ALTOCUMULUS: Scalable Scheduling for Nanosecond-Scale Remote Procedure Calls

Jiechen Zhao, Iris Uwizeyimana, Karthik Ganesan, Mark C. Jeffrey, Natalie Enright Jerger

Department of Electrical and Computer Engineering

University of Toronto

Toronto, Canada

{*jc.zhao, iris.uwizeyimana, karthik.ganesan*}@*mail.utoronto.ca*, {*mcj,enright*}@*ece.utoronto.ca*

Abstract—**Online services in modern datacenters use Remote Procedure Calls (RPCs) to communicate between different software layers. Despite RPCs using just a few small functions, inefficient RPC handling can cause delays to propagate across the system and degrade end-to-end performance. Prior work has reduced RPC processing time to less than 1 μs, which now shifts the bottleneck to the *scheduling* of RPCs. Existing RPC schedulers suffer from either high overheads, inability to effectively utilize high core-count CPUs or do not adaptively fit different traffic patterns. To address these shortcomings, we present ALTOCUMULUS,[1] a scalable, software-hardware co-design to schedule RPCs at nanosecond scales. ALTOCUMULUS provides a proactive scheduling scheme and low-overhead messaging mechanism on top of a decentralized user runtime. ALTOCUMULUS also offers direct access from the user space to a set of simple hardware primitives to quickly migrate long-latency RPCs. We evaluate ALTOCUMULUS with synthetic workloads and an end-to-end in-memory key-value store application under real-world traffic patterns. ALTOCUMULUS improves throughput by 1.3-24.6$\times$ under a 99th percentile latency <300 μs and reduces tail latency by up to 15.8$\times$ on 16-core systems over current state-of-the-art software and hardware schedulers. For 256-core systems, integrating ALTOCUMULUS with either a hardware-optimized NIC or commodity PCIe NIC can improve throughput by 2.8$\times$ or 2.7$\times$, respectively, under 99th percentile latency <8.5 μs.**

Keywords-**Remote procedure calls, Scheduling, Datacenters, Networked systems, Load balancing, Migration, Queuing theory**

I. INTRODUCTION

Distributed online services have adopted a multi-tiered software architecture running on thousands of datacenter machines. Communication between tiers uses a common API, Remote Procedure Calls (RPCs), which allows each system to call functions or access data on another system as though they were local. RPCs enable a high degree of flexibility and programmer productivity as they can call functions running on different operating systems and software stacks. RPCs have now become so ubiquitous that significant CPU time is spent handling RPCs in modern datacenters. Recent studies from Google and Meta show that RPC software accounts for 6-12% of their total CPU cycles [29], [60].

---

[1]**Auto**matic **C**oncurrent **M**igration **L**oad-balancing **S**trategy (Auto-CuMuLuS), homophonic with "altocumulus" as a type of clouds in meteorology, fragmented to separate patches or nodes.
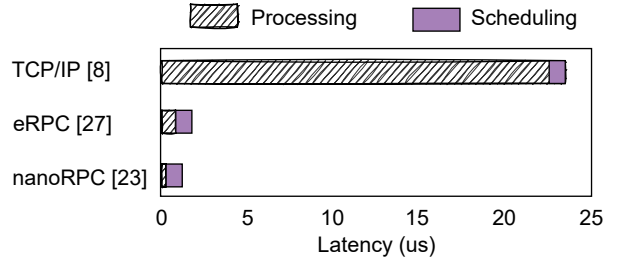

Figure 1: On-CPU latency for different RPC stacks.

Along with their increased use, RPC messages are also getting smaller. For example, in DeathStarBench [19], 75% of RPC requests are smaller than 512B while more than 90% of RPC responses are smaller than 64B [36]. Therefore, the efficiency of handling small RPCs becomes the performance determinant. However, on-CPU[2] RPC latency suffers due to high per-packet overheads for small messages [27], [45]. For instance, distributed applications spend up to 90% of the on-CPU time executing the RPC stack, as opposed to the application's business logic [19], [52].

A plethora of research has proposed ways to improve RPC stack processing efficiency by tackling one or more components of the stack, both in software [1], [8], [15], [26], [27], [48], [50], [53] and hardware [3], [5], [11], [20], [23], [31], [36], [51], [52], [61], [65]. Fig. 1 illustrates this impact, showing the time spent on a server handling a 300B RPC message. We distinguish time spent in RPC stack processing (extracting the RPC request from the network packet) vs. RPC scheduling (mapping the RPC request handler to some core). Prior work successfully reduces RPC stack processing latency from 10s of μs to sub-1 μs. Fig. 1 shows the standard TCP/IP protocol spending significant time processing network packets, while recent work, such as eRPC [27] and nanoRPC [23], implement more optimized network protocols to reduce this time significantly. In this paper, we address the new RPC system bottleneck which has shifted from processing to scheduling.

At the same time, datacenter networks are getting faster,

---

[2]We focus on the latency of processing an RPC from the time it arrives at the CPU to the time the response leaves the CPU.

going from 100 Gigabit Ethernet (GbE) a few years ago [16] to 400 GbE now [4], with 1 TbE on the horizon [2]. With higher network speeds, CPUs must service more RPCs while maintaining processing latency. One approach is to use CPUs with more cores. However, systems usually have to sacrifice CPU utilization at even moderate loads to guarantee low latency. For example, 36-64% of cycles of a 8-12 core CPU are wasted when handling small RPC messages at μs-scale latency [17], [54]. Underutilizing CPUs avoids unpredictable queuing delays and costly scheduling operations that can lead to increased RPC processing time. However, modern servers with 64+ cores demand a more robust approach to support high network bandwidth and guarantee sub-1μs RPC latency with high CPU utilization.

Prior work proposes various RPC scheduling designs and implementations in both software [8], [17], [26], [48], [53] and hardware [11], [23], [61]. However, these approaches do not meet one or more of the following requirements for RPC scheduling:

1) **Performance**: Small RPCs require nanosecond-scale scheduling. Software scheduling overhead increases a state-of-the-art RPC stack's latency by up to 25$\times$ (Sec. II-C).
2) **Scalability**: Efficient utilization of high core-count CPUs is crucial for both performance and cost-efficiency (Fig. 13).
3) **Adaptability**: Hardware-based schedulers cannot adapt to various loads and arrival patterns. For example, one hardware scheduler suffers up to 15.8$\times$ end-to-end latency increase for a highly-varied service time pattern (Fig. 10).

Based on these criteria, a fundamentally new and scalable approach is required to meet the demands of sub-1μs RPCs.

To address this need, we propose ALTOCUMULUS, a **proactive** migration-based system which uses a queuing theory based model to predict which RPCs are likely to experience high latency. We then migrate such *critical* RPCs from heavily loaded to lightly loaded cores *before* they negatively impact end-to-end latency. This is in stark contrast to prior work where critical RPCs are identified **after** they have violated end-to-end latency requirements and are simply dropped [14], [21]. ALTOCUMULUS achieves high **performance** without unnecessarily dropping packets.

To **scale** to high core-count servers, ALTOCUMULUS' runtime splits physical cores into *groups*. Each group consists of a single centralized manager core that dispatches RPCs to its several worker cores for processing. Our approach differs from prior work that uses a globally centralized manager to allocate RPCs to all worker cores; a centralized manager can become the performance bottleneck for 40+ GbE traffic [26], [48]. Our proactive migrations occur between manager cores, which saves scheduling traffic compared to prior work that balances load between all worker cores [53].

To achieve both performance and scalability, we offload migrations in ALTOCUMULUS to a hardware mechanism,

which allows for quick and efficient proactive migrations and avoids the high scheduling overhead of prior work. Because the mechanism is implemented based on lightweight hardware primitives with *direct register*-level access from the user level, ALTOCUMULUS provides cloud providers with **adaptability**. Different from prior art that seeks adaptive scheduling for task-parallel workloads [24], [57], ALTOCUMULUS allows adaptability for RPCs that exist in complex and unpredictable cloud environments while guaranteeing strict μs-level latency deadlines. Together, ALTOCUMULUS eliminates overheads incurred by current techniques and enables high-throughput, scalable and adaptive handling of sub-1μs RPCs.

II. BACKGROUND AND MOTIVATION

We provide background on measuring performance in datacenters, how RPCs are handled and why RPC scheduling is critical for high-throughput on sub-1μs latency RPCs. Next, we present an analysis of prior work on RPC scheduling and motivate the necessity of providing a software-hardware co-design to address limitations of prior work.

A. Measuring Datacenter Performance

Quantifying datacenter performance is more complex than using a single metric such as overall throughput or queries serviced per unit time. Cloud service providers (CSPs) must balance competing objectives such as: 1) maximizing the number of users they can service, 2) minimizing the latency for users and 3) minimizing running costs. CSPs must guarantee certain performance criteria to customers, such as minimizing server down-time or setting a maximum latency for user queries. These requirements are codified by CSPs as part of their Service Level Objectives (SLO). Setting a maximum allowable latency is particularly important as it directly influences both end-user experience and CSP profits [14].

Quantifying SLO. For a given hardware configuration, a CSP could set SLO to be the highest latency a user would tolerate for a specific application. However, this can lead to significant resource under-utilization as only a small fraction of users ($\sim$1%) might experience this worst case latency. Instead CSPs provide a *probabilistic* guarantee of the latency that 99% of users would experience, referred to as the 99th percentile latency. Therefore, we focus on 99th percentile latency as the key SLO metric which affects CSP performance and profits [14]. CSPs then try to service as many users as possible without violating this latency constraint. We use the metric of 'throughput@SLO' to measure the number of user requests that can be serviced without violating this SLO requirement [61]. To understand the challenge of RPC scheduling (which is the focus of our work), we next describe how an RPC is typically handled in a CPU server.
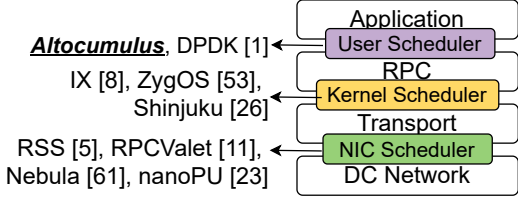
Figure 2: RPC system stack and different scheduler implementations in related work.

