The NIC Needs an Operating System

Wonsup Yoon (UT Austin) Daehyeok Kim (UT Austin)

Abstract

Modern NICs have evolved into shared, multi-tenant computers, integrating fixed-function and programmable accelerators over common on-chip memory, compute, DMA engines, and interconnect. As cloud providers increasingly offload stateful data-plane functions onto these accelerators, they also introduce a new class of runtime resource-contention problems. Our measurements on NVIDIA ConnectX-7 show that contention over shared on-chip memory inflates a co-located tenant's packet-processing tail latency by up to 2.2×, even when the interfering workload executes on a different accelerator. We argue that the root cause is architectural: existing NIC control planes provision accelerator state but lack an OS-like resource-management layer that observes, arbitrates, and manages shared resources at runtime.

We propose NicOS, a NIC-native resource-management architecture that separates reusable hardware mechanisms from operator-defined policies. To make the vision concrete, we instantiate NicOS for on-chip memory with *Accelerator Virtual Memory (AVM)*, which models memory as a shared pool of named consumers and exposes a common observe-and-act interface across accelerators. A software prototype on NVIDIA BlueField-3 demonstrates feasibility; even emulated in software, NicOS mitigates the resulting tail-latency inflation. We conclude by discussing the hardware primitives and research challenges this vision requires.

1 Introduction

The NIC is no longer a simple packet-delivery device; it has become a *small, shared computer* inside the network path. Modern NICs contain fixed-function accelerators, programmable processors, on-chip memory, DMA engines, and interconnects, all multiplexed across tenants. Cloud providers increasingly move stateful data-plane functions, *e.g.*, network virtualization, IPSec, RDMA, off the host CPU and onto these on-path accelerators [15, 16, 19] (*e.g.,* NVIDIA ConnectX [13], Broadcom Thor2 [4], AMD Pollara [3], Intel IPU [10]).

Consolidating many tenants onto one chip makes them share its physical resources, and that sharing is not free. Our measurements on NVIDIA ConnectX-7 (CX-7) NICs show the consequence of this sharing. Once a co-located tenant's working set exceeds the shared on-chip memory's capacity, an otherwise-unchanged victim's packet-processing latency inflates by up to 2.21×, regardless of whether the neighbor runs on the same accelerator or a different one. For latency-sensitive datacenter applications such as key-value stores [24, 29], RPC [25], and disaggregated memory and storage [27, 36], this NIC-side jitter can inflate end-to-end tail latency [21, 23, 34, 35].

This "noisy-neighbor" problem has been long studied in the cloud [11, 20, 33], but it has received little attention on the NIC, despite its performance impact. We expect it to become more prevalent as more cloud operators adopt programmable NICs [1, 2, 14], where custom functions can be offloaded onto shared on-chip compute and memory.

A natural question is whether today's NIC software stack already provides the mechanisms needed to manage this contention. It was built to configure state, not to manage the resources that state contends for. Operators set up flow rules, RDMA context, and security associations on fixed-function engines, or deploy code on programmable engines such as CX-7's data-path accelerator (DPA). These interfaces report what was configured, not which tenant is consuming shared memory, compute, or interconnect at runtime, and they offer no common view of that consumption across accelerators. Reconstructing such a view on the host, after the fact, is too indirect and too slow, since the contention arises inside the NIC's silicon.

The root cause is the absence of a layer that arbitrates the NIC's shared resources at runtime. Provisioning resources ahead of time is no longer sufficient once multiple tenants dynamically share limited hardware resources under changing workloads and deployment-specific objectives. At that point, resource allocation becomes a runtime systems problem rather than a hardware configuration problem. CPU-based systems crossed this boundary decades ago, which is why operating systems and hypervisors arbitrate shared resources, even the last-level cache [37], instead of leaving those decisions to fixed hardware policies. Modern NICs now face the same architectural shift: many tenants dynamically share on-chip memory, compute, and interconnect, yet contention is still resolved by whatever implicit cache, queue, or eviction rule the hardware enforces. Although NIC firmware performs device-specific control, it largely implements static vendor-defined policies rather than dynamically managing the resources shared across fixed-function and programmable accelerators.

