Building Distributed Systems Using Programmable Networks

Ming Liu

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Arvind Krishnamurthy, Chair

Luis Ceze

Ratul Mahajan

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Building Distributed Systems Using Programmable Networks

Ming Liu

Chair of the Supervisory Committee:
Professor Arvind Krishnamurthy
Computer Science & Engineering

The continuing increase of data center network bandwidth, coupled with a slower improvement in CPU performance, has challenged our conventional wisdom regarding data center networks: how to build distributed systems that can keep up with the network speeds and are high-performant and energy-efficient? The recent emergence of a programmable network fabric (PNF) suggests a potential solution. By offloading suitable computations to a PNF device (i.e., SmartNIC, reconfigurable switch, or network accelerator), one can reduce request serving latency, save end-host CPU cores, and enable efficient traffic control.

In this dissertation, we present three frameworks for building PNF-enabled distributed systems: (1) IncBricks, an in-network caching fabric built with network accelerators and programmable switches; (2) iPipe, an actor-based framework for offloading distributed applications on SmartNICs; (3) E3, an energy-efficient microservice execution platform for SmartNIC-accelerated servers. This dissertation presents how to make efficient use of in-network heterogeneous computing resources by employing new programming abstractions, applying approximation techniques, co-designing with end-host software layers, and designing efficient control-/data-planes. Our prototyped systems using commodity PNF hardware not only show the feasibility of such an approach but also demonstrate that it is an indispensable technique for efficient data center computing.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am extremely grateful to Arvind Krishnamurthy, my thesis advisor, for giving me the right amount of freedom and guidance during my graduate studies. Arvind is a fearless academic, visionary, and system architect. He always encouraged me to look for challenging problems, explore root causes, and refine my ideas in a constructive way. I treasure our whiteboard discussions on various topics. Above all else, he has been endlessly enthusiastic and supportive of my research and me. His engaging arguments and strong feedback have contributed greatly to this dissertation. I hope and look forward to continued collaboration with him in the future.

The genesis of this thesis can be traced back to my first visit at UW CSE when Luis Ceze and Arvind Krishnamurthy pitched the in-network computing project to me, which leads to the IncBricks project. I am indebted to Luis for answering my various computer architecture related questions, which improves my understanding of programmable network hardware. I've also been lucky to have Simon Peter as my co-advisor out of UW. Simon is a solid system researcher, who taught me system building principles, evaluation methodology, and performance analysis. I am grateful for their encouragement and feedback.

I am grateful to the other members of my committee: Ratul Mahajan, whose unique insights on data center management make me think about how to deploy programmable networks in a cost-efficient way, and Radha Poovendran, who suggested how to organize the presentation, especially the iPipe actor scheduler.

The Allen School is an amazing environment for collaboration, and I want to thank everyone I've worked with during my time here: Liang Luo, Naveen Kr. Sharma, Phitchaya Mangpo

Phothilimthana, Tianyi Cui, Henry Schuh, Kaiyuan Zhang, Tapan Chugh, and Chenxingyu Zhao. Tom Anderson and Xi Wang have been excellent mentors to me and I cherish the sage advice I received from them. I would also like to thank my colleagues from NetLab, Sampa, and Syslab with whom I spent my time as a graduate student at UW CSE. There are too many of these wonderful people to list here. I enjoyed our group meetings, reading seminars, and lunch discussions.

I am grateful to my industry collaborators. Jacob Nelson is my default consultant for high-performance computing clusters. Kishore Atreya taught me lots of hardware architecture details of Cavium programmable network products. I learned how to build a robust Paxos system in enterprise clouds from Karan Gupta, Rishi Bhardwaj, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar.

A huge thanks to Elise Dorough and Melody Kadenko for taking care of all administrative tasks seamlessly. I'm especially thankful for Melody, who helped me order various hardware. With her support, we haved upgraded the Zookeeper cluster network, built the 100Gbps hometown cluster, and assembled different programmable network elements.

Finally, I am deeply indebted to my dear wife Jingyi Sun for her love and understanding through my graduate years. I am grateful to my parents Weicheng Liu and Xiuyun Sun, for their unconditional love and support. Thanks to my son Harold for the joy and the happiness he brings to me during our many moments together.

Dedication

To my family

1 Introduction

Today's data center holds many essential services that we use every day, such as distributed databases [124, 129, 111, 8], cloud storage [119, 133, 114], data analytics [250, 109, 31], and machine learning workloads [191, 126, 101]. They are deployed on thousands of servers within the data center, connected via a Clos network topology [152, 212]. These applications run 24/7, support tens of thousands of users, and have pervaded almost every aspect of our lives, ranging from social networks (e.g., Facebook [53], Twitter [93], LinkedIn [65]), shopping (e.g., Amazon [28], eBay [50]), transportation (e.g., Uber [94], Lyft [66]), to work productivity (e.g., Google Docs [56], Microsoft Office 365 [73]), and entertainment (e.g., Netflix [75], Spotify [90], TikTok [92]). Many of these popular workloads store a large amount of user data, receive high-volume read/update request traffic, require high availability and strong consistency, and strive to provide interactive user experiences.

Generally, two major factors affect the performance of these distributed systems: one is how fast do internal nodes communicate with each other; the other one is how efficient does the end-host perform computations. Recently, these two factors exhibit distinct growth trends within the data center. First, network speeds are increasing dramatically. Figure 1.1 presents the x86 servers breakdown of prominent data centers based on the Ethernet speed for the last 20 years [74, 11, 21]. Apparently, we entered the 40G network era in the last five years, and the 100G networks are going to be deployed soon. Second, the end-host server CPU performance has stagnated. As shown in Figure 1.2, the single-thread performance improvements over the last ten years have slowed down. The growth in terms of the number of logical cores is also increasing slowly. These two trends indicate that as more traffic comes into the server, one has to use more CPU cores to

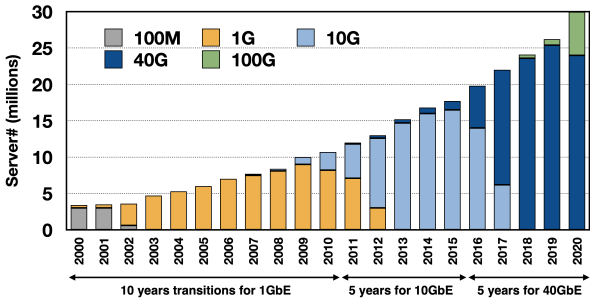


Figure 1.1: The x86 servers breakdown based on the Ethernet speed over the last 20 years.

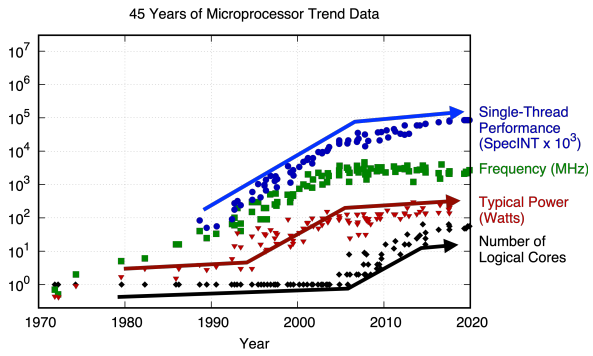


Figure 1.2: 45 years of Microprocessor trend.

handle it [149, 102]. For example, a recent microservice characterization study [149] shows that, for a social networking application, more than 36% of the total execution time is used for network protocol processing. Google's auto-sharding system Slicer [102] reports that more than 20% of the CPU cycles are used for data marshaling. As a result, a significant fraction of the server's computing is just utilized for keeping up with the basic needs of network protocol processing and application request pre-processing, leaving limited computing resources for application execution.

The fact that network bandwidths continue to increase dramatically, significantly outpacing the modest improvements in CPU performance, has challenged our conventional wisdom regarding data center architectures that server CPUs have abundant computing headroom to handle networking requests. The high-volume microsecond-scale network I/O raises the question: **how can we build a low-latency and energy-efficient computing system that can keep up with the network speeds?**

This dissertation affirmatively answers this question by designing systems that can offload computations of distributed applications to the emerging programmable network fabric (PNF). PNF is a domain-specific acceleration architecture for the data center that can accelerate packet processing tasks. It presents computing capabilities across the packet communication path, including fully-programmable SmartNICs at the network edge, match-action based reconfigurable switches in the network, and domain-specific network accelerators attached to the switch. PNF hardware

usually encloses ingress/egress pipelines for header parsing/deparsing, power-efficient computing engines for packet manipulation, onboard traffic control for flow directing and load balancing, diverse memory regions to save flow states, and programmable DMA engines to interact with hosts. Therefore, by offloading suitable computations into the network, one can 1) serve application requests within the network with low latency and high throughput, saving end-host cores; 2) improve end-host energy efficiency by increasing the per-packet computing density; and 3) deploy new network protocols that enable efficient packet processing.