Figure 3: High request throughput (load) with low-latency requests requires low scheduling overhead (shown in ns).

B. RPC Handling on a CPU Server

Fig. 2 depicts the different layers of the RPC system stack.[3] An RPC request arrives as a network packet from the datacenter (DC) network, e.g., Ethernet or InfiniBand, shown as the lowest layer. The network interface card (NIC) parses the packet's header and the on-NIC scheduler dispatches the packet to a CPU to run its transport layer. Receive Side Scaling [5] (RSS) is a commonly used on-NIC scheduler, where requests are distributed evenly across per-core queues [8], [13], [25], [30], [39], [48], [49], [50], [53], [54]. RSS scales well with increasing core count as its dispatch decisions are agnostic to core load: each core polls its private queue without synchronization. The transport layer uses protocols (e.g., TCP) to handle network interfacing.

Before passing the request to the RPC layer, some kernel schedulers use load balancing policies to schedule requests to cores [44]. The most common policy is work stealing [17], [48], [53], [54], where idle cores pull requests from other busy cores. Next, the RPC layer does RPC header parsing, requested function identification, message payload deserialization, etc. [52]. The requested function is then called in the application. In some systems, user-level schedulers are implemented to decide which core runs the requested RPC function. For instance, high-performance key-value store (KVS) applications maintain cache locality by binding the application and request handling to the same core [38], [39]. RPC responses are created by the application and traverse the RPC stack, going through the same operations in reverse order. All scheduler designs between layer-pairs can co-exist

[3]Some implementations bypass some layers. For example, user-space networking may bypass the kernel scheduler.
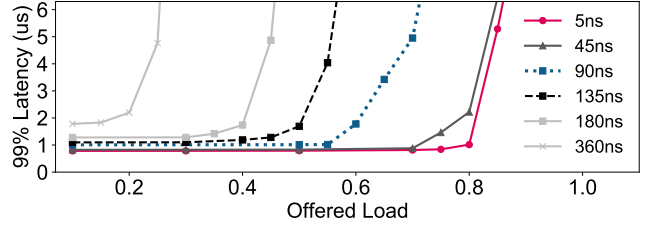
in some systems. We now describe why RPC scheduling is a major bottleneck for modern server CPUs.

C. Why Does RPC Scheduling Matter Now?

To demonstrate the importance of RPC scheduling time, we perform a discrete event simulation of a 64-core system. To show end-to-end *scheduling overhead* of sub-1 μs RPCs, we combine the overheads due to all the layers in Fig. 2. Our results in Fig. 3 show that reducing scheduling latency from 360ns to 5ns can improve throughput by $\sim3\times$ for a 99th percentaile tail latency of 5 μs. We use 45ns and 360ns in this experiment to represent the levels of time of a memory access and a work-stealing operation commonly used in scheduling [54], respectively. We tweak the overhead from 45ns to 360ns using numbers as a multiple of 45ns. We observe that even a few extra nanoseconds due to scheduling can significantly hurt tail latency of nanosecond-scale RPCs. Next, we provide some background on RPC schedulers detailed in prior work.

D. RPC Scheduling in Practice

In Table I, we categorize several state-of-the-art scheduler implementations according to three aspects: the scheduling scheme, scheduling implementation, and communication mechanism. Scheduling schemes determine which request is dispatched to which core and whether this decision is made in a centralized or decentralized manner. Communication mechanisms represent the communication channel a scheduling operation relies on after decisions are made.

Kernel-based c-FCFS. Centralized first-come-first-served (FCFS) scheduling (*c-FCFS*) uses one dedicated CPU core

Table I: Comparison of ALTOCUMULUS with prior art.

Prior work	Scalability bottleneck	Communication mechanism	Scheduling scheme	Scheduling manager
ZygOS [53]	high s/w stealing rate	shared caches	d-FCFS with work stealing	s/w, kernel-based
IX [8]	imbalance		d-FCFS	s/w, kernel-based
Shinjuku [26]	imbalance, dispatcher throughput		c-FCFS	
eRSS [55]	imbalance, interconnects	PCIe	d-FCFS	h/w, NIC RSS
nanoPU [23]	register file size, NoC	register files		
RPCValet [11]	limited cohe. domain size, mem. b/w	shared caches	c-FCFS	h/w, NIC-based
Nebula [61]	limited coherence domain size			
Altocumulus	mis-prediction penalty, NoC	migration channel & shared caches	global d-FCFS, local c-FCFS	h/w, SLO-aware user-level

(a) Kernel-based c-FCFS **(b) d-FCFS + stealing** **(c) NIC-driven c-FCFS**
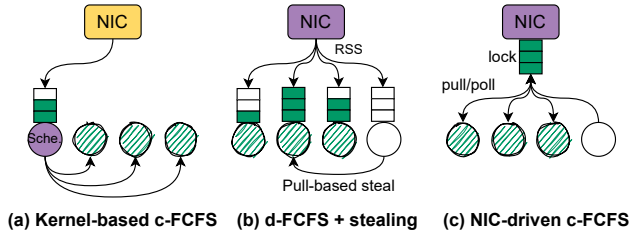
Figure 4: Common request scheduling models. Blocks in purple indicate schedulers.

as the scheduler (Fig. 4(a)). However, this single core can become a significant performance bottleneck. For instance, the centralized manager core in Shinjuku [26] can only handle 5M requests/s, or 2.5 Gbps and 41 Gbps of Ethernet traffic for 64B and 1024B requests, respectively. Due to lock contention and synchronization overhead on the centralized queue, current kernel-based schedulers are limited to a minimum scheduling interval of 5μs [26], [48].

NIC RSS-based d-FCFS. Another approach is to use RSS on the NIC to schedule requests. Despite being scalable, RSS scheduling does not factor in each core's load, leading to significant imbalance and therefore unpredictable tail latency increases [7], [53]. Scheduling using distributed queues such as RSS is called distributed FCFS scheduling, or *d-FCFS*.

d-FCFS + work stealing. ZygOS [53] enables load balance in RSS-based d-FCFS designs: idle cores with empty queues steal requests from other heavily loaded cores (Fig. 4(b)). However, ZygOS incurs significant overheads for two key reasons. First, ZygOS triggers work stealing with simple policies such as randomly selecting a queue to steal, leading to 60% of requests being moved across cores, which wastes communication bandwidth [53]. Worse, such intrinsic communication requirements are triggered without considering the SLO. Second, to find and fetch pending requests from other cores, work stealing requires 2 to 3 cache misses. This incurs 200-400 ns of inter-thread communication [54] or even ~1 μs interrupts [26], making it unsuitable for sub-1 μs RPCs.

NIC-driven c-FCFS. This model features a centralized queue managed by the NIC (Fig. 4(c)), which alleviates the bottleneck of Fig. 4(a). However, the packets must still use the slow PCIe bus to move from the NIC to the CPU. RPCValet [11] and Nebula [61] bypass this overhead with NICs that share the same memory space as the cores. However, remote cache accesses still limit the throughput in high core count systems [6]. NIC-to-core transfers are also restricted to the same coherence domain, whose size is limited due to hardware complexity [18]. While hardware RPC schedulers improve performance over kernel-based scheduling, they are not as adaptable to different arrival patterns and loads, which we elaborate on next.

Scheduling adaptability V.S. programming efforts. On-NIC schedulers typically require specialized hardware to implement a fixed scheduling policy. For instance, Nebula and nanoPU implement the Join-bounded-shortest-queue (JBSQ(n)) policy in hardware [23], [33], [61]. Similarly, other hardware-based schedulers [11] or work stealing [35] cannot adapt to varying input loads and request patterns.

Kernel schedulers (e.g., IX, ZygOS, and Shinjuku) are adaptive but require significant development and maintenance effort [43]. Therefore, application developers often build a user-level bespoke framework or dataplane system for each application class. However, these dataplanes typically use Linux-incompatible APIs [8], [26], [39], or rely on syscalls [22] whose performance is far from nanoseconds.

III. ALTOCUMULUS OVERVIEW

ALTOCUMULUS prevents SLO violations by *proactively* migrating potentially SLO-violating RPC requests instead of detecting that an SLO violation has already occurred. For this, we must first predict which requests are likely SLO violations and migrate them to lightly-loaded cores. We develop a model based on queuing theory, which uses **statistical queue length distributions** to predict potential SLO violations (Sec. IV). To perform migration, we employ a **fast hardware-assisted scheduling mechanism**, which uses register-level messaging to quickly move requests across cores (Sec. V). ALTOCUMULUS synergistically combines the effectiveness of SLO-aware migration policy and hardware-based mechanism altogether through a software runtime (Sec. VI). We now describe the different components of ALTOCUMULUS.

A. System Components

Fig. 5 shows a high-level overview of the entire ALTOCU-MULUS system. The software has an offline component which calculates a prediction model, which is then fed to the online components in the proactive scheduling scheme. The offline component takes as input the number of cores (k), request service time distribution (μ), message arrival patterns (λ), and the SLO target (detailed in Sec. IV) to generate the model.
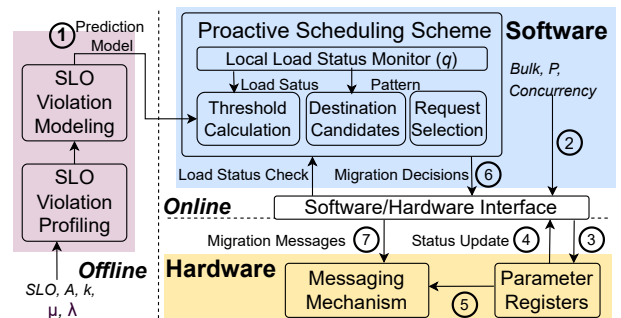

Figure 5: ALTOCUMULUS system overview.