This calls for a runtime resource-management layer that separates reusable hardware mechanisms from operator-defined policy. We propose **NicOS**, a NIC-native OS architecture for multi-tenant resource management. NIC vendors expose reusable observability and actuation mechanisms once, making shared NIC resources visible and controllable across

accelerators. Operators write policies on top, expressing priorities, guaranteed minimums, caps, and fairness without asking the vendor to bake each one into silicon. These policies run on the NIC because contention must be handled close to the resource, at the silicon boundary where it arises.

We make the vision concrete for on-chip memory, with *Accelerator Virtual Memory (AVM)*. AVM is not the whole NicOS design; it is one instantiation of the abstraction and interface a NicOS-managed resource needs. It models the NIC's on-chip memory as a single pool of named consumers spanning every accelerator, and exposes *observe* and *act* interfaces that let policy report and relieve contention over that namespace. Because the namespace names state by owner and object, *e.g.,* a flow table, IPSec security association, or RDMA address translation table, the same policy can cover intra- and inter-accelerator contention.

We demonstrate feasibility with a software prototype on an NVIDIA BlueField-3 equipped with CX-7. The prototype emulates the observe and act mechanisms with NVIDIA's DOCA API [6], steering low-priority traffic off contested on-chip memory and onto on-NIC ARM cores whose state lives in on-NIC DRAM. Even this software emulation cuts a victim's tail latency by 1.4×, and hardware-native support would recover more of the inflation.

We conclude with the open questions we must answer to realize this vision (§8): what belongs in NIC silicon, how the abstraction generalizes across the NIC's other shared resources, and how operators should program and safely compose the policies that run on it.

2 Noisy-Neighbor Problem on the NIC

The NIC's shared resources span on-chip memory, compute, and the interconnects between them. We focus on on-chip memory, which is fundamentally constrained. It is implemented as small, fast SRAM that keeps the data path at line rate, and because SRAM is expensive and die area is limited, that same small pool has to be shared across every accelerator on the chip. On-chip memory is also present in virtually every NIC, making it the most broadly relevant resource and the one we can measure on commodity silicon today.

2.1 Measurement on NVIDIA CX-7 NICs

We study NVIDIA ConnectX-7 (CX-7), a widely deployed NIC whose silicon is also embedded in the BlueField-3 DPU. On CX-7 silicon, every accelerator's context is held in a single, shared on-chip memory, the Interconnect Context Memory (ICM) cache. This context includes flow entries, IPSec security associations, and RDMA address translation entries, among others. The context is installed through the control-plane API into host or on-NIC DRAM and cached in the ICM cache on first access.

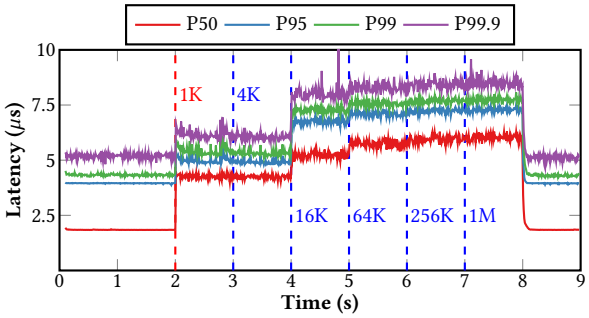

Figure 1: Primary's packet-processing latency over time, while the neighbor's traffic also runs through HWS. The neighbor's traffic starts at 2 seconds, and its flow count increases every second.

Measurement setup. We target two accelerators: (1) **Hardware Steering (HWS)** [9], a programmable match-action packet-processing subsystem used to execute a network function (NF) chain (Firewall → NAT → Metering), and (2) **Data Path Accelerator (DPA)** [8], comprising 16 programmable RISC-V cores (256 hardware threads in total) capable of executing user-defined C programs.

