RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs

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

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

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

Abstract

Modern online services come with stringent quality requirements in terms of response time tail latency. Because of their decomposition into fine-grained communicating software layers, a single user request fans out into a plethora of short, µs-scale RPCs, aggravating the need for faster inter-server communication. In reaction to that need, we are witnessing a technological transition characterized by the emergence of hardware-terminated user-level protocols (e.g., InfiniBand/RDMA) and new architectures with fully integrated Network Interfaces (NIs). Such architectures offer a unique opportunity for a new NI-driven approach to balancing RPCs among the cores of manycore server CPUs, yielding major tail latency improvements for µs-scale RPCs.

We introduce *RPCValet*, an NI-driven RPC load-balancing design for architectures with hardware-terminated protocols and integrated NIs, that delivers near-optimal tail latency. *RPCValet*'s RPC dispatch decisions emulate the theoretically optimal single-queue system, without incurring synchronization overheads currently associated with single-queue implementations. Our design improves throughput under tight tail latency goals by up to 1.4×, and reduces tail latency before saturation by up to 4× for RPCs with µs-scale service times, as compared to current systems with hardware support for RPC load distribution. *RPCValet* performs within 15% of the theoretically optimal single-queue system.

ACM Reference Format:
Alexandros Daglis*, Mark Sutherland, and Babak Falsafi. 2019. *RPCValet*: NI-Driven Tail-Aware Balancing of µs-Scale RPCs. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304070

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304070

1 Introduction

Modern datacenters deliver a breadth of online services to millions of daily users. In addition to their huge scale, online services come with stringent Service Level Objectives (SLOs) to guarantee responsiveness. Often expressed in terms of *tail* latency, SLOs target the latency of the slowest requests, and thus bound the slowest interaction a user may have with the service. Tail-tolerant computing is one of the major ongoing challenges in the datacenter space, as long-tail events are rare and rooted in convoluted hardware-software interactions.

A key contributor to the well-known "Tail at Scale" challenge [15] is the deployment of online services' software stacks in numerous communicating *tiers*, where the interactions between a service's tiers take the form of Remote Procedure Calls (RPCs). Large-scale software is often built in this fashion to ensure modularity, portability, and development velocity [26]. Not only does each incoming request result in a wide fan-out of inter-tier RPCs [10, 23], each one lies directly on the critical path between the user and the online service [6, 16, 29, 50]. The amalgam of the tail latency problem with the trend towards ephemeral and fungible software tiers has created a challenge to preserve the benefits of multi-tiered software while making it tail tolerant.

To lower communication overheads and tighten tail latency, there has been an intensive evolution effort in datacenter-scale networking hardware and software, away from traditional POSIX sockets and TCP/IP and towards lean user-level protocols such as InfiniBand/RDMA [21] or dataplanes such as IX and ZygOS [7, 47]. Coupling protocol innovations with state-of-the-art hardware architectures such as Firebox [4], Scale-Out NUMA [43] or Mellanox's BlueField Smart-NIC [37], which offer tight coupling of the network interface (NI) with compute logic, promises even lower communication latency. The net result of rapid advancements in the networking world is that inter-tier communication latency will approach the fundamental lower bound of speed-of-light propagation in the foreseeable future [20, 50]. The focus of optimization hence will completely shift to efficiently handling RPCs at the endpoints as soon as they are delivered from the network.

The growing number of cores on server-grade CPUs [36, 38] exacerbates the challenge of distributing incoming RPCs to handler cores. Any delay or load imbalance caused by

* This work was done while the author was at EPFL.

this initial stage of the RPC processing pipeline directly impacts tail latency and thus overall service quality. Modern NIC mechanisms such as Receive-Side Scaling (RSS) [42] and Flow Direction [24] offer load distribution and connection affinity, respectively. However, the key issue with these mechanisms, which apply static rules to split incoming traffic into multiple receive queues, is that they do not truly achieve *load balancing* across the server's cores. Any resulting load imbalance after applying these rules must be handled by system software, introducing unacceptable latency for the most latency-sensitive RPCs with µs-scale service times [47, 53].

In this paper, we propose *RPCValet*, a co-designed hardware and software system to achieve *dynamic* load balancing across CPU cores, based on the key insight that on-chip NIs offer the ability to monitor per-core load in real time and steer RPCs to lightly loaded cores. The enabler for this style of dynamic load balancing is tight CPU-NI integration, which allows fine-grained, nanosecond-scale communication between the two, unlike conventional PCIe-attached NIs. To demonstrate the benefits of our design, we first classify existing load-distribution mechanisms from both the hardware and software worlds as representative of different queuing models, and show how none of them is able to reach the performance of the theoretical best case. We then design a minimalistic set of hardware and protocol extensions to Scale-Out NUMA (soNUMA) [43], an architecture with on-chip integrated NIs, to show that a carefully architected system can indeed approach the best queuing model's performance, significantly outperforming prior load-balancing mechanisms. To summarize, our contributions include:

- *RPCValet*, an NI-driven dynamic load-balancing design that outperforms existing hardware mechanisms for load distribution, and approaches the theoretical maximum performance predicted by queuing models.
- Hardware and protocol extensions to soNUMA for native messaging support, a required feature for efficient RPC handling. We find that, in contrast to prior judgment [43], native messaging support is not disruptive to the key premise of NI hardware simplicity, which such architectures leverage to enable on-chip NI integration.
- An *RPCValet* implementation on soNUMA that delivers near-ideal RPC throughput under strict SLOs, attaining within 3–16% of the theoretically optimal queuing model. For µs-scale RPCs, *RPCValet* outperforms software-based and RSS-like hardware-driven load distribution by 2.3–2.7× and 29–76%, respectively.

The paper is organized as follows: §2 outlines the performance differences between multi- and single-queue systems, highlighting the challenges in balancing incoming RPCs with short service times among cores. §3 presents *RPCValet*'s design principles, followed by an implementation using soNUMA as a base architecture in §4. We detail our methodology in §5 and evaluate *RPCValet* in §6. Finally, we discuss related work in §7 and conclude in §8.

2 Background

2.1 Application and technology trends

Modern online services are decomposed into deep hierarchies of mutually reliant tiers [26], which typically interact using RPCs. The deeper the software hierarchy, the shorter each RPC's runtime, as short as a few µs for common software tiers such as data stores. Fine-grained RPCs exacerbate the tail latency challenge for services with strict SLOs, as accumulated µs-scale overheads can result in a long-tail event.

To mitigate the overheads of RPC-based communication, network technologies have seen renewed interest, with the InfiniBand fabric and protocol beginning to appear in datacenters [21] due to its low latency and high IOPS. With networking latency approaching the fundamental limits of propagation delays [20], any overhead added to the raw RPC processing time at a receiving server critically impacts latency. For example, while InfiniBand significantly reduces latency compared to traditional TCP/IP over Ethernet, InfiniBand adapters still remain attached to servers over PCIe, which contributes an extra µs of latency to each message [33, 43].

Efficiently handling µs-scale RPCs requires the elimination of these µs-scale overheads, which is the goal of fully integrated solutions (e.g., Firebox [4], soNUMA [43]). Such architectures employ lean, hardware-terminated network stacks and integrated NIs to achieve sub-µs inter-server communication, representing the best fit for latency-sensitive RPC services. NI integration enables rapid fine-grained interaction between the CPU, NI, and memory hierarchy, a feature leveraged previously to accelerate performance-critical operations, such as atomic data object reads from remote memory [14]. In this paper, we leverage NI integration to break existing tradeoffs in balancing RPCs across CPU cores and significantly improve throughput under SLO.