The software runtime uses the model and based on the current system load (A), calculates the migration threshold ①. The runtime also adjusts several migration-related parameters ②, which are then stored in *parameter registers* (PRs) ③. The runtime periodically checks its local load status (q) by reading registers ④. If migration is required, the runtime selects the source and destination queues and generates a request to migrate through the hardware messaging mechanism ⑤⑥. The mechanism conducts each messaging operation according to the parameters stored in the PRs ⑦. At the end of each period (P), the load status of each manager core is shared to all the other managers and to the runtime to inform future migration decisions. We now elaborate on each component of ALTOCUMULUS.

Software runtime. The software runtime monitors the current status of request queues and periodically predicts potential SLO violations (Sec. IV-A). We implement our runtime as a decentralized system, consisting of distributed manager cores (*global d-FCFS*), each running the software runtime. Each manager core then communicates with a subset of the worker cores (*local c-FCFS*), to schedule RPC requests to them. Migrations in ALTOCUMULUS only happen between manager cores. The runtime is described in Sec. VI.

Hardware messaging. Once the runtime predicts that an SLO violation is likely to occur, it communicates to the hardware to migrate requests between manager cores. This is done via messages, detailed in Sec. V.

Microarchitecture. Fig. 6 shows the additional hardware in each manager core tile. To provide low-overhead messaging, ALTOCUMULUS messages are sent and received directly through specialized *migration registers* (MRs) instead of using memory-mapped buffers. Hardware modules required for messaging include: 1) MRs that store *descriptors* to RPC messages in message arrival order, while the entire RPC message is in the LLC, 2) a migrator that performs register-to-register data movement, 3) PRs to hold runtime parameters, 4) a pair of send/receive FIFOs and 5) a controller to manage the operation of the added hardware.

System parameters. ALTOCUMULUS is configured through several key parameters, which we list below:

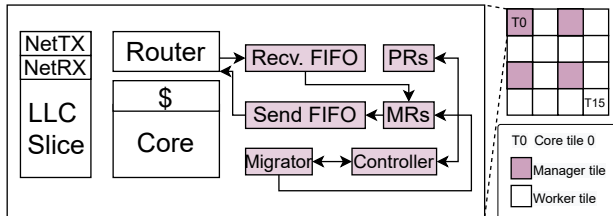1) $Concurrency$ determines the number of concurrent flows

Figure 6: Microarchitecture for each manager core. Purple modules indicate hardware modifications.

per migration, where each flow goes to a separate destination manager core. We set this to be $\frac{n}{4}$, $\frac{n}{2}$ or n, where n is the number of manager cores.

2) $Period$ determines the time interval between two migration decisions. The runtime periodically checks migration-related parameters to decide if migrations are necessary. We sweep P from 10 ns to 1000 ns.

3) $Bulk$ is the maximum number of requests we batch per migration operation to reduce overhead. We consider batches of 8-40 requests.

We now detail our predictive model for SLO violations.

IV. PROACTIVE MIGRATIONS

In this section, we answer the following questions:

- How can we effectively predict SLO violations?
- What are the trade-offs between minimizing migration cost and maximizing SLO violation prediction accuracy?

A. Statistical Characterization & Modeling

The crux of ALTOCUMULUS' proactive scheduling is to predict and prevent potential SLO violating RPCs by migrating them to less loaded cores that can process them within the SLO target. We first characterize the queue length at which SLO violations begin to occur, which we call the *threshold*.

Threshold characterization. As queuing is the root cause of long tail latency, a straightforward approach is to use the queue length to predict potential violations.[4] For a k-core c-FCFS system under a certain load, there is a threshold queue length T, after which SLO violations would begin to occur. A naive approach is to directly use $T = \frac{SLO\ Target}{Average\ latency}$. SLO is typically set to have the 99th percentile latency to be within $L\times$ of the average latency [14], [53], [61]. If all the worker cores are busy and the manager core's queue length exceeds $k \times L + 1$, any new requests will likely violate SLO. So one option might be to set the threshold to be $k \times L + 1$.

This naive model is not optimal as it does not take the *statistical distributions* of queuing delay, per-core service time and request arrival patterns into account. To demonstrate this, we perform an analysis using a cycle-accurate discrete event simulation of a 64-core system. We set L=10 and use a Poisson distribution for the request arrival pattern, similar to prior work [14]. Figs. 7(a)-(c) plots the ratio of SLO violations ($\frac{\#\ SLO\ Violations}{\#\ Total\ Requests}$) at a given queue length for three widely-used service time distributions: Fixed, Uniform and Bi-modal [26], [53], [61] for a single system load (0.99). Figs. 7(a)-(c) demonstrate that:

1) Queue length is a good metric to capture the *trend* of SLO violations and the ratio of SLO violations increases sharply once the queue length exceeds a specific value.

---

[4]Existing software systems such as Intel DPDK (Data Plane Development Kit) [1] offer rich APIs to monitor queue length at runtime.
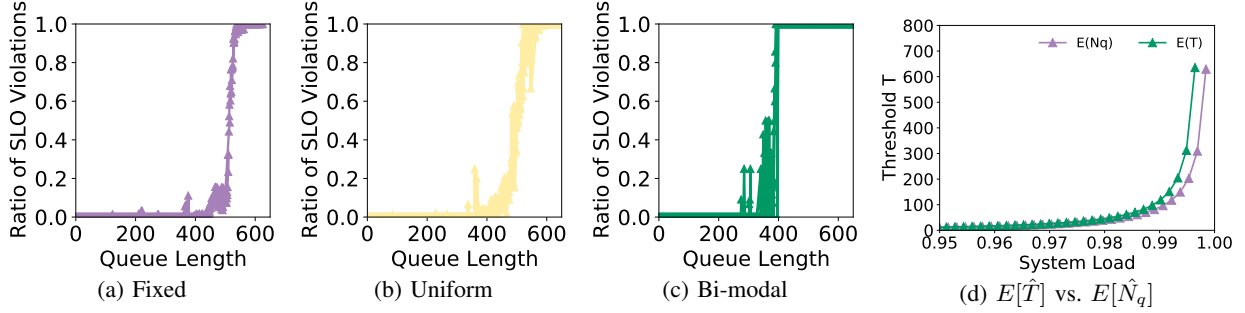
Figure 7: SLO violation prediction analysis on a 64-core system. (a,b,c) shows prediction accuracy for different request service time distributions for load = 0.99. (d) Example of $E[\hat{T}]$ based on linear transformation of $E[\hat{N}_q]$ for Fixed distribution (a=1.01, c=0.998 and b=d=0).

2) The first few SLO violations occur when the queue is at moderate occupancy (shown as points above the x-axis). This is due to variations in request arrival intervals and service time.

3) For all three distributions, nearly 100% of requests violate SLO once T is set to $k \times L + 1$ (i.e., 64×10 + 1 = 641). However, using this as the threshold would not catch SLO violations which occur for smaller queue lengths.

How then can we dynamically select the threshold T based on the current system load to accurately predict which RPCs will violate SLO?

ALTOCUMULUS prediction model. We observe through our simulations that, at moderate loads, early SLO violations happen when the local queue's length is greater than the length of the other queues. Each manager thread holds an independent N_q random variable representing the total number of RPC requests waiting in the local queue. For each manager thread, we model the expected value of queue length ($E[\hat{N}_q]$) using queuing theory, which naturally embeds statistical information and system states that affect tail latency. Specifically, we leverage the Erlang-C formula $C_k(A)$, which expresses the probability that an arriving request must queue, to model $E[\hat{N}_q]$:

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

$C_k(A)$ depends on the system load (A), which is a function of the request arrival Poisson distribution with rate λ and per-core service time μ. We use this analytical model to derive an expected value for T, called $E[\hat{T}]$. T is the queue length when the **first SLO-violating request** arrives for a given system load. First, we measure these T values in simulation, similar to Fig. 7 but sweeping across all system loads. We then observe that $E[\hat{T}]$ can be modeled as a linear transformation of $E[\hat{N}_q]$:

$$E[\hat{T}] = a \times E[c \times \hat{N}_q + d] + b \tag{2}$$

a, b, c and d are constants that are empirically determined based on factors such as the service time distribution.

Fig. 7(d) shows an example of a linear transformation of $E[\hat{N}_q]$ to derive an accurate SLO violation threshold $E[\hat{T}]$. The modeled $E[\hat{T}]$ is verified with our simulation results. The $E[\hat{T}]$ model is fed to our runtime, which uses the current system load to calculate T every period. If the local manager's queue length exceeds T, all queued RPCs greater than T are predicted to violate SLO and will be selected for migration. We explain the entire software runtime in Sec. VI.

Trade-off between prediction accuracy and migration effectiveness. Our goal is to avoid all SLO violations while minimizing false positive predictions. We define *prediction accuracy* as the ratio of correctly predicted SLO violations to the total number of SLO violations. In the above analysis, we derive T as the queue length when the first actual SLO-violating request arrives for a given system load. This queue length represents the lower bound on T (i.e., T_{lower_bound}). Thus T_{lower_bound} would be 121, 80, and 268 for Fig. 7(a), (b) and (c) when the system load is 0.99. While setting T=T_{lower_bound} would save all SLO-violating requests, this would cause a significant amount of unnecessary migration traffic as not all of these migrated requests would violate SLO. On the other hand, to maximize migration effectiveness, we can set T to be $k \times L + 1$, which we denote as T_{upper_bound}. By doing so, every migration triggered prevents SLO violations. Unfortunately, in this case, we miss a non-trivial number of violations and suffer from low prediction accuracy. Given this trade-off, we opt to design a flexible modeling framework to accurately model different selections for T.

V. HARDWARE MESSAGING MECHANISM

In this section, we answer the following questions:

- How does hardware support ALTOCUMULUS messages?
- What is the hardware cost of our mechanism?

A. Direct Register Messaging Mechanism

ALTOCUMULUS uses four message types to interface between the software runtime and the hardware. Fig. 8
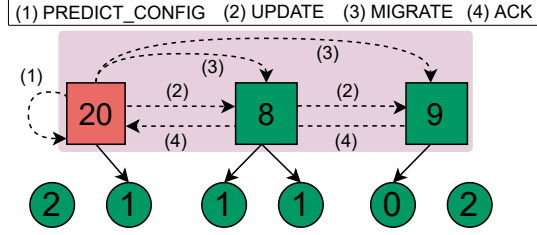
Figure 8: ALTOCUMULUS protocol with hardware message paths (dotted arrows). SLO violations happen in the manager.

