidth-intensive workload by limiting LLC interference.

Integration with datacenter networks. NEBULA requires the following features from the underlying network stack: (i) native protocol support for an RPC-oriented transport, (ii) hardware-terminated transport, and (iii) a lossless link layer. The two latter requirements are being addressed by recent works targeting datacenter-scale RDMA deployments [26], [53], while native support for RPC-oriented transports is an active research direction [14], [41]. A NEBULA-compliant protocol requires the addition of NACK messages which are sent by the receiver upon encountering deep server-side queues, and passed back to the RPC layer when they return to the sender. We argue that similar messages already exist in high-performance networking solutions (e.g., an RDMA NIC generates a local CQ entry after a remote memory write is performed).

IX. RELATED WORK

Leaky DMA. ResQ [35] encounters the "leaky DMA" problem, which is similar to the bandwidth interference problem we study. ResQ's authors observe that when deploying co-located NFV workloads using DPDK, the LLC space required by DDIO exceeds its default limit, and memory traffic multiplies. ResQ ameliorates memory traffic by statically limiting DPDK's buffer allocation to 10% of the LLC. In contrast, our work establishes a mathematical bound on the number of required buffers, based on queuing theory. While we share the intuition that limiting outstanding buffers can reduce excessive memory traffic, NEBULA targets hardware-terminated transports and demonstrates that mere KBs of buffering space are sufficient to handle µs-scale RPCs. NEBULA also maintains inter-core load balancing, a factor beyond ResQ's scope.

The leaky DMA problem is also observed in the context of the Shenango runtime scheduler for latency-sensitive datacenter workloads [54]. Shenango foregoes zero-copy I/O to hand buffers back to the NIC as soon as possible, increasing LLC reuse by DDIO. In contrast, NEBULA maintains zero-copy I/O, because we find that all useful packet buffers can be accommodated in the LLC when operating under a tight SLO.

RPCs over connected transports. HERD [55], FaSST [31] and FaRM [30] all propose optimizations in order to alleviate the scalability issues of InfiniBand's connected transports. FaSST uses solely unconnected datagram transports to reduce the number of Queue Pairs that must be cached in the NIC [31], while FaRM accepts the inherent performance loss of some connection sharing [30]. Storm [32] shows that at rack scale (i.e., up to 64 nodes), the newest generation of ConnectX-5 NICs have significantly improved in scalability, and hence designs a transaction API prioritizing RDMA operations which require connected transport. Our work targets datacenter-scale deployments and applies to any software with µs-scale RPCs.

eRPC [33] improves buffer scalability by using multi-packet receive queue descriptors introduced in Mellanox ConnectX-4 NICs. These descriptors keep NIC-resident state constant with the number of connected nodes. However, eRPC's server-side buffering state still scales with the number of connections, as it allocates memory for each pair of connected threads ([33]:§3.1). We show that although server memory is plentiful, µs-scale RPCs require buffers to be kept to LLC-resident sizes in order to avoid memory bandwidth interference and meet tight SLOs.

Compute hardware/NIC co-design. The importance of KVS in datacenters has resulted in multiple proposals for KVS-optimized hardware [36], [56]. SABRes [57] leverages coherent on-chip NIC integration to introduce a NIC extension for atomic remote object access, alleviating software overheads in KVS using one-sided RDMA reads for low latency. Li et al. [36] study MICA's performance characteristics and propose a bespoke CPU for KVS whose parameters we adopt for our baseline. Their work also observes the primacy of in-LLC buffer management, and empirically sizes the LLC based on simulation to minimize miss rate. They do not observe the same DRAM bandwidth contention as we do, because the amount of network state in their system would not scale with the number of communicating servers due to its use of UDP [39]. Achieving similar throughput with an order of magnitude lower latency, as we attempt to do, requires hardware-terminated network stacks and brings back the challenge of scaling dedicated per-endpoint state. NEBULA therefore begins with a hardware-terminated protocol, and demonstrates that packet buffering state should be sized based on SLO rather than system scale.