2.2 Load Balancing: Theory

To study the effect of load balancing across cores on tail latency, we conduct a first-order analysis using basic queuing theory. We model a hypothetical 16-core server after a queuing system that features a variable number of input queues and 16 serving units. Fig. 1 shows three different queuing system organizations. The notation *Model* $Q \times U$ denotes a queuing system with Q FIFOs where incoming messages arrive and U serving units per FIFO. The invariant across the three illustrated models is $Q \times U = 16$. The 16×1 system cannot perform any load balancing; incoming requests are uniformly distributed across 16 queues, each with a single serving unit. 1×16 represents the most flexible option that achieves the best load balancing: all serving units pull requests from a single FIFO. Finally, 4×4 represents a middle ground: incoming messages are uniformly distributed across four FIFOs with four serving units each.

To evaluate different queuing organizations, we employ discrete event simulations modeling Poisson arrivals and four

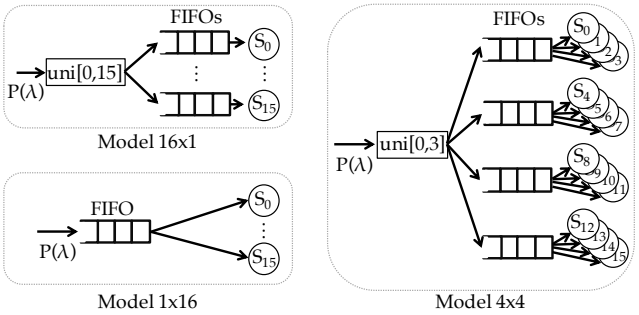


Figure 1. Different queuing models for 16 serving units (CPU cores). P(λ) stands for Poisson arrival distribution.

different service time distributions: fixed, uniform, exponential, and generalized extreme value (GEV). Poisson arrivals are commonly used to model the independent nature of incoming requests. §5 details each distribution's parameters.

Fig. 2a shows the performance of five queuing systems $Q \times U$ with (Q, U) = (1,16), (2,8), (4,4), (8,2), (16,1), for an exponential service time distribution. The system's achieved performance is directly connected to its ability to assign requests to idle serving units. As expected, performance is proportional to U. The best and worst performing configurations are 1×16 and 16×1 respectively, while 2×8, 4×4 and 8×2 lie in between these two.

Fig. 2b and 2c show the relation of throughput and 99th percentile latency for the two extreme queuing system configurations, namely 1×16 and 16×1. As seen in Fig. 2a, 1×16 significantly outperforms 16×1. 16×1's inability to assign requests to idle cores results in higher tail latencies and a peak throughput 25–73% lower than 1×16 under a tail latency SLO at 10× the mean service time $\bar{S}$. In addition, the degree of performance degradation is affected by the service time distribution. For both queuing models, we observe that the higher a distribution's variance, the higher the tail latency (TL) before the saturation point is reached, hence $TL_{fixed} < TL_{uni} < TL_{exp} < TL_{GEV}$. Also, the higher the distribution's variance, the more dramatic the performance gap between 1×16 and 16×1, as is clearly seen for GEV.

The application's service time distribution is beyond an architect's control, as it is affected by numerous software and hardware factors. However, they can control the queuing model that the underlying system implements. The theoretical results suggest that systems should implement a queuing configuration that is as close as possible to a single-queue (1×16) configuration.

2.3 Load Balancing: Practice

A subtlety not captured by our queuing models is the practical overhead associated with sharing resources (i.e., the input queue). In a manycore CPU, allowing all the cores to pull incoming network messages from a single queue requires synchronization. We refer to this RPC dispatch mode as "pull-based". Especially for short-lived RPCs, with service times of a few µs, such synchronization represents significant overhead. Architectures that share a pool of connections between cores have this pitfall; common examples include using variants of Linux's `poll` system call, or locked event queues supported by `libevent`.

An alternative approach for distributing load to multiple cores, advocated by recent research, is dedicating a private queue of incoming network messages to each core [7, 45]. Although this design choice corresponds to a rigid $N \times 1$ queuing model (N being the number of cores), it completely eschews overheads related to sharing (i.e., synchronization and coherence), delivering significant throughput gains. By leveraging RSS [42] inside the NI, messages are consistently distributed at arrival time to one of the N input queues. This ultimately results in a different mode of communication: instead of the cores *pulling* messages from a single queue, the NI hardware actively *pushes* messages into each core's queue. We refer to this load distribution mode as "push-based".

FlexNIC [30] extends the push-based model by proposing a P4-inspired domain-specific language, allowing software to install match-action rules into the NI. Despite their many differences, both FlexNIC and RSS completely rely on decisions based on the RPC packets' header content. Whether configured statically or by the application, push-based load distribution still fundamentally embodies a multi-queue system vulnerable to load imbalance, as no information pertaining to the system's current load is taken into account. §2.2's queuing models demonstrate the effect of this imbalance as compared to a system with balanced queues.

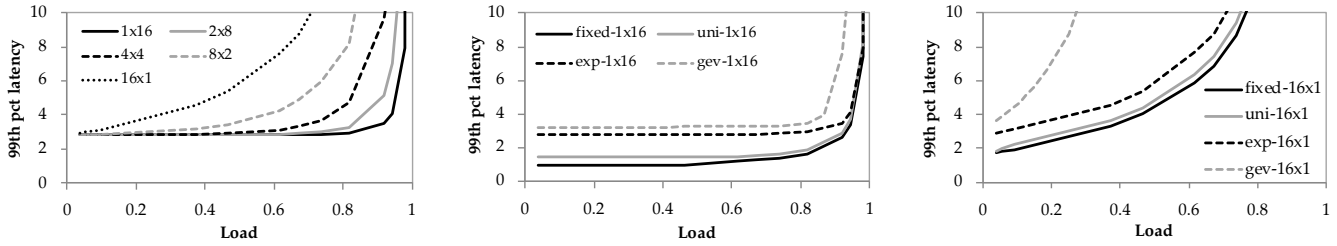
The two aforementioned approaches to load distribution, pull- and push-based, represent a tradeoff between synchronization and load imbalance. In this paper, we leverage the on-chip NI logic featured in emerging fully integrated architectures such as soNUMA [43] to introduce a novel push-based NI-driven load-balancing mechanism capable of breaking that tradeoff by making dynamic load-balancing decisions.

3 *RPCValet* Load-Balancing Design

This section describes the insights and foundations guiding *RPCValet*'s design. Our goal is to achieve a synchronization-free system that behaves like the theoretical best single-queue model. We begin by setting forth our basic assumptions about the underlying hardware and software, then explain the roadblocks to achieving dynamic load balancing, and conclude with the principles of *RPCValet*'s design.

3.1 Basic Architecture

We design *RPCValet* for emerging architectures featuring fully integrated NIs and hardware-terminated transport protocols. We target these architectures for two reasons. First, an important class of online services exhibits RPCs with service times that are frequently only a few µs long. For example,



(a) Exponential service time. **(b)** Model 1×16. **(c)** Model 16×1.

Figure 2. Tail latency as a function of throughput for different queuing systems and service time distributions. Y-axis values are shown as multiples of the mean service time $\bar{S}$.

the average service time for Memcached [2] is ~ 2µs [47]. Even software with functionality richer than simple data retrieval can exhibit µs-scale service times: the average TPC-C query service time on the Silo in-memory database [53] is only 33µs [47]. Software tiers with such short service times necessitate network architectures optimized for the lowest possible latency, using techniques such as kernel bypass and polling rather than receiving interrupts.

Second, unpredictable tail-inducing events for these short-lived RPCs often disrupt application execution for periods of time that are comparable to the RPCs themselves [6]. For example, the extra latency imposed by TLB misses or context switches spans from a few hundred ns to a few µs. At such fine granularities, any load-balancing policy implemented at the distal end of an I/O-attached NI is simply too far from the CPU cores to adjust its load dispatch decisions appropriately. Therefore, we argue that mitigating load imbalance at the µs level requires µs-optimized hardware.