We pair two tenant workloads: a latency-sensitive *primary* tenant, whose working set fits comfortably in the ICM cache, and a *neighbor* tenant whose ICM footprint we scale. These two tenants meet in two scenarios: intra-accelerator contention, where HWS processes both tenants' traffic, and inter-accelerator contention, where HWS carries only the primary's traffic while the neighbor's workload runs on the DPA, randomly accessing its allocated memory region.

We drive the traffic workloads with an open-loop generator whose inter-packet gaps follow a Poisson process, running on a separate server connected to the target CX-7 NIC. To measure the neighbor's impact on the primary tenant, we record each primary packet's processing latency using CX-7's hardware timestamping.

Intra-accelerator. We fix the primary's flow count at 7000, small enough that its flow entries fit comfortably in the ICM cache, while varying the neighbor's flow count from a thousand to a million. The primary and neighbor offer loads of 12 Gbps and 84 Gbps, respectively. As Figure 1 shows, the primary's latency rises as the neighbor's flow count grows, and its p99 inflates by up to 3.3 µs (1.64×). We confirm the cause with DOCA's telemetry API [7], which shows the ICM cache miss rate climbing over the same interval as the neighbor's entries evict the primary's.

Inter-accelerator. Here, the primary remains the same HWS tenant as in the previous experiment, while the neighbor instead runs a DPA workload in which 128 threads randomly access an allocated memory region. The memory region accessed from DPA threads varies from 10 MB to

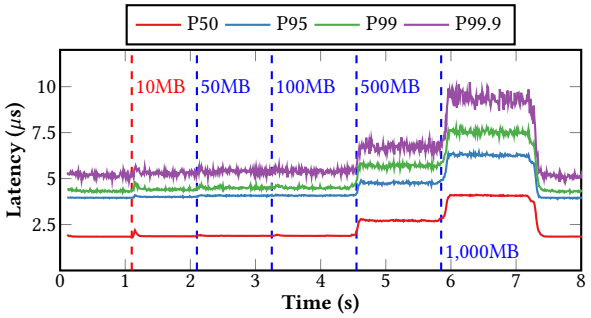

Figure 2: Primary's packet-processing latency over time, while the neighbor runs a DPA workload. The DPA threads start at 1 second, and the size of the memory region they access grows every 10 M accesses.

1000 MB. As Figure 2 shows, the primary's p99 latency rises by up to 4.9 µs (2.21×). The ICM miss rate again climbs in step, confirming that even a neighbor on a different accelerator evicts the primary's entries from the shared cache.

Key takeaways. These experiments show that on-chip memory contention is not confined to tenants sharing the same accelerator. Whether the neighbor's workload runs through the same HWS pipeline or a different engine, it contends for the same physical SRAM, evicting the primary's flow entries and inflating both median and tail latency. As NICs consolidate more accelerators, longer network-function chains, and larger working sets onto the same chip, contention for these shared resources is likely to intensify, further amplifying this latency inflation. Even today, the inflation reaches several microseconds, precisely the regime where latency-sensitive systems have already engineered away much of the host-side jitter [21, 23, 34, 35]. As a result, the NIC itself increasingly becomes the bottleneck determining end-to-end tail latency.

2.2 Missing Abstraction and Interface

Existing mechanisms for multi-tenant NICs do not yet address this case. Prior work either statically partitions programmable NIC resources [22] or schedules and multiplexes programmable functions [26, 30, 31], but the contention we observe arises underneath these abstractions, in the physical resources shared across fixed-function and programmable accelerators.

The natural alternative would be to rely on the NIC's existing control-plane interfaces. However, these interfaces were designed to provision accelerator state, not to manage the shared physical resources that state contends for at runtime. Operators install, delete, and cap flow rules, RDMA queue pairs, and security associations on fixed-function accelerators, while DPA programs allocate and free memory directly. What occupies the on-chip memory, however, is determined by the hot working set rather than the configured state: many cold rules may consume little cache space, while a small hot workload can dominate it. Without direct visibility into the runtime usage of these shared resources, software can only infer contention indirectly through coarse proxies.