PNF is a practical instantiation of the active networks idea [245, 244] from the 1990s. In an active network, the router and switch can perform customized computations on the message flowing through them. One typical realization [255] is using the active capsule approach, where traditional passive packets are replaced with arbitrary miniature programs. They are encapsulated as transmission frames and can be executed at each node along the communication path. In contrast, programmable networks rely on passive packets, with computations triggered at different PNFs based on pre-defined execution conditions (such as parsing results, matched rules, etc.). More importantly, PNF is a modest proposal. It only allows a limited amount of computation and state so that one can process traffic at line rate.

The key challenge of using a PNF is how to make efficient use of these in-network heterogeneous computing resources. There is a significant gap between application software integrated interfaces and the computing capabilities of programmable networking hardware. First, a PNF device lacks generic programming models or abstractions. A SmartNIC or network accelerator is programmed using low-level primitives or APIs within the firmware [79]. A programmable switch is programmed using domain-specific languages such as P4 [116]. Essentially, PNF hardware has multiple heterogeneous computing units along with diverse non-cache-coherent memory regions. There is no unified programming model fit for all scenarios. One has to take into account the execution characteristics of offloaded workloads and hardware capabilities to program these devices effectively.

Second, one should stay within the hardware constraints since a PNF device has limited computing capabilities and limited memory space. For example, a programmable switch favors match-

action processing, while SmartNICs and network accelerators can perform general-purpose per-packet execution and domain-specific per-message acceleration. When a PNF is overloaded, one would observe an increase of the tail latency, packet drops, or even bandwidth loss. In terms of memory, a programmable switch has tens of megabytes SRAM, whereas a SmartNIC/network accelerator encloses several gigabytes of DRAM. To use them effectively, one must consider how to layout the application working set and balance the trade-off between data structure maintenance overheads and offloading benefits.

Third, due to the hardware constraints, we need to figure out how to co-design application logic between programmable network hardware and end-host servers. This requires support from both the control-plane and the data-plane. Specifically, the control-plane manager should add PNF awareness when performing resource allocation and generating execution plans. On the data-plane side, it should guarantee high PNF utilization and performance isolation. The data-plane orchestrator should (1) schedule computations across the PNF and hosts based on runtime usage; (2) tolerate packet processing tasks with different latency distributions, traffic micro-bursts, and synchronous/asynchronous completion signal delivery scenarios.

In this dissertation, we address these challenges by performing detailed hardware characterization, proposing new programming models and abstractions, and employing efficient control-/data-plane systems. We build real systems using commodity PNF hardware and evaluate the benefits/trade-offs of such offloading.

1.1 Main Contribution

My thesis is that distributed systems can keep up with network speeds and achieve high performance and better energy efficiency by being co-designed with programmable network fabric. This dissertation's main contribution is to *bridge the gap between increasing network bandwidths and stagnating CPU computing capabilities by co-designing distributed systems with programmable network fabrics*. It develops principles and techniques about using in-network heterogeneous computing resources in an efficient way. Specifically, we first understand PNF hardware from three

perspectives (i.e., how the interconnect fabric feeds data to different in-network computing engines, how much stateful and stateless packet processing can a packet do without impacting line rate, how much parallelism along the pipeline that the programmable hardware exposes). Second, when developing the programming system, we explore an execution model that is tailored to the underlying hardware architecture, design the data structure layout model that is appropriate for the PNF memory hierarchy, and apply approximation techniques to reduce the application working sets. Finally, we add PNF awareness into the control-plane and provide a flexible allocation policy. The data-plane orchestrator is able to tolerate packet processing tasks with different latency distributions, traffic micro-bursts, and synchronous/asynchronous completion signal deliveries. To demonstrate the feasibility of such solutions, we have designed, implemented, and evaluated the following PNF-enabled distributed systems using commodity PNF devices:

- **IncBricks** [248], is an in-network caching fabric with basic computing primitives. It is a hardware-software co-designed system that supports caching in the network using a programmable network middlebox. Its hardware part combines a programmable switch (i.e., Cavium XPliant) and a network accelerator (i.e., OCTEON SoC) as a programmable middlebox. Its software part runs a distributed cache-coherent key-value store along with a topology-aware source routing protocol. IncBricks applies NoSQL key-value store APIs. As a key-value store accelerator, our prototype lowers request latency by over 30% and doubles throughput for 1024 byte values in a small-scale cluster configuration. When doing computations on cached values, IncBricks achieves three times more throughput and a third of the latency of client-side computation.
- **iPipe** [247], is an actor-based framework for offloading distributed applications on a SmartNIC. It is designed based on our characterization observations of a SmartNIC from four perspectives (i.e., traffic control, computing units, onboard DRAM, and host communication). iPipe introduces an actor programming model, where each actor has its own self-contained private state and communicates with other actors via messages. The framework provides a distributed memory object abstraction and enables actor migration. The central piece of iPipe is a hybrid scheduler combining FCFS and DRR-based processor sharing that can tolerate tasks with variable execution costs and maximize NIC compute utilization. Using iPipe, we build a real-time data

analytics engine, a distributed transaction system, and a replicated key-value store. Based on our prototyped system with commodity SmartNICs, we show that when processing 10/25Gbps of application bandwidth, NIC-side offloading can save up to 3.1/2.2 beefy Intel cores, along with 23.0/28.0μs latency savings.

- **E3** [249], is a microservice execution platform on SmartNIC-accelerated servers. By offloading suitable microservices to a SmartNIC's low-power computing units, one can improve server energy-efficiency without latency loss. E3 follows the design philosophies of the Azure Service Fabric microservice platform and extends key system components. It employs three key techniques: an ECMP-based load balancer at the ToR switch to balance network request traffic among SmartNICs, a hierarchical communication-aware microservice placement algorithm to balance computation demands, a data-plane orchestrator to detect and eliminate SmartNIC overloading. Our prototyped system using Cavium LiquidIO shows that SmartNIC offload can improve cluster energy-efficiency up to 3$\times$ and cost-efficiency up to 1.9$\times$ at up to 4% latency cost for common microservices, including real-time analytics, an IoT hub, and virtual network functions.
- **Characterizing PNF-based disaggregated storage.** We conduct a detailed characterization study on a Broadcom Stingray PS1100R platform to understand its performance characteristics, computing capabilities when serving line-rate network/storage traffic, and potential computing headroom for offloading. Our gathered observations provide some guidelines on how to build efficient storage applications on such disaggregated infrastructures.

1.2 Lessons Learned

We summarize some general lessons of using PNFs to build high-performant and energy-efficient distributed systems that go beyond the specific contributions of each of the above individual PNF-enabled systems.

#1: Approximation is a useful technique to address the hardware resource constraints of using PNFs. For example, one should take advantage of probabilistic data structures (like count-

min sketch, bloom filter) to save application/flow states in a sublinear memory space. One should decompose the application logic into small modules (where each of them can fit into a PNF unit) and spread the whole computation across a number of execution stages and/or devices.

#2: PNFs can be used to build a disaggregated processing system that would be suitable for future data center settings. IncBricks provides a concrete example of co-designing the switch and accelerator logic in a disaggregated setting. Switches can do certain things effectively, such as application-level monitoring and traffic control over all terabits of traffic. It can forward the interesting subset of traffic to the network accelerator. Accelerators can maintain interesting (non-trivial) data structures with a reasonable amount of memory. It can also handle the modest amounts of traffic forwarded to it by the switch. One can also extend the ingress/egress processing pipelines of a programmable switch to SmartNICs to realize the big switch abstraction. This form of co-design and disaggregation allows for the independent scaling of different types of hardware accelerations.

#3: Before we determine an optimal code deployment strategy, we must characterize PNF devices from multiple perspectives, e.g., traffic control, computing units, onboard memory, and host communication. Specifically, there are three important questions to answer: how much stateful and stateless packet processing can a packet do without impacting line rate; how much execution parallelism along the pipeline the programmable hardware exposes; how the interconnection fabric feeds data to an in-network acceleration engine. All of these factors determine the effectiveness of code-offloading.

We provide a concrete case study of this approach in the context of SmartNICs. To enable efficient SmartNICs offloading, one should consider computing headrooms, sensitivity to traffic workloads, and execution isolation. SmartNICs usually have two structural designs (on-path v.s. off-path), which expose different offloading capacity. The size and amount of offloading depends on the traffic profile (such as packet size distribution, flow size distribution, and packet interval, etc.). Further, when offloading, one should consider how to do performance isolation and security/privacy protection.

#4: When using PNFs to design an energy-efficient system, one should schedule compu-

tations to an execution engine that offers the best application performance given the power budget. There are three considerations regarding this. First, PNF offloading might not always be the optimal design decision even though PNFs usually enclose power-efficient computing engines. Second, the interesting performance metrics that will drive the system design are throughput (i.e., requests per second) and latency SLA/SLO. Under a given power budget, one should find a solution that offers the highest throughput without violating the SLA/SLO. Third, energy-efficiency is also related to the amount of incoming traffic, which determines the request load. For example, a line-rate energy-efficient system solution would not be optimal for the unloaded case.

