Exposing RDMA NIC Resources for Software-Defined Scheduling

Yibo Huang
University of Michigan
Ann Arbor, Michigan, USA
yiboh@umich.edu

Yiming Qiu
University of Michigan
Ann Arbor, Michigan, USA
UC Berkeley
Berkeley, California, USA
yimingq@umich.edu

Yunming Xiao
University of Michigan
Ann Arbor, Michigan, USA
yunming.xiao@u.northwestern.edu

Archit Bhatnagar
University of Michigan
Ann Arbor, Michigan, USA
architb@umich.edu

Sylvia Ratnasamy
UC Berkeley
Berkeley, USA
sylvia_ratnasamy@berkeley.edu

Ang Chen
University of Michigan
Ann Arbor, Michigan, USA
chenang@umich.edu

Abstract

Remote Direct Memory Access (RDMA) is emerging as a critical utility for large-scale datacenters, delivering significant performance improvements over the traditional TCP networking stack. Recent studies indicate that numerous applications can benefit from RDMA integration, and RDMA hardware resources are being shared among these diversifying applications. However, today's RDMA frameworks mostly view their software and hardware stacks as two independent subsystems, making it difficult for developers to align the performance objectives of RDMA applications with the limited resources in RDMA hardware.

We are developing a framework called SwiftRDMA, with the vision of enabling *software-defined RDMA scheduling*. SwiftRDMA views RDMA resource sharing as a scheduling problem. SwiftRDMA pinpoints the root causes of RDMA resource contentions and SLO violations, linking them to a set of trackable signals and controllable actions. A software scheduler then translates various operator demands into scheduling policies, which leverage the exposed signals and actions to achieve intended performance objectives. We describe our progress so far, and demonstrate the potential benefits of our approach.

CCS Concepts

• **Networks** → **Data center networks**; **Transport protocols**; • **Hardware** → *Networking hardware*; • **Software and its engineering** → *Operating systems.*

Keywords

RDMA, RNIC, software-defined scheduling, software-hardware co-design, scheduling

APNET 2025, Shang Hai, China

ACM ISBN 979-8-4007-1401-6/25/08
https://doi.org/10.1145/3735358.3735359

ACM Reference Format:
Yibo Huang, Yiming Qiu, Yunming Xiao, Archit Bhatnagar, Sylvia Ratnasamy, and Ang Chen. 2025. Exposing RDMA NIC Resources for Software-Defined Scheduling. In *9th Asia-Pacific Workshop on Networking (APNET 2025), August 07–08, 2025, Shang Hai, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3735358.3735359

1 Introduction

Remote Direct Memory Access (RDMA) has been widely adopted in modern datacenters to provide high-performance networking with minimal CPU overhead [1, 6, 7]. As the scale of RDMA deployment grows, the range of workloads that share RDMA-enabled servers has also expanded. Notable applications consist of model inference and training, distributed storage, and graph computations [2, 8, 9, 13, 36, 38], each possessing unique performance characteristics and service-level objectives (SLOs). To improve the hardware utilization, leading datacenter operators such as Alibaba and Google are colocating diverse workloads on the same end hosts [20, 31, 32]. This growing trend has led to significant concerns about the coordination of multiple workloads on shared RDMA hardware, with recent work attempting to avoid inter-workload resource contention via better performance isolation [16, 18, 29] and message scheduling [33, 39] mechanisms.

However, existing efforts lack a principled design for how RDMA software can best utilize the underlying RDMA hardware to meet diverse application performance goals. Today, application developers generally perceive RDMA hardware (e.g., RNICs) as opaque entities that are hard to control, while the hardware treats applications as non-cooperative parties vying for resources. This creates a context gap which can lead to a mismatch between an application's performance goals and how the RNIC handles the application's demands. Developers may want to cooperatively schedule applications based on their demands, thus optimizing overall performance, but the RNIC only provides performance isolation interfaces that trade efficiency for fairness; developers may have customized application quality of service (QoS) objectives [4, 20], but such intentions are oblivious to the RNIC internals. This context gap could easily introduce resource under-utilization or over-subscription.

In this position paper, we argue for a vision that we call *software-defined RDMA scheduling*. Our insight is that *RDMA resource sharing can be modeled as a scheduling problem, where applications coordinate with each other to optimize for global objectives.* As a concrete

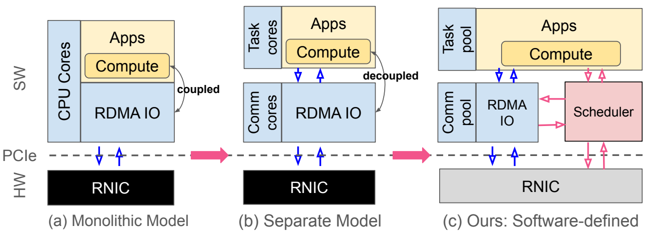

Figure 1: Existing RDMA frameworks vs. our vision—*Software-defined RDMA Scheduling.*

example, best-effort workloads can yield RNIC resources to latency-sensitive applications during traffic bursts to achieve SLO-aware scheduling. To realize this vision, RNIC should expose controllable actions (e.g., QP reuse) and useful signals (e.g., cache miss rate) to the scheduler, and the scheduler should be able to meet global (e.g., hardware utilization) and application-specific (e.g., QoS) demands by automatically tuning action knobs according to relevant signals. This would allow applications to share RNIC resources in a *context aware* and *cooperative* manner, bringing significant performance improvements while meeting customized developer requirements.