Daglis et al. [40] studied the on-chip latency implications of the VIA/RDMA network programming model [58], and proposed NIC decomposition and passing messages between NIC components to avoid multi-hop coherence interactions. We have similar insights regarding the on-chip data path of RPC payloads under µs-scale SLOs, and propose a solution that uses the NIC's role in RPC dispatch to accurately prefetch payloads without cache coherence modifications.

Latency-optimized systems software. The need for low latency has led systems designers to aggressively limit queuing in the transport and RPC protocol stacks themselves [45], [59]. Kogias et al. [45] also observe that limiting server-side queuing is critical for µs-scale RPCs, and use TCP flow control to limit the number of requests per connection based on the application's SLO. NEBULA performs buffer management for hardware- rather than software-terminated protocols.

CacheDirector [60] improves RPC latency by modifying DDIO to steer the header of each network packet into the LLC tile closest to the core that will process the packet. We go further by steering the whole packet all the way into the core's L1 cache. In the past, placing network data in L1 caches has been avoided due to pollution concerns, which NEBULA addresses by steering only a single RPC at a time.

Thomas et al. [61] observe that at 100Gbps+, packet inter-arrival times drop below DRAM latency, and that applications performing memory accesses will inevitably backpressure the NIC and lead to dropped packets. They propose CacheBuilder, a slab allocator using existing cache partitioning technology to guarantee the application's dataset is LLC- rather than memory-resident. CacheBuilder therefore only benefits applications with LLC-resident datasets, whereas NEBULA considers SLO-constrained applications with memory-resident datasets.

NIC-to-core data steering. NEBULA employs a prefetch hint

mechanism to steer the payload of an incoming RPC to its target core. Payload prefetches differ from regular cache accesses that are triggered by front-side demand accesses, because they are initiated from the back-side of the cache. Our work chooses to use prefetch hints because payload prefetches are transformed into regular front-side initiated operations, keeping both the cache controllers and coherence protocol unmodified. Intel's DCA mechanism also leverages prefetch hints to load data into the LLC [62]. Other mechanisms such as direct injection (e.g., DDIO's write-allocate/update policy [19]) and Curious Caching [63] have been proposed to support back-side initiated operations. NEBULA could be trivially adapted to use these mechanisms if supported by the cache controller.

X. CONCLUSION

In hyperscale datacenters, the combination of growing network bandwidth and µs-scale software layers is shifting performance bottlenecks to server endpoints. In particular, this work shows that the vast bandwidth of future NICs can interfere with application traffic at the server's memory controllers, due to network buffer bloat resulting from non-scalable connection-oriented buffer provisioning. However, for software with tight latency constraints, queuing theory in fact indicates that only limited server-side queuing is useful, and therefore network buffer provisioning can be shrunk to sizes trivially accommodated by the server's LLC. Following this insight, we propose NEBULA, a co-design of the network protocol and server hardware that actively keeps all network buffers LLC-resident, mitigating memory bandwidth interference. NEBULA thus alleviates the bottleneck of deployment scale for latency-critical software layers, hastening the path to adoption of latency-optimized hardware-terminated network stacks.

NEBULA demonstrates that nascent RPC-oriented transport protocols serve as powerful enablers for future NIC architectures to actively cooperate with network-intensive applications in achieving their performance goals. NEBULA uses the NIC's integral role in load balancing to minimize tail latency by actively triggering prefetches in the cache hierarchy before RPCs begin processing, without requiring changes to the caches or coherence protocol. Our evaluation shows that this capability delivers performance benefits of $1.13\times$ for a Key-Value Store application, as packets are directly placed into the L1 cache rather than the LLC. Overall, NEBULA improves the throughput under SLO of µs-scale RPCs by up to $2.19\times$.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their constructive feedback, as well as Dmitrii Ustiugov, Ed Bugnion, Marios Kogias, and Adrien Ghosn for their invaluable advice during this work. We also thank the members of the PARSA group for their unwavering support.

