LogNIC: A High-Level Performance Model for SmartNICs

Zerui Guo
University of Wisconsin-Madison
USA

Jiaxin Lin
The University of Texas at Austin
USA

Yuebin Bai
Beihang University
China

Daehyeok Kim
The University of Texas at Austin and Microsoft
USA

Michael Swift
University of Wisconsin-Madison
USA

Aditya Akella
The University of Texas at Austin
USA

Ming Liu
University of Wisconsin-Madison
USA

ABSTRACT

SmartNICs have become an indispensable communication fabric and computing substrate in today's data centers and enterprise clusters, providing in-network computing capabilities for traversed packets and benefiting a range of applications across the system stack. Building an efficient SmartNIC-assisted solution is generally non-trivial and tedious as it requires programmers to understand the SmartNIC architecture, refactor application logic to match the device's capabilities and limitations, and correlate an application execution with traffic characteristics. A high-level SmartNIC performance model can decouple the underlying SmartNIC hardware device from its offloaded software implementations and execution contexts, thereby drastically simplifying and facilitating the development process. However, prior architectural models can hardly be applied due to their limited capabilities in dissecting the SmartNIC-offloaded program's complexity, capturing the nondeterministic overlapping between computation and I/O, and perceiving diverse traffic profiles.

This paper presents the **LogNIC** model that systematically analyzes the performance characteristics of a SmartNIC-offloaded program. Unlike conventional execution flow-based modeling, LogNIC employs a *packet-centric* approach that examines SmartNIC execution based on how packets traverse heterogeneous computing domains, on-/off-chip interconnects, and memory subsystems. It abstracts away the low-level device details, represents a deployed program as an execution graph, retains a handful of configurable parameters, and generates latency/throughput estimation for a given traffic profile. It further exposes a couple of extensions to handle multi-tenancy, traffic interleaving, and accelerator peculiarity. We demonstrate the LogNIC model's capabilities using both commodity SmartNICs and an academic prototype under five application scenarios. Our evaluations show that LogNIC can estimate performance bounds, explore software optimization strategies, and provide guidelines for new hardware designs.

CCS CONCEPTS

• **Hardware → Networking hardware**; • **Networks → Network performance modeling**.

KEYWORDS

SmartNIC, Architectural Modeling, Programmable Networks

ACM Reference Format:
Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. 2023. LogNIC: A High-Level Performance Model for SmartNICs. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3614291

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614291

1 INTRODUCTION

The last few years have seen a rising number of SmartNICs [1, 3, 5, 7, 9, 12, 14] deployed in the public clouds, enterprise clusters, and edge data centers [4, 23, 25, 26, 37, 61]. Recent surveys [22, 27] from Dell'Oro project that the SmartNIC market size will exceed 1.6 billion US dollars by 2026, occupying more than one-third of the total network adapter market. SmartNICs, staying on the communication path, provide in-network computing capabilities for traversed packets, making the network data-plane more reconfigurable and flexible at the server edge. Researchers have built a series of SmartNIC-assisted solutions for different system layers–such as networking stack [30, 64, 72], NFV and network middleboxes [33, 37, 50, 53], distributed applications [35, 52, 56, 59, 66, 71, 75], and disaggregated storage [38, 47, 61, 62, 78]–and demonstrated considerable latency savings, throughput increases, and overall energy/cost efficiency improvements.

However, it is notoriously challenging and overly arduous to build an efficient SmartNIC-assisted system. First, a SmartNIC is an accelerator-rich heterogeneous device incorporating diverse on-chip/off-chip interconnects (Figures 1 and 8). To maximize the offloading benefits, one should develop a systematic characterization and understanding of its capabilities and limitations. For example, the actual throughput of a SmartNIC off-chip accelerator

(e.g., a compression engine) depends on its execution parallelism, aggregated interconnect bandwidth between NIC cores and the engine, and the delivery rate of submission/completion signals. Second, one should carefully refactor the application logic across the heterogeneous components in a SmartNIC, such as figuring out the execution model (e.g., run-to-completion v.s. pipeline), partitioning the data layout, redesigning data structures, and adapting communication interfaces pertaining to data transfer patterns. Further, the problem is exacerbated when considering diverse traffic profiles (e.g., packet/flow size distribution), cross-platform portability, and multi-tenancy. Therefore, the development process is tedious and usually involves many design-implement-test iterations.

We argue that a high-level performance model–that decouples the underlying SmartNIC hardware device from its offloaded software implementations and execution contexts–will help simplify and facilitate the SmartNIC development process. Such a model could allow programmers to easily explore various design spaces, identify application performance bottlenecks and suggest optimization opportunities, enhance system portability, and even provide early-stage insights/guidances on next-generation SmartNIC design.

The architecture community has a long history of developing systematic and analytical models for different computing systems [28, 34, 40, 41, 73, 76]. However, we observe that none of them can be effectively applied in the context of SmartNIC computing. First, a SmartNIC-offloaded program consists of multiple functional eclectic kernels that present different computing and I/O intensities, while existing models mainly focus on one application type. Second, unlike other computing substrates, the compounding effect among on-/off-chip interconnects, heterogeneous execution engines, and memory subsystems, complicates analyzing the data movement and execution flow on a SmartNIC. Third, the performance analysis is tightly coupled with traffic profiles. This significantly contrasts assumptions on existing models where the system input is fixed and well-defined.

In this paper, we propose the **LogNIC** model that systematically analyzes the performance characteristics of a SmartNIC-offloaded program under a given traffic profile. Our key idea is a *packet-centric modeling approach* that examines SmartNIC execution behaviors based on how packets transmit over different hardware entities, in contrast to existing models which center around execution flow. LogNIC consists of four major components: (1) system interface, where we abstract away the SmartNIC device details using a hardware model, represent an offloaded program as a software execution graph, and encode execution contexts as model parameters; (2) throughput modeling, estimating the achievable throughput by considering both the computing capacity of each triggered hardware engine and bandwidth limits of traversed interconnects/memory; (3) latency modeling that captures the execution latency of a compute engine, data movement overheads, and internal queueing effect; (4) model extensions that handle mixed execution graphs, interleaved traffic, and specialized accelerators. Overall, LogNIC takes an application execution graph and a set of device/traffic parameters as inputs, and operates in either (a) an estimation mode that outputs the expected latency and throughput of the given SmartNIC program under a traffic profile; or (b) an optimizer mode that generates a subset of parameters that can satisfy the stipulated performance bounds.

We evaluate the LogNIC model using three commodity SmartNICs (i.e., Marvell 25GbE LiquidIO-II CN2360 SmartNIC, NVIDIA/ Mellanox 100GbE BlueField-2 DPU, and Broadcom 100GbE Stingray PS1100R) and an academic prototype (PANIC [55]) under five application scenarios. We first validate the model accuracy by estimating the performance bounds, and then use the model to explore software optimization strategies and guide new hardware design. Our results are summarized as follows:

- In terms of model validation, LogNIC delivers accurate throughput estimation of six bump-in-the-wire acceleration scenarios on the LiquidIO-II SmartNIC and identifies their performance bottleneck on the data path. We also use the model to estimate the latency of an NVMe-oF target process on the Broadcom Stingray. Our results show that LogNIC can estimate the latency for three different I/O patterns with less than a 1% error rate.
- When using the LogNIC model to optimize SmartNIC-offloaded programs, it helps tune the execution parallelism of Microservice-based applications on the LiquidIO-II card and decide the network function placement on the BlueField2. For example, using LogNIC suggested parallelism, one can achieve up to 36.4% throughput increase and 22.8% latency saving across five Microservice workloads when compared with heuristics-based approaches.
- LogNIC can also be used for hardware design space exploration. For example, we use it to determine an optimal hardware resource provision of PANIC for three scenarios: sizing the request queue of an accelerator, steering traffic at the central scheduler, and configuring the hardware parallelism.

2 MOTIVATION

This section first describes the hardware architecture of a SmartNIC and highlights contemporary SmartNIC use cases. We then discuss the need for a high-level performance model and why existing models are inadequate.

2.1 SmartNIC Hardware Architecture

SmartNICs extend a traditional NIC with extra computation capabilities using CPUs, FPGAs, or other domain-specific accelerators (e.g., crypto engines). A SmartNIC can execute general-purpose or domain-specific programs closer to network I/Os than the host CPU. Figure 1 presents a high-level overview of the SmartNIC architecture, which captures most of today's SmartNICs regardless of the computing substrates (programmable ASIC, Multicore SoC, FPGA) and deployment models (inline or bump-in-the-wire). A SmartNIC, staying on the packet communication path, mainly consists of the following four functional components:

- **Parser/Deparser**: It (1) analyzes the packet header based on a predefined parsing graph and generates necessary metadata upon an incoming packet and (2) rebuilds the packet header when a packet leaves the pipeline. Packets could come from both RX/TX ports and the host PCIe bus.
- **RX/TX pipeline**: It contains a set of match action engines and performs packet-level header manipulations. The enclosed SRAM or TCAM table stores thousands of matching rules with fast lookup support. Some SmartNICs [3, 7, 12, 14] also provide a

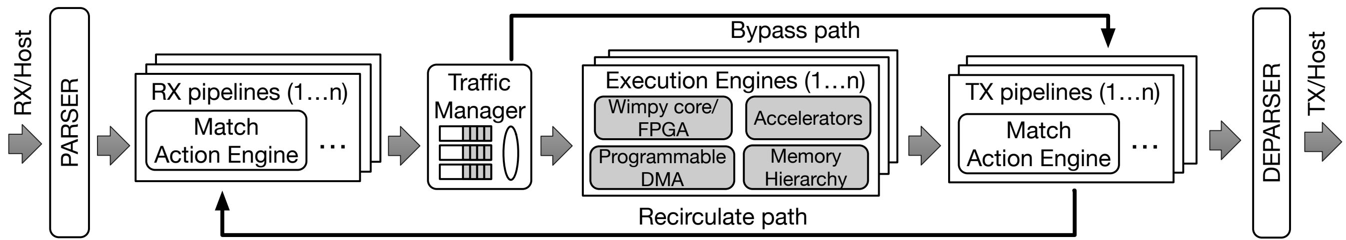


Figure 1: An overview of the SmartNIC architecture.

recirculating path such that a packet can reenter the processing pipeline to gain more execution cycles.