In pursuing this vision, we are faced with a set of domain-specific challenges. To begin with, we must identify why today's RDMA deployments struggle to meet various demands by pinpointing the root causes of RDMA resource contentions [14] and SLO violation. This necessitates a thorough understanding of the intrinsics related to RDMA NIC hardware. Moreover, we need to link these root causes to trackable signals and controllable actions that the software stack can use for efficient scheduling. Since we aim at commodity RDMA stack (meaning no reliance on specialized ASICs such as DPUs or FPGAs, nor changes to RNIC driver or verb semantics), we cannot easily instrument customized signals or actions out of the box. This is compounded by the fact that, unlike CPU scheduling, RDMA NICs do not expose direct actions to allocate its resource units, requiring us to systematically explore indirect ones. Finally, it is crucial for the proposed scheduler to automatically convert developer objectives (such as application QoS requirements) into hardware scheduling strategies. This enables the scheduler to effectively interpret incoming signals and swiftly enforce actions to adapt to evolving workload dynamics. This process should also remain transparent to existing applications, so that no drastic changes are needed to adopt the framework.

The rest of this paper outlines our technical roadmap to address these challenges, along with some initial evidence from our prototype implementation. Concretely, we (1) provide an architecture overview of software-defined RDMA scheduling with software-hardware co-design (§2.3); (2) give a comprehensive dissection into RNIC resource contention points, and innovatively link them to a set of novel signals and actions as the basic building blocks (§3); and (3) demonstrate a case study where software demands could be translated into hardware scheduling policies (§4).

2 Motivation

In this section, we discuss the trend of RDMA resource sharing, highlight the problems of current efforts, and outline the workflow for software-defined RDMA scheduling.

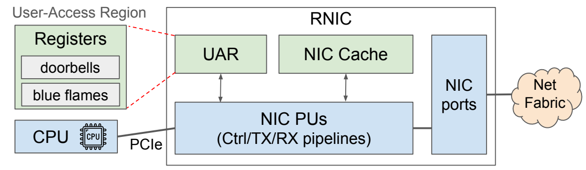

Figure 2: The limited internal resources of RDMA NIC (RNIC) subsystems and their interactions.

2.1 Background: RDMA resource sharing

To improve hardware efficiency, datacenter operators [20, 31, 32] are increasingly colocating multiple workloads on the same RDMA-enabled end hosts. These collocated workloads present diverse Service-Level Objectives (SLOs). On the one hand, high-performance data stores such as Redis [30] are typically classified as latency sensitive applications, which come with strict latency requirements (e.g., 99th percentile tail latency of $\leq 50\mu s$). On the other hand, machine learning model (e.g., DLRM [11, 24]) inference prioritizes latency-bounded throughput, demanding higher throughput under a less strict tail latency constraint (e.g.,$\leq$100ms). This is further complicated by low-priority best-effort workloads that do not have performance objectives, yet still consume large amounts of resources if not controlled.

Inherently, when colocated RDMA workloads share resources, they encounter a "noisy neighbors" issue: Colocated workloads make concurrent invocations to RDMA control verbs (e.g., `ibv_create_qp()`) and data verbs (e.g., `ibv_post_send()`), which compete for shared RNIC resource units, causing significant performance interference in terms of latency and bandwidth. As Figure 2 shows, RNIC contains many hardware components, such as NIC processing units (PU) that drive control, send, and receive pipelines, user access region (UAR) that manages doorbell and blue flame registers, NIC cache that preserves frequently accessed RDMA context metadata, and NIC ports that interact with network fabric. All of these components can independently cause resource contentions. As an example, suppose we are colocating Redis workload with DLRM inference, the latter, which requires many fan-out requests to remote parameter servers, has to establish large amounts of RDMA connections, occupying a major portion of NIC cache space and making it hard for the former to meet its SLO.

The challenge of RDMA resource sharing is well recognized in both industry and academia, particularly in multitenant scenarios, where multiple virtual machines belonging to different users are hosted on the same end host and utilize a common RNIC. This has led to extensive work on RNIC performance isolation [16, 18, 29], which attempts to guarantee fairness among tenants in terms of bandwidth or latency. These methods operate on the premise that different tenants are non-cooperative parties or even malicious competitors unaware of each other's demand. Our work instead focuses on scheduling workloads in an SLO-aware manner so that they can share RNIC resources cooperatively.

2.2 State-of-the-art & limitations

How to orchestrate the interactions between application compute logic and RDMA hardware has been a long-standing discussion. As shown in Figure 1(a), early-day RDMA applications [6, 15, 19, 21] leveraged a *monolithic model*, where the compute logic and

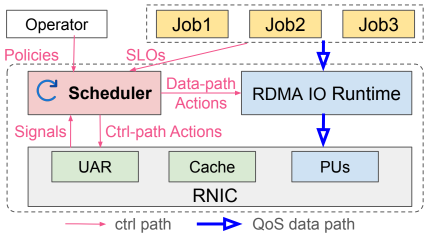

Figure 3: Software-defined RDMA scheduling.