Table II: Runtime messages. T: SLO violation threshold, q: queue depth, Q_D: a vector of migration destination queues.

Name	Description	Input Format
PREDICT_CONFIG	Configure registers to adjust migration parameters	<reg_addr reg_value>
MIGRATE	Proactively dequeue RPCs from Tail to destination queue(s)	S, Q_D, *MR[Tail]
UPDATE	Broadcast local queue length to all other managers	<q>
ACK/NACK	Acknowledge the completion/discard of a MIGRATE message	-

provides an example of our protocol with messages between manager cores (squares). For local scheduling, worker cores (circles) can queue at most 2 requests, inheriting the same algorithm as existing hardware schedulers [23], [61]. For each core, the number shown represents the number of queued RPC requests. The red manager core has 20 pending requests and is migrating RPC requests to the other two lightly-loaded cores. The different message types are listed at the top of Fig. 8 and summarized in Table II. We now describe each message type.

PREDICT_CONFIG. The manager cores periodically (every period P) execute the software runtime (Sec. VI) to update the runtime parameters. The manager cores then use this message type to internally update their parameter registers (PRs), which are read generating other messages. PRs store period P, maximum batch size $Bulk$, queue length vector q, migration threshold T, and the $Concurrency$. Unlike the other messages, PREDICT_CONFIG messages are not sent across manager cores but are only used within cores.

MIGRATE. This message is used to migrate RPCs predicted to violate SLO to a different manager core. To reduce overhead, we migrate several RPC requests at once. MIGRATE messages involve the following: 1) determining the size of each MIGRATE message (S), which contains a number of requests (req_num), 2) message generation, 3) dequeuing requests locally and 4) enqueuing requests to remote managers. MIGRATE messages have both send and receive paths. To determine req_num when sending a MIGRATE message, the local controller uses $Concurrency$ and $Bulk$ stored in PRs to calculate $\frac{Bulk}{Concurrency}$. Then the controller hands over message generation to the local

migrator. Each message has a header that stores req_num, local manager core ID src_mid, remote manager core ID dst_mid determined by the manager thread. The tail pointer points to *MR[Tail], and is maintained by the migrator. The migrator then reads req_num RPC pointers from its local MRs, enqueues them into the send FIFO, and injects them into the NoC. The send FIFO occupancy is monitored by the controller to prevent overflowing the FIFO.

To receive a MIGRATE message, the controller first parses the header containing req_num and src_mid, and validates if dst_mid equals the local manager core ID. Then the MIGRATE message payload is stored in the receive FIFO if the FIFO is not full. Next, the controller gives permission to the migrator to dequeue and move req_num entries from the receive FIFO's head to the local MR for scheduling.

ACK/NACK. When sending MIGRATE messages, once the source migrator receives an ACK from the dst_mid, it invalidates those req_num entries in its local MRs. When MIGRATE messages are received, the destination manager's controller issues an ACK to src_mid manager after receiving the entire MIGRATE message. If the destination manager does not have free receive FIFO slots or MR entries, it drops the received MIGRATE message and returns a NACK to the source to reject this migration. For simplicity, the source core does not replay this rejected migration message.

UPDATE. UPDATE messages are triggered periodically to broadcast the local queue length (q) to all other manager cores. The hardware messaging latency at NoC speed (3 ns per hop) for such synchronization is superior to software based updates through shared caches.

B. Discussion

Design optimizations. To make the hardware messaging fast and efficient, we implement the following optimizations:

1) Our mechanism only operates between manager core tiles. The direct *register* messaging bypasses the cache coherence protocol which would add unpredictable contention and prohibitive delays for nanosecond-scale RPCs.

2) To avoid moving RPCs until they need to be processed, each manager only stores the *descriptor* (pointer and connection information) of each message (14B), while the actual message is stored in the network buffer (in the LLC or memory).

3) The mechanism batches multiple descriptors for every migration to reduce data movement. It also supports concurrent migrations between multiple source-destination pairs to improve migration throughput.

4) We allow a request to be migrated at most once. This restriction saves unnecessary traffic and avoids livelocks and migration-induced interconnect contention.

Bounded MR and FIFO. Each MR does not need to store an entire RPC message that requires up to 2KB [23]. Instead, messages stay in the LLC [61] and each MR only stores

Algorithm 1: Software runtime executing on each manager core.

Input: $Concurrency$, number of queues N, queue length monitors $q[0, 1, ...N-1]$, network receive queues $NetRX[0, 1, ...N-1]$

1 every *period* **ns:**
2 $q \leftarrow Update(q)$
3   Use the prediction model to calculate T
4   **for** $j \leftarrow 0$ **to** $N-1$ **do**
5 $pattern, Q_D \leftarrow$
         $\text{predict}(T, q, Concurrency)$
6   **for** $i \leftarrow 0$ **to** $Q_D.size()$ **do**
7   MIGRATE message size $S \leftarrow$
           $\frac{Bulk}{Concurrency}$
8   **if** $q[j] - S < q[Q_D[i]] + S$ **then**
9   continue
10   **else**
11 $NetRX[j].tail\_dequeue()$
12   Trigger one MIGRATE message
13 $NetRX[Q_D[i]].tail\_enqueue()$

an 8B pointer to the RPC message and a 48-bit IP address per IP port. Each MR therefore consumes just 14B. Since we use a decentralized manager, MRs in each manager core tile can be bounded regardless of the total number of cores in the system. According to Eqn. 1, the mean of $E[\hat{N}_q]$ for each group equals 11 when system load is near 1, therefore each group contains one 154B MR (11×14B). For FIFOs, small send/receive FIFOs suffice, since migration is not continuously used. In our runtime, send FIFOs that hold 16 entries are sufficient to cover message bursts, which means 16×14B = 224B per FIFO.

Message Ordering. UPDATE, MIGRATE, ACK/NACK messages go through the NoC. We need to preserve ordering both in each manager core and during the message transfer through the NoC between sender and receiver. In each manager core, we employ FIFOs for send/receive buffers. In the NoC, we can either use deterministic routing or implement a flow control protocol with reordering at the endpoints. We opt for deterministic routing since the NoC is often lightly loaded [58]. We use one extra virtual network [12] to route ALTOCUMULUS packets in the NoC.

VI. SOFTWARE RUNTIME

We now introduce the software runtime for our proactive migration-based RPC scheduling system. The runtime is decentralized and runs on each manager core. Each manager core owns a network receive (NetRX) queue across all its local worker cores and dispatches requests to worker cores. The software runtime periodically: 1) Executes SLO violation prediction, 2) determines the migration destination NetRX
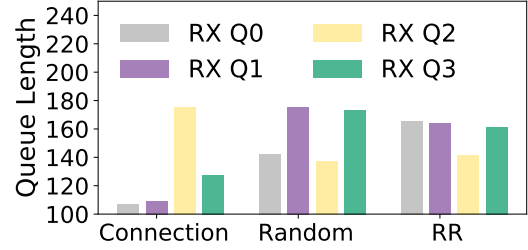

Figure 9: Snapshot of temporal load imbalance across 4 network receive queues (256-core d-FCFS system with each queue managing a 64-core c-FCFS system).

queue, 3) classifies the distribution pattern, 4) synchronizes system states across manager cores, and 5) triggers MI-GRATE messages. Algorithm 1 details the runtime operation.

Parameter register configuration. The software runtime configures migration parameters to the PR registers in every manager cores through the software-hardware interface.

Periodical proactive prediction. The queue length monitor vector q is periodically updated across all manager cores through UPDATE messages (line2 of Algorithm 1). Each manager core runs SLO violation prediction (line5) every P ns. The inputs of predict() are read from the PRs. For the n-th manager core, predict() checks two conditions: 1) whether $q[n]$ exceeds T and 2) if the queue length distribution based on q matches any of the patterns below. If either returns true, predict() returns a Q_D that records a set of migration destination queue ID(s) and the pattern.

Pattern classification. We classify the patterns we observe for q into three categories.

1) **Hill**: We identify q as a Hill pattern when the length of the longest queue is larger than the length of the second longest queue by $Bulk$. This pattern triggers several MIGRATE messages to move pending work from the longest queue to other (shorter) queues.
2) **Valley**: This pattern occurs when the length of the shortest queue is smaller than the length of the second shortest queue by $Bulk$. When detecting a Valley pattern, each manager core triggers one MIGRATE message from itself to the shortest queue.
3) **Pairing**: This pattern triggers M messages, the Mth message is sent from the Nth longest queue to the shortest queue, the $(M-1)^{th}$ message from the $(N-1)^{th}$ longest queue to the second shortest queue, etc. Pairing occurs when there is a gradual imbalance in queue lengths.

Fig. 9's three bar groups correspond to Hill, Pairing, and Valley. We model different policies such as randomly selected (random), round-robin (RR), and connection-based (i.e., requests from a certain network connection are steered to a specific network receive queue, which is the policy that RSS uses). This result shows the queue lengths for 4 NetRX queues at the cycle when the first 10 SLO violations occur.

We see that in each case there is a noticeable difference in queue lengths. Each manager's pattern classification gives the same `pattern` result because q is synchronized across all manager cores every period.

Migration destination(s) determination. For each pattern, the number of migration destination queues is determined by $Concurrency$. The queue ID of destination(s) is selected according to the above pattern classification algorithm.

Migration message generation. `line7` of Algorithm 1 generates MIGRATE messages a size of S. The number of messages to be generated per migration is equal to $Concurrency$.

Migration messaging. We forbid the migration if the condition in `line8` of Algorithm 1 is met. This prevents a migration that would result in the migrated message experiencing longer queuing than if migration had not occurred. Otherwise, we leverage our hardware mechanism to dequeue the tail of `NetRX`, send MIGRATE messages out, and have the destination queue enqueue the migrated contents at its tail.