The critical feature of our µs-optimized hardware is a fully integrated NI with direct access to the server's memory hierarchy, eliminating costly roundtrips over traditional I/O fabrics such as PCIe. Each server registers a part of its DRAM in advance with a particular *context* that is then exported to all participating servers, creating a partitioned global address space (PGAS) where every server can read/write remote memory in RDMA fashion. The architecture's programming model is a concrete instantiation of the Virtual Interface Architecture (VIA) [18], where each CPU core communicates with the NI through memory-mapped queue pairs (QPs). Each QP consists of a Work Queue (WQ) where the core writes entries (WQEs) to be processed by the NI, and a Completion Queue (CQ), where the NI writes entries (CQEs) to indicate that the cores' WQEs were completed. For more details, refer to the original VIA [18] and soNUMA [43] work.

3.2 NI Integration: The Key Enabler

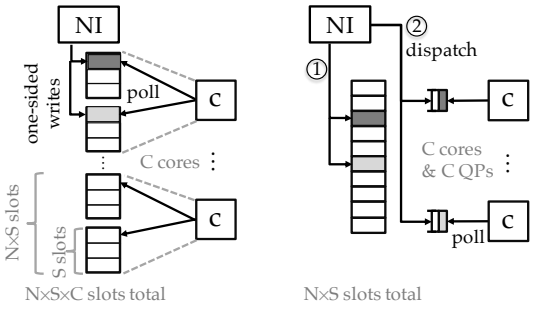
The NI's integration on the same piece of silicon as the CPU is the key enabler for handling µs-scale events. By leveraging the fact that such integration enables fine-grained real-time (nanosecond-scale) information to be passed back and forth between the NI and the server's CPU, the NI has the ability to respond to rapidly changing load levels and make *dynamic* load-balancing decisions. To illustrate the importance of ns-scale interactions, consider a data serving tier such as Redis [3], maintaining a sorted array in memory. Since the implementation of its sorted list container uses a skip list to provide add/remove operations in $O(log(N))$ time, an RPC to add a new entry may incur multiple TLB misses, stalling the core for a few µs while new translations are installed. While this core is stalled on the TLB miss(es), it is best to dispatch RPCs to other available cores on the server.

An integrated NI can, with proper hardware support, monitor each core's state and steer RPCs to the least loaded cores. Such monitoring is implausible without NI integration, as the latency of transferring load information over an I/O bus (e.g., ~ 1.5µs for a 3-hop posted PCIe transaction) would mean that the NI will make delayed—hence sub-optimal, or even wrong—decisions until the information arrives.

The active feedback of information from the server's compute units (which are not restricted to CPU cores) to the NI can take many forms, ranging from monitoring memory hierarchy events to metadata directly exposed by the application. Regardless of the exact policy, the underlying enabler for *RPCValet*'s ability to handle µs-scale load imbalance is that load dispatch decisions are driven by an integrated NI.

3.3 Design Principles

Our design goal is to break the tradeoff between the load imbalance inherent in multi-queue systems and the synchronization associated with pulling load from a single queue. To begin, we retain the VIA's design principle of allocating a single virtual interface (identical to a QP in IB/soNUMA terminology) to each participating thread, which is critically important for handling µs-scale RPCs. Registering independent QPs with the NI helps us achieve the goal of eliminating synchronization, as each thread polls on its own QP and waits for the arrival of new RPCs. This simplifies the load-balancing problem to simply choosing the correct QP to dispatch the RPC to. By allowing the NI to choose the QP at message arrival time, based on one of the many possible heuristics for estimating per-core load, our design achieves the goal of synchronization-free push-based load balancing.



(a) Emulated messaging. **(b)** *RPCValet.*

Figure 3. Emulated messaging versus *RPCValet*.

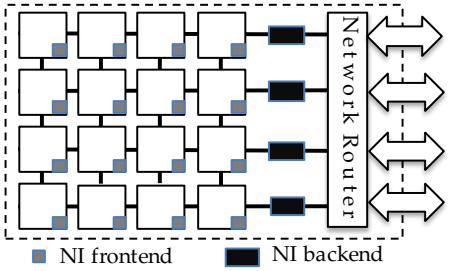


Figure 4. Manycore NI architecture.

Unfortunately, realizing such a design with our baseline architecture (§ 3.1) is not possible, as existing primitives are not expressive enough for push-based dispatch. In particular, architectures with on-chip NIs such as soNUMA [43] do not provide native support for *messaging* operations, favoring RDMA operations for hardware simplicity that facilitates NI integration. These RDMA operations (a.k.a. "one-sided" ops) enable direct read/write access of remote memory locations, without involving a CPU at the remote end. Hence, a reception of a one-sided op is not associated with a creation of a CPU notification event by the NI.

Messaging can be *emulated* on top of one-sided ops by allocating shared bounded buffers in the PGAS [17, 27, 43], into which threads directly place messages using one-sided writes. Fig. 3a illustrates the high-level operation of emulated messaging. As emulated messaging is performed in a connection-oriented fashion from thread to thread, each RPC-handling thread allocates N bounded buffers, each with S message slots; N is the number of nodes that can send messages. Each of the C cores polls at the head slots of its corresponding N buffers for incoming RPCs.

The fundamental drawback of such emulated messaging is that the sending thread implicitly determines which thread at the remote end will process its RPC request, because the memory location the RPC is written to is tied to a specific thread. The result is a multi-queue system, vulnerable to load imbalance. Although it may be possible to implement some form of load-aware messaging (e.g., per-thread client-server flow control), such mechanisms will have little to no benefit due to the relatively high network round-trip time for load information to diffuse between the two endpoints, especially when serving short-lived RPCs.

A key reason why, in the case of emulated messaging, the NI at the destination cannot affect the a priori assignment of an incoming RPC to a thread is that the protocol does not enable the NI to distinguish a "message" (i.e., a one-sided write triggering two-sided communication) from a default one-sided op. Protocol support for native messaging with innate semantics of two-sided operations overcomes this limitation and enables the NI at the message's destination node to perform push-based load balancing. Fig. 3b demonstrates *RPCValet*'s high-level operation. The NI first writes every incoming message into a *single* PGAS-resident message buffer of $N \times S$ slots, as in the case of emulated messaging. Then, the NI uses a selected core's QP to notify it to process the incoming RPC request. In effect, *RPCValet* decouples a message's arrival and location in memory from its assignment to a core for processing, thus achieving the best of both worlds: the load-balancing flexibility of a single-queue system, and the synchronization-free, zero-copy behavior of partitioned multi-queue architectures. Fig. 3b demonstrates how NI-driven dynamic dispatch decisions result in balanced load, in contrast to Fig. 3a's example.

In conclusion, *RPCValet* requires extensions to both the on-chip NI hardware and the networking protocol, to first provide support for native messaging and, second, realize dynamic load-balancing decisions. In the following section, we describe an implementation of an architecture featuring both of these mechanisms.

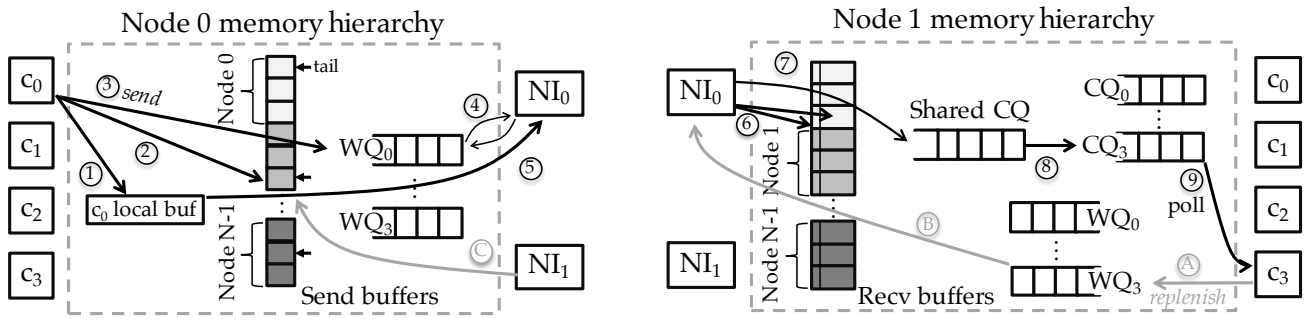
4 *RPCValet* Implementation

In this section, we describe our *RPCValet* implementation as an extension of the soNUMA architecture [43], including a lightweight extension of the baseline protocol for native messaging and support for NI-driven load balancing.