RDMA I/O are tightly coupled on the same set of run-to-completion CPU cores. This model offers ultra-low latency as it minimizes the context switch between software and hardware, at the cost of lower resource efficiency: The RDMA I/O has to wait for the compute logic to complete, resulting in head-of-line blocking.

Consequently, modern-day RDMA application deployments [12, 13, 38] are instead dominated by what we call a *separate model* as shown in Figure 1(b), where the application compute logic runs on a set of task cores, and RDMA I/O runs on a separate set of communication cores. This allows for the efficient reuse of RDMA software resources across multiple applications, thereby removing head-of-line blocking caused by the execution of compute logic. Nonetheless, the vanilla separate model still exhibits considerable drawbacks in addressing hardware resource contentions and fulfilling application-specific needs. This stems from its inability to provide adequate coordination among application workloads, RDMA I/O and RNIC resources, leaving a substantial amount of untapped performance potential.

To improve this status quo, recent work has explored how the software stack can actively control the behavior of the RNIC hardware [33, 39], so that applications can share the underlying RNIC resources more efficiently. For instance, projects on RDMA message scheduling [33] attempted to reorder or slice work requests (WR) in the RNIC drivers so that messages with higher priority could be put into the RNIC before others. As another example, efforts on RDMA performance isolation [16, 18, 35, 40] aimed to rate limit traffic from different tenants or applications so that they do not oversubscribe to RNIC resources reserved for other parties. However, they typically require substantial changes to existing RDMA practices, either by changing the RNIC driver and the RDMA verb semantics, or by making use of hardware accelerators such as SmartNICs [40], programmable switches [18], and FPGAs [16, 35].

2.3 SwiftRDMA: An extensible framework for software-defined RDMA scheduling

As illustrated in Figure 1(c), SwiftRDMA is designed to establish a development framework in which a central scheduler orchestrates the interaction between application workloads, RDMA I/O, and RNIC resources. Figure 3 further provides an overview of the SwiftRDMA system architecture. The scheduler initially takes a set of user-defined policies (e.g., maximizing throughput or ensuring QoS) as input. It then gathers RNIC signals and application SLOs to understand the hardware contention level along with workload requirements. Once the scheduler decides that the current RNIC state or I/O runtime no longer meets user-defined policies, it invokes a set of actions to actively navigate away from the situation. Specifically,

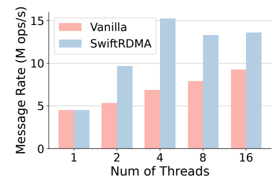

Figure 4: SwiftRDMA mitigates UAR contention (§3.1).

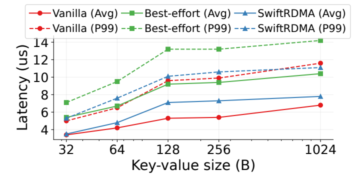

Figure 5: SwiftRDMA mitigates inter-QP RNIC port contention (§3.4).

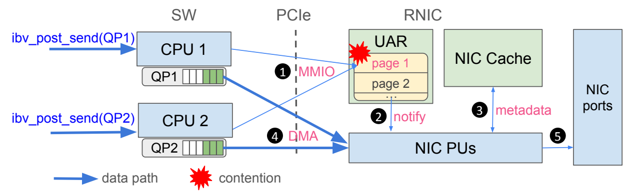

Figure 6: Data data verbs cause severe UAR contention.

the scheduler leverages RDMA control verbs (e.g., ibv_create_qp) to enforce control-path actions (e.g., load-balancing QPs across UARs), and manipulates RDMA data verbs (e.g., ibv_post_send) to roll out data-path actions (e.g., controlling enqueue rate of workloads). Together, this forms a reasoning loop that enables context-aware and cooperative RDMA resource sharing.

3 Linking RNIC Contentions to Signals and Actions

In this section, we identify 5 types of RNIC contentions and link their root causes to the corresponding trackable signals and controllable actions, leveraged as the basic building blocks of software-defined RDMA scheduling.

3.1 Cause 1: UAR Contention

User Access Region (UAR) is a critical but scarce memory region provided by RNICs. It is mapped into the software layer and allows CPUs to access RNIC resources, such as ringing DoorBells, from userspace. The UAR contains a limited number of pages. For example, only 16 UAR pages are exposed in every Mellanox ConnectX-6 RNIC device context [22]. Each time an applications uses `ibv_create_qp()` to create an RDMA Queue Pair (QP), the RNIC places the QP in a UAR page, the index of which is typically decided in a random or round-robin manner.

Root cause. Concurrent QPs sharing the same UAR page can introduce non-negligible data-path overhead when they are driven by different CPU cores. As Figure 6 shows, QP1 and QP2 share UAR page1 but are driven by CPU1 and CPU2, respectively. When QP1 and QP2 concurrently post work requests via data verb `ibv_post_send()`, their CPUs first issue Memory-Mapped IO (MMIO) write to UAR page1 to ring the DoorBell, so that RNIC PU is notified to fetch QPs' context metadata and then read Work Requests' (WR) payloads via DMA. However, this causes multiple CPUs to compete for the same UAR page protected by a lock [23], causing workload performance degradation. In our key value store benchmark, the throughput of a `get` workload is reduced by up to 57% due to UAR contention.