Walk-through example. Consider a system where $Bulk$=40 and $Concurrency$=4. If q is [30, 30, 70, 30] for 4 `NetRX` queues, the runtime identifies this as a Hill pattern. The manager core of the 3rd queue triggers one MIGRATE message to move 10 RPC request descriptors to each of the other queues and Q_D will be set to $\{0, 1, 3\}$). Once complete, the manager core of the 3rd queue will receive 3 ACK messages in total. At the end of this period, q is updated in all manager cores and q' is now: [40+X_1,40+X_2,40+X_3,40+X_4], where X_1-X_4 are the number of new requests that have arrived at each `NetRX` queue and have not yet been dispatched to any core.

Software-Hardware Interface. As discussed in Sec. V, ALTOCUMULUS messages use register-level read/write, queue operations such as enqueue, dequeue and queue status checking for FIFOs and MRs. One option for communication between the user runtime and hardware is using standard `x86` model specific registers (MSR) to move register-level data. This does not require adding any new hardware or any ISA extensions. However, this interface uses syscalls such as `rdmsr` and `wrmsr`, rather than directly issuing micro-ops to the hardware from the user space. We find that these syscalls take ~100 CPU cycles on Sandybridge-EP servers.

We add instructions (shown in Table III) to allow the runtime to directly communicate with the hardware. When a manager core receives a message, it can issue the corresponding instruction directly to perform a register level read/write. Specifically, `altom_update` and `altom_predict_config` can perform register-level data movement to implement UPDATE and PREDICT_CONFIG messages. `altom_status` grabs the arguments required by the MIGRATE message from PRs (parameter registers). Implementing atomic queue operations between FIFOs and

Table III: Custom ALTOCUMULUS instructions

Instruction	Description
`altom_send r1, r2, r3`	Send local MR offset (r1) content to MR entry ID (r2) with a batch size (r3)
`altom_status r3, r4, r5`	Returns local header, tail, and threshold pointers
`altom_update r6, q<n,1>`	Update local rx queue depth (r6) to all managers (vector reg of length n, stride 1)
`altom_predict_config r7`	Update migration related registers

the MR can also be costly if we rely on blocking atomic operations provided by the CPU [59]. Therefore, we leverage our migrator hardware module (Fig. 6) to allow for reads and writes asynchronously with the CPU. This offers significant speed-ups compared to using MSRs, as shown in Sec. IX-D.

Programmer guidelines. To use ALTOCUMULUS' prediction model, the programmer of the software runtime must know several parameters, which we now briefly list. First, they must know k to calculate Eq. 2. They must know how many manager cores the system has (N) for Algorithm 1. Lastly, they must also set an SLO value, which is typically provided by the client or determined by the cloud providers.

With these parameters known, the programmer must now pick the optimum values for other parameters such as $Period$, $Bulk$ and $Concurrency$. For μs-scale RPCs, an optimal $Period$ is typically less than 1 μs. If RPCs' on-CPU time involves a long latency operation (e.g., a PCIe transfer (200-800 ns [46]), QPI latency (150-250 ns [6]) or inter-thread communication (400-800 ns [54]), the $Period$ is typically dominated by this long latency. Second, a larger $Period$ usually couples with a larger $Bulk$. Third, $Concurrency$ is usually maximized to be N.

VII. METHODOLOGY

In this section, we outline the methodology we use to evaluate ALTOCUMULUS. We start by providing an overview of the different configurations we evaluate followed by a description of our simulation environment.

A. Configurations

We evaluate ALTOCUMULUS with both software and hardware scheduling techniques, which we list below.

Emulated Commodity RSS NIC. A commodity server with a modern NICs with Receive Side Scaling (RSS) mechanism [5]. As in prior work [61], our implementation spreads requests to cores uniformly.

ZygOS and Shinjuku. State-of-the-art software-optimized solutions on a commodity server. Both rely on traditional network stacks, such as TCP/UDP. Unlike ZygOS, Shinjuku separates networking threads and dispatcher on a dedicated

CPU core. Both use SLO-unaware software-based scheduling operations such as work stealing and 5 μs preemption.

Nebula and nanoPU. State-of-the-art network-compute co-design with hardware schedulers. For extremely low network latency, both implement hardware-terminated network stacks [20], [23], [61] that offload NIC RX/TX queuing operations in hardware. To reduce NIC-to-core message transfer overhead, both tightly integrate NIC and CPU with a hardware-based communication mechanism–Nebula leveraging cache coherence protocol and nanoPU, direct messaging to a core's special register file. Both use centralized hardware scheduler based on JBSQ algorithm to manage all cores–every time a core's local queue has less than 2 queued requests, the central scheduler pushes the head of its queue to that core.

AC_{int}. ALTOCUMULUS on hardware-terminated integrated NICs. Each group contains 1 manager core and 15 worker cores, where the manager core only runs the software runtime and the other 15 cores carry out RPC processing. Similar to Nebula and nanoPU, within each group, this design employs their centralized hardware JBSQ scheduler and offloads NIC RX/TX operations in hardware. Our design then rebalances requests across multiple groups.

AC_{rss}. ALTOCUMULUS runtime on top of a commodity CPU with RSS NIC attached through PCIe, which adds pressure from 1) load imbalance from using RSS and 2) long latency from PCIe. In each group, each manager core runs the software runtime and handles traditional networking threads and request dispatch, similar to Shinjuku. The design requires a minimum of 70 cycles to move a message to a worker through the cache coherence protocol [26].

B. Simulation Environment

We simulate all designs with a Pin-based [41] in-house simulator extended from Zsim [56]. We model 16 cores, as the prior work we compare against is optimized for this scale. We model the NIC, NoC and QPI in our simulator and faithfully quantify their queuing effects. We model the NIC with memory-mapped queue pairs (QPs) that receive RPC messages and send RPC responses. We integrate techniques that reduce NIC-to-core transfer overhead proposed by Nebula and nanoPU when evaluating those two baselines. Ethernet MAC, serial I/O and transport interpretation time on the NIC are set to be ∼30ns in total [23]. Each NoC packet has a per-hop latency of 3ns. We model QPI with point-to-point latency of 150ns [6]. PCIe latency is 200-800ns depending on data size [46].

Load generator. We evaluate both synthetic and real-world traces. First, we generate Poisson-based synthetic traces. Second, we use a request arrival pattern based on a regression model trained in the public cloud [9]. The regression model also integrates features that encode temporal information about the period for which we are generating batches. Batches
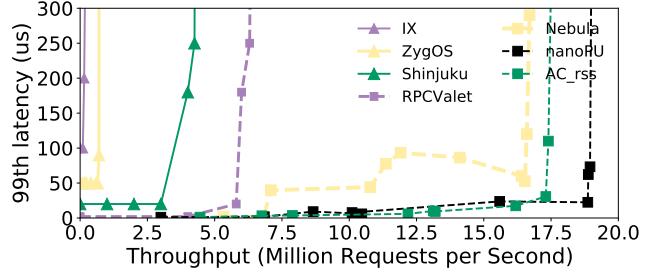

Figure 10: Comparisons against prior work with synthetic workloads (16-core, Bi-modal distribution)

are particularly common and widely explored for RPCs [27]. This model can accurately capture 82% and 92% of actual request arrivals in Microsoft Azure and Huawei Cloud, respectively [9].

Evaluation metrics. We evaluate the performance of all designs in terms of throughput@SLO. Unless otherwise stated, the SLO is a 99th percentile latency target of 10× the average RPC service time [14], [53], [61]. We evaluate the effect of saving SLO violations for SLO target ratios other than 10 in Fig. 13(c). Our measurements are server-side: each RPC's latency measurement begins when it is received by the NIC, and ends when its buffers are freed upon completion.

VIII. EVALUATION

We now present an evaluation of our design, starting with a comparison against state-of-the-art schedulers using synthetic workloads (Sec. VIII-A). We also explore varying group sizes, migration parameters and evaluate scaling to higher core counts in Sec. VIII-B-VIII-E. Finally, we evaluate an end-to-end application with realistic traces in Sec. IX.

A. Comparing ALTOCUMULUS and State-of-the-art RPC Scheduling Systems

We compare ALTOCUMULUS with state-of-the-art scheduling systems in Table I. They span both software (IX [8], ZygOS [53], Shinjuku [26]) and hardware (RPCValet [11], Nebula [61] and nanoPU [23]) techniques. The request service time follows a Bimodal distribution where 99.5% of the requests are 0.5 μs and 0.5% 500 μs [26]. This high dispersion service time pattern represents the scenarios where long/short RPCs co-exist, e.g., GET/SET vs SCAN in key-value stores and databases. For this experiment, we set SLO target to be 300 μs. We use AC_{rss} to demonstrate how our design can correct the initial load imbalance caused by RSS.

Fig. 10 demonstrates tail latency vs. throughput. Though ZygOS outperforms RSS-based IX by using software-based work stealing, both suffer from head-of-line blocking when long requests exist (even though rare). Shinjuku addresses these issues through fast preemption and centralized load balancing and achieves up to 5× better throughput@SLO.
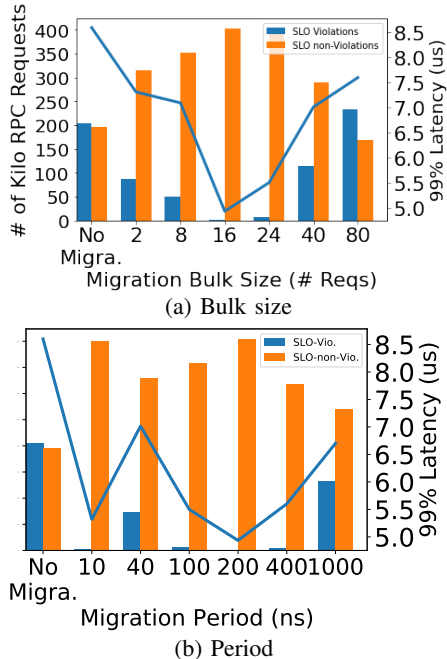
Figure 11: Impact of migration granularity and period on throughput (bars) and 99th percentile latency (line).