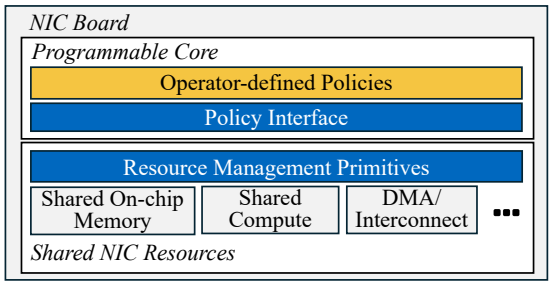

Figure 3: NicOS architecture. The blue components comprise NicOS. Operator-defined policies run over the NicOS policy interface, while NIC vendors provide resource-management primitives for the shared NIC resources.

Even a runtime view is insufficient if every accelerator keeps a private namespace. An operator can ask HWS about flow-table state or the DPA about allocated memory regions, but no common operation can ask whether tenant B's DPA state is evicting tenant A's HWS entries, because the two engines do not name their consumers in the same space. Bridging this gap requires a resource-management layer with a shared abstraction for the physical resource, a common namespace for its consumers, and actions that can change placement or allocation wherever contention arises.

3 NicOS Vision

We propose NicOS as the NIC-resident resource-management layer this gap calls for, governing the shared on-chip resources across tenants and accelerators (Figure 3). Like a conventional operating system, NicOS separates *mechanism* from *policy* [17, 18]. Rather than embedding a single resource-management policy into the NIC, NicOS exposes reusable mechanisms over which operators define their own policies. Vendors expose reusable resource-management primitives that make shared resources observable and controllable through a common abstraction, while operators implement NIC-resident policies that consume those primitives to express priorities, guarantees, caps, and fairness objectives. This separation allows the same mechanisms to support many deployment-specific resource-management policies.

Policies execute on a small programmable runtime integrated on the NIC so they can observe and react to contention where it arises. Running policy at the NIC avoids the latency of involving the host while keeping the underlying mechanisms reusable across deployments. For example, one deployment may protect the tail latency of latency-critical HWS traffic, while another prioritizes weighted sharing or caps a DPA workload's memory footprint, all using the same underlying mechanisms.

At the heart of NicOS is a resource abstraction that names *consumers* of a physical resource rather than private accelerator structures. For on-chip memory, a consumer might be one tenant's HWS flow table, DPA memory region, IPSec state, or RDMA translation state. Once policies address these consumers through a shared namespace, the same control loop can observe contention, identify who is affected and who is responsible, and choose an appropriate action. The action depends on the resource: placement and allocation for on-chip memory, scheduling for compute, and arbitration for DMA and interconnect.

In §4, we instantiate NicOS for on-chip memory as one concrete realization of this architecture.

4 Instantiating NicOS for On-Chip Memory

Separating mechanism from policy requires a common abstraction over the shared physical resource, one that both vendor-provided mechanisms and operator-defined policies can observe and control. We instantiate this abstraction for on-chip memory with *Accelerator Virtual Memory (AVM)*. On-chip memory is a natural first target because it is present in virtually every NIC, shared across accelerators, and, as shown in §2, already exhibits measurable cross-accelerator contention on commodity hardware.

4.1 Resource Abstraction

AVM models the NIC's on-chip memory as a single shared resource rather than as accelerator-specific structures. This abstraction follows directly from the measurements in §2: regardless of whether contention originates from HWS, DPA, or another accelerator, it ultimately competes for the same physical SRAM. Treating memory as separate HWS flow tables, DPA heaps, or RDMA tables therefore obscures the resource actually under contention.

Instead, AVM views each tenant's state on an accelerator as a *consumer* of a shared memory pool. Each consumer occupies some fraction of the pool, referred to as its *occupancy*. The object being managed is therefore the shared memory itself rather than the accelerator-specific structures that happen to reside in it. This unified view allows one policy to reason uniformly across tenants and accelerators.