- **Traffic manager or NIC switch**: It performs QoS traffic control and steers traffic to the NIC engine. Some SmartNICs are also equipped with a bypassing path that can forward packets directly to the TX pipeline. This feature is used when a packet requires no deep packet inspection or flow/application-level computation.
- **Execution engine or intellectual property (IP) block**: It performs complicated packet processing using a general-purpose wimpy multi-core processor (such as Mellanox BlueField [12], Broadcom Stingray [1], Marvell LiquidIO [7], Fungible FAC [3]), embedded flow engines (like Pensando DSC [14], Netronome Agilio [9]), FPGAs [18–21], or domain-specific accelerators. Its memory hierarchy usually encloses the core-local L1 cache, self-managed scratchpad, L2 cache and DRAM. Some SmartNICs also hold programmable DMA engines [7, 14, 21].

There are two types of SmartNICs: (1) *On-path* SmartNICs have execution engines on the packet communication path and handle all inbound/outbound traffic between Ethernet ports and the PCIe host. Examples include LiquidIO, Agilio, Pensando DSC, and Fungible FAC; (2) *Off-path* SmartNICs, which expose themselves as a second network endpoint. It can deliver traffic flows to host CPUs directly (without entering the SmartNIC SoC) based on forwarding rules installed on a *NIC switch*. Packets targeting the host go from the traffic manager to TX pipelines via the bypass path (Figure 1), while the rest follow the default path and would trigger execution engines on the SmartNIC. BlueField and Stingray are examples of off-path SmartNICs. NIC vendors for both types are also improving the programmability of the NIC switch. SmartNICs can also be categorized based on the computing substrate: (1) *Pipeline NICs* place multiple offloads in a pipeline to enable packets to be processed by a chain of functions [20, 37]; (2) *Manycore NICs* load balance packets across many embedded CPU cores which control the processing of packets as needed for different offloads [1, 7, 9, 12]; (3) *RMT NICs* use the on-NIC reconfigurable match+action (RMT) pipeline to implement NIC offloading [9, 14].

2.2 SmartNIC-assisted Systems

SmartNICs have become a new computing substrate at end-hosts, assisting their computation. Many SmartNIC-assisted systems have been developed across different layers recently.

- **Networking stack and NFV:** ClickNP [53] proposes a modular framework to implement network functions over an FPGA SmartNIC. AccelNet [37] deploys the virtual networking layer onto SmartNICs and provides an SDN interface for the control-plane. AccelTCP [64] partitions the kernel monolithic TCP stack, and offloads connection management functionalities to the NIC. FlexTOE [72] accelerates the SmartNIC TCP performance via fine-grained data-path parallelization and segment reordering.
- **Application layer:** KV-Direct [52] extends RDMA primitives with vector operations to build a high-throughput in-memory key-value store. E3 [59] exploits the use of Multicore-SoC SmartNICs for energy-efficient Microservice execution. Xenic [71] develops a SmartNIC-accelerated distributed transaction system via an asynchronous aggregated execution model, flexible point-to-point communication strategies, and a NIC-side customized data store. Floem [66] applies a data-flow programming model that simplifies design space explorations. iPipe [56] uses an actor framework to offload distributed applications and employs a hybrid scheduler to optimize the tail latency. NICA [35] proposes an ikernel abstraction (tightly coupled with the network stack) and uses it to deploy application logic on FPGA-based SmartNICs. hXDP [33] then develops a compiler that parallelizes and translates eBPF bytecode to an FPGA SmartNIC. Researchers [75] characterize an off-path SmartNIC from the communication-path perspective and derive an optimization guideline to help designers exploit multiple paths of SmartNICs when offloading distributed applications.
- **Storage layer:** LineFS [47] decomposes the distributed file system operations into execution stages and offloads CPU-intensive tasks to a customized NIC pipeline. Gimbal [62] models the SmartNIC as a software storage switch that provides performance isolation for multiple co-located tenants through a couple of networking techniques. LEED [38] builds an energy-efficient, distributed, and persistent key-value store over SmartNIC JBOFs. It tackles their architectural imbalance via two design principles: (a) trading excessive I/O bandwidth for scarce SmartNIC core computing cycles and memory capacity; and (b) making scheduling decisions as early as possible to streamline the request execution flow.

Cloud providers further deploy infrastructure management services over SmartNICs to reduce the server virtualization overheads and enable bare-metal clouds [23–25].

2.3 The Need for a New Performance Model

While SmartNICs can bring us many benefits, building an efficient SmartNIC-assisted system is notoriously challenging and overly arduous. In particular, programmers should first develop a solid understanding and detailed characterization of the underlying SmartNIC hardware architecture. For example, as shown in our case study (§4.2), the actual throughput of a SmartNIC off-chip compression engine depends not only on its execution parallelism but also on

Architecture Model	Target Domain
LogP [34]	Parallel algorithms for MPP machines
Roofline [76]	Compute kernels for multi-core/many-core processors
LogCA [28]	Compute kernels for domain-specific accelerator
Gables [40]	Smartphone applications for mobile SoC
Accelerometer [73]	Microservices for domain-specific accelerators
LogNIC	**In-network computing for SmartNICs**

Table 1: A list of architecture models.

the aggregated interconnect bandwidth between NIC cores and the accelerator, and the delivery rate of submission/completion signals. Second, one has to carefully refactor the application logic across heterogeneous computing engines, on-/off-chip interconnects, and memory subsystems, and explore an optimal offloading strategy based on their capabilities and limitations (§4.4, §4.5). Third, we need to dissect the relationship between SmartNIC execution behavior and traffic characteristics. For example, an optimized implementation (targeting MTU-sized traffic) would show inferior performance when running under 64B packets. Such a development process is tedious and involves many design-implement-test iterations. We argue that a high-level performance model, which decouples the underlying SmartNIC device from the actual software implementation and execution contexts, could significantly facilitate this process with the following four benefits:

- **Performance analysis.** It is vital to estimate how much latency and throughput an offloaded program can achieve and what the potential bottlenecks are. A high-level performance model can deliver such hints without actually deploying the program. It can show how the application behaves when varying the traffic input.
- **Design space exploration.** A model can abstract the SmartNIC as a parameterized hardware graph, where developers explore various design options by splitting the application logic and mapping it over the hardware graph. In this case, one could figure out the execution model (e.g., run-to-completion v.s. pipeline), redesign data structures and partition the data layout, and adapt communication interfaces pertaining to data transfer patterns.
- **Guiding new hardware design.** SmartNIC vendors continuously upgrade existing IP blocks or add new hardware features to accelerate certain workloads. Instead of hinging on a cycle-level simulator or prototyping the actual hardware design, one could use the model to answer some early-stage questions, such as whether it's worthwhile to integrate an IP into the SoC, where it should be placed, and how many expected performance improvements.
- **Implementation portability.** SmartNICs embody great architectural diversity (§2.1), presenting different capabilities in terms of execution parallelism across various IPs, memory subsystem performance, and interconnect speed. A performance model could estimate how a ported application behaves on the target SmartNIC, whether there are potential bottlenecks on the data path, and how to refactor the software design to maximize performance.