Nebula and nanoPU have a 3.9-4.4× throughput improvement over Shinjuku, as they employ hardware-based scheduler and hide NIC-core communication latency with a fine-grained JBSQ algorithm. Since Nebula's JBSQ only makes decisions based on per-core request number, it lacks awareness of long requests. Therefore, even at low load, its tail latency is up to 4.7× worse than Shinjuku. nanoPU avoids this drawback of JBSQ scheduler by piggybacking a preemption mechanism on each core. AC_{rss} has 24.6× improvement over ZygOS due to effective migration and low-overhead direct register messaging. We outperform Nebula by a factor of 1.05× and 15.8× for throughput and 99th percentile latency. Although we dedicate one core as the manager – sacrificing 6.25% potential throughput – we still deliver 92.5% throughput and similar 99th percentile latency compared to the best-performing case of nanoPU that gets rid of RSS. While nanoPU requires hardware changes both in the core microarchitecture and the NIC, ALTOCUMULUS only modifies the manager core tiles.

B. Group Size Exploration

ALTOCUMULUS 'groups' cores such that each group has one manager core and one or more worker cores. As the manager cores do not service RPC requests, smaller groups suffer from higher overhead due to the manager core. On the other hand, large groups makes more effective use of all cores, but suffer from the drawback of earlier work where the single manager core becomes the bottleneck. Fig. 12(a)

explores different group configurations for a 64-core system. For AC_{int}, group sizes of 16 and 32 cores achieve the highest throughput@SLO. A group size of 64 gets degraded throughput because of variance in remote cache access latency. In AC_{rss}, for group sizes larger than 16, a manager core itself can become the throughput bottleneck. The reason is that a manager's throughput has a theoretical upper bound of 28 MRPS, considering 70 cycles@2GHz per message. To support both designs, we choose to use 16 cores per group.

C. Migration Parameter Exploration

We now perform two sensitivity studies to understand how the choice of design parameters affects ALTOCUMULUS performance. We model NIC bandwidth to be 1.6 TbE and use a 256-core system, with 16 manager cores, each assigning work to 16 worker cores. In Fig. 11(a), we show the number of SLO violations and 99th percentile tail latency vs. $Bulk$, for a migration period of 200 ns. When $Bulk$=16, we can eliminate all SLO violations (left axis). We also see that 99th percentile tail latency (right axis) strongly correlates with SLO violations.

In Fig. 11(b), we show that varying the migration period from 10 ns to 400 ns does not significantly affect either the rate of SLO violations or the 99th percentile latency. We can understand why this is the case as follows. For a 1.6 TbE NIC, the time between packets is on average ∼2.5 ns. Thus for 16 queues, it would take $16 \times Bulk$ number of packets to fill the queues such that the longest queue would reach the threshold for migration. This works out to $2.5\text{ns} \times 16 \times Bulk = $ ∼640 ns, which is comparable to the mean service time of an RPC, which is ∼630 ns in our experiment. If migrating every 1000 ns, Fig. 11(b) indicates ∼1/3 of migration opportunity is lost and it fails to recover 150K out of 400K RPCs that violate SLO. The anomaly in the 40 ns case is explained in Sec. VIII-D.

D. Migration Effectiveness Breakdown

We replay 400K RPCs from the baseline, and compare the two cases to evaluate the effectiveness of ALTOCUMULUS' migration implementation. We split predicted SLO violations into four groups, which we show in Fig. 12(c), where:
1) `Eff.` means migrations that saves SLO violations.
2) `InEff. w/o harm` means the migrated RPC request did not violate SLO either before or after migration.
3) `InEff. w/o benefit` means this SLO-violating RPCs would still violate SLO after migration.
4) `False`, harmful mis-predictions, where an SLO-satisfying RPC becomes an SLO-violating one after migration.

`Eff.` can significantly improve 99th percentile latency because the worst case SLO violations are recovered. `InEff. w/o harm` can also reduce queuing delay of requests because we migrate them to a shorter queue. `False`
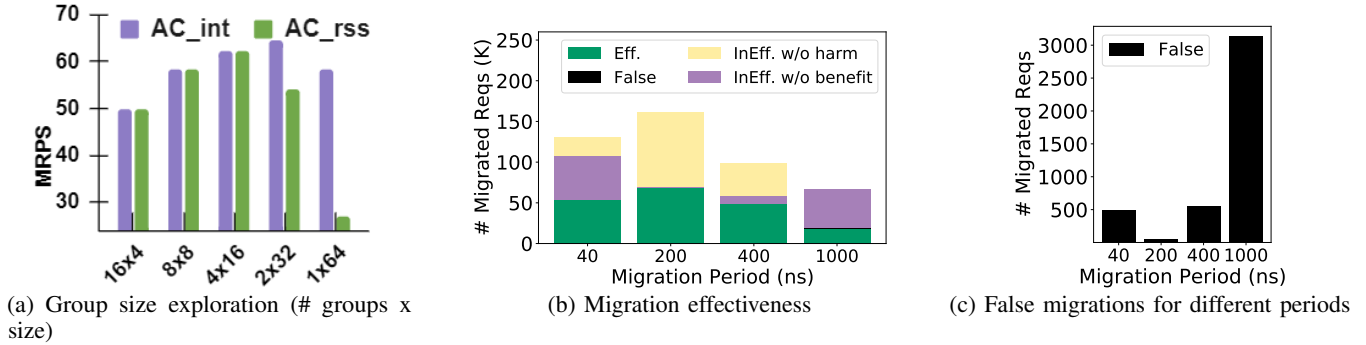
(a) Group size exploration (# groups x size)

(b) Migration effectiveness

(c) False migrations for different periods

Figure 12: ALTOCUMULUS sensitivity analysis.

and `InEff. w/o benefit` cause unnecessary scheduling traffic, and `False` even increases extra SLO violations and thus adversely affects throughput@SLO.

In Fig. 12(a), we observe that 130K, 161K, 108K, 69K RPCs have experienced migration for the 4 migration periods tested. First, migrating too lazily (1000 ns) fails to save 47% RPCs that are queued too deep (InEff. w/o benefit). Migrating too eagerly (40 ns) results in 41% of migration operations that fail to impart any positive benefit, as such frequent scheduling produces migrations faster than they can be consumed leading to new contention effects in the system.

Fig. 12(b) shows that we can eliminate **nearly all** mispredictions and save all RPCs to satisfy the SLO. At best, among 161K migrated RPCs, 70K of them are effective migration, saving SLO-violations. The effective ratio is 42%. The rest (58%) of the migration messages significantly reduce queuing time because they rebalance system loads across managers and do not cause any undesirable SLO violations. Sec. VIII-E discusses the overhead of those ineffective migrations and the way to further reduce such overhead.

Effectiveness of preventing SLO violations. Fig. 12(c) shows only the false migrations from Fig. 12(b). Since we evaluate 99th percentile latency, the slowest 1% of RPCS (4K) contribute disproportionately to tail latency. With a period of 200ns, we falsely migrate only 53 RPCs. We successfully eliminate >99.8% of SLO violations and improves tail latency.

E. Migration Overhead

Migration traffic. The results in Fig. 12(c) use the prediction model that selects T based on the first SLO violation, which delivers the highest accuracy by sacrificing migration effectiveness as discussed in Sec. IV. ALTOCUMULUS' approach can provide flexibility to balance this trade-off and save migration traffic. Although the model used in Fig. 12(c) results in 58% ineffective migration, the number of requests involved migration is smaller than the 70% of messages moved by ZygOS [53]. ZygOS also moves the entire message up to thousands of bytes, whereas we only move 14B message

pointers which significantly reduces scheduling related traffic.

Latency cost. The migration happens well before a migrated request gets processed. Thus, migration latency is off the critical path and does not hurt tail latency. Instead, Fig. 11 illustrates that migration can actually reduce the queuing time of migrated requests. The SLO prediction overhead added on each RPC consists of the operations needed in Algorithm 1. These operations (and their cycle counts) are: 1) 3ns per hop in the NoC to send update messages between managers, 2) 2 multiplications (7 cycles) and 2 additions (1 cycle) to calculate T, 3) at most 3 comparisons (2 cycles), one against T and the other three against $Bulk$ for each $pattern$. For a 2 GHz CPU, this gives a worst-case prediction latency is 18ns. For 256-core system, migration latency is less than 50ns. ALTOCUMULUS enables migrations as frequent as every ∼50ns without saturating system throughput or adding significant latency on the critical path.

IX. END-TO-END APPLICATION

In the previous section, we evaluated ALTOCUMULUS using a synthetic workload. We now show that ALTOCUMULUS effectively reduces RPC scheduling latency for an end-to-end application, namely MICA, under real-world traces.

A. MICA over an Altocumulus RPC system

We evaluate MICA, an in-memory key-value store [39], which is the end-to-end application evaluated in prior work, such as HERD [28], Nebula [61] and nanoPU [23]. MICA is implemented as a library with an API that allows distributed applications to GET and SET key-value pairs. We port MICA to our RPC handlers, in which requests generated by our load generator are drained from pre-allocated message buffers. We copy the descriptors of network messages between in-memory buffers and register files of ALTOCUMULUS microarchitecture. After completion, the RPC handler enqueues RPC responses to pre-allocated response message buffers. Each RPC handler follows a run-to-completion model. Each manager thread can enqueue/dequeue its `NetRX` and read

the number of waiting RPC requests within a few cycles. Message buffers are read/written at shared cache speed. We evaluate ALTOCUMULUS using two optimized network protocols, where eRPC stack lowers RPC latency down to 850 ns [27] and nanoRPC, within 40 ns [23].

B. MICA Configurations

The MICA key-value store implementation is optimized for multicore architecture with partitioned DRAM stores. We use EREW (exclusive read, exclusive write) mode of MICA, in which each core owns one key partition. EREW has the highest performance in most cases [39] because there are no concurrency control overheads, making MICA scale linearly with CPU cores. In our implementation, we map each key partition to each manager thread instead of mapping them to each core. From each manager to its local worker cores, we assume each worker core has the entire replica of the dataset and therefore it is possible to balance load to any local worker core if available. We use the default MICA hash bucket count (2M) and circular log size (4GB). We deploy an 819MB dataset owned by each manager, comprising 1.6M 16B/512B key/value pairs. Query mixes are 50/50 GET/SET. For a SET, the core loads the value to be written from the LLC [61] (i.e., a remote cache read) or the main memory [1] (i.e., a DRAM access) and then write it to the DRAM-resident MICA log. GETs first must fetch the value from the MICA log, and write it to the response message buffer, usually taking longer delay than SETs.