APPENDIX A

This appendix details the queuing model used in §III (Fig. 1). We model bandwidth consumption by assuming that accesses reaching memory are (a) NIC writes of incoming RPCs into receive buffers, deemed to be misses (described next), and (b) all key/value accesses to the datastore (no cache locality). To model the extra bandwidth of dirty writebacks (Fig. 1, ④), we double an access' memory bandwidth utilization *iff* the access is both a write and determined to be a miss.

Assuming abundant interleaving among requests from clients, we model the LLC miss ratio for the network buffers as:

$$MR_{buf} = 1 - min\left(\frac{LLC\ capacity}{Recv.\ buf.\ size}, 1\right) \qquad (2)$$

Thus, the DRAM traffic generated by the NIC writing incoming RPC payloads (Fig. 1, ①) is $BW_{NIC} \times MR_{buf}$, where BW_{NIC} is the incoming network bandwidth. Eqn 2 optimistically assumes that the NIC can use the entire LLC, whereas Intel servers place a default (but configurable) limit at 10% of the LLC size [19].

We use a service time $S = t_{fixed} + (N_{acc} \times AMAT)$, where t_{fixed} is the CPU processing time for the RPC, N_{acc} is the number of serialized memory accesses per RPC and $AMAT$ is the average memory access time, which under zero load is 45ns, but is affected by queuing conditions. We measured the zero-load service time to be 630ns in our experimental setup (see §VI) and set $N_{acc} = 2$ because MICA first synchronously accesses its hash index, and then moves multiple cache blocks in/out of the data store in parallel due to the MLP of an out-of-order core. Thus, $t_{fixed} = 540$ns and each request's resulting service time is solely affected by $AMAT$, which changes as a function of system load.

REFERENCES

[1] Y. Gan *et al.*, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems." in *Proc. 24th Int. Conf. Architectural Support for Prog. Languages and Operating Sys. (ASPLOS-XXIV)*, 2019, pp. 3–18.

[2] G. Kakivaya *et al.*, "Service fabric: a distributed platform for building microservices in the cloud." in *Proc. 2018 EuroSys Conf.*, 2018, pp. 33:1–33:15.

[3] A. Sriraman and T. F. Wenisch, "µTune: Auto-Tuned Threading for OLDI Microservices." in *Proc. 13th Symp. Operating Sys. Design and Implementation (OSDI)*, 2018, pp. 177–194.

[4] A. G. Greenberg *et al.*, "VL2: a scalable and flexible data center network." in *Proc. ACM SIGCOMM 2009 Conf.*, 2009, pp. 51–62.

[5] A. Singh *et al.*, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network." in *Proc. ACM SIGCOMM 2015 Conf.*, 2015, pp. 183–197.

[6] M. Handley *et al.*, "Re-architecting datacenter networks and stacks for low latency and high performance." in *Proc. ACM SIGCOMM 2017 Conf.*, 2017, pp. 29–42.

[7] Ethernet Alliance. (2020) The Ethernet Alliance 2020 Roadmap. [Online]. Available: https://ethernetalliance.org/technology/2020-roadmap/

[8] InfiniBand Trade Association. (2019) InfiniBand Roadmap. [Online]. Available: https://www.infinibandta.org/infiniband-roadmap/

[9] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan, "Attack of the killer microseconds." *Commun. ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[10] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's Time for Low Latency." in *Proc. 13th Workshop Hot Topics in Operating Sys. (HotOS-XIII)*, 2011.

[11] K. Kaffes *et al.*, "Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency." in *Proc. 16th Symp. Networked Sys. Design and Implementation (NSDI)*, 2019, pp. 345–360.

[12] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks." in *Proc. 26th ACM Symp. Operating Sys. Princ. (SOSP)*, 2017, pp. 325–341.