UAR-aware QP scheduling. Resource domain is an efficient RDMA resource management abstraction that contains a set of RDMA resources (e.g., UAR, QP) and allows them to be accessed from the same execution context. In our case, the resource domain is leveraged as a control knob to make newly created RDMA QPs bound with a specific in-RNIC UAR page. Every CPU core for RDMA IO runs over a resource domain and exclusively uses the context-specific UAR page to manage its QPs. Next, we present the initial evidence on the effectiveness of the proposed solution. Clients synthesize the key-value `get` workload with 512B key-value size over 64 RDMA QPs to key-value servers. We vary the number of threads each pined with one CPU core to evenly drive QPs. We measure the aggregate message rate (that is, `get` operations per second). The paper uses the same testbed, which includes two interconnected servers each equipped with 32 Intel Xeon CPUs, 128GB memory, and a 100Gbps Mellanox CX-5 RNIC. The vanilla case with the worst-case UAR contention is used as the baseline. As Figure 4 illustrates, SwiftRDMA outperforms vanilla by up to 2.22×, significantly mitigating the contentions over UAR.

Signals and actions. To resolve the runtime load imbalance across different UAR pages, we first identify three available signals that can be combined to indicate imbalances: (i) the congestion level (e.g., doorbell ringing) per UAR page, (ii) the number of active RDMA QPs per UAR page, and (iii) the Work Request rate (e.g., WRs per second) per QP. The first signal, which can be detected from the doorbell congestion event in RNIC event queues by enabling event reporting [22], reveals the rate of doorbell ringing on UAR pages. If this rate is higher than the rate that the RNIC can handle, then it becomes a contention source. The second signal, which can be indirectly tracked using a connection counter per resource domain, manifests the load pressure by counting the number of connections. The third signal is used mainly to identify the most overloaded QP as a target for taking actions, which can be computed from the per QP WR counter.

To mitigate the UAR load imbalance, we abstract away two critical actions: (i) assign new QPs into a UAR page with lighter load for latency sensitive workloads, and (ii) migrate an overloaded QP from high-load UAR page to low-load UAR page. For example, when a QP from latency sensitive workload is scheduled to be created, we could take the first action based on signals (i) and (ii), ensuring its service-level latency objective. When an overloaded QP is detected running on a high-load UAR page using signals (i)/(ii)/(iii), the second action could be taken to identify a low-load UAR page (using signal (i)) and then migrate the QP to it.

3.2 Cause 2: RNIC Cache Contention

The on-chip cache in RNIC is another performance-critical but scarce resource. When processing send work requests (WR) from QPs, the RNIC PU directly interacts with the cache to query the QP context metadata and memory region (MR) information from QP Context (QPC) table and Memory Translation/protection Table (MTT) respectively. A cache hit avoids the overhead of RNIC PU fetching metadata from host memory. Besides QPC and MTT, RNIC cache also needs to maintain other essential metadata, including Completion Queue Context (CQC), and Event Queue Context (EQC). However, the RNIC cache size is quite limited (the cache size of commodity Mellanox CX-5 is ~2MB[13]), and cannot scale well as the number of RDMA entities (e.g., QP, MR, CQ) grow.

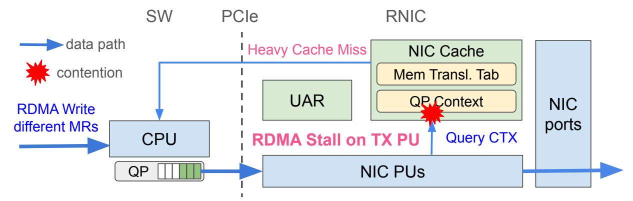

Figure 7: RDMA data verbs cause severe RNIC cache contention.

Root cause. As Figure 7 shows, suppose that a large amount of RDMA QPs and Memory Regions (MRs) have been used and overflow the RNIC cache [34]. When processing a RDMA Write WR from a QP, the PUs first attempt to query the context metadata from the NIC cache but fails, leading to large amount cache misses. The metadata have to be fetched from host via DMA read, introducing an extra PCIe round-trip. Meanwhile, RDMA PU is stalled to wait for the metadata. In one of our microbenchmarks, the throughput of the workload degrades from 96.6Gbps to 48Gbps as the cache miss rate increases from 17.2% to 49.1% [14].

Practical mitigation. RNIC cache contention mainly results from MTT and QPC cache misses. The principle of mitigating cache contention is to reduce the number of cached RDMA objects, thus reducing cache consumption. In terms of MTT cache, instead of using 4KB page size by default, huge pages (e.g., 2MB and 1GB) are well known to significantly reduce virtual-physical memory translation entries. Our framework adopts a huge page pool by default. In terms of QPC cache, QP reuse is often leveraged to reduce the number of QPs. For instance, QPs that have identical source and destination pairs might be optimized by combining them into a single QP. However, QP reuse may hurt workload SLOs by introducing queuing delays, thereby requiring a trade-off between reducing cache miss rate and optimizing queuing delay.