4.1 Scale-Out NUMA with Manycore NI

soNUMA enables rapid remote memory access through a lean hardware-terminated protocol and on-chip NI integration. soNUMA deploys a QP interface for CPU-NI interaction (§3.1) and leverages on-chip cache coherence to accelerate QP-entry transfers between the CPU and NI.

Fig. 4 shows soNUMA's scalable NI architecture for manycore CPUs [13]. The conventionally monolithic NI is split into two heterogeneous parts, a frontend and a backend. The frontend is the "control" component, and is collocated with each core to drastically accelerate QP interactions. The backend is replicated across the chip's edge, to scale the NI's capability with growing network bandwidth, and handles all data and network packets. Pairs of frontend and backend entities, which together logically comprise a complete



(a) Sender (source). **(b)** Receiver (destination).

Figure 5. Messaging illustration. Node 0 (sender) sends a message to node 1 (receiver).

NI, communicate with special packets over the chip's interconnect. Our *RPCValet* implementation relies on such a Manycore NI architecture.

4.2 Lightweight Native Messaging

We devise a lightweight implementation of native messaging as a required building block for dynamic load-balancing decisions at the NI. A key difficulty to overcome is support for multi-packet messages, that must be reassembled by the destination NI. This goal conflicts with soNUMA's stateless request-response protocol, which unrolls large requests into independent packets each carrying a single cache block payload. Emulated messaging (see §3.3) does not require any reassembly at the destination, because all packets are directly written to the bounded buffer specified by the sender.

One workaround to avoid message reassembly complications would be to limit the maximum message size to the link layer's MTU. Prior work has adopted this approach to build an RPC framework on an IB cluster [27]. Such a design choice may be an acceptable limitation for IB networks which have a relatively large MTU of 4KB. However, fully integrated solutions with on-chip NIs will likely feature small MTUs (e.g., a single cache line in soNUMA), so limiting the maximum message size to the link-layer MTU is impractical.

Our approach to avoiding the hardware overheads associated with message reassembly is keeping the buffer provisioning of the emulated messaging mechanism, which allows the sender to determine the memory location the message will be written to. Therefore, soNUMA's request-response protocol can still handle the message as a series of *independent* cache-block-sized writes to the requester-specified memory location. While this mechanism may seem identical to one-sided operations, we introduce a new pair of `send` and `replenish` operations which expose the semantics of multi-packet messages to the NI—it can then distinguish true one-sided operations from messaging operations, which are eligible for load balancing. The NI keeps track of packet receptions belonging to a `send`, deduces when it has been fully received, and then hands it off to a core for processing.

Fig. 5 shows the delivery of a message from Node 0 to Node 1 in steps. Completing the message delivery requires the execution of a `send` operation on Node 0 and a `replenish` operation on Node 1. Fig. 5 only shows NI backends; NI frontends are collocated with every core. We start with the required buffer provisioning associated with messaging.

Buffer provisioning. We introduce the notion of a *messaging domain*, which includes N nodes that can exchange messages and is defined by a pair of buffers allocated in each node's memory, the `send` buffer and the `receive` buffer. The `send` buffer comprises $N \times S$ slots, as described in §3.3.

Fig. 5a illustrates a `send` buffer with S=3 and different shades of gray distinguishing the `send` slots per participating node. Each `send` slot contains bookkeeping information for the local cores to keep track of their outstanding messages. It contains a *valid* bit, indicating whether the `send` slot is currently being used, a pointer to a buffer in local memory containing the message's payload, and a field indicating the size of the payload to be sent. A separate in-memory data structure maintains the head pointer for each of the N sets of `send` slots, which the cores use to atomically enqueue new `send` requests (not shown).

The `receive` buffer, shown in Fig. 5b, is the dual of the `send` buffer, where incoming `send` messages from remote nodes end up, and is sized similarly ($N \times S$ `receive` slots). Unlike `send` slots, `receive` slots are sized to accommodate message payloads. Each `receive` slot also contains a counter field, used to determine whether all of a message's packets have arrived. The counter field should provide enough bits to represent the number of cache blocks comprising the largest message; we overprovision by allocating a full cache block (64B), to avoid unaligned accesses for incoming payloads.

Overall, the messaging mechanism's memory footprint is $32 \times N \times S + (max\_msg\_size + 64) \times N \times S$ bytes. We expect that for current deployments, that number should not exceed a few tens of MBs. Systems adopting fully integrated solutions will likely be of contained scale (e.g., rack-scale systems), featuring a few hundred nodes, hence bounding the N parameter. In addition, most communication-intensive

latency-sensitive applications send small messages, bounding *max_msg_size*. For instance, the vast majority of objects in object stores like Memcached are <500B [5], while 90% of all packets sent within Facebook's datacenters are smaller than 1KB [49]. Finally, given the low network latency fully integrated solutions like soNUMA deliver, the number of concurrent outstanding requests S required to sustain peak throughput per node pair would be modest (a few tens). Dynamic buffer management mechanisms to reduce memory footprint are possible, but beyond the scope of this paper.

Importantly, a fixed *max_msg_size* does not preclude the exchange of larger messages altogether. A *rendezvous* mechanism [51] can be used, where the sending node's initial message specifies the location and size of the data, and the receiving node uses a one-sided read operation to directly pull the message's payload from the sending node's memory.

Send operation. Sending a message to a remote node involves the following steps. First, the core writes the message in a local core-private buffer (Fig. 5a, ①), updates the tail entry of the `send` buffer set corresponding to the target node (e.g., Node 1) ② and enqueues a `send` operation in its private WQ ③. The `send` operation specifies a messaging domain, the target node id, the remote `receive` buffer slot's address, a pointer to the local buffer containing the outgoing message, and the message's size. The target `receive` buffer slot's address can be trivially computed, as the number of nodes in the messaging domain, the number of `send`/`receive` slots per node, and the *max_msg_size* are all defined at the messaging domain's setup time. The NI polls on the WQ ④, parses the command, reads the message from the local memory buffer ⑤, and sends it to the destination node.

At the destination, the NI writes each `send` packet directly in the local memory hierarchy, into the specified `receive` slot, and increments that `receive` slot's counter (Fig. 5b, ⑥). When the counter matches the `send` operation's total packet count (contained in each packet's header), the NI writes a message arrival notification entry in a *shared CQ* ⑦. The shared CQ is a memory-mapped and cacheable FIFO where the NI enqueues pointers to received `send` requests. When it is time for a dispatch decision, the NI selects a core and assigns the head entry of the shared CQ to it by writing the `receive` slot's index, contained in the shared CQ entry, into that core's corresponding CQ ⑧. This is a crucial step that enables *RPCValet*'s NI-driven dynamic load balancing, which we expand in §4.3. Finally, the core receives the new `send` request ⑨ polling the head of its private CQ, then directly reads the message from the `receive` buffer and processes it.

Replenish operation. A `replenish` operation always follows the receipt of a `send` operation as a form of end-to-end flow control: a `replenish` notifies the `send` operation's source node that the request has been processed and hence its corresponding `send` buffer slot is free and can be reused. In Fig. 5b's example, when core 3 is done processing the `send` request, it enqueues a `replenish` in its private WQ Ⓐ. The `replenish` only contains the target node and the target `send` buffer slot's address, trivially deduced from the `receive` buffer index the corresponding `send` was retrieved from. The NI, which is polling at the head of core 3's WQ, reads the new `replenish` request Ⓑ and sends the message to node 0. When the `replenish` message arrives at node 0, the NI invalidates the corresponding `send` buffer slot by resetting its *valid* field (Fig. 5a, Ⓒ), indicating its availability to be reused. In practice, a `replenish` operation is syntactic sugar for a special remote write operation, which resets the *valid* field of a `send` buffer slot.