4.2 State Namespace

To manage these consumers, policies need a common naming scheme independent of any particular accelerator. AVM therefore addresses each consumer using a `(tenant, object)` selector, where `tenant` identifies the owner of the state and `object` identifies the accelerator-visible structure containing it, *e.g.*, `hws.flows`, `dpa.mem`, `ipsec.sa`, or `rdma.mtt`. Selectors also support wildcards, such as `("A", *)` for all of tenant A's consumers or `(*, "hws.flows")` for all HWS flow tables.

Primitive	Meaning
observe signals — read as `ev.read(sig, sel)`	
`occupancy`	Share of pool capacity the consumer holds
`misses`	Miss rate the consumer incurs
`demand`	The consumer's access rate / working-set churn
`p99`	The consumer's tail latency
observe queries — invoked with an explicit argument	
`contention(pool)`	Aggregate pressure on a shared pool, *e.g.*, its overall miss rate
`evictor(sel)`	The consumer displacing victim `sel`
act operations — invoked on a selector	
`pin(sel)`	*Placement:* hold resident in the fast tier
`demote(sel, tier)`	*Placement:* move to a slower tier as pressure builds
`promote(sel, tier)`	*Placement:* restore to a faster tier once pressure clears
`cap(sel, frac)`	*Allocation:* bound the share of the pool the consumer may hold
`weight(sel, w)`	*Allocation:* share the pool proportionally among competitors

Table 1: NicOS's observe signals, queries, and act operations.

The object is the natural management granularity because it preserves the distinction between a tenant's different accelerator states without exposing accelerator-specific implementation details. For example, a policy can independently manage tenant A's HWS flow table and DPA memory while remaining agnostic to how either accelerator organizes its internal state.

4.3 Mechanism and Policy Interface

Mechanism. Given the resource abstraction and namespace, AVM requires a small set of hardware primitives that observe and control each consumer at runtime. The *observe* primitives expose per-consumer signals such as occupancy, misses, and latency, much as CPU cache-monitoring hardware does per core [37]. The *act* primitives control two aspects of a consumer: its *placement*, determining which memory tier holds its state, and its *allocation*, determining how much of the shared pool it may occupy.

Policy interface. Policies are event-driven programs over `(tenant, object)` selectors. Table 1 summarizes the proposed interface. A policy registers handlers over observed signals (*e.g.*, `@nicos.on(pred)`); when triggered, it queries additional resource state through the *observe* interface and invokes *act* operations to adapt placement or allocation. Selectors may refer to an individual consumer or a wildcarded group, allowing policies to react to pool-wide contention before identifying and managing the responsible consumers.

The interface intentionally exposes policy-level operations rather than accelerator-specific controls. Each accelerator implements these operations through its own driver. For example, `demote()` may steer an HWS flow table to a software path, remap a DPA memory region to DRAM, or park RDMA state in a slower tier. Policies therefore operate over a single abstraction while accelerator-specific implementation details remain hidden behind the driver.

4.4 Example Policies

The following examples illustrate how the same policy interface naturally expresses both intra- and inter-accelerator resource-management policies.

Intra-accelerator contention. The policy in Listing 1 mitigates the intra-accelerator contention shown in §2. It protects the primary tenant while allowing lower-priority tenants to use the fast memory tier whenever capacity is available. When contention arises, the policy progressively demotes lower-priority tenants' flow tables to on-NIC DRAM until the primary's latency SLO is restored. Unlike static partitioning, memory is allocated dynamically according to runtime demand rather than fixed reservations.

Listing 1: Demoting lower-priority tenants' flow tables to protect the primary tenant.

```
@nicos.on(contention(ICM) > τ) # ICM pool contended
def protect_hws(ev):
    vip = ("primary", "hws.flows") # protected tenant
    for t in LOW_TO_HIGH: # tenants, low priority first
        if ev.read("p99", vip) <= SLO: # primary recovered?
            break
        nicos.demote((t, "hws.flows"), tier=DRAM)
```

Inter-accelerator contention. The inter-accelerator scenario in §2 uses the same policy interface but addresses contention spanning multiple accelerators. Because all consumers share a common namespace, a policy can identify the specific consumer responsible for contention even when the victim and the culprit reside on different accelerators. For example, if tenant B's DPA memory evicts tenant A's HWS flow entries, the policy caps only tenant B's memory consumer rather than throttling all DPA traffic. The policy is written once and reused across accelerators; only the consumer selector changes.