Signals and actions. We recognize multiple available signals to directly or indirectly indicate RNIC cache contentions: (i) QPC cache miss rate and MTT cache miss rate, (ii) the total number of active QPs, and (iii) tail latency of work request completion. The first signal, which can be queried from RNIC hardware counters using tools, shows the current contention level of the RNIC cache in terms of amount of RDMA entities. The latter two signals can be indirectly obtained through connection counter and completion events in RDMA IO runtime, pointing to global RDMA QP and work request states. Signal (iii) can help judge whether workload SLOs are violated and whether a violation stems from RNIC cache contention combined with the other two signals.

We further define two actions for cache contention mitigation: (i) QP reuse, and (ii) QP scale up. When the scheduler detects a high QPC cache miss rate (e.g., ≥30%) under total QP number exceeding a warning threshold (e.g., 512), and increased work request completion tail latency, it could take action (i) by merging multiple QPs to reduce the RNIC QPC cache contention guided by target workloads' SLOs; by contrast, when the QPC cache miss rate is low (e.g., ≤15%) with a small amount of total QPs below a safety threshold but the WR tail latency is approaching target workloads' SLOs, the scheduler could take action (ii) to create new QPs for higher

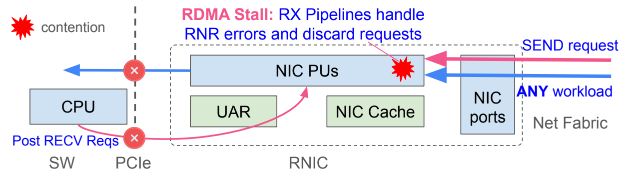

Figure 8: RX pipeline PU contention due to error handling, such as Receive-Not-Ready (RNR).

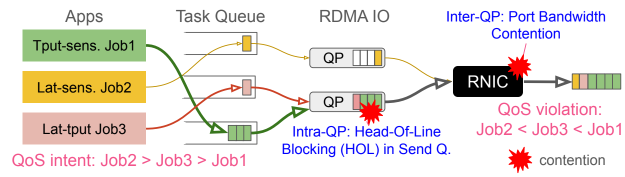

Figure 9: (1) Intra-QP send Q. contention—head-of-line (HOL) blocking, and (2) inter-QP port BW contention.

parallelism and evenly distribute tasks to them, thus reducing WR tail latency.

3.3 Cause 3: RX Pipeline PU Contention

Root cause. A third cause arises from an RNIC mechanism for handling the Receive-Not-Ready (RNR) error in SEND/RECV primitives. Specifically, when using SEND/RECV, the receiver-side CPU must post RECV requests for the target QP before the data arrives. The RNIC's RX pipeline processes these requests, storing pointers to available buffers. Upon receiving a SEND request, the RX pipeline checks for a matching RECV request. If a valid RECV request is available, the data is placed in the pre-registered buffer. However, if no RECV request has been posted in advance, the RNIC cannot accept the data, triggering an RNR error.

When an RNR error occurs, the RX pipeline PUs discard the incoming SEND request and send an RNR NACK, prompting the sender to retry after a backoff. Crucially, such error handling consumes RX pipeline processing cycles. As Figure 8 shows, this leads to resource contention and potentially stalling the entire RX pipeline. As a result, not only is the workload associated with the RNR error affected, but other workloads may also experience delays. Prior studies [14] have demonstrated that RNR on one QP can drastically reduce the bandwidth of another QP, dropping it from over 90 Gbps to just 0.018 Gbps – only 0.02% of its original bandwidth.

Signals and actions. We identify two key signals for tracking the RX pipeline PU contention: (*i*) the length of RECV queues and (*ii*) the RNR hardware counter in RNIC. The former is an indirect signal that reflects delays in processing RECV requests, where both excessively high and low values can be problematic, while the latter is a direct signal that indicates whether an RNR error has occurred.

Our action plan is as follows: If the RECV queue length falls below a predefined low threshold, it indicates a high likelihood of RNR errors due to insufficient available RECV requests. Similarly, an increasing RNR hardware counter signals that RNR errors have already occurred. In either case, we proactively allocate and post a batch of new RECV requests using `ibv_post_recv()`. Conversely, if the RECV queue length stays above a high threshold for too long, some QPs may be blocked from posting RECVs. To prevent this, we selectively remove excess RECV requests to free up resources and sustain RX pipeline health.

3.4 Cause 4: Inter & Intra QP Contention

Root cause. The final cause stems from QP contention, which can occur both within a single QP (intra-QP contention) and across multiple QPs (inter-QP contention). To illustrate this issue, we consider three jobs with different QoS requirements, where Job1 and Job3 share the same QP, while Job2 uses a separate QP, as shown in Figure 9. For simplicity, we focus on the sender side.

At the RDMA I/O level, intra-QP contention arises because a QP processes requests in a FIFO manner. Since Job1 and Job3 share the same QP, their requests are processed sequentially based on posting order, irrespective of their QoS requirements. This can lead to head-of-line (HOL) blocking, where a low-priority request delays a high-priority one, disrupting the intended QoS for both jobs.