4.3 NI-driven Dynamic Load Balancing

With the NI's newly added ability to recognize and manage message arrivals, we now proceed to introduce NI-driven dynamic load balancing. Load-balancing policies implemented by the NIs can be sophisticated and can take various affinities and parameters into account (e.g., certain types of RPCs serviced by specific cores, or data-locality awareness). Implementations can range from simple hardwired logic to microcoded state machines. However, we opt to keep a simple proof-of-concept design, to illustrate the feasibility and effectiveness of load-balancing decisions at the NIs and demonstrate that we can achieve the load-balancing quality of a single-queue system without synchronization overheads.

Fig. 5b's step ⑧ is the crucial step that determines the balancing of incoming requests to cores. In *RPCValet*, the receiving node's NI keeps track of the number of outstanding `send` requests assigned to each core. Receiving a `replenish` operation from a core implies that the core is done processing a previously assigned `send`. Allowing only one outstanding request per core and dispatching a new request *only after* receiving a notification of the previous one's completion corresponds to true single-queue system behavior, but leaves a small execution bubble at the core. The bubble can be eliminated by setting the number of outstanding requests per core to two. We found that introducing a small multi-queue effect is offset by eliminating the bubble, resulting in marginal performance gains for ultra-fast RPCs with service times of a few 100s of nanoseconds.

A challenge that emerges from the distributed nature of a Manycore NI architecture is that the otherwise independent NI backends, each of which is handling `send` message arrivals from the network, need to coordinate to balance incoming load across cores. Our proposed solution is simple, yet effective: centralize the last step of message reception and dispatch. One of the NI backends—henceforth referred to as the *NI dispatcher*—is statically assigned to handle message dispatch to all the available cores. Network packet and data handling still benefit from the parallelism offered by the Manycore NI architecture, as all NI backends still independently handle incoming network packets and access memory

directly. However, once an NI backend writes all packets comprising a message in their corresponding receive buffer slots, it creates a special message completion packet and forwards it to the NI dispatcher over the on-chip interconnect. Once the NI dispatcher receives the message completion packet, it enqueues the information in the shared CQ, from which it dispatches messages to cores in FIFO order as soon as it receives a `replenish` operation. As all the incoming messages are dispatched from a single queue to all available cores, *RPCValet* behaves like a true single-queue queuing system.

Having a single NI dispatcher eschews software synchronization, but raises scalability questions. However, for modern server processor core counts, the required dispatch throughput should be easily sustainable by a single centralized hardware unit, while the additional latency due to the indirection from any NI backend to the NI dispatcher is negligible. From the throughput perspective, even an RPC service time as low as 500ns corresponds to a new dispatch decision every ~31/8ns for a 16/64-core chip, respectively. Both dispatch frequencies are modest enough for a single hardware dispatch component to handle, especially for our simple greedy dispatch implementation. The same observation also holds for more sophisticated dispatch policies if their hardware implementation can be pipelined. Latency-wise, the indirection from any NI backend to the NI dispatcher costs a couple of on-chip interconnect hops, adding just a few ns to the end-to-end message delivery latency. In case of exotic system deployments where the above assumptions do not hold, an intermediary design point is possible where each NI backend can dispatch to a limited subset of cores on the chip. As an example of this design point, we also implement and evaluate a 4×4 queuing system in §6.

4.4 soNUMA Extensions for *RPCValet*

We now briefly summarize the modifications to soNUMA's hardware to enable *RPCValet*, including the necessary protocol extensions for messaging and load balancing. Load balancing itself is transparent to the protocol and only affects a pipeline stage in the NI backends.

Additional hardware state. Most of the state required for messaging (i.e., `send/receive` buffers) is allocated in host memory. The only metadata kept in dedicated SRAM are the `send` and `receive` buffers' location and size, as they require constant fast access. On each node, the maintained state per registered soNUMA *context* includes a memory address range per node and a QP per local core. In total, we add 20B of stored state per context, including: the base virtual addresses for the `send/receive` buffers, the maximum message size (*max_msg_size*), the # of nodes (N) in the messaging domain, and the # of messaging slots (S) per node.

Hardware logic extensions. soNUMA's NI features three distinct pipelines for handling Request Generations, Request Completions, and Remote Request Processing, respectively [43]. We extend these pipelines to support the new messaging primitives and load-balancing functionality. Receiving a new `send` or `replenish` request is very similar to the reception of a remote write operation in the original soNUMA design. To support our native messaging design, we add a field containing the total message size to the network layer header; this is necessary so the NI hardware can identify when all of a message's packets have been received.

We add five new stages to the NI pipelines in total. A new stage in Request Generation differentiates between `send` and `replenish` operations, and operates on the messaging domain metadata. All other modifications are limited to the Remote Request Processing Pipeline, which is only replicated across NI backends. When a `send` is received, the pipeline performs a fetch-and-increment operation to the corresponding *counter* field of the target `receive` buffer slot (§4.2, "Send operation"). The next stage checks if the counter's new value matches the message's length, carried in each packet header. If all of the `send` operation's packets have arrived, the next stage enqueues a pointer to the corresponding `receive` buffer slot in the shared CQ.

The final stage added to the Remote Request Processing pipeline, *Dispatch,* keeps track of the number of outstanding requests assigned to each core and determines when and to which core to dispatch `send` requests to from the shared CQ. A core is "available" when its number of outstanding requests is below the threshold defined; in our implementation, this number is two. Whenever there is an available core, the *Dispatch* stage dequeues the shared CQ's first entry and sends it to the target core's NI frontend, where the Request Completion pipeline writes it into the core's private CQ. The complexity of the *Dispatch* stage is very simple for our greedy algorithm, but varies based on the logic and algorithm involved in making load-balancing decisions. Finally, after completing the request, the core signals its availability by enqueuing a `replenish` operation in its WQ, which is propagated by the core's NI frontend to the NI backend that originally dispatched the request.

In summary, the additional hardware complexity is modest, thus compatible with architectures featuring ultra-lightweight protocols and on-chip integrated NIs, such as soNUMA. Given the on-chip NI's fast access to its local memory hierarchy, it is possible to virtualize most of the bulky state required for the messaging mechanism's `send` and `receive` buffers in the host's memory. The dedicated hardware requirements are limited to a small increase in SRAM capacity, while the NI logic extensions are contained and straightforward.

5 Methodology

We now detail our methodology for evaluating *RPCValet*'s effectiveness in balancing load transparently in hardware.

System organization. We model a single tiled 16-core chip implementing soNUMA with a Manycore NI, as illustrated

Cores	ARM Cortex-A57-like; 64-bit, 2GHz, OoO 3-wide dispatch/retirement, 128-entry ROB, TSO
L1 Caches	32KB 2-way L1d, 48KB 3-way L1i, 64-byte blocks 2 ports, 32 MSHRs, 3-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 2MB total 16-way, 1 bank/tile, 6-cycle latency
Coherence	Directory-based Non-Inclusive MESI
Memory	50ns latency, 4×25.6GBps (DDR4)
Interconnect	2D mesh, 16B links, 3 cycles/hop

Table 1. Flexus simulation parameters.

in Fig. 4. The modeled chip is part of a 200-node cluster, with remote nodes emulated by a traffic generator which creates synthetic `send` requests following Poisson arrival rates, from randomly selected nodes of the cluster. The traffic generator also generates synthetic replies to the modeled chip's outgoing requests. We use Flexus [54] cycle-accurate simulation with Table 1's parameters.