5 A Software Prototype

We built a software prototype on a BlueField-3 (BF-3) [12], which combines CX-7 silicon with on-NIC ARM cores and DRAM. The NicOS *manager*, the agent that runs the operator's policy, executes on BF-3's ARM cores, faster to react than an equivalent host implementation. CX-7 lacks the two hardware primitives NicOS's design calls for, so we emulate them with existing DOCA APIs [6]: its telemetry API [7] stands in for the observability hook, reporting the ICM cache's hit and miss rates, and its flow API [5] stands in for the placement hook, steering traffic between the HWS and ARM paths. We implement the intra-accelerator mitigation policy of Listing 1, using the DSCP field to classify traffic into priority classes.

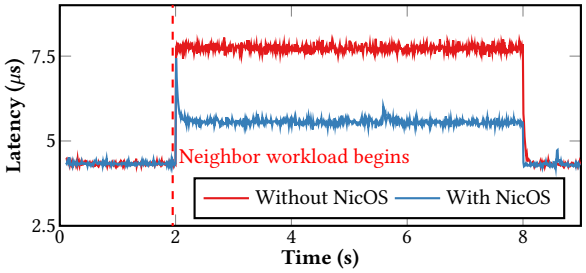

Figure 4: Primary tenant's p99 latency over time. Without NicOS (red), on-chip memory contention inflates the tail; with NicOS (blue), steering the neighbor to the ARM path recovers most of it.

Observability. The manager reads DOCA's aggregate ICM cache miss rate and treats sustained high misses as memory contention. The signal is coarse but reflects our observability primitive, since evictions and ICM cache misses rise together as memory pressure grows. Being aggregate, however, it tells NicOS that the pool is contended but not which consumer is responsible.

Placement. To relocate the low-priority class's state out of the contested SRAM, the HWS NF chain is replicated on an on-chip hardware path and an ARM path whose state lives in on-NIC DRAM. As the placement primitive, NicOS steers lower-priority classes to the ARM path as contention rises and promotes them back as it clears, moving whole classes between two fixed paths. We realize this steering with an indirection table in HWS, where each class maps to a single entry that forwards it to the HWS or ARM path. A hardware-native placement primitive that could pin, demote, and promote individual consumers across memory tiers would let policy act at the granularity where contention arises.

6 Preliminary Evaluation

We evaluate the effectiveness of our prototype by measuring its latency reduction, reaction time, and resource overhead.

Tail-latency reduction. We repeat the intra-accelerator contention experiment of §2, fixing the neighbor's flow count at 2 million and assigning the primary tenant high priority and the neighbor low priority. Figure 4 shows how the primary's p99 latency evolves over time, with and without NicOS. NicOS reduces the primary's p99 latency by 2.2 µs (1.4×) relative to the unmanaged baseline.

Residual I/O contention. Ideally the primary's latency would return to its pre-contention baseline once NicOS steers

the neighbor away, but Figure 4 shows it settles about 1.1 µs above it. NicOS removes the on-chip memory contention, yet other shared resources remain contended, and the on-chip RX/TX path is one of them. Its hardware I/O scheduler and per-class receive buffers provide only best-effort fairness across co-located traffic, which NicOS's memory management does not address. This residual motivates extending NicOS to the other shared resources.

Reaction time. The time from contention onset to applied steering is 152 µs, of which roughly 50 µs is the telemetry query and 5.3 µs the flow-entry update over a kernel-bypass path; the remaining delay comes from the software control loop on the ARM cores and DOCA API overhead. Actuation is relatively cheap, while the cost lies in observing contention and running policy in software. Hiding the latency spikes during this reaction window will require hardware-native observe-and-act.