2.4 Inadequacies of Existing Models

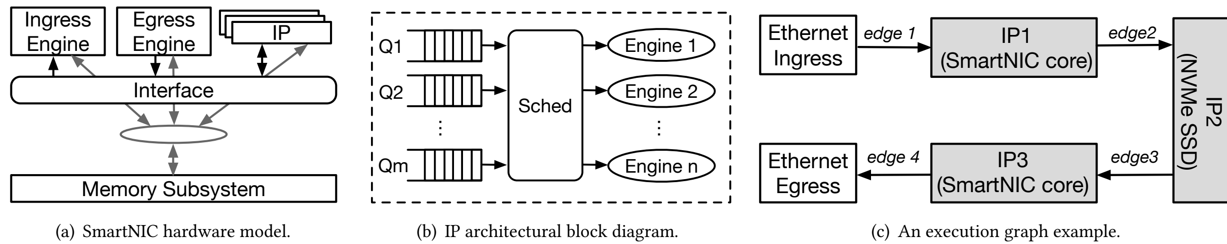
The computer architecture community has proposed a series of analytical models (Table 1) to understand the performance bounds and explore SW/HW optimizations for various computing systems [28, 34, 40, 41, 73, 76]. The LogP [34] model characterizes a parallel machine using four parameters: the number of processors (P), the communication bandwidth (g), the communication delay (L), and the communication overhead (o). The Roofline model [76] estimates the performance bound of a compute kernel running on a multi-core or many-core processor. LogCA [28] suggests which optimizations may alleviate the performance bottlenecks of general-purpose hardware accelerators. The Gables model [40] refines and retargets the Roofline to analyze how an application kernel executes on a mobile SoC. Accelerometer [73], extending the LogCA to analyze concurrency-induced performance bounds, quantifies the impact of hardware acceleration on microservice-based workloads. However, none of these models can capture the execution behavior of a SmartNIC-assisted program and analyze its performance characteristics for the following three reasons:

- **The complexity of a SmartNIC-offloaded program.** A SmartNIC program consists of multiple functional eclectic kernels that present different execution and I/O intensities, such as packet header manipulation, transport protocol processing, and per-packet or per-message or per-flow domain-specific execution. Hence, the model should describe both the characteristics of each individual kernel and their interactions, whilst existing models mainly focus on one compute-intensive or memory-intensive workload across one or multiple hardware substrates. The Gables [40] model is the closest one that might be applicable to our scenario, but it fails at capturing the I/O behavior of an IP.
- **The nondeterministic overlapping between computation and I/O.** Differing from many other accelerators, a SmartNIC stays close to the packet communication path, consumes all or partial incoming traffic based on stipulated flow rules, and triggers predefined packet-dependent computations over various IP blocks. There are two types of I/Os affecting the SmartNIC behavior: external I/Os (from wire and PCIe) and internal I/Os (happening within the SoC). In terms of memory, there are several non-cache-coherent regions (e.g., scratchpad SRAM, TCAM, packet buffer, DRAM) that allow passing packet contents, data-plane metadata, as well as temporary computing results. The compounding effect between diverse I/O interconnects and heterogeneous parallel execution engines complicates modeling the execution flow over a SmartNIC, whereas existing models shed little light on this.
- **The agnosticism of traffic profiles.** The performance characteristics of a SmartNIC application highly depend on the incoming traffic profile, such as packet size, inter-frame gap, flow size distribution, and burst degree. In contrast, existing models (such as LogCA [28] and Accelerometer [73]) assume that the system input is fixed and deterministic. Further, the problem could be even exacerbated by traffic-induced different execution paths. For example, a firewall module can be realized by either a match-action table or a regular expression engine based on traffic demand, embodying different performance bottlenecks.

3 THE LOGNIC MODEL

3.1 Overview

LogNIC is an analytical model that takes SmartNIC hardware devices, software application implementations, and traffic profiles in predefined formats as inputs and generates performance estimation through white-box modeling (Figure 4-a). LogNIC consists of the



(a) SmartNIC hardware model. (b) IP architectural block diagram. (c) An execution graph example.

Figure 2: The hardware and software representation of LogNIC. Figure 2(c) depicts a SmartNIC-offloaded NVMe-oF target program, where edges 1/4 go through the SoC interconnect and edges 2/3 use SoC interconnect and DRAM.

following four major components: (1) **system interface**, abstracting away the SmartNIC device details using a hardware model (§3.2), represents a SmartNIC program as a software execution graph (§3.3), and encodes execution input/contexts as model parameters (§3.4); (2) **throughput modeling**, estimating the achievable throughput by considering both the computing capacity of each triggered hardware engine and bandwidth limits of traversed interconnects/memory based on the execution graph (§3.5); (3) **latency modeling**, which captures not only the execution latency at a compute engine but also data movement overheads, and internal queueing effect (§3.6); (4) **model extension**, which handles mixed execution graphs due to multi-tenancy, interleaved traffic, and specialized accelerators (§3.7). §3.8 will describe the overall workflow.

Key Idea. LogNIC employs a new *packet-centric modeling* technique, a variant of data flow modeling [31, 74]. It analyzes the SmartNIC execution by examining how packets transmit over different hardware entities (e.g., IP blocks, on-/off-chip interconnects, and non-cache-coherent memory regions). In LogNIC, a SmartNIC-offloaded program is abstracted as a directed acyclic graph (DAG) where packets flow through, and its edges/vertices are the hardware components. This is in contrast to prior *computation-centric* modeling approaches [28, 34, 40, 73, 76], where they view compute units as first-class citizens and analyze system performance by tracing the execution flow, i.e., chaining running activities of all IP blocks on the performance path.

3.2 Hardware Model of a SmartNIC

LogNIC models a SmartNIC using four architectural components (Figure 2(a)): (1) ingress/egress engines, which move I/O traffic between wire/PCIe and the SmartNIC; (2) N intellectual property (IP) blocks, which could be a general-purpose CPU, a domain-specific accelerator, a digital signal processor (DSP), or even an onboard GPU; (3) an interface (adopted from LogCA [28] and Gables [40]), the communication layer among different IPs; (4) the memory subsystem with an interface shared by IPs, such as DRAM, scratchpad, SRAM, and TCAM. The ingress/egress component includes one or multiple I/O ports (or channels/lanes) used for data transmission. All IPs operate concurrently and communicate with each other via either the interface or memory. Similar to prior work [28, 73], LogNIC uses the interface abstraction to provide the intuition for the performance and communication overhead of moving data across different IPs, which can be mapped to any high-bandwidth on-chip interconnect. To facilitate inter-IP data movements via an external DRAM (if the device employs one), we assume that the DRAM provides multiple-megabyte buffering/rate-matching capabilities.

IP Architecture. We model each IP block as Figure 2(b) including m input queues, a request scheduler, and n parallel execution engines. Each queue has k entries. The scheduler works in a (weighted) round-robin fashion and dispatches a request when an IP engine becomes available. The actual execution parallelism (i.e., the number of concurrent active engines) would impact the queueing delay. Each engine operates independently and shares the bandwidth to/from the interface and memory. Since a physical IP would be shared by different stages of one program or multiple applications (§3.7), we partition an IP into multiple virtual instances (virtual IPs), where each has m' queues (with k' entries per queue) and n' engines. This can be achieved via either temporal or spatial multiplexing, where the model provides a parameter.

Extended Roofline of an IP.. We repurpose the Roofline [76] model to profile the computing performance of ingress/egress and IPs given its capability to analyze parallel engines. The conventional Roofline model determines the attainable performance based on (1) the arithmetic intensity of the kernel, defined as the number of floating-point operations per byte of memory traffic; (2) the DRAM bandwidth, capping the system peak throughput and denoting the performance ceiling. We make two extensions to capture IPs in a SmartNIC. First, LogNIC adds multiple bandwidth ceilings to represent input data from different sources, such as SoC interconnects or memory hierarchy. Second, we replace the arithmetic intensity parameter with the *packet intensity*, defined as the number of executed IP-specific operations (which could be floating-point arithmetic computations for a CPU/GPU, checksum computations for a crypto accelerator, or packet manipulations for a match-action engine) per packet transmission, which is also size-dependent. The Roofline of a virtual IP can be obtained in the same way.

3.3 Software Representation of a Program

Execution Graph. LogNIC lets programmers express a SmartNIC offloaded programs as a directed graph $G=(V,E)$. A vertex (v_i) represents an IP or an ingress/egress engine, and an edge (e_{ij}) represents the data movement from v_i to v_j via a communication medium (which could be either interface or memory subsystem). When data traverses more than one medium type, such as the SmartNIC core

Category	Parameter	Symbol	Description	Source
Hardware	Interface bandwidth	BW_{INTF}	The maximum communication bandwidth over an interface	SPEC
	Memory bandwidth	BW_{MEM}	The maximum data transfer rate over a memory hierarchy	SPEC
	IP-IP bandwidth	BW_{mn}	The communication bandwidth between two IPs	CHAR
Software	Data transfer ratio	$\delta_{e_{ij}}$	The relative data transfer percentage across an edge e_{ij}	CONF
	Edge medium usage	$\alpha_{e_{ij}}/\beta_{e_{ij}}$	The bandwidth usage over an edge e_{ij} via interface/memory	CONF
	Ingress granularity	g_{in}	The data transfer granularity at an ingress engine	CONF
	Overhead	O_i	The computation transfer overhead from a node to the next one	CHAR
	Node partition	γ_{v_i}	The multiplexing percentage of an execution engine v_i	CONF
	IP throughput	P_{v_i}	The computing throughput of a physical IP node	CHAR
	IP parallelism degree	D_{v_i}	The parallelism of a (virtual) IP node in the execution graph	CONF
	IP queue capacity	N_{v_i}	The queue capacity of a (virtual) IP node in the execution graph	CONF
Traffic	Ingress bandwidth	BW_{in}	The data serving rate to the SmartNIC	CONF
	Packet size distribution	$dist_{size}$	The packet size distribution of the incoming traffic	CONF
Output	Throughput	$P_{attainable}$	The estimated throughput of the target application on the SmartNIC	N/A
	Latency	$T_{attainable}$	The estimated latency of the target application on the SmartNIC	N/A

Table 2: Description of the LogNIC parameters. SPEC=Specification. CHAR=Characterization. Conf=Configurable. Specification/Configurable/Characterization means the data source comes from the hardware manual, user inputs, and profiling.

issuing a DMA read that copies data from DRAM to an accelerator through PCIe, its performance parameters should consider all types. Hence, we divide a SmartNIC program into a series of computing kernels that run across various IPs.