1.3 Published Work

Part of the work presented in this thesis has been published in the following papers:

- Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. **IncBricks: Toward In-Network Computation with an In-Network Cache**. *Proceedings of Architecture Support for Programming Languages and Operating Systems (ASPLOS), 2017*.
- Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. **Offloading Distributed Applications onto SmartNICs using iPipe**. *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM), 2019*.
- Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. **E3: Energy-Efficienct Microservices on SmartNIC-Accelerated Servers**. *Proceedings of USENIX Annual Technical Conference (ATC), 2019*.

1.4 Organization

The rest of this dissertation is organized as follows:

Chapter 2 provides background information. It first presents an overview of the programmable network fabric and then describes the hardware architectures and software stacks of three PNF devices (i.e., programmable switch, SmartNIC, network accelerator) in detail. Next, we discuss the challenges of PNF offloading.

Chapter 3 describes the design and implementation of our first PNF-enabled system. It motivates the need for in-network caching and shows how to combine a programmable switch and a network accelerator efficiently. We also describe how to address the cache coherence issues under a multi-path FatTree topology.

Chapter 4 first presents a SmartNIC characterization study. Based on our gathered insights, it introduces the iPipe framework and covers its different components (e.g., actor programming model, distributed memory object, an actor scheduler). We then show how to use the framework to develop three distributed applications.

Chapter 5 describes how we build a microservice execution platform on SmartNIC-accelerated servers. We first discuss the benefits/challenges of microservice offloading quantitatively and then describe how to address them by extending key system components of Azure Service Fabric.

Chapter 6 presents a characterization study of SmartNIC-based disaggregated storage. It describes the experimental methodology, presents our observations, and discusses some open problems of applying it for system design.

Finally, Chapter 7 summarizes the work performed in this dissertation, discusses the limitations of our work, and ends with directions for future work.

2 Background

The increasing data center network bandwidths and the stagnating CPU performance has motivated the use of programmable network hardware to accelerate networked applications. By offloading suitable computations onto intermediate networking devices, one can keep up with networking speeds, and achieve high-performance and energy-efficiency. This thesis presents how to realize such goals for distributed applications. In this chapter, we first describe the hardware architectures and software system stacks of contemporary programmable networking devices, and then discuss the challenges of using them for system designs.

The remainder of this chapter is organized as follows. Section 2.1 presents a high-level overview of a programmable network fabric. Sections 2.2 – 2.4 describe programmable switch, SmartNICs, and network accelerators in detail. Section 2.5 talks about the offloading challenges from three perspectives (i.e., programming abstraction, hardware constraints, control-plane/data-plane co-design), respectively. Section 2.6 concludes this chapter by summarizing our findings.

2.1 Programmable Network Fabric

Programmable network fabric (PNF) is one kind of emerging domain-specific hardware that computing architects bring into the data center to increase its computing density. It targets packet processing tasks. PNF is a practical realization of past academic proposals, such as active networks [244, 245] from the 1990s. However, instead of injecting and running general program fragments (or capsules) at each network router/switch along the path, PNF only allows for a limited amount of computing and state so that one can process traffic at line rate.

A PNF hardware stays on the data plane, and is able to perform stateless/stateful computa-

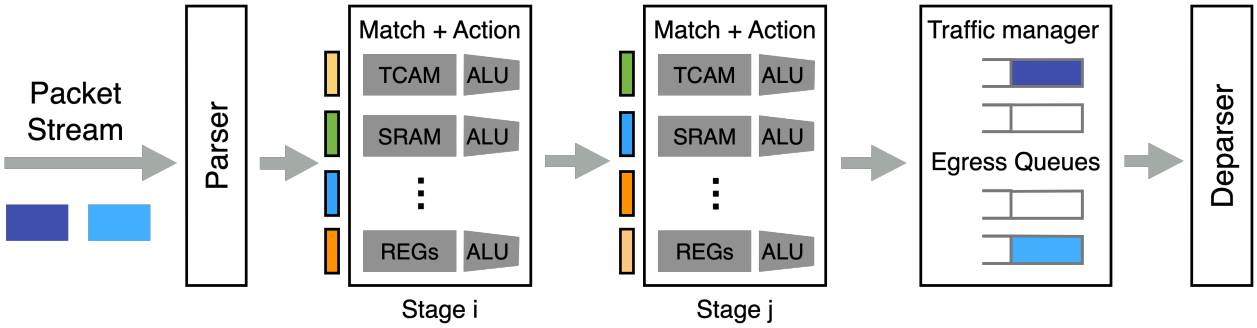


Figure 2.1: Architectural block diagram for a programmable switch.

tions on traversing packets. It usually comprises of five functional components: (a) ingress/egress pipelines that parse/deparse the packet header, generate packet metadata, and modify certain fields based on installed rules; (b) computing engines, which could be reconfigurable match-action tables, networking processor, general-purpose multi-core/many-core processor, or domain-specific accelerators; (c) traffic control modules, directing and load-balancing networking flows from Ethernet MAC ports to different execution units. It might also enclose internal priority queues with flexible scheduling primitives. (d) diverse memory units, ranging from fast scratchpad SRAM to slow onboard L2/DRAM; (e) programmable DMA engines, which move data between two units via high-bandwidth interconnects. Generally, a programmable network fabric can be a programmable switch in the network, a fully programmable SmartNIC at the networking edge, and a network accelerator along the communication path.

2.2 Programmable Switches

Programmable (or reconfigurable) switches provide a match+action (M+A) processing model. It matches on arbitrary predefined packet header fields and then conducts simple packet processing actions. Figure 2.1 presents its hardware architecture. The switch begins packet processing by extracting relevant packet headers via a user-defined parse graph and generating per-packet metadata. Next, the extracted header fields along with the metadata are passed onto a pipeline

of stages. Each stage has user-defined M+A tables (in TCAM or SRAM) for packet matches as fixed-function switches. Unlike a traditional switch, programmable switches also make registers available so that one can maintain and mutate state as packets traverse the switches. Further, each stage has small ALUs for performing a limited set of operations on packet headers or register contents. After traversing the pipeline stages, packets are deposited in an egress queue. The data-plane program on a switch can also control which queue is used for egress. Some switches like the Cavium XPliant [7], are also equipped with an embedded service CPU, which can asynchronously perform some computations on the data plane. In summary, a programmable switch provides the following hardware features that enable flexible application offloading: (1) a limited amount of stateful memory (such as counters, meters, and registers) that can be used to maintain states across packets; (2) computation primitives (e.g., addition, bit-shifts, max/min), which are executed upon extracted packet header fields, metadata, and stateful memory; (3) in-network traffic telemetry, reporting egress queue length, concurrent flow characteristics, etc.; (4) packet scheduling primitives, which can be used to design customized QoS-aware enqueue/dequeue service disciplines; (5) communication interfaces via the local control-plane CPU for bookkeeping tasks.

As with traditional fixed-function switches, the system stack of a programmable switch has three levels (i.e., management, control, and data). A management plane carries the administration traffic, which is used for configuring and monitoring devices. Its programming interfaces are mainly CLI [47], WebGUI [97], SNMP [89], OpenConfig [84]. A control plane is responsible for populating the routing/forwarding tables, and determining the packet forwarding path. It comprises three layers: protocol stacks, which support a range of routing protocols like OSPF [83], EIGRP [52], BGP [39], IS-IS [63]); platform libraries, including program-dependent table-level APIs, fixed-function APIs for port/MMU management; device drivers that interact with the switch ASIC. A data plane defines how to forward traffic from ingress to egress. A programmable switch usually provides domain-specific languages such as P4 [116] that allow developers building flexible and customizable packet processing pipelines. It then compiles to a sequence of architecture-dependent instructions, links with the system libraries and ABI, and generates an executable firmware that can be loaded.

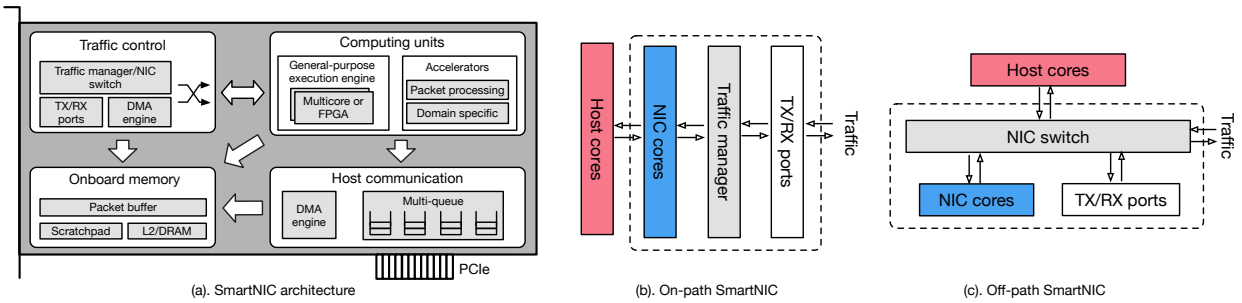


Figure 2.2: Architectural block diagram for a SmartNIC and packet processing for the two types of SmartNICs.