C. Scalability

Fig. 13(a) shows that all our evaluated configurations scale effectively with increasing number of cores under a Poisson arrival distribution and a fixed 850 ns service time per request with the eRPC stack [27]. Fig. 13(a) also evaluates real-world traffic under which commodity RSS NIC and Nebula achieve limited throughput@ SLO as they cannot adapted to varied request times and arrival patterns.

In contrast, we demonstrate our adaptability using two configurations. AC_{int_subopt} uses optimal migration parameters for synthetic traces (Sec. VIII-C), i.e., $Period$=200 ns, $Bulk$=16 and $Concurrency$=8. Although not optimal for real traffic, it still increases throughput@SLO over Nebula by a factor of 2.3 and 1.5, with 128 and 256 cores, respectively. By tuning migration parameters, AC_{int_opt} realizes near-linear scalability across core counts, achieving 2.8-7.4$\times$ throughput when the ratio of SLO violations is within 5% range. AC_{int_opt} under realistic traffic loses 13.6-15.4% throughput@SLO compared to that under synthetic traces. First, due to more complex traffic patterns, ALTOCUMULUS' prediction accuracy decreases from 99.8% to 96%. Second, since MICA is in EREW mode, our design undergoes application-level concurrency overhead as some migrated RPCs have to perform an additional remote cache access to the key's owner core.

D. Adaptability

We mix three types of RPCs: 0.5% ~50 μs SCAN, and 99.5% ~50 ns GET/SET based on nanoRPC stack [23]. The baseline (Nebula) and our 4-manager ALTOCUMULUS are all set to be 64 cores because large core count needs cross QPI bus, whose latency is detrimental for 50 ns GET/SET. We only initialize the ALTOCUMULUS runtime when throughput reaches 250 MRPS, which is the point at which Nebula starts to violate SLO. ALTOCUMULUS settings including AC_{rss}-ISA and AC_{rss}-MSR to compare our custom ISA instructions versus x86 MSR instructions to implement our mechanism.

Comparison against Nebula. Fig. 14 reports 99th percentile latency (left, logscale) and SLO violation (right) results after 10,000 RPCs complete. At low load, Nebula can maintain low 99th percentile latency within 300 ns due to the effectiveness of Nebula's NIC-managed hardware scheduler design and request prefetch. However, once throughput reaches 250 MRPS, Nebula's 99th percentile latency begins to fluctuate unpredictably and increases to 15μs, *320$\times$* worse than an average GET/SET request handling time. Fig. 14 on the right shows that up to 47% requests violate the SLO at ~300 MRPS due to head-of-line blocking in Nebula. In contrast, the ALTOCUMULUS scheduler, particularly AC_{rss}-ISA, does not experience these fluctuations and achieves a much lower 99th percentile latency We also see a more gradual increase in the rate of SLO violations up to 700 MRPS, achieving a 2.5$\times$ throughput improvement over Nebula when 99th percentile latency reaches SLO.

Comparison against kernel scheduling. Fig. 14 shows that before the ALTOCUMULUS runtime has started, our two configurations experience high tail latency. The generic RSS-based kernel-scheduler we model can cause severe queueing effects, resulting in a high 99th percentile latency of ~40μs at even low load. Software-only schedulers are inefficient at handling sub-1 μs RPCs on a manycore system, under complex arrival patterns.

Custom ISA instructions vs. MSR. Fig. 14 shows at high load, AC_{rss}-MSR reaches 91% of the max throughput that AC_{rss}-ISA delivers (for 99th percentile latency < 1 μs). AC_{rss}-ISA also provides more stable tail latency than AC_{rss}-MSR, as instructions are much faster than the ~100s cycles taken by rdmsr/wrmsr syscalls. ALTOCUMULUS sees a 2.5$\times$ higher throughput over state-of-the-art hardware-based system.

E. Case studies using ALTOCUMULUS

In this section, we present three case studies to show the versatility of ALTOCUMULUS.

Case study 1. First, we show how ALTOCUMULUS components can be used with an integrated-NIC based system, such as $Nebula$, to improve throughput@SLO. The baseline
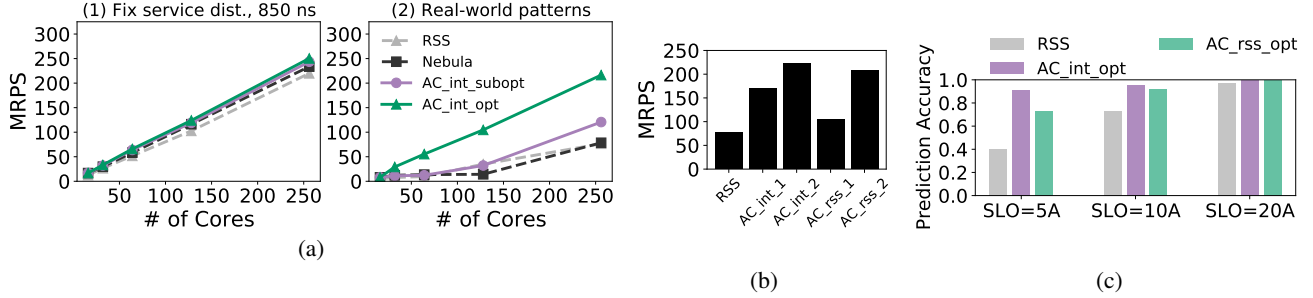
Figure 13: (a) Throughput@SLO and SLO prediction accuracy for MICA using real-world traffic patterns (256 cores). (b) Throughput@SLO for case studies 1 and 2 with the following configurations: RSS, AC_{int_rt}, AC_{int_rt+msg}, AC_{rss_syn} and AC_{rss_rw}. (c) Prediction accuracy while varying the ratio of SLO to average service time (A).
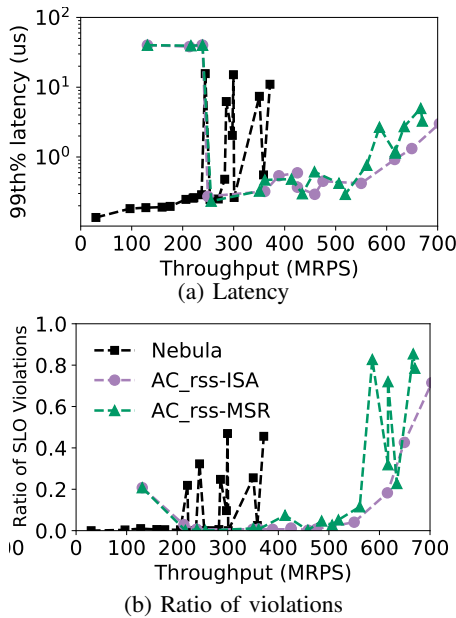

Figure 14: Nebula vs. two implementations on MICA with real-world traffic (nanoRPC, 64 cores).

for our comparison is a commodity server with RSS NIC attached via PCIe. We evaluate a scale-out 256-core $Nebula$ system, where each 16-core Nebula is a single group. We optimistically assume that all 256 cores use a single coherence domain (as required by Nebula). AC_{int_rt} integrates our decentralized runtime on each of the 16 manager cores. We also evaluate AC_{int_rt+msg} which adds both our runtime and hardware messaging on top of the scale-out Nebula system.

Fig. 13(b) shows the throughput comparison between the baseline RSS system and the two modified $Nebula$ systems. $Nebula_{rt}$ improves throughput@SLO over the baseline RSS system by 2.2×. AC_{int_rt+msg} further improves this by a factor of 1.3×. We get a best case throughput of 251 MRPS, achieving 83.3% of the ideal throughput (which is 301 MRPS for 850 ns requests with 256 cores).

Case study 2. We now show how ALTOCUMULUS' parameters can be tuned for the AC_{rss} system, which uses a commodity CPU with an RSS NIC attached via PCIe. We explore two variations of AC_{rss}, namely AC_{rss_syn} and AC_{rss_rw}, shown in Fig. 13(b). AC_{rss_syn} is tuned for synthetic traces but still achieves a 1.4× speedup over the baseline RSS system. Using tuned parameters for the real-world traffic, AC_{rss_rw} achieves a 2.7× throughput@SLO improvement. Interestingly, performance only degrades by 7% from AC_{int_rt+msg} to AC_{rss_rw}, primarily because our design is resilient to the long queuing delays that would be caused by RSS load imbalance and long PCIe latency.

Case study 3. The final case study explores the effect of changing the SLO target on prediction accuracy. Recall that, until now we have used an SLO target of $10A$, where A is the average service time. For this experiment, we also evaluate SLO=$5A$ and SLO=$20A$ with $A = 850ns$ and $load$=0.9. We evaluate using two configurations: AC_{int_opt} and AC_{rss_opt}, both tuned separately for maximum performance. Furthermore, for AC_{rss_opt} we integrate the recent feature that allows RSS to re-configure its request-to-core mapping to adapt to load imbalance, but only at a frequency of every 20 μs [7].

Fig. 13(c) shows the comparison between a baseline RSS system and the two AC systems described above. At SLO=$5A$, AC_{int_opt} and AC_{rss_opt} achieves 2.3 and 1.8× prediction accuracy increase over the baseline RSS, respectively. Our technique works better for the stricter SLO=$5A$ case as other systems are not able to effectively load balance at such a strict latency constraint.

At SLO=$10A$, AC_{rss_opt} saves 1.3× more SLO violations than the baseline RSS system. For SLO=$20A$, as the target is not demanding anymore, all approaches are able to realizes >95% prediction accuracy for this relaxed SLO target. This shows that ALTOCUMULUS is ideally suited to systems with demanding SLO targets (e.g., $<= 10A$ target) requirements, which is pivotal in modern cloud [14], [27], [53].