An Example. Consider an NVMe-oF target application over the Broadcom Stingray PS1100R SmartNIC [1]. The program (1) receives RDMA packets from an ingress Ethernet port, (2) executes the NVMe-over-RDMA target-side protocol and fabricates NVMe commands at the NIC core, (3) issues NVMe reads/writes and waits for responses from the SSD drive, (4) builds NVMe-oF response packets and forwards them to the egress port. Its execution graph is shown in Figure 2(c), where IP1 and IP3 are both SmartNIC cores, and are responsible for submission and completion path handling, respectively. Some SmartNICs (such as Fungible FAC200) employ a hardware-accelerated NVMe-oF implementation, whose IPs in the graph are domain-specific accelerators.

3.4 Model Parameters

LogNIC aims to retain the model simplicity and only requires necessary parameters. It comprises four types of parameters (Table 2): (1) **Hardware** describes the performance characteristics of each hardware block and interface/DRAM; (2) **Software** captures the execution behavior of an offloaded program; (3) **Traffic** details target traffic profile; (4) **Output** generates model analyzing results with a focus on throughput and latency. Parameter calibration is platform and implementation dependent. For example, one can obtain hardware parameters from the device specification or offline microbenchmark characterizations. They are fixed and can be reused. Software and traffic parameters are provided by users, where some might require programmers to understand the packet execution flow and testing environment. LogNIC also exposes configurable parameters for its model optimizer (§3.8) to explore the trade-off of different design alternatives.

3.5 Throughput Modeling

LogNIC estimates the attainable throughput by analyzing the computing throughput of each triggered IP and the bandwidth capacity of the traversed interface/memory based on the program's execution graph. Any hardware entities along the data-plane would confine the application performance on a SmartNIC. To simplify the description, we make two assumptions for now: (1) the incoming traffic has only one type of fixed-sized packets running at a given rate BW_{in} (i.e., the I/O rate of the ingress engine); (2) each IP block operates in a work-conserving fashion, and a packet can be processed by any engine. We will generalize this in §3.7.

Our derivation process works as follows. For a given time T, the total amount of data entering a SmartNIC is $W = T \times BW_{in}$. First, for an IP[i] (v_i in the graph), the maximum amount of work that it receives is the aggregated bandwidth across its incoming edges that feed the data: $T \times \sum BW_{e_{ji}}$. Since an IP might not process all the input data, we have $T \times \sum BW_{e_{ji}} \leq W$ and thus introduce a configurable parameter $\delta_{e_{ij}}$ across each edge (e_{ij}) to represent the relative data transfer percentage: $T \times BW_{e_{ji}} = \delta_{e_{ij}} \times W$. Hence, the minimum computation time for this IP to process the workload is the task size divided by its peak performance (which is P_{v_i}):

$$T_{IP[i]} = \frac{T \times \sum BW_{e_{ji}}}{P_{v_i}} = \frac{\sum(T \times BW_{e_{ji}})}{P_{v_i}} = \frac{W \times \sum \delta_{e_{ji}}}{P_{v_i}} \quad (1)$$

Next, the time to move data across the interface and memory is the total transferred data size (during time T) across all incoming edges divided by their bandwidth. The data traversed an interface and memory edge is $\alpha_{e_{ij}} \times W$ and $\beta_{e_{ij}} \times W$, respectively. We add a parameter because an IP block might only use partial packets for computation or partition traffic across multiple edges. One can decide its value based on the execution strategy graph and data granularity of an edge. So the data movement time for an interface and memory edge is:

$$T_{INTF} = \frac{\sum(\alpha_{e_{ij}} \times W)}{BW_{INTF}}; T_{MEM} = \frac{\sum(\beta_{e_{ij}} \times W)}{BW_{MEM}} \quad (2)$$

Finally, the maximum attainable throughput of a SmartNIC is inversely proportional to the maximum of times at each component (e.g., IP execution at each vertex, data movement across an edge, and data serving rate at interface/memory):

$$\begin{aligned} P_{attainable} &= \frac{W}{max(..., T_{IP[i]}, ..., T_{BW_{e_{ij}}}, ..., T_{INTF}, T_{MEM})} \\ &= min(..., \frac{W}{T_{IP[i]}}, ..., \frac{W}{T_{BW_{e_{ij}}}} ..., \frac{W}{T_{INTF}}, \frac{W}{T_{MEM}}) \end{aligned} \quad (3)$$

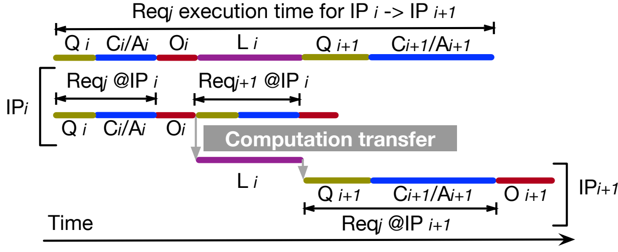


Figure 3: Modeling Req_j execution for $IP_i \rightarrow IP_{i+1}$.

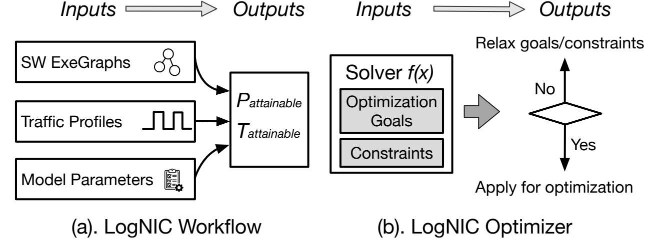


Figure 4: LogNIC workflow and optimizer.

Via algebra and re-expanding terms (using equations1–3), $P_{attainable}$ is independent of T and equals to:

$$min(..., \frac{P_{v_i}}{\sum \delta_{e_{ji}}}, ..., \frac{BW_{e_{ij}}}{\delta_{e_{ij}}}, ..., \frac{BW_{INTF}}{\sum \alpha_{e_{ij}}}, \frac{BW_{MEM}}{\sum \beta_{e_{ij}}}) \tag{4}$$

Discussion. Our derivation procedure is inspired by the Gables[40] model and uses its interconnect extension to model SoC interconnects and memory hierarchies of a SmartNIC. We eradicate the unified operational intensity parameter and replace it with IP-specific packet intensity, meaning how much traffic the IP could handle during computation. LogNIC allows the "fan-out" connectivity between nodes to represent task partition. In this case, an IP vertex in the execution graph has multiple outgoing edges where each one is equipped with a data transfer fraction factor ($\alpha_{e_{ij}}$ or $\beta_{e_{ij}}$).

3.6 Latency Modeling

Latency analysis is challenging due to (1) computation transfer across heterogeneous hardware domains; (2) the compounding effect between execution pipeline and IP parallelism; (3) traffic-induced intra-IP queueing and the resulting head-of-line blocking issue. LogNIC employs an incremental approach: it starts with examining the execution time between two consecutive IPs, then extends to all edges in an execution graph, and finally incorporates the queueing effect. We make the same assumptions as the throughput case.

Figure 3 illustrates the latency breakdown for the basic component of an execution graph (i.e., $IP_i \rightarrow IP_{i+1}$). The execution time of a request at an IP block mainly includes two components: queueing time within the block (Q) and execution time at the engine (C). To generalize the description, similar to LogCA [28] and Accelerometer [73] models, we temporarily add a tunable parameter (called acceleration or A) to explore the effect of an IP kernel optimization. When transferring computations across IPs, there are two other latency components: one is the preparation overhead (O) to trigger the computation at the next IP, such as passing parameters, setting up the initialization and completion signals, etc.; the second one is the time of moving data via either interface or memory, equaling to the data granularity divided by bandwidth (i.e., $\frac{g}{BW}$). The overhead (O) is independent of the granularity and computing parallelism. An engine will immediately fetch the next request via the scheduler when it's available. Hence, the execution time of $IP_i \rightarrow IP_{i+1}$ is:

$$T_{IP_i \rightarrow IP_{i+1}} = Q_i + \frac{C_i}{A_i} + O_i + \frac{g}{BW} + Q_{i+1} + \frac{C_{i+1}}{A_{i+1}} \tag{5}$$

Next, the latency for a particular path (P_k) is an accumulation of all its edge segments including the queueing and execution at the last IP block (IP_{n-1}).

$$T_{P_k} = \sum_{e_{ij} \in P_k} (Q_i + \frac{C_i}{A_i} + O_i + \frac{g_{e_{ij}}}{BW_{e_{ij}}}) + (Q_{n-1} + \frac{C_{n-1}}{A_{n-1}}) \tag{6}$$

We further expand the above equation by (a) representing the data communication granularity of each edge by considering the traversed data over interface and memory, and (b) calculating the computing time of each IP using aggregated data across ingress edges and its execution throughput.

$$\begin{aligned} \frac{g_{e_{ij}}}{BW_{e_{ij}}} &= \frac{g_{in} \times \alpha_{e_{ij}}}{BW_{INTF}} + \frac{g_{in} \times \beta_{e_{ij}}}{BW_{MEM}} \\ \frac{C_i}{A_i} = \frac{D_{v_i} \times \overline{g_{e_{ji}}}}{P_{v_i}} &= \frac{D_{v_i} \times g_{in} \times \sum \delta_{e_{ji}}}{P_{v_i} \times indegree(v_i)} \end{aligned} \tag{7}$$

When an execution graph presents multiple paths, the execution time for the whole graph (or application) is the weighted average across T_{P_k}, where weight (w_{P_k}) is calculated using traffic partition parameters (e.g., $\delta_{e_{ij}}$, $\alpha_{e_{ij}}$ and $\beta_{e_{ij}}$).

$$T_{attainable} = \sum (w_{P_k} \times T_{P_k}) \tag{8}$$

Finally, we dissect the queueing delay (Q) of an IP under a given system input (BW_{in}). As stated before, we are considering a work-conserving system without queueing prioritization which processes one traffic profile. LogNIC employs two techniques to facilitate the modeling. First, since the execution path would enclose multiple disjoint queues from different IPs, to facilitate our modeling, we apply the *virtual shared queue* abstraction [67], concatenating these queues as a single logic one. The queue synchronization overhead is then merged into the request-pulling phase. Second, we apply the M/M/1/N queue [29] to capture the queueing delay based on the observations: (1) the request arrival of most data center traffic follows the Poisson process; (2) the job service time of an IP presents the exponential distribution [28, 36, 63, 69].