[13] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs." in *Proc. 24th Int. Conf. Architectural Support for Prog. Languages and Operating Sys. (ASPLOS-XXIV)*, 2019, pp. 35–48.

[14] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2P2: Making RPCs first-class datacenter citizens." in *Proc. 2019 USENIX Annual Tech. Conf. (ATC)*, 2019, pp. 863–880.

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

[16] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA." in *Proc. 19th Int. Conf. Architectural Support for Prog. Languages and Operating Sys. (ASPLOS-XIX)*, 2014, pp. 3–18.

[17] A. Eisenman *et al.*, "Reducing DRAM footprint with NVM in facebook." in *Proc. 2018 EuroSys Conf.*, 2018, pp. 42:1–42:13.

[18] K. T. Lim *et al.*, "Disaggregated memory for expansion and sharing in blade servers." in *Proc. 36th Int. Symp. Comp. Architecture (ISCA)*, 2009, pp. 267–278.

[19] (2012) Intel Data Direct I/O Technology. Intel Corp. [Online]. Available: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html

[20] R. L. Sites. (2017) Finding Datacenter Software Tail Latency. [Online]. Available: https://memento.epfl.ch/event/ic-colloquium-finding-datacenter-software-tail-lat

[21] J. Dean and L. A. Barroso, "The tail at scale." *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[22] Data Plane Development Kit. The Linux Foundation Projects. [Online]. Available: https://www.dpdk.org

[23] A. Belay *et al.*, "The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane." *ACM Trans. Comput. Syst.*, vol. 34, no. 4, pp. 11:1–11:39, 2017.

[24] S. Peter *et al.*, "Arrakis: The Operating System Is the Control Plane." *ACM Trans. Comput. Syst.*, vol. 33, no. 4, pp. 11:1–11:30, 2016.

[25] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture." in *Proc. 49th Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2016, pp. 7:1–7:13.

[26] C. Guo *et al.*, "RDMA over Commodity Ethernet at Scale." in *Proc. ACM SIGCOMM 2016 Conf.*, 2016, pp. 202–215.

[27] S. Karandikar *et al.*, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud." in *Proc. 45th Int. Symp. Comp. Architecture (ISCA)*, 2018, pp. 29–42.

[28] Intel Omni-Path Architecture Driving Exascale Computing and HPC. Intel Corp. [Online]. Available: https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html

[29] Gen-Z Consortium: Computer Industry Alliance Revolutionizing Data Access. [Online]. Available: http://genzconsortium.org/

[30] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory." in *Proc. 11th Symp. Networked Sys. Design and Implementation (NSDI)*, 2014, pp. 401–414.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs." in *Proc. 12th Symp. Operating Sys. Design and Implementation (OSDI)*, 2016, pp. 185–201.

[32] S. Novakovic *et al.*, "Storm: a fast transactional dataplane for remote data structures." in *Proc. 12th ACM Int. Conf. Sys. and Storage (SYSTOR)*, 2019, pp. 97–108.

[33] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be General and Fast." in *Proc. 16th Symp. Networked Sys. Design and Implementation (NSDI)*, 2019, pp. 1–16.

[34] Infiniband Trade Association. (2006) InfiniBand Architecture Release 1.2.1. (Date last accessed 10 June 2019). [Online]. Available: https://www.infinibandta.org/ibta-specifications-download

[35] A. Tootoonchian *et al.*, "ResQ: Enabling SLOs in Network Function Virtualization." in *Proc. 15th Symp. Networked Sys. Design and Implementation (NSDI)*, 2018, pp. 283–297.

[36] S. Li *et al.*, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform." in *Proc. 42nd Int. Symp. Comp. Architecture (ISCA)*, 2015, pp. 476–488.

[37] Linley Group, "Xeon Scalable Reshapes Server Line," *Microprocessor Report*, July 2017.