Further contention arises within the RNIC due to inter-QP port bandwidth (BW) contention. Modern RNICs support QP-level traffic classes with priority queuing mechanisms [26], enabling differentiation between workloads with varying QoS requirements. However, our experience is that application developers often fail to properly configure or be aware of RNIC priorities, resulting in unintended QoS degradation despite the available hardware support.

Potential of proper inter-QP QoS. We evaluate the benefits of using proper inter-QP QoS to and mitigate port BW contention. The client machine generates latency-sensitive key-value `get` workloads with various key-value sizes to the server machine. Meanwhile, we continuously generate 64KB requests and responses between these two machines as background best-effort (BE) workloads. Our baselines include (1) *Vanilla* only running key-value workload, and (2) *w/ BE* running two workloads with the same traffic class (TC) of QPs by default. SwiftRDMA assigns higher TC to the QPs used in the key value workload when co-locating it with the BE workload. As Figure 5 shows, SwiftRDMA achieves near Vanilla's latency and reduces average and P99 latency than baseline *w/ BE* by up to 35% and 25%, respectively.

Signals and actions. We identify three signals for tracking the intra-QP send queue contention: (*i*) bandwidth utilization, (*ii*) queueing latency of the QP, and (*iii*) the tail latency for high-priority traffic. The first signal, which can be retrieved from hardware counters, indicates whether resources are fully utilized. For instance, low bandwidth utilization suggests that resources are underutilized, potentially pointing to send queue contention. The second and third signals require computation, where high queueing latency and increased tail latency for high-priority traffic suggest impending contention and QoS degradation. To mitigate these issues, we introduce priority task queues before applications write to the QP send queues, as illustrated in Figure 9. If bandwidth utilization remains low while queueing latency and tail latency increase, we then act to dynamically adjust the weights of traffic within the shared QP to prioritize critical workloads.

For inter-QP port bandwidth contention, we track two key signals: (*i*) bandwidth utilization and (*ii*) tail latency for high-priority traffic. Unlike intra-QP contention, high bandwidth utilization in this case signals potential port bandwidth saturation, which can lead to contention among QPs. Note that various forms of inter-QP contention exist, but port bandwidth contention – our primary focus here – only manifests when utilization is high. The second signal further confirms whether this high utilization negatively impacts high-priority workloads (*e.g.,* latency-sensitive ones). If both signals are high, our action is then to assign high-priority traffic to elevated traffic classes supported by the RNIC, ensuring better QoS differentiation.

4 Case Study: A QoS Scheduling Policy

As illustrated in Figure 3, the central scheduler is a core design of SwiftRDMA: it optimizes RDMA resource sharing by monitoring both software and hardware signals while considering application SLOs. This is achieved by enforcing pre-defined policies through adaptive actions. Policies, defined as state-action pairs, bridge application intent to concrete actions. A state is determined by job SLOs and real-time system signals, while actions involve control-path and data-path adjustments, as detailed in Section 3. Policies are created by operators (e.g., cloud providers) or users, and the scheduler enforces them to achieve the intended behavior.

We demonstrate a case study of a QoS scheduling policy. Using the setup from Section 3.4, we consider three jobs with different SLOs: throughput-sensitive (TS) job j_1, latency-sensitive (LS) job j_2, and latency-throughput-balanced (LT) job j_3. Here, the application QoS intention is $j_2 > j_3 > j_1$. Note that SLOs can be expressed in different ways, such as priority levels or concrete specifications (e.g., the 99th percentile latency should remain below $10\mu s$), as long as they align with policy definitions.

Let u_p denote the bandwidth utilization for port p and L_{qp} represent the queueing latency of QP qp. The 99th percentile latency of job j is denoted as L_j^{99}, with $L_j^{99'}$ representing its last reading. Let w_j represent job j's weight in the task queue w_j, and c_j denote its assigned traffic class.

To mitigate intra-QP head-of-line (HOL) blocking, we have the following formally-defined policy:

$$\begin{cases} \textbf{STATE:} & (TS=\{j_1\}, LS=\{j_2\}, LT=\{j_3\}) \wedge \\ & ((\exists j \in LT \cup LS, L_j^{99} > L_j^{99'}) \\ & \wedge\ (j \in p, u_p < 80\%)\ \wedge\ (j \in qp, L_{qp} > 5\mu s)) \\ \textbf{ACTION:} & w_j \leftarrow w_j + 1 \end{cases}$$

This policy first specifies job classifications and their SLOs, then outlines the example conditions under which intra-QP HOL contention is detected. When the specified state is met, the corresponding action is to increase the affected job's task queue weight to prioritize its processing. This adjustment may be applied iteratively until HOL blocking is resolved, ensuring that latency-sensitive and latency-throughput workloads take precedence over throughput-sensitive ones.

Similarly, a policy for resolving inter-QP port bandwidth contention can be written as:

$$\begin{cases} \textbf{STATE:} & (TS=\{j_1\}, LS=\{j_2\}, LT=\{j_3\}) \wedge \\ & ((\exists j \in LS, L_j^{99} > L_j^{99'})\ \wedge\ (j \in p, u_p > 80\%)) \\ \textbf{ACTION:} & c_j \leftarrow c_j + 1 \end{cases}$$

Policy compilation. The above policies should be automatically compiled from high-level application QoS intentions and contention criteria, translating abstract user-defined goals into enforceable scheduling decisions. In future work, we aim to explore this compilation process, ensuring efficient and automated policy generation. Moreover, real-world scenarios often involve multiple simultaneous contention issues in Section 3, which we plan to investigate further.