Hence, the average queueing delay of an IP is then determined by three parameters (i.e., average number of requests in the queue L, effective arrival rate λ_e, and request service rate μ). Without node partition, the queueing delay is:

$$Q = \frac{L}{\lambda_e} - \frac{1}{\mu} \tag{9}$$

Following the M/M/1/N queueing mode, the probability of k requests in the queue and queue utilization (ρ) is:

$$Pro_k = \frac{\rho^k}{\sum_{n=0}^{N} \rho^n} = \frac{\rho^k - \rho^{k+1}}{1 - \rho^{N+1}}; \rho = \frac{\lambda}{\mu} \tag{10}$$

So the average number of requests in the queue $L = \sum_{n=0}^{N}(n \times Pro_n)$. Since the queue has a fixed capacity N, λ_e depends on the IP overall arrival rate and queue reject rate: $\lambda_e = \lambda(1 - Pro_N)$. We can view Pro_N as the packet dropping rate.

Now considering a specific IP (v_i) under the given rate BW_{in}, during the certain service time (e.g., $T = \frac{g_{in}}{BW_{in}}$), λ, μ, and ρ are calculated as follows:

$$\lambda = \frac{indegree(v_i)}{D_{v_i} \times T} = \frac{BW_{in} \times indegree(v_i)}{D_{v_i} \times g_{in}}$$
$$\mu = \frac{1}{\frac{C_i}{A_i}} = \frac{P_{v_i} \times indegree(v_i)}{D_{v_i} \times g_{in} \times \sum \delta_{e_{ji}}};\ \rho = \frac{BW_{in} \times \sum \delta_{e_{ji}}}{P_{v_i}} \tag{11}$$

Now combining equations 9–11, one can derive the final queueing delay as follows:

$$Q = \frac{L}{\lambda_e} - \frac{1}{\mu} = \frac{1}{\mu} \times (\frac{\rho}{1-\rho} - \frac{N\rho^N}{1-\rho^N}) \tag{12}$$

When an IP is partitioned and shared among vertices in the execution graph, LogNIC introduces a parameter γ_{v_i} to represent the resource partition ratio and updates the Q slightly.

Discussion. Compared with other architectural models (e.g., LogP [34], LogCA [28], and Accelerometer [73]) that mainly focus on analyzing the execution flow of a request, LogNIC examines how concurrent packets traverse different hardware blocks on a SmartNIC SoC during computation offloading. Thus, it is able to not only estimate the average processing latency, but also explore the impact of data movement granularity, IP parallelism and load balancing, queue sizing and queueing discipline, offering optimization insights.

3.7 LogNIC Generalization

We have described how LogNIC models the latency and throughput of a single SmartNIC application under one traffic profile. This section relaxes the aforementioned assumptions and introduces three extensions to handle mixed execution graphs, interleaved traffic, and non-work-conserving IPs.

- **Extension #1: Consolidate multiple execution graphs.** A SmartNIC allows multiple tenants to offload different programs concurrently. LogNIC preserves the composability propriety inherently because (1) it provides the node partition parameter (γ_{v_i}) to virtualize an IP; (2) it characterizes the IP-IP bandwidth (BW_{mn}) independently. This frees us from handling the resource contention issue. When performing throughput/latency modeling, LogNIC firstly splits W across different graphs and associates each with a weight parameter (w_{G_i}). It then computes the weighted average of data transfer percentage over each edge (e.g., $\sum w_{G_i} \times \alpha$), and obtains the $P_{attainable}/T_{attainable}$ for the entire SmartNIC.
- **Extension #2: Tolerate diverse traffic profiles.** LogNIC allows different execution graphs processing distinct traffic patterns (e.g., packet size distribution). When an application consumes multiple packet sizes, we apply different execution graphs since the per-IP execution time C, data transfer ratio $\delta_{e_{ij}}$, and overhead O_i would vary. Even though LogNIC partitions the queue capacity of an IP for multiple graphs, it also needs to accommodate the request service rate μ (proportional to the traffic demand) to capture the queueing effect of different-sized packets. With the composability support, we estimate the throughput and latency as the weighted average across different types $\sum dist_{size} \times P_{attainable}$ and $\sum dist_{size} \times T_{attainable}$ (Equations 3 and 8).
- **Extension #3: Accommodate IP non-determinism.** There are two factors affecting an IP performance: one is the queue-core mapping, which decides the execution parallelism of a request; the other one is whether the computation resource is work-conserving or not, impacting its efficiency. To handle the first one, LogNIC makes the IP actual parallelism configurable, impacting both the IP maximum performance (P_{v_i}) and the request delay. It provides an opportunity to use the LogNIC model to explore an optimal parallelism degree for a request, and then infer the suggested queue-core mapping. Regarding the second one, we employ the traffic shaping technique to unify the IP operating mode. Essentially, all IPs are work-conserving. For non-work conserving ones, LogNIC will add a specialized hardware block (called *rate limiter*) in front of the IP at its incoming edge. This rate limiter IP only performs enqueue/dequeue operations, but encloses a fixed-sized queue to capture the computation resource idleness. The queue capacity depends on the rate limiting degree. As a result, T_{P_k} and $T_{IP[i]}$ are slightly modified.

3.8 LogNIC Workflow and Optimizer

The LogNIC model takes three inputs (i.e., software execution graphs, traffic profiles, and model parameters) and derives attainable performance (Figure 4-a). The model spawns two independent threads to estimate the throughput and latency for each offloaded program and the overall. LogNIC is realized in Python. We represent the execution graph as a directed graph in NetworkX [10]. The modeling procedure uses the NumPy [11] for latency and throughput calculation. The optimizer uses the SciPy functions [16].

LogNIC Optimizer. LogNIC exposes a couple of configurable parameters (Table 2) that can guide system optimization, e.g., computation partitioning, load-aware scheduling, concurrency tuning, etc. To demonstrate this capability, we develop an interactive optimizer to explore the system design space (Figure 4-b). It works as follows. First, the optimizer defines the objective function (such as minimize $P_{attainable}$ or maximize $T_{attainable}$)) as well as the system constraints (e.g., device bus speed, IP parallelism, average latency bound). Second, we provide an interface for developers to prioritize different design alternatives by assigning weights and encoding them as another set of constraints. Finally, the optimizer uses an off-the-shelf solver to find a satisfactory solution. The suggested result might not be the optimal one if a local optimization algorithm (like the Nelder-Mead method [8]) is chosen. If the solver is unable to find a solution, one can further relax the optimizing goals or constraints. Our solver applies the SLSQP algorithm (from the SciPy library [16]) that combines the Han-Powell quasi-Newton method [15] and the BFGS update [2] of the B-matrix, where the step-length algorithm uses an L1-test function.

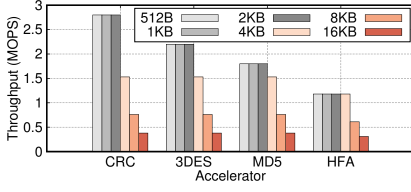


Figure 5: Accelerator throughput varied with its data access granularity.

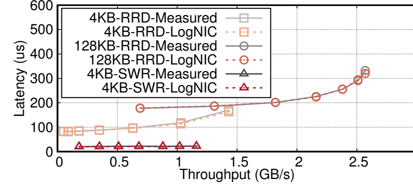


Figure 6: Latency varied with the throughput under three I/O profiles.

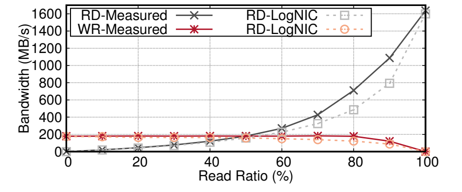


Figure 7: 4KB random IO performance varied with the read ratio.

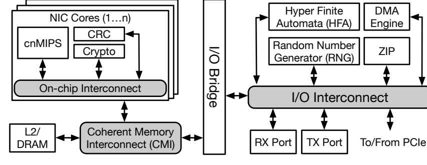


Figure 8: The hardware model of the LiquidIOII CN2360 SmartNIC.

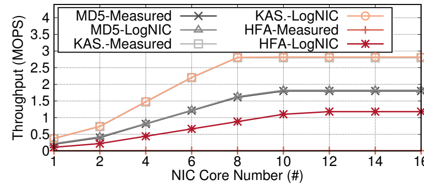


Figure 9: Throughput varied with the IP1 parallelism under line rate.