[38] ——, "Centriq Aces Scale-Out Performance," *Microprocessor Report*, November 2017.

[39] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage." in *Proc. 11th Symp. Networked Sys. Design and Implementation (NSDI)*, 2014, pp. 429–444.

[40] A. Daglis, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, "Manycore network interfaces for in-memory rack-scale computing." in *Proc. 42nd Int. Symp. Comp. Architecture (ISCA)*, 2015, pp. 567–579.

[41] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout, "Homa: a receiver-driven low-latency transport protocol using network priorities." in *Proc. ACM SIGCOMM 2018 Conf.*, 2018, pp. 221–235.

[42] M. Hao *et al.*, "MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface." in *Proc. 26th ACM Symp. Operating Sys. Princ. (SOSP)*, 2017, pp. 168–183.

[43] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store." in *Proc. 2012 ACM SIGMETRICS Int. Conf. Meas. and Model. of Comp. Sys.*, 2012, pp. 53–64.

[44] M. Zukerman. (2018, Dec.) Introduction to Queueing Theory and Stochastic Teletraffic Models. arXiv. ArXiv:1307.2968.

[45] M. Kogias and E. Bugnion, "Flow control for Latency-Critical RPCs." in *Proc. 2018 Workshop Kernel-Bypass Networks (KBNETS@SIGCOMM)*, 2018, pp. 15–21.

[46] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, 1984.

[47] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network." in *Proc. ACM SIGCOMM 2015 Conf.*, 2015, pp. 123–137.

[48] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[49] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques." in *Proc. 2011 IEEE/ACM Int. Conf. Comp.-Aided Design (ICCAD)*, 2011, pp. 694–701.

[50] QEMU: the FAST! processor emulator. [Online]. Available: https://www.qemu.org/

[51] T. F. Wenisch *et al.*, "SimFlex: Statistical Sampling of Computer System Simulation." *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[52] Y. Yuan, Y. Wang, R. Wang, and J. Huang, "HALO: accelerating flow classification for scalable packet processing in NFV." in *Proc. 46th Int. Symp. Comp. Architecture (ISCA)*, 2019, pp. 601–614.

[53] R. Mittal *et al.*, "Revisiting network support for RDMA." in *Proc. ACM SIGCOMM 2018 Conf.*, 2018, pp. 313–326.

[54] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads." in *Proc. 16th Symp. Networked Sys. Design and Implementation (NSDI)*, 2019, pp. 361–378.

[55] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services." in *Proc. ACM SIGCOMM 2014 Conf.*, 2014, pp. 295–306.

[56] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing SoC accelerators for memcached." in *Proc. 40th Int. Symp. Comp. Architecture (ISCA)*, 2013, pp. 36–47.

[57] A. Daglis *et al.*, "SABRes: Atomic object reads for in-memory rack-scale computing." in *Proc. 49th Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2016, pp. 6:1–6:13.

[58] D. Dunning *et al.*, "The Virtual Interface Architecture." *IEEE Micro*, vol. 18, no. 2, pp. 66–76, 1998.

[59] M. Alizadeh *et al.*, "Data center TCP (DCTCP)." in *Proc. ACM SIGCOMM 2010 Conf.*, 2010, pp. 63–74.

[60] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostic, "Make the Most out of Last Level Cache in Intel Processors." in *Proc. 2019 EuroSys Conf.*, 2019, pp. 8:1–8:17.

[61] S. Thomas, R. McGuinness, G. M. Voelker, and G. Porter, "Dark packets and the end of network scaling." in *Proc. 2018 Symp. Architectures for Networking and Commun. Sys. (ANCS)*, 2018, pp. 1–14.

[62] A. Kumar and R. Huggahalli, "Impact of Cache Coherence Protocols on the Processing of Network Traffic." in *Proc. 40th Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2007, pp. 161–171.

[63] D. Chiou, "Extending the reach of microprocessors: column and curious caching." Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999.