Microbenchmark. We use a multithreaded microbenchmark that emulates different service time distributions, where each thread executes the following actions in a loop: (i) spins on its CQ, until a new `send` request arrives; (ii) emulates the execution of an RPC by spending *processing time X*, where *X* follows a given distribution as detailed below; (iii) generates a synthetic RPC reply, which is sent back to the requester using a `send` operation with a 512B payload; and (iv) issues a `replenish` corresponding to the processed `send` request, marking the end of of the incoming RPC's processing. The overall *service time* for an emulated RPC (i.e., the total time a core is occupied) is the sum of steps (ii) to (iv).

RPC processing time distributions. To evaluate *RPCValet* on a range of RPC profiles, we utilize processing time distributions generated with three different methods. First, we develop an RPC processing time generator that samples values from a selected distribution. We experiment with four different distributions: fixed, uniform, exponential, and GEV. Fixed represents the ideal case, where all requests take the same processing time. GEV represents a more challenging case with infrequent long tails, which may arise from events like page faults or interrupts. Uniform and exponential distributions fall between fixed and GEV in terms of impact on load balancing, as established in Fig. 2. For our synthetic processing time distributions, we use 300ns as a base latency and add an extra 300ns *on average*, following one of the four distributions. The parameters we use for GEV are (location, scale, shape) = (363, 100, 0.65), which result in a mean of 600 cycles (i.e., 300ns at 2GHz) [1]. Fig. 6a illustrates the PDFs of the four resulting processing time distributions.

Second, we run the HERD [27] key-value store and collect the distribution of the RPCs' processing times. We use a dual-socket Xeon E5-2680 Haswell server and pin 12 threads on an equal number of a single socket's physical cores. The second socket's cores generate load. Our parameters for HERD are:

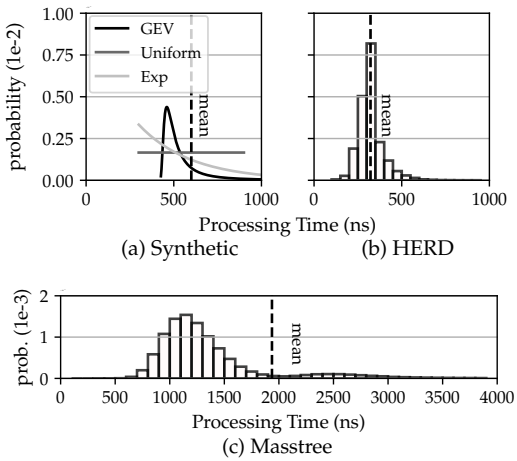


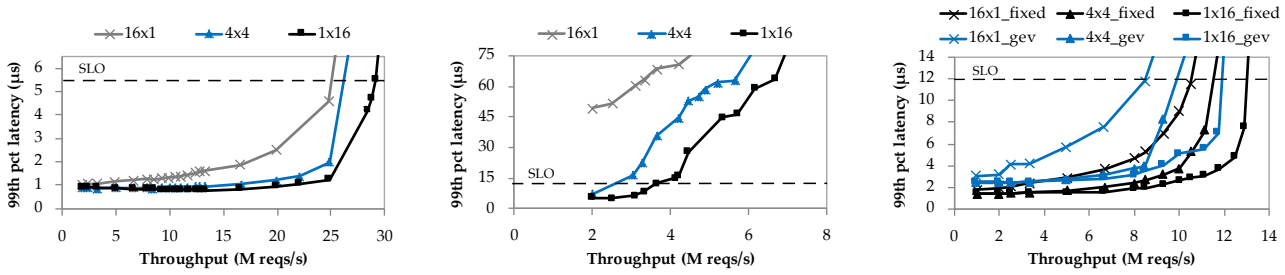
Figure 6. Modeled RPC processing time distributions.

95/5% read/write query mix, uniform key popularity, and a 4GB dataset (256MB per thread). Fig. 6b displays a histogram of HERD's RPC processing times *after* the request has exited the network, which have a mean of 330ns.

Finally, we evaluate the Masstree data store [40], which stores key-value pairs in a trie-like structure and supports ordered scans in addition to `put/get` operations. Ordered scans are common in database/analytics applications and compete with latency-critical operations for CPU time when accessing the same data store. To collect RPC processing times, we use the same platform and dataset we used for HERD and load the server with 99% single-key `gets`, interleaved with 1% long-running `scans` which return 100 consecutive keys. The resulting distribution for `gets` is shown in Figure 6c and has an average of 1.25 µs. The runtime of `scans` is 60–120 µs (not shown in Fig. 6c due to the X-axis bounds).

Load-balancing implementations. We first compare the performance of two *RPCValet* variants, 1×16 and the less flexible 4×4. In 4×4, each NI backend is limited to balancing load across the four cores corresponding to its on-chip network row. We also consider a 16×1 system, representing partitioned dataplanes where every incoming message is assigned to a core at arrival time without any rebalancing. 16×1 is the only currently existing NI-driven load distribution mechanism. Next, we compare the best-performing hardware load-balancing implementation, 1×16, to a software-based counterpart. In our software implementation, NIs enqueue incoming *send* requests into a single CQ from which all 16 threads pull requests in FIFO order. We use an MCS queue-based lock [41] for the shared request queue.

We assume a 99th percentile Service Level Objective (SLO) of $\leqslant 10\times$ the mean service time $\bar{S}$ we measure in each experiment and evaluate all configurations in terms of throughput under SLO. We measure each request's latency as the time from the reception of a `send` message until the thread that services the request posts a `replenish` operation.



(a) HERD. **(b)** Masstree. **(c)** Synthetic distributions.

Figure 7. Load balancing with three different hardware queuing implementations.

6 Evaluation

6.1 Load Balancing: Hardware Queuing Systems

Fig. 7a shows the performance of HERD with each of the three evaluated NI-driven load-balancing configurations. With a resulting $\bar{S}$ of ~ 550ns, 1×16 delivers 29MRPS, 1.16× and 1.18× higher throughput than 4×4 and 16×1, respectively. 1×16 consistently delivers the best performance, thanks to its superior flexibility in dynamically balancing load across all 16 available cores. In comparison, 4×4 offers limited flexibility, while 16×1 offers none at all. The flexibility to balance load from a single queue to multiple cores not only results in higher peak throughput under SLO, but also up to 4× lower tail latency before reaching saturation load. Conversely, lower tail means that the throughput gap between *RPCValet* and 1×16 would be larger for SLOs stricter than the assumed $10 \times \bar{S}$. Note that data points appearing slightly lower at mid load as compared to low load in Fig. 7a is a measurement artifact: for low arrival rates, the relatively small number of completed requests during our simulation's duration results in reduced tail calculation accuracy.

Fig. 7b shows the tail latency of Masstree's `get` operations with each queuing configuration. We set the SLO for Masstree at 10× the service time of the `get` operations, equalling 12.5µs; we do not consider the `scan` operations latency critical. Due to interference from the `scans`, 16×1 cannot meet the SLO even for the lowest arrival rate of 2MRPS, while even 4×4 quickly violates the SLO at 3MRPS. 1×16 delivers 4.1MRPS at SLO, outperforming 4×4 by 37%. Under a more relaxed SLO of 75µs, *RPCValet*'s 1×16 configuration delivers 54% higher throughput than 16×1 and 20% higher than 4×4. In the presence of long-running `scans` that occupy cores for many µs, *RPCValet* leverages occupancy feedback from the cores to eliminate excess queuing of latency-critical `gets` and improve throughput under SLO.

Fig. 7c shows the results for two of our synthetic service time distributions, fixed and GEV. The results for uniform and exponential distributions fall between these two, are omitted for brevity, and are available in [12]. The results follow the expectations set in §2.2. For the fixed distribution, 1×16 delivers 1.13× and 1.2× higher throughput than 4×4 and 16×1 under SLO, respectively. For GEV, the throughput improvement grows to 1.17× and 1.4×, respectively. Similar to HERD results, in addition to throughput gains, *RPCValet* also delivers up to 4× lower tail latency before saturation.