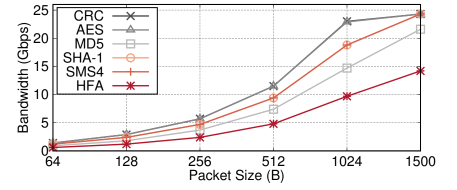


Figure 10: Achieved bandwidth varied with the packet size under line rate.

4 EVALUATION

Our evaluations aim to answer the following questions:

- How accurate is the LogNIC model in terms of performance estimation and system bottleneck analysis? How does LogNIC tolerate different traffic profiles? (§4.2–§4.3)
- Can we use the LogNIC model to optimize a SmartNIC-offloaded program? If so, why and how does the model provide such a capability? (§4.4–§4.5)
- Can we use the LogNIC model to guide a new SmartNIC design? If so, how does the model explore the hardware design space and what configurations can the model suggest? (§4.6)

4.1 Experimental Setup

Our testbed comprises Supermicro x86 servers, connected to an Arista 7160-32CQ switch. Each server has two Intel Xeon processors and 64/92GB memory, equipped with either a dual-port 100GbE NVIDIA Mellanox ConnectX-5 NIC (for traffic generation) or one of the two SmartNICs: 25GbE LiquidIO-II CN2360 (enclosing 16×1.5GHz cnMIPS cores and 4GB DRAM) and 100GbE BlueField-2 DPU [12] (equipped with 8×2.5GHz ARM A72 cores and 16GB DRAM). We install the Broadcom Stingray PS1100R SmartNIC on a standalone PCIe carrier board. It includes an 8-core 3.0GHz ARM A72 CPU, 8GB DDR4-2400 DRAM, FlexSPARX accelerators, and a 100GbE NetXtreme NIC. We run either CentOS 7.4 or Ubuntu 20.04 on all the machines.

4.2 Case Study #1: Inline Acceleration

Case Description. SmartNICs enclose a range of domain-specific accelerators and support bump-in-the-wire (inline) acceleration. We take an accelerator-rich SmartNIC (i.e., LiquidIO-II CN2360) as an example. Figure 8 depicts its overall device model. The SmartNIC program extends a basic UDP echo server. A NIC core (IP1) first pulls incoming packets from the RX port via the I/O interconnect, performs the L3/L4 packet processing, and then triggers an accelerator (IP2) for execution. After catching the completion signal, the NIC core (IP3) fabricates the response and sends it back via the TX port. We consider both (1) on-chip cryptographic units (i.e., CRC, MD5, 3DES, AES, SMS4 [17], and KASUMI [6]) that uses on-chip interconnect for data movement; and (2) off-chip application-specific engines, such as (de)compression (ZIP) and hyper finite automata (HFA), where the NIC-accelerator communication is performed via CMI (coherent memory interconnect) and I/O interconnect.

LogNIC Analysis. LogNIC delivers accurate throughput modeling. As shown in Figure 9, for the MTU-sized traffic, on average across all cases, the difference between estimated and measured results for all three accelerators (i.e., MD5, KASUMI, HFA) is less than 0.1%. Further, LogNIC can identify three factors that bound the overall system throughput (Equation 4). The first one is P_{IP1} (IP1's computing capacity), which impacts the number of issued accelerator calls. We configure such an experiment by gradually increasing the IP1's parallelism and measuring the achieved throughput. As shown in Figure 9, the experimental results match our hypothesis. MD5/KASUMI/HFA require 9/8/11 NIC cores to max out the performance, respectively. The number of required cores is different among these engines because they present different computing transfer overheads (i.e., O_{IP1}) in terms of accelerator preparation and parameter transmission.

The second one is the accelerator throughput (P_{IP2}). Since the LiquidIO-II SmartNIC provides no programming interface to tweak an accelerator's computing performance, to explore the impact of this factor, we instead vary the packet size of input traffic and measure the echo server bandwidth, equivalent to changing the data feed rate to an accelerator. The achieved bandwidth is close to $MIN(P_{IP2} \times Pkt_{size}, 25Gbps)$, as depicted in Figure 10. Third, BW_{MEM} and $BW_{Interface}$ are two deciding factors from the communication perspective. Specifically, we examine the cache-coherent memory interconnect to the crypto engine and HFA I/O fabric,

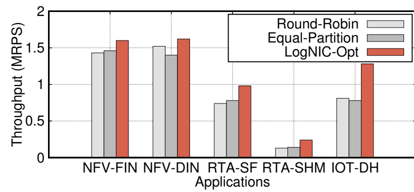


Figure 11: Throughput comparison among three allocation schemes.

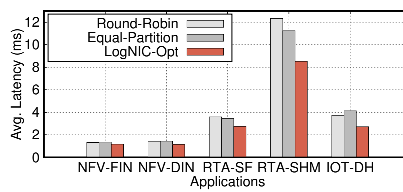


Figure 12: Average latency comparison among three allocation schemes.

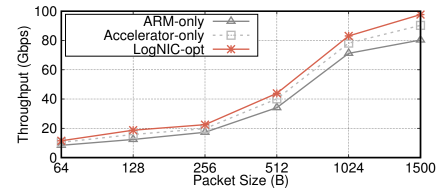


Figure 13: Throughput varied with the packet size among three placements.

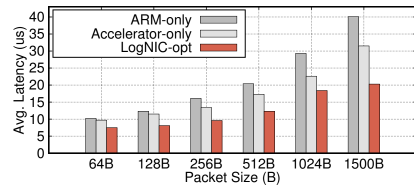


Figure 14: Latency comparison varying the packet size from 64B to 1500B.

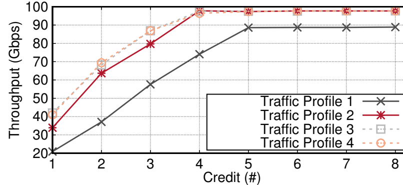


Figure 15: Measured bandwidth varied with the number of provisioned credits.

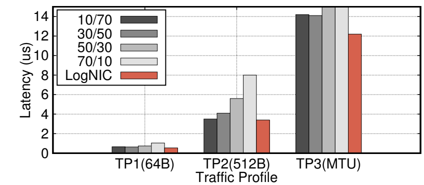


Figure 16: Latency comparison among static and LogNIC suggested partitions.

whose bandwidth is 50Gbps and 40Gbps, respectively. We configure an experiment that maximizes the accelerator performance and varies the data access granularity (i.e., $\alpha_{e_{ij}}$ and $\beta_{e_{ij}}$). As reported in Figure 5 (that uses 1KB-sized traffic), the throughput starts to drop as the data fetching size is larger than 4KB due to the memory/IO bandwidth limit. For example, under the 16KB access granularity, CRC, 3DES, MD5, and HFA only achieves 13.6%, 17.3%, 21.2%, and 25.8% of the maximum, respectively. Note that for all the experiments, the accelerator submission and completion are handled by the same NIC core, indicating that IP3 holds the same parallelism as IP1.

Takeaway. LogNIC can locate the performance bottleneck by considering the computing capability of all involved IPs and the interconnect/memory bandwidth.

4.3 Case Study #2: NVMe-oF Target

Case Description. A SmartNIC JBOF (just a bunch of flash) is an emerging cost-effective storage appliance for disaggregated storage. We consider the target-side NVMe-oF [13] protocol (NVMe-over-RDMA in our case) running over a Broadcom Stingray PS1100R JBOF. The application consists of RDMA networking stack processing on the NIC core, NVMe protocol handling, I/O submission/completion path coordination, and SmartNIC-SSD interaction.

LogNIC Analysis. One challenge when applying LogNIC (Equation 12) in this example is that the SSD internal details (e.g., command queue and write cache) and execution conditions are hidden from the programmers, complicating our initial parameter configurations. To remedy this, we characterize the latency/throughput as increasing the IO depth and use the curve fitting technique to figure out model parameters for the SSD. Next, we feed a single type of I/O traffic and vary the ingress rate. Figure 6 reports the comparison between measured latency and model estimated result for 4KB random read (4KB-RRD), 128KB random read (128KB-RRD), and 4KB sequential write (4KB-SWR) scenarios. The predicted differences are only 0.89%, 0.24%, and 2.75%, respectively. Next, we use the model to evaluate the aggregated bandwidth under a read/write mixed traffic scenario for a fragmented SSD (preconditioned with random writes). Figure 7 presents the 4KB random I/O performance as we vary the read ratio. Our estimated performance is 14.6% lower than the characterized one for both reads and writes. The misprediction rate increases slightly because random write IOs trigger SSD garbage collection operations that consume internal bandwidth, thereby, affecting both read/write performance. This cannot be captured by the LogNIC model.

Takeaway. LogNIC generates accurate latency/throughput estimations for various traffic profiles. It can describe a complicated IP without exposing its architectural internals.

4.4 Case Study #3: Parallelism Tuning

Case Description. E3 [59] is a SmartNIC-based Microservice execution platform developed for the LiquidIO CN2360 card. In E3, each Microservice runs as a multi-threaded process either on the SmartNIC or the host. An incoming request is forwarded to an available core in a round-robin fashion, and then triggers its service chain execution (defined as a dataflow graph). There is an orchestrator agent that continuously monitors the SmartNIC overloading status (by examining the traffic manager queue length) and migrates the Microservice to the host side if reaching an empirical threshold. By default, E3 leverages the inter-request parallelism to maximize system throughput without exploring the intra-request parallelism opportunities. We use the LogNIC optimizer to determine an optimal number of NIC cores allocated for a Microservice-based application.