Manager footprint. The manager runs on a single ARM core, and its two-level indirection table occupies negligible on-chip memory, so the manager itself does not become a noisy neighbor.

7 Related Work

Programmable NIC resource management. Prior work on NIC multi-tenancy targets programmable NICs. FairNIC [22] statically partitions programmable cores, cache, and accelerators across tenants, while PANIC [30] and SuperNIC [31] schedule and isolate offload chains on programmable SmartNICs. OSMOSIS [26] goes further, dynamically multiplexing a programmable SmartNIC's compute and I/O among tenants under tenant-level QoS. All of these manage resources *within* the programmable execution environment. NicOS instead targets the physical resources shared *underneath* both fixed-function and programmable accelerators, exposing one abstraction and mechanism/policy interface over shared on-chip memory, compute, DMA, and interconnect, independent of any single accelerator's implementation.

RDMA performance isolation. Another line of work studies performance interference in RDMA. It characterizes the NIC's hidden microarchitectural resources [28], enforces isolation at a programmable PCIe switch [32], or shapes interfering traffic in host-side software [38]. These efforts focus on the RDMA engine, whereas our measurements show contention crossing accelerators through the shared on-chip memory. NicOS therefore treats these shared physical resources as first-class objects rather than reasoning about each accelerator in isolation.

8 Future Directions

We conclude by outlining several open research questions toward realizing the NicOS vision.

Memory architecture. Today's NICs largely treat on-chip SRAM as a hardware-managed cache for accelerator state, such as CX-7's ICM cache. Our measurements suggest that this caching model may become increasingly inefficient as NICs consolidate heterogeneous accelerators with rapidly changing working sets. NicOS instead exposes placement primitives that naturally support treating on-chip SRAM and host or on-NIC DRAM as an explicitly managed memory hierarchy. Should future NICs then continue to rely primarily on hardware-managed caches, or should memory residency become a software-managed resource?

Beyond memory. We instantiate NicOS for on-chip memory, where the abstraction is a shared resource pool, a common namespace over its consumers, and observe and act primitives that let policy adjust placement and allocation at runtime. The same pattern should extend to the NIC's other shared resources. Programmable compute could be exposed as dynamically allocatable execution contexts rather than statically assigned cores, DMA bandwidth as a schedulable resource rather than a fixed reservation, and on-chip interconnect as an arbitrable medium across competing traffic classes. For each, the open problem is finding the minimal mechanisms that enable such runtime management without baking deployment-specific policy into silicon.

Another challenge is that these resources are not independent. Relieving contention on one often shifts it to another, since moving state from SRAM to DRAM eases memory pressure but adds DMA traffic, and throttling compute frees memory at the cost of throughput. Should a NIC then manage each resource independently, or coordinate them across the chip, balancing latency, throughput, fairness, and utilization without becoming prohibitively complex?

Policy programming model. This paper sketches one possible policy interface through event-driven observe-and-act programs, but it is unlikely to be the only viable model. What is the right abstraction for expressing NIC resource-management policies? Should policies specify what resource objectives to achieve, or how resources should be managed? How expressive can policies be while still meeting the NIC's tight latency and compute budget and remaining predictable enough that a misbehaving policy cannot destabilize the data path? The policy programming model may matter as much as the underlying hardware mechanisms.

Policy composition. Multiple policies may coexist over the same mechanisms, from vendor policies for safety to operator policies for deployment goals, and eventually per-tenant policies. How should they compose, and how should conflicts be resolved while vendor policies still guarantee safe operation? Just as operating systems evolved from fixed schedulers to extensible resource managers, NicOS raises the question of safe composition between vendor mechanisms and deployment-specific policies.

References

[1] Agilio CX SmartNICs - Netronome. https://www.netronome.com/products/agilio-cx/. Accessed: 2026-07-01.

[2] Alveo U50 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html. Accessed: 2026-07-01.

[3] AMD Pensando Pollara 400 AI NIC. https://www.amd.com/en/products/network-interface-cards/pensando.html. Accessed: 2026-07-01.