Several programmable switches are available today, such as Barefoot Tofino [37]/Tofino2 [38], Cavium XPliant [7], and Intel FlexPipe [60]. They are mainly used as the ToR (top of the rack) switches and are able to deliver tens of terabits per second traffic. We have seen an increasing number of deployments of such switches from hyperscale cloud and service provider networks to enterprise and high-performance computing clusters.

2.3 Programmable NICs (SmartNICs)

A SmartNIC consists of four major parts (as shown in Figure 2.2-a): (1) computing units, including a general-purpose execution engine (like ARM/MIPS/RISC multi-core processor), along with accelerators for packet processing (e.g., deep packet inspection, packet buffer management) and specialized functions (e.g., encryption/decryption, hashing, pattern matching, compression); (2) onboard memory, enclosing fast self-managed scratchpad and slower L2/DRAM; (3) traffic control module that transfers packets between TX/RX ports and the packet buffer, accompanied by an internal traffic manager or NIC switch that delivers packets to NIC cores; (4) DMA engines for communicating with the host.

Table 2.1 lists the HW/SW specifications of eight commercial SmartNICs. They represent different design trade-offs regarding performance, programmability, and flexibility. The first two LiquidIOII SmartNICs enclose an OCTEON [44] processor with a rich set of accelerators but run in

SmartNIC model	Vendor	Processor	BW	L1/L2/DRAM	Deployed SW	Nstack	To/From host
LiquidIOII CN2350 [43]	Marvell	cnMIPS 12 core, 1.2GHz	2× 10GbE	32KB / 4MB / 4GB	Firmware	Raw packet	Native DMA
LiquidIOII CN2360 [43]	Marvell	cnMIPS 16 core, 1.5GHz	2× 25GbE	32KB / 4MB / 4GB	Firmware	Raw packet	Native DMA
BlueField1 1M332A [67]	Mellanox	ARM A72 8 core, 0.8GHz	2× 25GbE	32KB / 1MB / 16GB	Full OS	Linux/DPDK/RDMA	RDMA
BlueField2 MBF2H332A-AEEOT [68]	Mellanox	ARM A72 8 core, 2.5GHz	2× 25GbE	32KB / 1MB/16GB	Full OS	Linux/DPDK/RDMA	RDMA
Stingray PS225 [42]	Broadcom	ARM A72 8 core, 3.0GHz	2× 25GbE	32KB / 16MB / 8GB	Full OS	Linux/DPDK/RDMA	RDMA
Agilio CX [76]	Netronome	NFP-4000	2× 25GbE	NA/4MB/2GB	Firmware	Raw packet	Native DMA
Innova-2 MNV303212A-ADLT [69]	Mellanox	Xilinx Kintex UltraScale XCKU15P	2× 25GbE	NA / NA / 8GB	Firmware	Raw packet	Native DMA
ADM-PCIE-9V3 [27]	Alpha Data	Xilinx Virtex UltraScale Plus VU3P-2	1× 100GbE	NA / NA / 2GB	Firmware	Raw packet	Native DMA

Table 2.1: Specifications of the eight SmartNICs used in this study. BW = bandwidth. Nstack = networking stack.

the context of a light-weight firmware. Programmers have to use native hardware primitives to process raw packets, issue DMA commands, and trigger accelerator computations. Netronome Agilio is also such a SmartNIC but encloses MicroFlow engine processors. BlueField 1/2 and Stingray cards run an ARM Cortex-A72 [34] processor and host a full-fledged operating system. They offer a lower barrier for application development, and one can use traditional Linux/DPDK/RDMA stacks to communicate with local and external endpoints. The BlueField card even has NVDIMM support for fault-tolerant storage. Some SmartNICs like Mellanox Innova [69], Alpha Data ADM-PCIE-9V3 [27], and Microsoft Catapult [146] also use FPGAs at the execution engine. Current SmartNICs typically have link speeds of 10/25/100 GbE, and 200/400 GbE units are starting to

appear. Generally, a SmartNIC is a bit more expensive than a traditional dumb NIC. For example, a 10/25GbE SmartNIC typically costs 100∼400$ more than a corresponding standard NIC [23].

Based on how SmartNIC cores interact with traffic, we further categorize SmartNICs into two types: ***on-path*** and ***off-path*** SmartNICs. The cores for on-path SmartNICs (Figure 2.2-b) are on the packet communication path and hold the capability to manipulate each incoming/outgoing packet. LiquidIOII CN2350/CN2360, Netronome Agilio, and FPGA-based ones are on-path SmartNICs. Off-path SmartNICs (Figure 2.2-c), deliver traffic flows to host cores (bypassing NIC cores) based on forwarding rules installed on a *NIC switch*. Mellanox BlueField 1/2 and Broadcom Stingray are examples of off-path SmartNICs. Both NIC vendors are further improving the programmability of the NIC switch (e.g., Broadcom TruFlow [118], Mellanox $ASAP^2$ [202]).

For both types of SmartNICs, host processing is the same as that with standard NICs. On the transmit path (where a host server sends out traffic), the host processor first creates a DMA control command (including the instruction header and packet buffer address) and then writes it into a command ring. The NIC DMA engine then fetches the command and data from host memory and writes into the packet buffer (located in the NIC memory). NIC cores on on-path SmartNICs pull in incoming packets (usually represented as work items), perform some processing, and then deliver them to TX/RX ports via the DMA engine. For off-path ones, packets are directly forwarded to either NIC cores or TX/RX ports based on switching rules. Receive processing (where a host server receives traffic from the SmartNIC) is similar but performed in the reverse order.

2.4 Network Accelerators

The last type of PNF we consider is a network accelerator. As shown in Figure 2.3, a network accelerator scales up the communication and computation capabilities of a SmartNIC. It usually has 4 to 8 communication ports, which can handle much more traffic than a SmartNIC. In terms of computing, a network accelerator has a many-core flow processor and more complicated domain-specific accelerators, like compression, encryption, and storage RAID engine. An important hardware piece on the accelerator is the programmable hardware dispatcher that can steer traffic to a

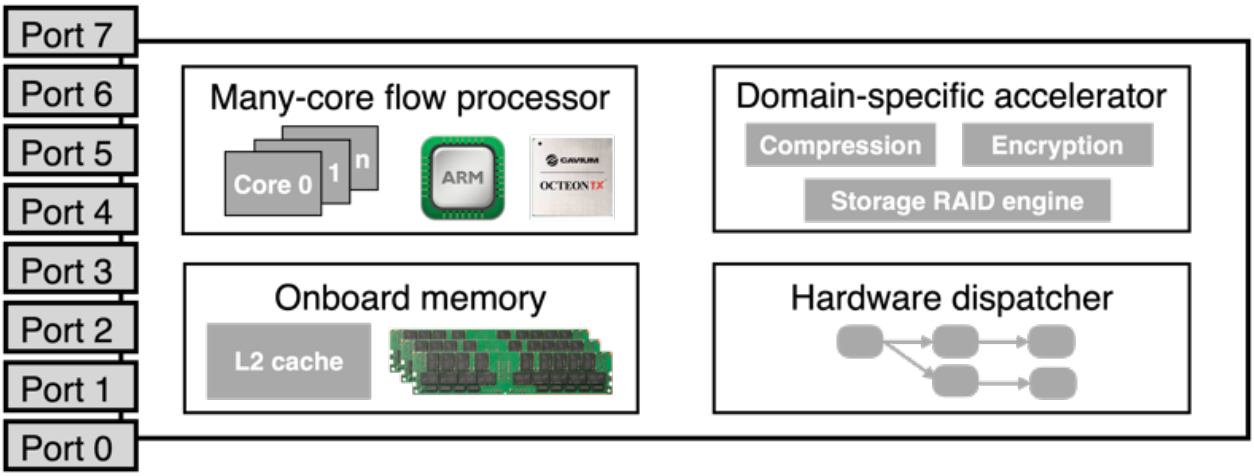


Figure 2.3: Architectural block diagram for a network accelerator.

particular computing unit more efficiently. Generally, a network accelerator is a standalone SoC board with an independent power supply, which is deployed close to a rack switch or router. Take the Cavium OCTEON SoC board [44] as an example. It is equipped with 32 cnMIPS [45] cores and diverse hardware acceleration units (such as packet-management/security/application/specialized accelerators). There is a high-speed coherent interconnect connecting all these execution engines that optimize data movements. The board has up to 8 I/O interfaces which serve more than 200GbE traffic.