LogNIC Optimization. Our optimization goal is maximizing overall throughput by identifying the right parallelism degree (P_{v_i} and D_{v_i}) at each vertex. We use the $P_{attainable}$ to explore the design space and take the original execution graph [59]. Based on the suggested configurations, we modify the E3 data-plane runtime that

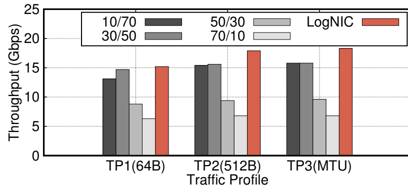


Figure 17: Throughput comparison among four static traffic partitions.

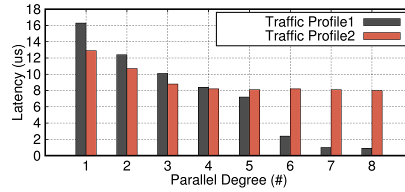


Figure 18: Latency varying the parallel degree for two traffic profiles.

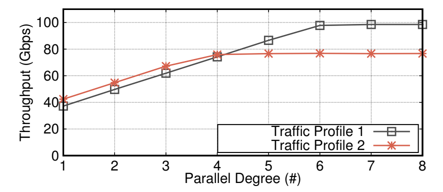


Figure 19: Throughput varying the parallel degree for two traffic profiles.

sets up the request-core mapping and enables the cross-core intra-request execution. We compare this with two other approaches: the default round-robin scheme (used by E3); an equal partition mechanism (that allocates NIC cores equally among all deployed Microservices). Our evaluation considers five E3 applications: flow monitoring (NFV-FIN), intrusion detection (NFV-DIN), spam filter (RTA-SF), server health monitoring (RTA-SHM), and IoT data hub (RTA-DH). Figures 11 and 12 report the throughput and average latency. With 80% traffic load, on average across these five workloads, our suggested allocation scheme achieves 34.8% and 36.4% throughput improvement, along with 22.4% and 22.8% latency savings over the other two. This is mainly because LogNIC can estimate the actual working set for each IP across the service chain and yields more accurate and fine-grained NIC core allocation.

Takeaway. The LogNIC optimizer suggests an optimal parallelism degree (D_{v_i}) of a vertex in the execution graph under given traffic profiles that results in higher throughput.

4.5 Case Study #4: Computation Placement

Case Description. This experiment exploits the use of LogNIC optimizer to place computations over a range of heterogeneous execution engines. We consider a network middlebox workload, consisting of five network functions (NFs): firewall gateway (FW)→L4 load balancer (LB)→deep packet inspection (DPI)→network address translation (NAT)→packet encryption (PE), which is deployed over the Mellanox BlueField-2 SmartNIC. Similar to the Broadcom Stingray, it is also an off-path Multicore-SoC card. All these functions can be implemented using the NIC ARM processor. However, the SmartNIC also provides hardware-accelerated Crypto, RegEx, Hashing, and Connection Tracking modules that can benefit most of the above NFs (except the DPI one). When placing them over different hardware domains, one not only needs to consider hardware limitations or computing constraints (such as a NAT with a large connection table will not fit into the NIC switch) but also the data movement overheads.

LogNIC Optimization. We apply the optimizer in two steps. First, we prepare the model parameters via offline characterizations for different-sized traffic. Next, we enumerate different combinations of the target execution graph and ask the optimizer to estimate the performance. For a given traffic profile, we choose the placement that offers the best throughput without over-subscribing the hardware resource. We compare it with two other schemes: *ARM-Only*, placing the NF on SmartNIC general-purpose cores; *Accelerator-only*, which takes a NIC accelerator as the first candidate (if possible). Figures 13 and 14 present the results. On average across different cases, our approach saves 37.9% and 27.3% latencies compared with the other two schemes, leading to 81.9% and 21.7% throughput improvements. This is because it considers the packet size-induced throughput difference and avoids costly off-chip communication if necessary.

Takeaway. The LogNIC optimizer chooses the fastest execution path for different-sized packets by examining the per-IP execution throughput (D_{v_i}), computation transfer overhead (O_i), and potential queueing delay (equation 12).

4.6 Case Study #5: Guiding SmartNIC Design

Case Description. SmartNICs are upgraded surprisingly fast to accommodate emerging application demands. This example demonstrates how to use LogNIC for hardware design space exploration. We take PANIC [55]–a recently proposed multi-tenant programmable NIC–as an example. PANIC has four major architectural components: (1) an RMT pipeline that parses the packet and generates the per-packet offloading chain; (2) a switching fabric, connecting and orchestrating traffic across all hardware components; (3) a central scheduler, which monitors the SoC communication and computation overloading status, prioritizes different execution paths, and provides traffic isolation guarantees; (4) compute units that perform acceleration executions. An incoming or outgoing packet first traverses the RMT pipeline to obtain an offloading descriptor and is forwarded to the scheduler through the switching entity. The scheduler combines pull-/push-based scheduling with a credit mechanism, and steers the packet in a load-aware fashion. Finally, the packet is injected into a chain of execution engines for various stateless/stateful packet processing. PANIC has a number of configurable knobs (such as accelerator parallelism, request queue size, switching capacity, etc.) to adapt to different offloading scenarios, where LogNIC could help.

LogNIC Optimization. LogNIC can be used to explore the hardware design when configuring hardware parameters. We consider three scenarios: configuring the queue size for saving hardware resources, enhancing the traffic steering capability to improve the throughput, and designing the optimal execution parallelism for performance maximization. The execution graphs are based on the Model 1 ("Pipelined Chain"), Model 2 ("Parallelized Chain"), and Mode 3 ("Hybrid Chain") from the original paper, where each compute unit is an IP. We use the same experiment setup as [55].

- **Scenario #1: Sizing the request queue of an accelerator.** The number of credits of a compute unit determines its internal queue capacity. Our goal is to identify the minimal amount of credits that saves the hardware resource without hurting throughput. We configure four mixed traffic profiles (under Model 1) that splits bandwidth across different-sized flows equally: profile 1 (64B/512B),

profile 2 (64B/512B/1024B), profile 3 (64B/256B/ 512B/1500B), and profile 4 (64B/128B/256B/1024B/1500B). The default configuration that the original paper suggests is 8 credits and exposes the credit allocation as a tunable factor. LogNIC models it using the node partition parameter (γ_{v_i}), and generates the optimal credit information as 5/4/4/4 credits for the above four profiles. The suggestions align well with our experimental results (Figure 15). Fewer credits reduce the latency. For example, for the first profile, compared with the 8 credits case, we observe a 21.8% latency drop when only using 5 credits that LogNIC suggests.

- **Scenario #2: Steering traffic at the central scheduler.** The LogNIC model can enhance the PANIC scheduler with accelerator awareness. We take Model 2 as an example and configure the computing throughput ratio of three accelerators (A1:A2:A3) as 4:7:3. Our experiment configures the traffic profile that comprises 20%, X%, (80-X)% traffic issuing to A1, A2, A3, respectively. We use the LogNIC to find an optimal X by using the BW_{in} parameter so that one can minimize the average execution latency. Figure 16 reports our empirical results by comparing four manually configured cases with the one suggested by LogNIC for 64B/512B/MTU-sized traffic. On average across all three traffic profiles, compared with the other four load balancing strategies, LogNIC reduces the average latency by 11.7%, 15.6%, 38.4%, and 57.2%, respectively. Figure 17 illustrates that LogNIC also achieves 16.3%, 11.4%, 84.8%, and 159.1% higher throughput. This is because the LogNIC optimizer steers traffic in proportion to an accelerator's computing capability.
- **Scenario #3: Configuring the IP hardware parallelism.** This example demonstrates the capability of the LogNIC model for tuning the accelerator's computing throughput. We use a modified version of Model 3, and set up three execution paths between ingress and egress: $IP1 \rightarrow IP3$, $IP1 \rightarrow IP4$, and $IP2 \rightarrow IP4$. We split the traffic between $IP1 \rightarrow IP3$ and $IP1 \rightarrow IP4$ in two ways (i.e., 50%/50%, 80%/20%) and measure the performance when increasing the parallel degree of IP4. The optimizer suggests the optimal parallel degree for traffic 1 and 2 is 6 and 4, which aligns with our empirical results Figures 18 and 19.

Takeaway: The LogNIC optimizer determines the minimal amount of resource provisioning and better execution pipeline that yield a more efficient hardware implementation.

4.7 Discussion

LogNIC entails the following limitations. First, some model parameters (in Table 2) might be inaccessible or difficult to characterize. For example, the IP-IP communication bandwidth (BW_{mn}) via an internal SoC interconnect cannot be directly measured if the SmartNIC vendor provides no APIs to measure it. An implicit library call that triggers an acceleration execution would hide the computation transfer overhead (O_i). Hence, developers have to provide conservative parameters that could affect the model's accuracy. Second, in addition to data movements, an IP can issue interface/memory accesses during its execution (such as data structure traversals), where the model does not have one dedicated parameter to capture this. We address this by reusing the edge medium usage parameters (i.e., $\alpha_{e_{ij}}$ and $\beta_{e_{ij}}$), and allow an IP to integrate its internal bandwidth usage into the ingress edge. Third, an IP can be complicated and opaque, where its architectural internal details are obscure. The SSD is such an example. Our experience suggests an alternative solution: empirically obtaining the latency vs. throughput graph as a whole and applying the curve-fitting technique to generate approximate parameters. Fourth, the LogNIC model optimizer cannot take the tail latency as the optimization goal or constraint since the model is unable to estimate the tail behavior. Lastly, to search the configuration parameters, BW_{in} should be given.