Scheduling loop. The scheduler runs in a continuous loop, where each iteration efficiently gathers runtime signals and evaluates predefined policies. It systematically checks for conditions that require action and, if met, executes the corresponding actions to adjust resource allocation. Our goal is to implement the scheduler using a single CPU core while achieving fast decision making as timely as possible.

5 Discussion

Interaction with virtualization and cluster-level scheduling. In modern datacenters, RDMA scheduling can interact with surrounding environments in complex ways. As an example, Single Root IO Virtualization (SR-IOV) [25] can affect SwiftRDMA by evenly partitioning a RNIC's resources into multiple virtual instances. If the partitions are known to be static, then SwiftRDMA can be applied seamlessly. But if the setting involves dynamic repartitioning of RNIC resources (e.g. due to container autoscaling), then its interaction with SwiftRDMA becomes an interesting direction for future works. Across machines/RNICs, cluster-level scheduling decisions may also affect the behavior of SwiftRDMA, even if SwiftRDMA focuses on lower-level scheduling. Thus, developing hierarchical techniques that integrate cluster-level scheduling with SwiftRDMA is also a promising direction for further exploration.

New scheduling policies. The software-defined RDMA scheduling framework should be extensible, supporting various policies across application workloads and RNIC hardware. In particular, as multi-path transport is increasingly adopted in AI/storage workloads [5], SwiftRDMA should enable software-defined multi-path hashing policies to enhance load balancing across RDMA links. The trend toward dual RNICs and ports in the host further introduces the need for workload-specific reliability policies, such as RNIC backup and TCP fallback [6, 17]. Extending SwiftRDMA from single RNIC scheduling to dual RNIC scheduling would be an interesting research endeavor. Moreover, SwiftRDMA should enable policies that narrow the gap between heterogeneous RNIC hardware— those from different generations or vendors. All of these go beyond the QoS guarantee policy that SwiftRDMA has tested, and we leave their development to future work.

Extending to other emerging hardware. While SwiftRDMA focuses on RDMA hardware resource scheduling, we believe that its core paradigm—software-defined scheduling for domain specific hardware resources— can be effectively extended to other emerging hardware platforms, such as XPU [28, 37], and PCIe interconnect [3, 10]. This paradigm converts hardware resource scheduling from traditional black-box modeling to a gray-box approach, substantially narrowing the context gap between low-level hardware behavior and high-level application objectives. As an example, it could turn GPU workload colocation into a scheduling problem: we start by profiling various forms of GPU contention at the microarchitecture level (e.g., streaming multiprocessors, high-bandwidth memory and GPU cache), accompanied by identifying accurate contention signals via GPU hardware counters and job-level metrics, then derive effective mitigation actions from composable CUDA programming abstractions. Therefore, we believe co-designing the software scheduler with other domain-specific hardware represents a broad direction for future exploration.

Integration with collective communication over RDMA. We reckon that as the RDMA-based collective communication frameworks (e.g., NCCL) continue to scale up in AI training and inference workloads, smarter scheduling mechanisms become increasingly crucial, especially for the purpose of addressing the RNIC contentions arising from collective operations. For example, NCCL employs several multi-GPU parallelization techniques, including those that involve multi-thread and multi-process [27] models, making SwiftRDMA well suited to mitigate related RNIC contentions and enhance the performance of collective primitives such as all-reduce. We plan to integrate SwiftRDMA into RDMA-based collective communication systems in the future.

6 Conclusion & Future Work

This paper presents SwiftRDMA, a software-defined RDMA scheduling framework that ensures co-located workloads meet their SLOs while maintaining hardware efficiency. RDMA resource management is challenging due to the opaque nature of commodity RNICs. To address them, SwiftRDMA dissects the root causes of various RNIC contentions, links them to various control signals and actions, and converts user demands into scheduling policies. The case study and preliminary results demonstrate the potential of SwiftRDMA in mitigating RNIC contention and meeting application SLOs.

For future work, we plan to: (1) define a declarative policy and task QoS programming model, (2) develop lightweight components for signal gathering, decision-making, and action enforcement, and (3) explore more use cases of extensible SwiftRDMA in a plug-and-play manner. *This work does not raise any ethical issues.*

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback and Xinhao Kong for his valuable comments. This work is partially supported by a VMware Early Career Faculty Grant, and NSF grants CNS-1942219, CNS-2106751, CNS-2107147, CNS-2214272.

References

[1] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. 2023. Empowering azure storage with {RDMA}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 49–67.

[2] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar K Panda. 2018. Accelerating tensorflow with adaptive rdma-based grpc. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 2–11.

[3] COMPUTE EXPRESS LINK CONSORTIUM, INC. 2025. *CXL® Specification.*

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

[5] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. 2024. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 57–70.

[6] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 519–533.

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

[8] Bobo Huang, Li Jin, Zhihui Lu, Ming Yan, Jie Wu, Patrick CK Hung, and Qifeng Tang. 2019. RDMA-driven MongoDB: An approach of RDMA enhanced NoSQL paradigm for large-Scale data processing. *Information Sciences* 502 (2019), 376–393.