Network accelerators provide a flexible software system stack for application offloading. The underlying firmware abstracts the hardware resources and exposes hardware threads, memory managers, signals/mutexes, and a set of libraries to use the accelerators. On the control-plane side, programmers can partition and configure the device. The data-plane is similar to that of a SmartNIC and is able to perform per-packet computation and maintain intermediate states across packets. One can build customized packet handlers (using provided libraries) into the firmware. Generally, there are two common issues of using a domain-specific accelerator on the board. One is how to feed data into the execution engine. Essentially, one needs to set up the memory channel for data movement and manage the inbound DMA facility (including both DMA queues and the instruc-

tion words). The other one is how to catch the completion signal. Since most onboard accelerators use synchronous execution, polling is the primary approach that helps reduce latency but wastes some NIC core cycles. The interrupt mechanism is more heavy-weight but offers the flexibility of asynchronous handling. Hence, one should design a hybrid approach and take the workload characteristics into account. Some emerging accelerators offer complex application logic (such as video encoding [198]) and use the asynchronous type. In such cases, an event handling mechanism would be more appropriate.

2.5 Challenges of PNF Offloading

The key challenge towards using a PNF is how to make efficient use of these in-network heterogeneous computing resources. There is a significant gap between application software integrated interfaces and the computing capabilities of programmable networking hardware.

First, a PNF device lacks generic programming models or abstractions. A SmartNIC or network accelerator is programmed using low-level primitives or APIs within the firmware. A programmable switch is programmed with some domain-specific languages (such as P4 [116]) for the data plane. A PNF hardware has multiple heterogeneous computing units along with diverse non-cache-coherent memory regions. There is no unified programming model fit for all scenarios. One has to take both the execution characteristics of offloaded workloads as well as the hardware capabilities. For example, network functions present great packet-level parallelism with light control flow. When offloading to a network accelerator or SmartNIC, data-flow [48] programming model would be effective. On the other hand, the actor model [25] is better for applications that are communication-intensive.

Second, one should stay within the hardware constraints since a PNF device has limited computing capabilities and limited memory space. Specifically, a programmable switch favors match-action processing, while the other two PNFs can perform general-purpose per-packet execution and domain-specific per-message (or even per-flow) acceleration. When overloading happens, one would observe an increase of the tail latency, packet drops, or even bandwidth loss. In terms

of memory, a programmable switch has tens of megabytes SRAM, where a SmartNIC/network accelerator encloses several gigabytes of DRAM. To use them effectively, one has to figure out how to layout the application working set. Generally, there are four models as described below. When designing a PNF-offloaded system, one has to balance the trade-off between data structure maintenance overheads and offloading benefits.

- **Partition,** that applies the key-based sharding approach;
- **Caching,** maintaining a software cache layer with application-dependent organization and replacement policies;
- **Assisted,** providing lookup indexes to accelerate the traversal of the main data structure;
- **Distributed shared memory**, which offers a unified memory space.

Finally, due to the hardware constraints, we need to figure out how to co-design application logic between programmable network hardware and end-host servers. This requires support from both the control-plane and the data-plane. Specifically, the control-plane manager should add PNF awareness when performing resource allocation and generating execution plans. On the data-plane side, it should guarantee high PNF utilization and performance isolation. The data-plane orchestrator should (1) schedule computations across PNF and hosts based on runtime usage; (2) tolerate packet processing tasks with different latency distributions, traffic micro-bursts, and synchronous/asynchronous completion signal delivery scenarios. For example, when using a programmable switch, one needs to first partition tables/meters/registers across different stages, and then design an execution chaining pipeline. The problem becomes much more challenging for SmartNICs/network accelerators due to their substantial computing resources, which would require a library OS support.

2.6 Summary

In this chapter, we presented the background for programmable network fabrics and describe three PNF devices (i.e., programmable switch, SmartNIC, network accelerator) in detail. We discussed their hardware architectures as well as software stacks. Different PNF devices make different

trade-offs between computing capabilities and bandwidth. A programmable switch is able to perform limited computing but at tens of terabits per second, while a SmartNIC can perform general-purpose computations at tens of gigabits per second. A network accelerator resembles a SmartNIC but can process hundreds of gigabits per second. We concluded this chapter by discussing the challenges of PNF offloading.

In this thesis, we address these challenges in the context of distributed applications. We perform detailed hardware characterizations, develop new programming models, and build efficient control-planes/data-planes. The next three chapters will present three PNF-enabled systems.

3 IncBricks: Caching Data in the Network

In this chapter, we will describe the first PNF-enabled system (called IncBricks) to accelerate distributed in-memory key-value stores. It is built with a programmable switch and a network accelerator. By caching data onto intermediate networking devices, one can (1) serve network requests on the fly with low latency; (2) reduce data center traffic and mitigate network congestion; and (3) save energy by running servers in a low-power mode. IncBricks is a hardware-software co-designed system that supports caching with basic computing primitives. The hardware part is a hybrid switch/network accelerator architecture that offers flexible support for offloading application-level operations. The software part is a distributed cache-coherent key-value store.

The rest of this chapter is organized as follows. Section 3.1 provides the background and motivation of in-network caching. Section 3.2 gives a high-level overview of our proposal. Section 3.3 describes the system architecture of IncBricks deployment. Sections 3.4 and 3.5 describe the design and implementation of its hardware and software components, respectively. Section 3.6 reports our prototyped system, experimental setup, and evaluations. Section 3.7 discusses related work. We conclude this chapter by discussing our learned lessons in Section 3.8.

3.1 Background

Networking is a core part of today's data centers. Modern applications such as big data analytics and distributed machine learning workloads [4, 5, 258, 191, 126] generate a tremendous amount of network traffic. Network congestion is a major cause of performance degradation, and many applications are sensitive to increases in packet latency and loss rates. Therefore, it is important to reduce traffic, lower communication latency, and reduce data communication overheads in com-

modity data centers.

Existing networks move data but don't perform computation on transmitted data, since traditional network equipment (including NICs, switches, routers, and middleboxes) primarily focuses on achieving high throughput with limited forms of packet processing. As discussed in the previous chapter, new classes of programmable network devices such as programmable switches (e.g., Intel's FlexPipe [60], Cavium's XPliant [7], the Reconfigurable Match-Action Table architecture [117] and network accelerators (e.g., Cavium's OCTEON [44]) have recently emerged. Programmable switches allow for application-specific header parsing and customized match-action rules, providing terabit packet switching with a light-weight programmable forwarding plane. Network accelerators are equipped with scalable low-power multi-core processors and fast traffic managers that support more substantial data plane computation at the line rate. Together, they offer in-transit packet processing capabilities that can be used for application-level computation as data flows through the network.

The key idea of this work is offloading a set of compute operations from end-servers onto programmable network devices (primarily switches and network accelerators) so that (1) remote operations can be served on the fly with low latency; (2) network traffic is reduced; (3) servers can be put in low-power mode (e.g., Intel C6 state) or even be turned off or removed, leading to energy and cost savings.

However, offloading computation to the network has three challenges. First, even though the hardware resources associated with programmable network infrastructure have been improving over time, there is limited compute power and little storage to support general data center computation or services (like a key-value store). For instance, the Intel FlexPipe [60] chip in the Arista 7150S switch [33] has a flexible parser and a customizable match-action engine to make forwarding decisions and control flow state transitions. The switch is programmed with a set of rules, and then it applies data-driven modifications to packet headers as packets flow through the switch. The 9.5 MB packet buffer memory on this switch is not exposed for storing non-packet data; even if it was, the bulk of it would still be needed to buffer incoming traffic from dozens of ports in the case of congestion, leaving limited space for other uses. Network accelerators have less severe

space and processing constraints, but that flexibility comes at the cost of reduced interface count and lower routing and traffic management performance when compared with switches. Second, data center networks offer many paths between end-points [152, 212, 224], and network component failures are commonplace, so keeping computations and state coherent across networking elements is complex. Finally, programmable network hardware should provide a simple and general computing abstraction that can be easily integrated with application logic in order to support a broad class of data center applications.

In this chapter, we address the challenges outlined above by building IncBricks. As a first step towards in-network computation, it provides a distributed caching service to accelerate data retrieval of data center workloads.

3.2 Solution Overview

IncBricks is an in-network caching fabric with basic computing primitives, based on programmable network devices. IncBricks is a hardware/software co-designed system, comprising *IncBox* and *IncCache*. IncBox is a hybrid switch/network accelerator architecture that offers flexible support for offloading application-level operations. We choose a key-value store as a basic interface to the in-network caching fabric because it is general and broadly used by applications. We build IncCache, an in-network cache for key-value stores, and show that it significantly reduces request latency and increases throughput. IncCache borrows ideas from shared-memory multi-processors and supports efficient, coherent replication of key-value pairs. To support more general offloading, we provide basic computing primitives for IncCache, allowing applications to perform common compute operations on key-value pairs, such as increment, compare and update, etc. We prototype IncBox using Cavium's XPliant switches and LiquidIO boards. Our real system evaluations demonstrate that (1) in-network caching can provide common data center service abstractions with lower latency and higher throughput than existing software-only implementations; and (2) by carefully splitting the computation across programmable switches, network accelerators, and end hosts, one can achieve fast and efficient data center network request processing.