[4] Broadcom Thor2 (BCM57608) 400G Ethernet NIC. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/p1400g. Accessed: 2026-07-01.

[5] DOCA Flow. https://networking-docs.nvidia.com/doca/archive/3-4-0/doca-flow. Accessed: 2026-07-01.

[6] DOCA Library APIs - NVIDIA Docs. https://docs.nvidia.com/doca/api/3-4-0/doca-libraries-api/index.html. Accessed: 2026-07-01.

[7] DOCA Telemetry Diag. https://networking-docs.nvidia.com/doca/archive/3-4-0/doca-telemetry-diag. Accessed: 2026-07-01.

[8] DPA Subsystem | DOCA. https://networking-docs.nvidia.com/doca/archive/3-4-0/dpa-subsystem. Accessed: 2026-07-01.

[9] Hardware Steering (HWS). https://networking-docs.nvidia.com/linuxkernelpatchforconnectx8andconnectx9/hardware-steering-hws. Accessed: 2026-07-01.

[10] Intel Infrastructure Processing Unit (Intel IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html. Accessed: 2026-07-01.

[11] Noisy Neighbor antipattern | Microsoft Learn. https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor. Accessed: 2026-07-01.

[12] NVIDIA BlueField-3 Networking Platform. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield. Accessed: 2026-07-01.

[13] NVIDIA ConnectX-7 NIC. https://www.nvidia.com/en-us/networking/ethernet-adapters/. Accessed: 2026-07-01.

[14] The next wave of Google Cloud infrastructure innovation: New C3 VM and Hyperdisk. https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu. Accessed: 2026-07-01.

[15] Alibaba Cloud Community. A Detailed Explanation about Alibaba Cloud CIPU. https://www.alibabacloud.com/blog/a-detailed-explanation-about-alibaba-cloud-cipu_599183. Accessed: 2026-07-01.

[16] Amazon Web Services. AWS Nitro System. https://aws.amazon.com/ec2/nitro/. Accessed: 2026-07-01.

[17] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. In *ACM SOSP*, 1995.

[18] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *ACM SOSP*, 1995.

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *USENIX NSDI*, 2018.

[20] Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. Tuna: Tuning unstable and noisy cloud applications. In *ACM EuroSys*, 2025.

[21] Joshua Fried, Idiris Angelopoulos, David E. Culler, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *USENIX OSDI*, 2020.

[22] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *ACM SIGCOMM*, 2020.

[23] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX NSDI*, 2019.

[24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM*, 2014.

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be general and fast. In *USENIX NSDI*, 2019.

[26] Mikhail Khalilov, Marcin Chrapek, Siyuan Shen, Alessandro Vezzu, Thomas Benz, Salvatore Di Girolamo, Timo Schneider, Daniele De Sensi, Luca Benini, and Torsten Hoefler. OSMOSIS: Enabling multi-tenancy in datacenter SmartNICs. In *USENIX ATC*, 2024.

[27] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote flash $\approx$ local flash. In *ACM ASPLOS*, 2017.

[28] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA microarchitecture resources for performance isolation. In *USENIX NSDI*, 2023.

[29] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI*, 2014.

[30] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *USENIX OSDI*, 2020.

[31] Will Lin, Yizhou Shan, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. Supernic: An fpga-based, cloud-oriented smartnic. In *ACM/SIGDA FPGA*, 2024.

[32] Jiaqi Lou, Xinhao Kong, Jinghan Huang, Wei Bai, Nam Sung Kim, and Danyang Zhuo. Harmonic: Hardware-assisted RDMA performance isolation for public clouds. In *USENIX NSDI*, 2024.

[33] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *USENIX OSDI*, 2018.

[34] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX NSDI*, 2019.

[35] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *ACM SOSP*, 2017.

[36] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM:High-performance,application-integrated far memory. In *USENIX OSDI*, 2020.

[37] Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A closer look at Intel Resource Director Technology (RDT). In *ACM RTNS*, 2022.

[38] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software multi-tenancy in hardware kernel-bypass networks. In *USENIX NSDI*, 2022.