[9] Bobo Huang, Li Jin, ZhiHui Lu, Xin Zhou, Jie Wu, Qifeng Tang, and Patrick CK Hung. 2019. BoR: Toward high-performance permissioned blockchain in RDMA-enabled network. *IEEE Transactions on Services Computing* 13, 2 (2019), 301–313.

[10] Yibo Huang, Yukai Huang, Ming Yan, Jiayu Hu, Cunming Liang, Yang Xu, Wenxiong Zou, Yiming Zhang, Rui Zhang, Chunpu Huang, et al. 2022. An ultra-low latency and compatible PCIe interconnect for rack-scale communication. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*. 232–244.

[11] Yibo Huang, Zhenning Yang, Jiarong Xing, Yi Dai, Yiming Qiu, Dingming Wu, Fan Lai, and Ang Chen. 2024. Disaggregating Embedding Recommendation Systems with FlexEMR. *arXiv preprint arXiv:2410.12794* (2024).

[12] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.

[13] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16.

[14] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. 2023. Understanding {RDMA} microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 31–48.

[15] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, et al. 2023. Flor: An open high performance {RDMA} framework over heterogeneous {RNICs}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 931–948.

[16] Yunkun Liao, Jingya Wu, Wenyan Lu, Xiaowei Li, and Guihai Yan. 2023. Optimize the TX Architecture of RDMA NIC for Performance Isolation in the Cloud Environment. In *Proceedings of the Great Lakes Symposium on VLSI 2023*. 29–35.

[17] Shengkai Lin, Qinwei Yang, Zengyin Yang, Yuchuan Wang, and Shizhen Zhao. 2024. LubeRDMA: A Fail-safe Mechanism of RDMA. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*. 16–22.

[18] Jiaqi Lou, Xinhao Kong, Jinghan Huang, Wei Bai, Nam Sung Kim, and Danyang Zhuo. 2024. Harmonic: Hardware-assisted {RDMA} Performance Isolation for Public Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1479–1496.

[19] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. 2019. X-RDMA: Effective RDMA middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.

[20] Artemiy Margaritov, Siddharth Gupta, Rekai Gonzalez-Alberquilla, and Boris Grot. 2019. Stretch: Balancing qos and throughput for colocated server workloads on smt cores. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 15–27.

[21] Mellanox. [n. d.]. libVMA: Linux user space library for network socket acceleration based on RDMA compatible network adaptors. https://github.com/Mellanox/libvma.

[22] Mellanox. 2020. Mellanox Adapters Programmer's Reference Manual (PRM). https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf.

[23] Mellanox. 2023. NVIDIA MLNX_OFED Linux Drivers. https://network.nvidia.com/products/infiniband-drivers/linux/mlnx_ofed/.

[24] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean

Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[25] Nvidia. 2023. Single Root IO Virtualization (SR-IOV). https://docs.nvidia.com/networking/display/ofed510660/single+root+io+virtualization+(sr-iov).

[26] NVIDIA. 2024. Quality of Service (QoS) in NVIDIA MLNX_OFED Documentation. https://docs.nvidia.com/networking/display/mlnxofedv497100lts/quality+of+service+(qos).

[27] NVIDIA Corporation. 2025. *NVIDIA Collective Communications Library (NCCL).*

[28] NVIDIA Corporation. 2025. *NVIDIA H100 Tensor Core GPU.*

[29] Dian Shen, Junzhou Luo, Fang Dong, Xiaolin Guo, Kai Wang, and John CS Lui. 2020. Distributed and optimal rdma resource scheduling in shared data center networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 606–615.

[30] valkey.io. 2024. Valkey: A flexible distributed key-value datastore that is optimized for caching and other realtime workloads. https://github.com/valkey-io/valkey.

[31] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems*. 1–17.

[32] Kangjin Wang, Ying Li, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou, Jie Yao, et al. 2022. Characterizing job microarchitectural profiles at scale: Dataset and analysis. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[33] Xizheng Wang, Shuai Wang, and Dan Li. 2023. sRDMA: A General and Low-Overhead Scheduler for RDMA. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*. 21–27.

[34] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. 2023. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1–14.

[35] Zilong Wang, Xinchen Wan, Luyang Li, Yijun Sun, Peng Xie, Xin Wei, Qingsong Ning, Junxue Zhang, and Kai Chen. 2024. Fast, Scalable, and Accurate Rate Limiter for RDMA NICs. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 568–580.

[36] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Xstore: Fast rdma-based ordered key-value store using remote learned cache. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–32.

[37] Yunming Xiao, Diman Zad Tootaghaj, Aditya Dhakal, Lianjie Cao, Puneet Sharma, and Aleksandar Kuzmanovic. 2024. Conspirator:{SmartNIC-Aided} Control Plane for Distributed {ML} Workloads. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 767–784.

[38] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. 2019. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[39] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. 2022. Justitia: Software {Multi-Tenancy} in Hardware {Kernel-Bypass} Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1307–1326.

[40] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. [n. d.]. White-Boxing RDMA with Packet-Granular Software Control. ([n. d.]).