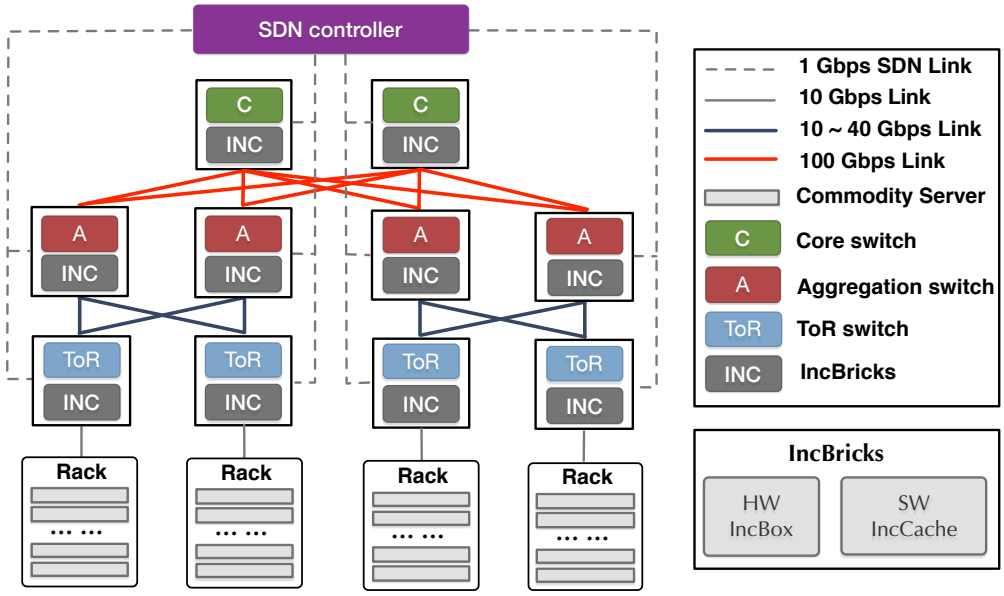


Figure 3.1: An overview of the IncBricks system architecture in a commodity data center network.

3.3 System Architecture

This section describes the IncBricks system architecture in the context of a data center. We first present the background of a commodity data center network. Then we discuss the internal architecture of IncBricks in detail and explain how to integrate IncBricks into existing networks.

3.3.1 Data center Network

Figure 3.1 shows a typical data center network [228, 152, 212], with a hierarchical topology reaching from a layer of servers in racks at the bottom to a layer of core switches at the top. Each rack contains roughly 20 to 40 nodes, connected to a Top of Rack (ToR) switch via 10 Gbps links. ToR switches connect to multiple aggregation switches (for both redundancy and performance) using 10–40 Gbps links, and these aggregation switches further connect to core switches with a higher bandwidth (e.g., 100 Gbps) link. To offer higher aggregate bandwidth and robustness, modern data centers create multiple paths in the core of the network by adding redundant switches. Switches

usually run a variant of ECMP [1] routing, hashing network flows across equal-cost paths to balance load across the topology. Multiple paths bring coherence challenges to our IncBricks system design.

Traditional Ethernet switches take an incoming packet and forward it based on a forwarding database (FDB). They comprise two parts: a data plane, which focuses on processing network packets at the line rate, and a control plane, which is used for configuring forwarding policies. The data plane consists of specialized logic for three core functions: (1) an ingress/egress controller, which maps transmitted and received packets between their wire-level representation and a unified, structured internal format; (2) packet memory, which buffers in-flight packets across all ingress ports; and (3) a switching module, which makes packet forwarding decisions based on the forwarding database. The control plane usually contains a low-power processor (e.g., an Intel Atom or ARM chip) that is primarily used to add and remove forwarding rules. SDN techniques enable dynamic control and management by making the control plane more programmable [220, 201].

3.3.2 Deployment

IncBricks is a hardware/software co-designed system that enables in-network caching with basic computing primitives. It comprises two components, IncBox and IncCache, shown in Figure 3.1. IncBox is a hardware unit consisting of a network accelerator co-located with an Ethernet switch. After a packet marked for in-network computing arrives at the switch, the switch will forward it to its network accelerator, which performs the computation. IncCache is a distributed, coherent key-value store augmented with some computing capabilities. It is responsible for packet parsing, hashtable lookup, command execution, and packet encapsulation.

Our IncBricks implementation highly depends on the execution characteristics of switches. For example, a core or aggregation switch with a higher transmit rate and more ports might enclose a larger hash table inside its IncCache compared to the hash table of a ToR switch. Moreover, IncBricks requires that the data center has a centralized SDN controller connecting to all switches so that different IncBricks instances can see the same global view of the system state.

3.4 IncBox: Programmable Network Middlebox

This section describes IncBox, our middlebox. We first discuss our design decisions and then show the internal architecture of IncBox.

3.4.1 Design Decisions

A design must support three things to enable in-network caching: (1) it must parse in-transit network packets and extract some fields for the IncBricks logic (**F1**); (2) it must modify both header and payload and forward the packet based on the hash of the key (**F2**); (3) it must cache key/value data and potentially execute basic operations on the cached value (**F3**). In terms of performance, it should provide higher throughput (**P1**) and lower latency (**P2**) than existing software-based systems for forwarding packets or processing payloads.

A network switch is an ideal place to do in-network computation since all packets are processed by its forwarding pipeline. However, traditional fixed-function switches struggle to meet **F1** since their packet parser configuration is static. Software switches (such as Click [179], RouteBricks [144], and PacketShader [156]) are able to perform dynamic header parsing, but they remain two orders of magnitude slower at switching than a hardware switch chip. This means they fail to meet **P1** and **P2** in terms of forwarding, not to mention header or payload modifications. Programmable switches seem a promising alternative. Their configurable parsers use a TCAM and action RAM to extract a customized header vector. During ingress processing, they could match different parts of a packet-based on user-defined rules and apply header field modifications. Nevertheless, programmable switches have two drawbacks: first, they can support only simple operations, such as read or write to a specified memory address, or primitive compute operations (e.g., add, subtract, shift) on counters. There is no support for more complicated operations on payloads. Second, the size of packet buffer (along with TCAM and action RAM) is on the order of a few tens of megabytes; most of that is needed to store incoming packet traffic and match-action rules, leaving little space for caching. They meet **F1** and **F2**, but to satisfy **F3**, **P1**, and **P2** in terms of payload-related operations, we must take advantage of other devices.

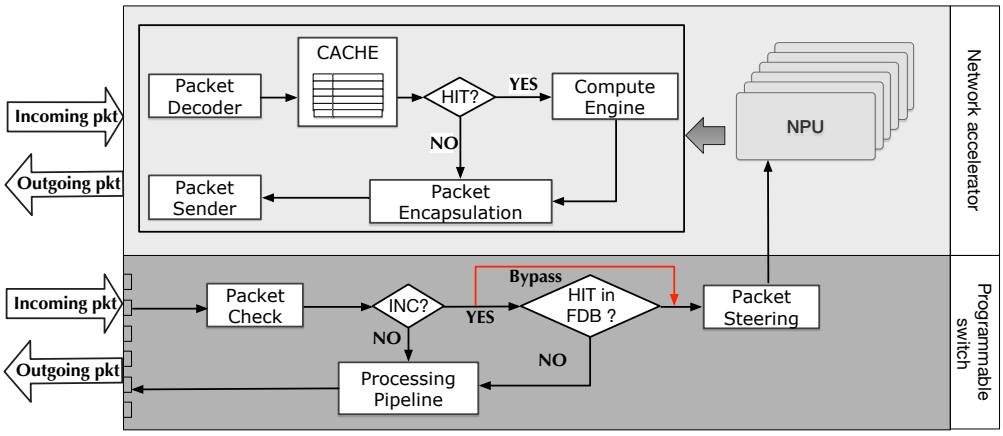


Figure 3.2: IncBox internal architecture.

We choose network accelerators to satisfy the rest of our requirements for three reasons. First, their traffic managers can serve packet data to a processing core in hundreds of nanoseconds (e.g., 200ns for our OCTEON board), which is significantly faster than kernel-bypass techniques [217]. Second, their multi-core processors are able to saturate 40Gbps–100Gbps of bandwidth easily, which is hard to achieve with general-purpose CPUs. Third, they support multiple gigabytes of memory, which can be used for caching. FPGA accelerators like Catapult [121] could also fit our scenario, but the general-purpose processors in the network accelerator are more flexible and easier to program compared with an FPGA.