In all of Fig. 7's experiments we set *RPCValet*'s number of outstanding requests per core to two (see §4.3). Reducing this to one marginally degrades HERD's throughput, because of its short sub-µs service times, but has no measurable performance difference in the rest of our experiments.

In conclusion, *RPCValet* significantly improves system throughput under tight tail latency goals. Implementations that enable request dispatch to all available cores (i.e., 1×16) deliver the best performance. However, even implementations with limited balancing flexibility, such as 4×4, are competitive. As realizing a true single-queue system incurs additional design complexity, such limited-flexibility alternatives introduce viable options for system designers willing to sacrifice some performance in favor of simplicity.

6.2 Hardware Versus Software Load Balancing

Fig. 8 compares the performance of *RPCValet* to a software implementation, both of which implement the same theoretically optimal queuing system (i.e., 1×16). The difference between the two is how load is dispatched to a core. Software requires a synchronization primitive (in our case, an MCS lock) for cores to atomically pull incoming requests from the queue. In contrast, *RPCValet* does not incur any synchronization costs, as dispatch is driven by the NI.

The software implementation is competitive with the hardware implementation at low load, but because of contention on the single lock, it saturates significantly faster. As a result, our hardware implementation delivers 2.3–2.7× higher throughput under SLO, depending on the request processing time distribution. A comparison between Fig. 7b and 8 reveals that the 1×16 software implementation is not only inferior to the 1×16 hardware implementation, but to all of the evaluated hardware implementations. The fact that even the 16×1 hardware implementation is superior to the software 1×16 implementation indicates that the software synchronization costs outweigh the dispatch flexibility they provide,

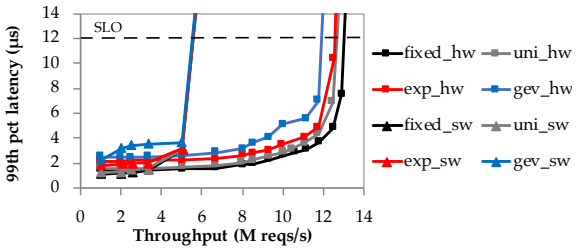


Figure 8. 1 × 16 load balancing: hardware vs. software.

a direct consequence of the µs-scale RPCs we focus on. In addition, we corroborate the findings of prior work on dataplanes [7, 47]—which effectively build a 16 × 1 system using RSS—showing that elimination of software synchronization from the critical path offsets the resulting load imbalance.

6.3 Comparison to Queuing Model

Our results in §6.1 qualitatively meet the expectations set by the queuing analysis presented in §2.2. We now quantitatively compare the obtained results to the ones expected from purely theoretical models, to determine the performance gap between *RPCValet* and the theoretical 1 × 16 system.

To make *RPCValet* measurements comparable to the theoretical queuing results, we devise the following methodology. We measure the mean service time $\bar{S}$ on our implementation; a part D of this service time is synthetically generated to follow one of the distributions in §5, and the rest, $\bar{S} - D$, is spent on the rest of the microbenchmark's code (e.g., event loop, executing `send` for the RPC response and `replenish` to free the RPC slot). We conservatively assume that this $\bar{S} - D$ part of the service time follows a fixed distribution. Using discrete-event simulation, we model and evaluate the performance of theoretical queuing systems with a service time $\bar{S}$, where $\frac{D}{\bar{S}}$ of the service time follows a certain distribution (fixed, uniform, exponential, GEV) and $\frac{\bar{S}-D}{\bar{S}}$ of the service time is fixed.

Fig. 9 compares *RPCValet* to the theoretical 1 × 16. The graphs show the 99th percentile latency as a function of offered load, with four different distributions for the D part of the service time. *RPCValet* performs as close as 3% to 1 × 16, and within 15% in the worst case (GEV). We attribute the gap between the implementation and the model to contention that emerges under high load in the implemented systems, which is not captured by the model. Furthermore, assuming a fixed service time distribution for the $\bar{S} - D$ part of the service time is a conservative simplifying assumption: modeling variable latency for this component would have a detrimental effect on the performance predicted by the model, thus shrinking the gap between the model and the implementation. In conclusion, *RPCValet* leaves no significant room for improvement; neither centralizing dispatch nor maintaining private request queues per core introduces performance concerns.

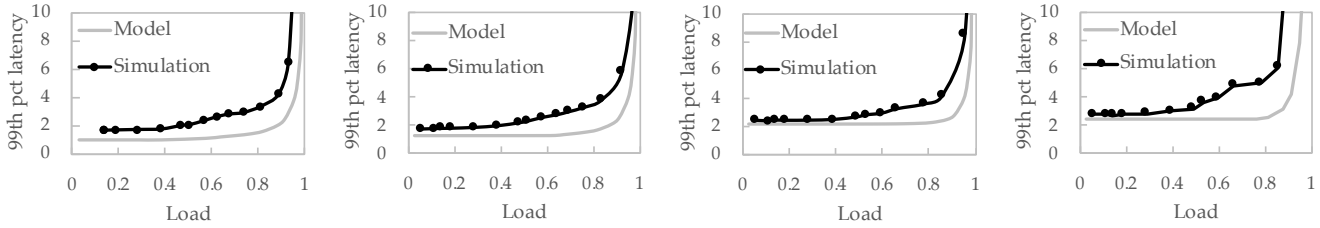
7 Related Work

Other Techniques to Reduce Tail Latency. Prior work aiming to control the tail latency of Web services deployed at datacenter scale introduced techniques that duplicate/hedge requests across multiple servers hosting replicated data [15]. The goal of such replication is to shrink the probability of an RPC experiencing a long-latency event and consequently affecting the response latency of its originating request. A natural side-effect of replication is the execution of more requests than strictly necessary, also necessitating extra server-side logic to reduce the load added by duplicated requests. As compared to ms-scale applications where the network RTT is a negligible latency contributor, applying the same technique for µs-scale applications requires a more aggressive duplication of requests, further increasing the generation of unnecessary server load. In contrast to such *client-side* techniques, *RPCValet*'s *server-side* operation offers an alternative that does not increase the global server load.

RPCValet improves tail latency by minimizing the effect of queuing. Queuing is only one of many sources of tail latency, which lie in all layers of the server's software stack. Therefore, no single solution can wholly address the tail challenge; a synergy of many techniques is necessary, each targeting specific issues in particular layers (e.g., IX [7] targets protocol and interrupt processing). However, despite the complex nature of the problem, managing on-server queuing is a *universal* approach that helps mitigate *all* sources of tail latency. Our work does not prevent straggler RPCs, but eliminates the chance that such stragglers will cascadingly impact the latency of other queued RPCs by providing a true single-queue system on each RPC-handling server. *RPCValet* is synergistic with techniques on both clients and servers to address specific sources of tail latency in the workflow of serving RPCs.

A range of prior work also leverages queuing insights to balance web requests within a datacenter, by mainly focusing on algorithmic aspects of load distribution among backend servers rather than a single server's cores. Examples of such algorithms are Join-Shortest-Queue [22], Power-of-d [9], and Join-Idle-Queue [39]. Pegasus [34] is a rack-scale solution where the ToR switch applies load-aware request scheduling by either estimating per-server load, or by leveraging load statistics reported directly by the servers. In the context of balancing µs-scale RPCs among a single server's cores, challenges such as dispatcher-to-core latency are of minor importance, because of the integrated NI's proximity. Our results show that single-queue behavior is feasible by deferring dispatch until a core is free, which is unattainable at cluster scale due to the latency of the off-chip network.