Through these three case studies, we provide some key

takeaways regarding our design:

1) ALTOCUMULUS is highly versatile and is able to improve throughput@SLO on a variety of systems.
2) Even for a simple baseline system such as a commodity CPU with a PCIe-attached RSS NIC, parameter tuning enables ALTOCUMULUS to reduce the impact of load imbalance and PCIe overhead caused by RSS.
3) Optimizing ALTOCUMULUS parameters for real-world traces requires tuning a few parameters (as shown in Sec. VI) to achieve high throughput.

X. RELATED WORK

We discussed RPC scheduling designs and implementations in Sec. II. We now highlight additional related work.

RPC stack optimization. DPDK [1] and PacketMill [15] reduce OS network transport overhead and memory copying via user-level specialization. eRPC [27] combines many software techniques to optimize small messages and congestion-free common cases. Recent research focuses on dedicating extra hardware to realize faster RPC stack processing. RPCValet [11] achieves NIC-core communication through shared caches, bypassing slow PCIe buses. Nebula [61] further shortens delays between messages and applications at L1 cache speed. Dagger [36] and Cerebros [52] offload software components required for commonly used RPC protocols to hardware. Optimus Prime [51] and Zerializer [65] propose data marshalling accelerators for (de)serialization ill-suited for CPUs. nanoPU [23] offers an ultra-low latency path between the NIC and the core by directly writing a message from the NIC to the core's register file.

We are the first to identify the bottleneck of RPC scheduling for μs-scale RPCs. *We expect our design to synergize with existing RPC stack optimization*: the more optimized the RPC stack, the more scheduling overhead affects the RPC throughput and latency, and *the more pivotal our design will be*.

CPU efficiency of RPC servers. Prior work implicitly acknowledges that guaranteeing μs-scale SLO comes at the cost of sacrificing CPU utilization. Existing Linux systems can only deliver μs-scale latency when keeping CPU utilization low and leaving enough idle cores available to handle incoming requests instantly [30], [37], [67]. Alternatively, by circumventing the kernel scheduler, kernel-bypass approaches such as ZygOS achieve μs-level latency at higher throughput [1], [8], [49], [50], [53]. Shenango addresses CPU wastage coming from spin-polling and core overprovisioning for peak load [48]. However, significant CPU cycles of even a small core-count CPU remain wasted when handling μs-scale requests [17], [54]. A plethora of work enables co-location of latency-critical and batch applications to improve system efficiency [17], [32], [34], [40], [42], [44], [64], [66], [67]. Our work focuses on another source of CPU waste – lack of effective and fast scheduling for μs-scale SLO. Mitigating such waste is beneficial to improve loads of latency critical requests under guaranteed μs-scale latency constraints such as RPC systems [17], [44], or leave more CPU cycles for handling batch applications as prior work proposed.

SLO and queuing theory in RPC systems. While prior work used queuing theory for load balancing in a distributed multi-server systems [33], [47], we employ it to balance loads across tens to hundreds of cores within a CPU server. Nebula [61] leveraged queuing theory to bound RPC buffer size to mitigate memory bandwidth bottleneck, while we leverage it to predict SLO violations with the help of discrete event simulation.

Architectural support for scheduling. Hardware accelerated scheduling has been proposed for traditional [10], [35], [57], [62], [63] and speculative [24] task-parallel programming models. Our messaging for scheduling is most similar to ADM [57]. However, since most scheduling designs in task-parallel systems target throughput but not latency, we augment the ADM mechanism with unique features for RPC systems. ALTOCUMULUS inherits direct register messaging for low latency RPC message transfer from nanoPU [23]. Unlike nanoPU that moves the entire message payload around, we only send the descriptor of a RPC message and thus mitigate overhead and boost migration efficiency.

XI. CONCLUSIONS & FUTURE WORK

We present ALTOCUMULUS, which uses a predictive model to migrate RPC requests that are likely to violate SLO to less busy cores. ALTOCUMULUS uses hardware to move requests at the register level and allows software a direct path to hardware via ISA extensions. The ALTOCUMULUS runtime runs as a software shim layer, whose migration messages are supported by a set of simple hardware primitives. Our two-tier scheduling scheme effectively scales to hundreds of cores for CPUs used in future datacenters.

The ALTOCUMULUS software-based hardware-assisted design opens up new opportunities in RPC systems and beyond. The flexibility provided by the ALTOCUMULUS software runtime can support a wide range of new scheduling policies, without requiring hardware or kernel scheduler modifications. In addition, our distributed software runtime offers the opportunity for isolating different applications, which we leave as a study for future work.

REFERENCES

[1] "Data Plane Development Kit. The Linux Foundation Projects." https://www.dpdk.org.

[2] "Ethernet alliance. (2020)," https://ethernetalliance.org/technology/2020-roadmap/.

[3] "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.

[4] "Marvell® octeon 10 dpu platform." https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf.

[5] "Microsoft corp. receive side scaling." http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx.

[6] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, "Mitosis: Transparently self-replicating page-tables for large-memory machines," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 283–300.

[7] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić, "RSS++: load and state-aware receive side scaling," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, 2019, pp. 318–333.

[8] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 49–65.

[9] S. Bergsma, T. Zeyl, A. Senderovich, and J. C. Beck, "Generating complex, realistic cloud workloads using recurrent neural networks," in *Proceedings of the 28th Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 376–391.

[10] E. Castillo, L. Alvarez, M. Moreto, M. Casas, E. Vallejo, J. L. Bosque, R. Beivide, and M. Valero, "Architectural support for task dependence management with flexible software scheduling," in *Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 283–295.

[11] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: NI-driven tail-aware balancing of μs-scale RPCs," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 35–48.

[12] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[13] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. d. Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 373–387.

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

[15] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, "PacketMill: toward per-core 100-Gbps networking," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 1–17.

[16] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: Smartnics in the public cloud," in *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 51–66.

[17] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 281–297.

[18] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence domain restriction on large scale systems," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015, pp. 686–698.

[19] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 3–18.

[20] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 202–215.

[21] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi, "MittOS: Supporting millisecond tail tolerance with fast rejecting SLO-aware OS interface," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 168–183.

[22] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "ghOSt: Fast & flexible user-space delegation of linux scheduling," in *Proceedings of the 28th Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 588–604.

[23] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown, "The nanoPU: A nanosecond network stack for datacenters," in *Proceedings of the 15th Symposium on Operating Systems Design and Implementation (OSDI)*, 2021, pp. 239–256.

[24] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-centric execution of speculative parallel programs," in *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, 2016, pp. 5:1–5:13.

[25] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level TCP stack for multicore systems," in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 489–502.

[26] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 345–360.

[27] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be general and fast," in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 1–16.

[28] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of the 2014 ACM SIGCOMM Conference*, 2014, pp. 295–306.

[29] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169.

[30] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, 2012, pp. 1–14.

[31] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A hardware accelerator for protocol buffers," in *Proceedings of the 54th International Symposium on Microarchitecture (MICRO)*, 2021, pp. 462–478.

[32] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, p. 729–742.

[33] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2P2: Making RPCs first-class datacenter citizens," in *Proceedings of the 2019 Annual Technical Conference (ATC)*, 2019, pp. 863–880.

[34] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi, "CuttleSys: Data-driven resource management for interactive services on reconfigurable multicores," in *Proceedings of the 53rd International Symposium on Microarchitecture (MICRO)*, 2020, pp. 650–664.

[35] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 162–173.

[36] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 36–51.

[37] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, 2014, pp. 1–14.

[38] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 476–488.

[39] H. Lim, D. Han, D. G. Andersen, and M. 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)*, 2014, pp. 429–444.

[40] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 450–462.

[41] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2005, p. 190–200.

[42] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, "Stretch: Balancing qos and throughput for colocated server workloads on smt cores," in *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 15–27.

[43] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: A microkernel approach to host networking," in *Proceedings of the 27th Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 399–413.

[44] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy, "Efficient scheduling policies for microsecond-scale tasks," in *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI)*, 2022, pp. 1–18.

[45] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proceedings of the 2018 ACM SIGCOMM Conference*, 2018, pp. 221–235.

[46] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proceedings of the 2018 ACM SIGCOMM Conference*, 2018, pp. 327–341.

[47] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "The case for rackout: Scalable data serving using rack-scale systems," in *Proceedings of the 7th Symposium on Cloud Computing (SoCC)*, 2016, pp. 182–195.

[48] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 361–378.

[49] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud storage system," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–55, 2015.

[50] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 4, pp. 1–30, 2015.

[51] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus prime: Accelerating data transformation in servers," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1203–1216.

[52] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the rpc tax in datacenters," in *Proceedings of the 54th International Symposium on Microarchitecture (MICRO)*, 2021, pp. 407–420.

[53] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 325–341.

[54] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: core-aware thread management," in *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 145–160.

[55] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun, "Elastic rss: Co-scheduling packets and cores using programmable nics," in *Proceedings of the 3rd Asia-Pacific Workshop on Networking (APNet)*, 2019, pp. 71–77.

[56] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 475–486.

[57] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 311–322.

[58] K. Sangaiah, M. Lui, R. Kuttappa, B. Taskin, and M. Hempstead, "SnackNoC: Processing in the communication layer," in *Proceedings of the 26th International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 461–473.

[59] M. Shan and O. Khan, "Accelerating concurrent priority scheduling using adaptive in-hardware task distribution in multicores," *IEEE Computer Architecture Letters*, vol. 20, no. 1, pp. 17–21, 2020.

[60] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 733–750.

[61] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis, "The NEBULA RPC-optimized architecture," in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 199–212.

[62] C. Torng, M. Wang, and C. Batten, "Asymmetry-aware work-stealing runtimes," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 40–52.

[63] M. Wang, T. Ta, L. Cheng, and C. Batten, "Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems," in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 173–186.

[64] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "Swap: Effective fine-grain management of shared last-level caches with minimum hardware support," in *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 121–132.

[65] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, "Zerializer: Towards zero-copy serialization," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS)*, 2021, pp. 206–212.

[66] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 607–618.

[67] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th European Conference on Computer Systems (EuroSys)*, 2013, pp. 379–391.