3.4.2 IncBox Design and Prototype

We carefully split IncBox's functionality across a programmable switch and a network accelerator. Figure 3.2 shows our design. The switch performs three tasks. The first is packet checking, which filters in-network caching packets based on the application header. If there is a match, the packet will be forwarded to a network accelerator. Otherwise, it will be processed in the original processing pipeline. The second is key hit checks, which determines whether the network accelerator has cached the key or not. This is an optimization between the two hardware units and can be bypassed depending on the switch configuration. The third is packet steering, which forwards the packet

to a specific port based on the hash value of the key. This will be used when there are multiple attached network accelerators, to ensure that the same key from the same network flow will always go the same accelerator, avoiding packet reordering issues. Since IncCache is based on the UDP protocol, packet reordering will cause incorrect results (e.g., a GET after SET request might fail since it could be processed out of order). We expect that more capabilities will be provided in the upcoming programmable switches (i.e., Barefoot's Tofino [37, 38]) due to deeper ingress pipelines and more flexible rewrite engines. For example, a switch could maintain a Bloom filter of keys cached by the network accelerator or could even probabilistically identify heavy-hitter keys that are frequently accessed. Implementing these operations fully in the switch data-plane would require reading state updated by previous packets; the Cavium switch we use requires intervention of its management processor to make state updated by one packet available to another.

The network accelerator performs application-layer computations and runs our IncCache system. First, it extracts key/value pairs and the command from the packet payload. Next, it conducts memory-related operations. If the command requires writing a new key/value pair, the accelerator will allocate space and save the data. If the command requires reading a value based on one key, the accelerator performs a cache lookup. If it misses, the processing stops and the network accelerator forwards the request further along its original path. If it hits, the network accelerator goes to the third stage and executes the command, both performing any operations on the value associated with the key as well as conducting cache coherence operations. Finally, after execution, the accelerator rebuilds the packet and sends it back to the switch.

Our IncBox prototype consists of (1) one 3.2 Tbps (32x100Gbps) Cavium XPliant switch and (2) one Cavium OCTEON-II many-core MIPS64 network accelerator card. We use two types of OCTEON-II devices: an OCTEON-II evaluation board with 32 CN68XX MIPS64 cores and 2GB memory, and a light-weight LiquidIO adapter, enclosing 8 CN66XX MIPS64 cores and 2GB memory. We program the switch by (1) adding packet parsing rules for the new packet header; (2) changing the ingress processing pipeline; and (3) modifying the forwarding database (FDB) handlers. We program the network accelerator using the Cavium Development Kit (CDK), which provides a thin data plane OS (OCTEON simple executive) along with a set of C-language hard-

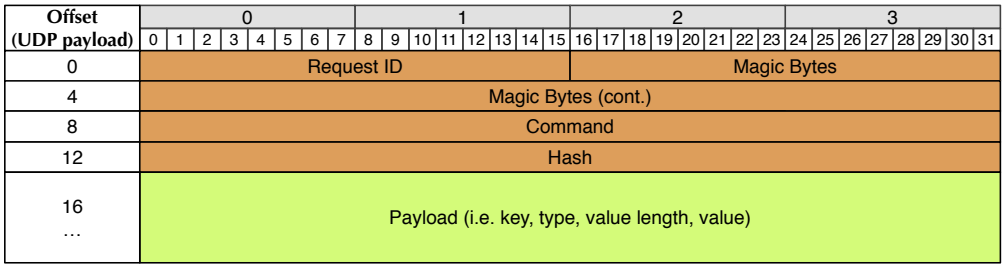


Figure 3.3: In-network computing packet format

ware acceleration libraries (such as compression/decompression, pattern matching, and encryption/decryption). Our software stack uses the hardware features of this network accelerator for fast packet processing.

3.5 IncCache: a Distributed Coherent Key-Value Store

This section describes the IncCache system design. IncCache is able to (1) cache data on both IncBox units and end-servers; (2) keep the cache coherent using a directory-based cache coherence protocol; (3) handle scenarios related to multipath routing and failures; and (4) provide basic compute primitives. A common design pattern in data center applications today is to have two distributed storage layers: a high-latency persistent storage layer (like MySQL), and a low-latency in-memory cache layer (like Memcached). IncCache accelerates the latter; when data requests go through the data center network, our IncCache will (1) cache the value in the hash table; (2) execute a directory-based cache coherence protocol; and (3) perform computation involving cached values.

3.5.1 Packet Format

Packet format is one of the key components enabling IncCache. There are three design requirements: (1) it should be a common format agreed to by the client, server, and switch; (2) it should be parsed by the network switch efficiently without maintaining network flow states; and (3) it should

be flexible and easily extended.

We target UDP-based network protocols, which have been shown to reduce client-perceived latency and overhead [213] and are also amenable for switch processing without requiring per-flow states. Our in-network computing messages are carried in the UDP payload; their format is shown in Figure 3.3 and described in detail here:

- **Request ID (2 bytes):** A client-provided value for matching responses with their requests.
- **Magic field (6 bytes):** Labels in-network cache packets, allowing switches to filter other UDP packets that should not be handled by IncCache.
- **Command (4 bytes):** Selects which operation to perform on the key/value of the payload.
- **Hash (4 bytes):** The hash of the key, used to guarantee ordering inside the network switch.
- **Application payload (variable bytes):** Key-value pairs, which also includes the type.

3.5.2 Hash Table Based Data Cache

We use a hash table to cache the key-value data on both network accelerators and end-host servers. To achieve high throughput, the hash table should (1) support concurrent accesses from multiple processing cores and (2) provide full associativity for cached keys. It should also respect the constraints of the hardware.

Fixed-size lock-free hash table We designed the hash table on the network accelerator to have a fixed size due to the limited memory space. It consists of a set of buckets, each containing an ordered linked list of entries. Each node in the bucket list has a unique hash code and serves as the head of an entry list. Items with the same hash code but different keys are stored on the same entry list, in sorted order based on the key. The hash table uses lock-free operations to access and modify its state [158, 204]. For set or delete operations, the hash table interacts with the memory allocator module to allocate and free data. Our allocator maintains a fixed number of data items and applies a least frequently used eviction policy.

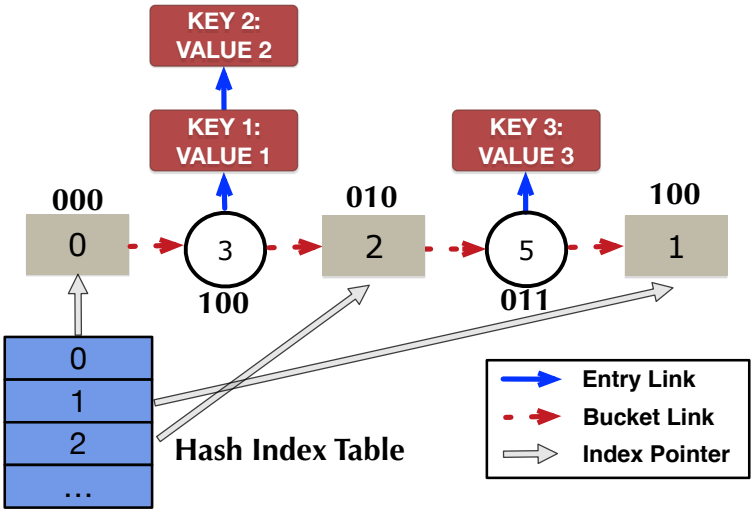


Figure 3.4: Bucket splitting hash table design. The prefix of a key's hash code is looked up in an index table, which points to (square) sentinel bucket nodes in the bucket list. Hash table entries are stored in entry lists from each bucket node. As the table expands, buckets are inserted to create (round) overflow bucket nodes and the index is extended, and more sentinel (square) nodes are added.

Extensible lock-free hash table On the server side, we implemented an extensible hash table due to the server's more relaxed memory constraints. It is also lock-free, using the split-ordered list technique [234]. Figure 3.4 shows the structure of the hash table. Search and insert are the two primary operations, and behave similarly. When looking for an item, the hash table first traverses the bucket list to match a hash code. If there is a match, the corresponding entry list is searched for an exact key match. In order to accelerate this lookup process, our hash table maintains a set of hash indexes that are shortcuts into the bucket list. We build the hash index by examining the prefix of the hash code of a given item. An insert operation follows a similar procedure, except that it atomically swaps in a new node (using compare-and-swap) if there is no match. To guarantee a constant time lookup, the hash table dynamically expands the size of the hash index and examines a progressively longer prefix of an item's hash code. The newer elements in the expanded hash index are ordered to appear after the elements that existed before the expansion, resulting in what