Load Distribution Frameworks. Most modern NICs distribute load between multiple hardware queues, which can be privately assigned to cores, through Receive Side Scaling (RSS) [42] or Flow Director [24]. Systems like IX [7] and



(a) 1 × 16 – fixed. **(b)** 1 × 16 – uniform. **(c)** 1 × 16 – exponential. **(d)** 1 × 16 – GEV.

Figure 9. *RPCValet* comparison to theoretical 1 × 16 queuing model. Y-axis values shown as multiples of the avg service time $\bar{S}$.

MICA [35] leverage these mechanisms to significantly boost their throughput under tail latency constraints. The disadvantage of RSS/Flow Director is that they blindly spread load across multiple receive queues based on specific network packet header fields, and are oblivious to load imbalance.

ZygOS [47] ameliorates the shortcomings of partitioned dataplanes, which suffer from increased tail latency under load imbalance. ZygOS introduces an intermediate shuffling layer where idle CPU threads can perform *work stealing* from other input queues. Due to the added synchronization overhead of work stealing, there is a measurable performance gap between ZygOS and the best single-queue system, inversely proportional to the RPC service times. *RPCValet* achieves the best of both worlds, offering single-queue performance without synchronization; instead of adding layers to *rebalance* load, we co-design hardware and software to implement a single-queue system.

The Shinjuku operating system [25] improves throughput under SLO by preempting long-running RPCs instead of running every RPC to completion. Their approach is particularly effective for workloads with extreme service time variability and CPUs with limited core count. Shinjuku preempts requests every 5–15µs, which is higher than the vast majority of our evaluated RPC runtimes. A system combining Shinjuku and *RPCValet* would rigorously handle RPCs of a broad runtime range, from hundreds of ns to hundreds of µs.

Programmable Network Interfaces. Offloading compute to programmable network processors is an old idea that has seen rekindled interest; FLASH [32] and Typhoon [48] integrated general-purpose processors with the NI, enabling custom handler execution upon message reception. NI-controlled message dispatch to cores has been proposed in the context of parallel protocol handler execution for DSMs to eschew software synchronization overheads [19, 46]. Programming abstractions such as PDQ [19] could be deployed as load-balancing decisions in *RPCValet*'s NI dispatch pipeline.

Today's commercial "SmartNICs" target protocol processing or high-level application acceleration, with the goal of reducing CPU load; they integrate either CPU cores (e.g., Mellanox's BlueField [37]), or FPGAs (e.g., Microsoft's Catapult [11, 33]). FlexNIC [30, 31] draws inspiration from SDN switches [8], deploying a match-action pipeline for line-rate header processing. The programmable logic in these SmartNICs could be leveraged to implement non-static load balancing, adding flexibility to RSS or Flow Director. However, our dynamic load balancing scheme relies on ns-scale interaction between the NI and CPU logic, which is only attainable through tight NI integration and CPU-NI co-design.

RPC Layers on InfiniBand NICs. Latency-critical software systems for key-value storage [27, 28], distributed transaction processing [17, 28], distributed durable data storage [44], and generalized datacenter RPCs [52], have already begun using RDMA NICs due to their low latency and high IOPS. All of these systems are fine-tuned to maximize RPC throughput given the underlying limitations of their discrete NICs and the IB verbs specification. We distinguish *RPCValet* from these software-only systems by our focus on balancing the load of incoming RPCs across the CPU cores. Furthermore, all of the above proposals are adversely affected by the shortcomings of PCIe-attached NICs, and use specific optimizations to ameliorate their inherent latency bottlenecks; this strengthens our insight that NI integration is the key enabler for handling RPCs in true single-queue fashion.

8 Conclusion

We introduced *RPCValet*, an NI-driven dynamic load-balancing mechanism for µs-scale RPCs. *RPCValet* behaves like a single-queue system, without incurring the synchronization overheads typically associated with single-queue implementations. *RPCValet* performs within 3–15% of the ideal single-queue system and significantly outperforms current RPC load-balancing approaches.

Acknowledgements

We thank Edouard Bugnion, James Larus, Dmitrii Ustiugov, Virendra Marathe, Dionisios Pnevmatikatos, Mario Drumond, Arash Pourhabibi, Marios Kogias and the anonymous reviewers for their precious feedback. This work was partially funded by Huawei Technologies, the Nano-Tera *YINS* project, the Oracle Labs *Accelarating Distributed Systems with Advanced One-Sided Operations* grant, and the SNSF's *Memory-Centric Server Architecture for Datacenters* project.

References

[1] Generalized Extreme Value distribution. https://www.wolframalpha.com/input/?i=MaxStableDistribution%5B363,100,0.65%5D.

[2] Memcached. http://memcached.org/.

[3] Redis. https://redis.io/.

[4] Krste Asanović. A Hardware Building Block for 2020 Warehouse-Scale Computers. *USENIX FAST Keynote*, 2014.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

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

[7] 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.

[8] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Computer Communication Review*, 44(3):87–95, 2014.

[9] Maury Bramson, Yi Lu, and Balaji Prabhakar. Randomized load balancing with general service time distributions. In *Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 275–286, 2010.

[10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[11] 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.

[12] Alexandros Daglis. Network-Compute Co-Design for Distributed In-Memory Computing . *EPFL PhD Thesis*, September 2018.

[13] Alexandros Daglis, Stanko Novakovic, Edouard Bugnion, Babak Falsafi, and Boris Grot. Manycore network interfaces for in-memory rack-scale computing. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 567–579, 2015.

[14] Alexandros Daglis, Dmitrii Ustiugov, Stanko Novakovic, Edouard Bugnion, Babak Falsafi, and Boris Grot. SABRes: Atomic object reads for in-memory rack-scale computing. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 6:1–6:13, 2016.

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

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[17] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[18] Dave Dunning, Greg J. Regnier, Gary L. McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, 1998.

[19] Babak Falsafi and David A. Wood. Parallel Dispatch Queue: A Queue-Based Programming Abstraction to Parallelize Fine-Grain Communication Protocols. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 182–192, 1999.

[20] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 249–264, 2016.

[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 202–215, 2016.

[22] Varun Gupta, Mor Harchol-Balter, Karl Sigman, and Ward Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Perform. Eval.*, 64(9-12):1062–1081, 2007.

[23] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito IV, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. SVE: Distributed Video Processing at Facebook Scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–103, 2017.

[24] Intel Corp. Introduction to Intel Ethernet Flow Director and Memcached Performance. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf.

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

[26] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the 2018 EuroSys Conference*, pages 33:1–33:15, 2018.

[27] 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.

[28] 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.

[29] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 158–169, 2015.

[30] 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.

[31] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas E. Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXI)*, pages 67–81, 2016.

[32] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 302–313, 1994.

[33] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 137–152, 2017.

[34] Jialin Li, Jacob Nelson, Xin Jin, and Dan R. K. Ports. Pegasus: Load-Aware Selective Replication with an In-Network Coherence Directory. *UW CSE Technical Report*, December 2018.

[35] 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.

[36] Linley Group. Epyc Relaunches AMD Into Servers. *Microprocessor Report*, June 2017.

[37] Linley Group. Mellanox Accelerates BlueField SoC. *Microprocessor Report*, August 2017.

[38] Linley Group. X-Gene 3 Up and Running. *Microprocessor Report*, March 2017.

[39] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert G. Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Eval.*, 68(11):1056–1071, 2011.

[40] 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.

[41] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

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

[43] 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.

[44] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[45] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[46] Ilanthiraiyan Pragaspathy and Babak Falsafi. Address Partitioning in DSM Clusters with Parallel Coherence Controllers. In *Proceedings of the 9th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 47–56, 2000.

[47] 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.

[48] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 325–336, 1994.

[49] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 123–137, 2015.

[50] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's Time for Low Latency. In *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011.

[51] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 32–39, 2006.

[52] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 306–324, 2017.

[53] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, 2013.

[54] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26(4):18–31, 2006.