5 RELATED WORK

5.1 Compiler Support for SmartNICs

Researchers have developed new programming models, abstractions, and utilities to assist SmartNIC programming. Floem [66] applies a data-flow programming model and provides abstractions to assign computation to hardware execution engines, perform logical-physical queue mapping, etc. NICA [35] proposes an ikernel abstraction that is tightly integrated with the network stacks, enabling deploy application logic on FPGA-based SmartNICs. hXDP [33] develops a compiler that translates eBPF bytecode to an FPGA SmartNIC that follows an extended eBPF Instruction-set Architecture. Clara [68] generates offloading insights of legacy NFs and predicts its key performance parameters (such as the number of compute instructions and memory accesses). LogNIC is orthogonal to these systems and can be integrated into their compiler backend for performance estimation.

5.2 Performance Analysis of Networked System

A number of network-based performance analysis utilities have been built. For example, Bolt [43] proposes the performance contract for network functions deploying over x86 machines based on the worst-case execution time (WCET) analysis technique. It relies on the worst-case execution time (WCET) analysis technique, identifies the performance critical variables (PCV), and generates contracts for the target server box. However, it cannot estimate the whole hardware performance bound and limits the software exploration interface. SLOMO [60] leverages the hardware performance counters to analyze the interference impact over shared computing resources, and build an application/ platform-dependent accurate prediction framework. PIX [42] proposes the performance interface abstraction for NFs, including the number of instructions, the number of memory operations, and the number of CPU cycles, which are specific to the CPU's ISA. PCIe-bench [65] models the PCIe bus behavior and analyzes its impact on devices. Collie [48] unearths performance anomalies in RDMA subsystems by simulated annealing to trigger RDMA-based performance and diagnostic counters to extreme value regions. Similar to these systems, LogNIC also requires an offline characterization phase to obtain the IP Roofline as model parameters.

5.3 Switch-accelerated System

Driven by the high-throughput computing capability of programmable RMT switches [32], people have co-designed rack-scale distributed systems with in-network primitives [39, 44–46, 49, 51, 54, 57, 58, 70, 77, 79, 80]. For example, NetCache [44] realizes the key-value store logic on the packet processing pipeline. NetLock [77] then runs a centralized lock manager on the switch

data-plane. Pegasus [54] develops an in-network coherence directory to track and manage the location of replicated objects, and enables load-aware forwarding and dynamic rebalancing. MIND [51] places the memory management logic in the network fabric for distributed shared memory. [80] proposes a rack-level microsecond-scale request scheduler to mitigate load imbalance and head-of-line block issues. ATP [49] and SwitchML [70] accelerate machine learning tasks by using switch-enabled primitives. We believe the LogNIC model can support programmable switches by designing a new set of system interfaces.

6 CONCLUSION

This paper proposes LogNIC, a high-level performance model for SmartNICs. It employs a packet-centric based modeling that captures the SmartNIC execution based on how packets transmit over different hardware entities. LogNIC hides the underlying device details, represents a SmartNIC-offloaded program as an execution graph, retains a handful of configurable parameters, and estimates latency/throughput for a given traffic profile. We evaluate the model using three commodity SmartNICs and an academic prototype under five case studies, and demonstrate its three capabilities i.e., identifying performance bounds, exploring optimization strategies, and guiding new hardware design. We believe LogNIC will be a useful utility for software developers and hardware architects when building SmartNIC-assisted solutions.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments and feedback. This work is supported in part by NSF grants CNS-2106199 and CNS-2212192.

REFERENCES

[1] 2022. Broadcom Stingray PS1100R. https://docs.broadcom.com/doc/PS1100R-PB.
[2] 2022. Broyden–Fletcher–Goldfarb–Shanno algorithm. https://en.wikipedia.org/wiki/Broyden-Fletcher-Goldfarb-Shanno_algorithm.
[3] 2022. Fungible FAC100. https://www.fungible.com/product/nvme-over-tcp-fungible-accelerator-cards/.
[4] 2022. Intel co-designed the Mount Evans SmartNIC with Google Cloud. https://www.intel.com/content/www/us/en/products/network-io/infrastructure-processing-units/asic/es2000asic.html.
[5] 2022. Intel IPUs and SmartNICs. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.
[6] 2022. KASUMI Algorithm. https://en.wikipedia.org/wiki/KASUMI.
[7] 2022. Marvell LiquidIO DPUs. https://www.marvell.com/documents/08icqisgkbtn6kstgzh4/.
[8] 2022. Nelder-Mead Method. https://en.wikipedia.org/wiki/Nelder-Mead_method.
[9] 2022. Netronome Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/.
[10] 2022. NetworkX Library. https://networkx.org/.
[11] 2022. NumPy Library. https://numpy.org/.
[12] 2022. NVIDIA BlueField-2 DPU. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.
[13] 2022. NVMe-oF Specification. https://nvmexpress.org/developers/nvme-of-specification/.
[14] 2022. Pensando DSC-100. https://pensando.io/products/dsc/.
[15] 2022. Quasi-Newton Method. https://en.wikipedia.org/wiki/Quasi-Newton_method.
[16] 2022. SciPy Library. https://scipy.org.
[17] 2022. SMS4 Algorithm. https://en.wikipedia.org/wiki/SM4_(cipher).
[18] 2022. The Alpha Data FPGA SmartNICs. https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/.
[19] 2022. The Intel FPGA SmartNIC. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/smartnic.html.
[20] 2022. The NVIDIA Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/.
[21] 2022. The Xilinx Alveo U25 SmartNIC. https://www.xilinx.com/products/boards-and-kits/alveo/u25.html.
[22] 2023. 100 Gbps and Higher-Speed Ports to Account for 38 Percent of Shipments. https://www.delloro.com/news/smart-nic-revenues-projected-to-reach-1-6-billion-by-2026/.
[23] 2023. AWS Nitro System. https://aws.amazon.com/ec2/nitro/.
[24] 2023. Introducing C3 machines with Google's custom Intel IPU. https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu.
[25] 2023. Project Monterey. https://blogs.vmware.com/vsphere/2020/09/announcing-project-monterey-redefining-hybrid-cloud-architecture.html.
[26] 2023. SmartNIC deployment at Tencent Cloud. https://www.theregister.com/2021/11/08/tencent_homebrew_silicon/.
[27] 2023. SmartNICs to make up 38% of network market by 2026. https://www.theregister.com/2022/08/11/smartnics_network_market/.
[28] Muhammad Shoaib Bin Altaf and David A. Wood. 2017. LogCA: A High-Level Performance Model for Hardware Accelerators. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.
[29] CJ Ancker Jr and AV Gafarian. 1963. Some queuing problems with balking and reneging. I. *Operations Research* 11, 1 (1963), 88–100.
[30] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*.
[31] B. Bhattacharya and S.S. Bhattacharyya. 2001. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing* 49, 10 (2001), 2408–2421.
[32] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*.
[33] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.
[34] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. 1993. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*.
[35] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.
[36] Deniz Ersoz, Mazin S Yousif, and Chita R Das. 2007. Characterizing network traffic in a cluster-based, multi-tier data center. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*.
[37] 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. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*.
[38] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. 2023. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1012–1027.
[39] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 33–47.
[40] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A Roofline Model for Mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[41] M.D. Hill and A.J. Smith. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput.* 38, 12 (1989), 1612–1630.
[42] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance Interfaces for Network Functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.
[43] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance Contracts for Software Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.
[44] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating*

Systems Principles.

[45] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication.*

[46] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. 2021. RedPlane: Enabling Fault-Tolerant Stateful in-Switch Applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference.*

[47] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.*

[48] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. 2022. Collie: Finding Performance Anomalies in RDMA Subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22).*

[49] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21).*

[50] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T. V. Lakshman. 2017. UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of the 2017 Symposium on Cloud Computing.*

[51] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.*

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

[53] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference.*

[54] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).*

[55] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).*

[56] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication.*

[57] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. 2020. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20).* 305–323.

[58] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems.* 795–809.

[59] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19).*

[60] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. 2020. Contention-Aware Performance Prediction For Virtualized Network Functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication.*

[61] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. 2022. From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference (Sigcomm 22).*

[62] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference.*

[63] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. 2010. Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. *SIGMETRICS Perform. Eval. Rev.* (2010).

[64] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20).*

[65] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.*

[66] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).*

[67] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles.*

[68] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.*

[69] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication.*

[70] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21).*

[71] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.*

[72] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22).*

[73] Akshitha Sriraman and Abhishek Dhanotia. 2020. *Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale.*

[74] Arthur H. Veen. 1986. Dataflow Machine Architecture. *ACM Comput. Surv.* 18, 4 (dec 1986), 365–396.

[75] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23).* 987–1004.

[76] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[77] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication.*

[78] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2022. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.*, Article 37 (jun 2022), 30 pages.

[79] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.* (2019).

[80] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).*