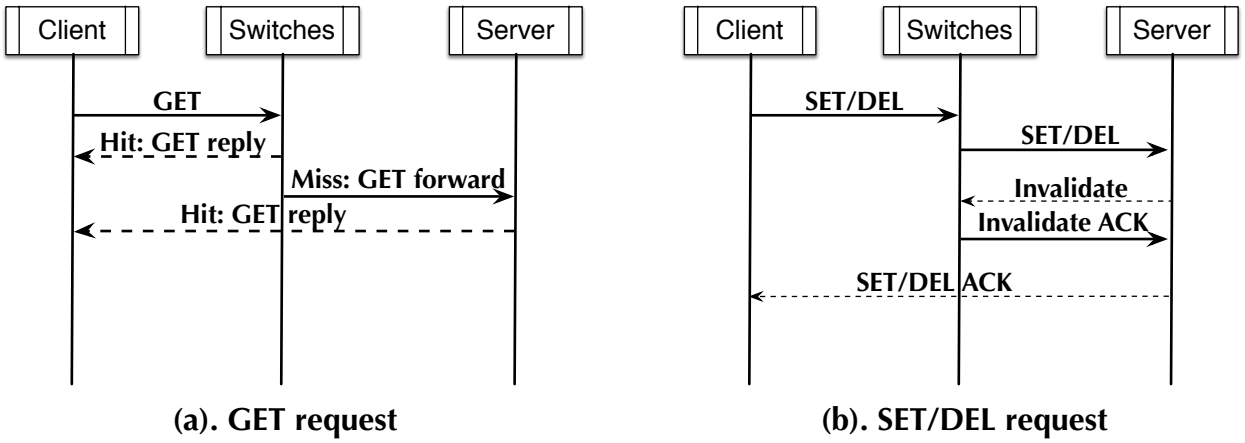


Figure 3.5: Packet flow for GET/SET/DELETE requests, showing how end-host servers and switches maintain cache coherence in a single path scenario.

is known as a recursively split-ordered list. By recursively splitting the hash indexes with this technique, we introduce finer-grained shortcuts in a lock-free manner, without having to copy the index.

3.5.3 Hierarchical Directory-based Cache Coherence

IncCache allows data to be cached in both IncBox units and servers. A cache coherence protocol is required to keep data consistent without incurring high overheads. Maintaining the sharers list (tracking who has the data) is a challenge since (1) servers do not know the details of the network topology, and (2) there is no shared communication media like the coherence bus in commodity processors.

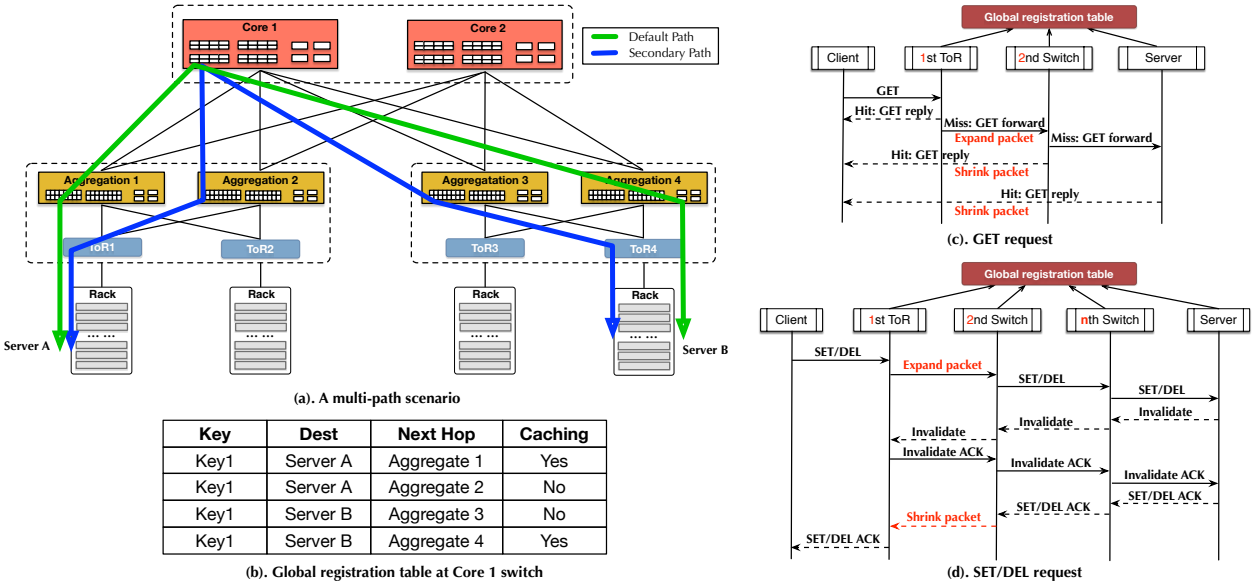
Therefore, we propose a hierarchical directory-based cache coherence protocol. Our key ideas are: (1) take advantage of the structured network topology by using a hierarchical distributed directory mechanism to record the sharers information; (2) decouple the system interface and program interface in order to provide flexible programmability; (3) support sequential consistency for high-performance SET/GET/DEL requests. We outline the scheme below, wherein we initially constrain

the network topology to a tree topology and then generalize it to a multi-rooted multi-path topology in the subsequent section.

We place the home directory at the end-host server. Note that in the tree topology, a source-destination pair uniquely determines the entire communication path and all of the switches that can potentially cache the key-value entry. Hence, the directory only needs to record the source address to be able to infer all other potential sharers along the path. Switches (including ToR/Aggregation/Core) will route an "in-network" labeled packet to the IncBox unit for further processing. For a GET request (Figure 3.5-a), if there is a hit, the IncBox replies with the key's value directly. If not, the GET request will be forwarded to the end-host server to fetch the data. For a GET reply (Figure 3.5-a), each IncBox on the path will cache the data in the hash table. For SET and DELETE operations (Figure 3.5-b), we use a write invalidation mechanism. On a SET or DELETE request, switches first forward the request to the home node and then the home node issues an invalidation request to all sharers in the directory. IncBox units receiving this request will delete the data (specified in the request) from the hash table. The IncBox at the client-side ToR switch will respond with an invalidation ACK. After the home node receives all the expected invalidation ACKs, it performs the SET or DELETE operation and then sends a set/delete ACK to the client.

3.5.4 Extension for Multi-rooted Tree Topologies

We discuss how to address the issue of multiple network paths between source-destination pairs, as would be the case in multi-rooted network topologies. As existing data centers provide high aggregate bandwidth and robustness through multiple paths in a multi-rooted structure, this creates a challenge for our original cache coherence design—recording only the source/destination address is not sufficient to reconstruct the sharers list. For example, GET and SET requests might not traverse the same path, and a SET invalidation message might not invalidate the switches that were on the path of the GET response. We adapt our original cache coherence protocol to record path-level information for data requests and also traverse predictable paths in the normal case, thus minimizing the number of invalidate or update messages for the cache coherence module.



Key	Dest	Next Hop	Caching
Key1	Server A	Aggregate 1	Yes
Key1	Server A	Aggregate 2	No
Key1	Server B	Aggregate 3	No
Key1	Server B	Aggregate 4	Yes

(b). Global registration table at Core 1 switch

Figure 3.6: Cache coherence for a multi-rooted tree topology. (a) presents a typical multi-path scenario. Specially, it shows two communication paths between server A and server B; (b) gives an example about the global registration table and designated caching; (c) and (d) shows how to use proposed methods to maintain coherence for GET/SET/DELETE requests.

Figure 3.6-a shows a multi-path scenario. It has three layers of switches: ToR, Aggregation, and Core. Each switch connects to more than one upper-layer switch for greater reliability and performance. We assume each switch routes based on ECMP. To handle the multi-path scenario, our key ideas are as follows. (1) ***Designated caching*** ensures that data is only cached at a set of predefined IncBox units for a given key. Any IncBox units between the designated IncBox units will not cache that key, meaning that routing between designated units can use any of the feasible paths. (2) ***Deterministic path selection*** ensures that for a given key and given destination, the end-host and IncBox units will choose a consistent set of designated caching IncBox units to traverse. (3) ***A global registration table*** (Figure 3.6-b), replicated on each IncBox, is used to identify the designated cache locations and designated paths for a given key and destination (indexed by a hash of the key rather than the key itself to bound its size). (4) ***A fault-tolerance mechanism*** uses non-

designated paths in the event of failures, but will not allow the data to be cached on the IncBox units traversed along the non-designated paths.

Our mechanism addresses the multi-path scenario by breaking a single (src, dest) flow into several shorter distance flows; for example, the path tuple (client, server) is transformed to a sequence of sub-paths: (client, designated ToR1) + (designated ToR1, designated Aggregation1/Core1) + ... + (designated ToR2, server). This means that we have to perform additional packet re-encapsulation operations at designated IncBox units along a designated path, which is not necessary in the single path case. The global registration table is preloaded on each IncBox and will be updated by a centralized SDN controller upon failures.

We illustrate our proposal using the example in Figure 3.6-a and Figure 3.6-b. The designated path between *Server A* and *Server B* for *Key1* is shown as the green/light colored path: *Server A* $\leftrightarrow$ *ToR1* $\leftrightarrow$ *Aggregation1* $\leftrightarrow$ *Core1* $\leftrightarrow$ *Aggregation4* $\leftrightarrow$ *ToR4* $\leftrightarrow$ *Server B*. Note that this is symmetric; communication in either direction uses the same path for a given key.

This designated path is always chosen for *Key1* when there are no failures. If *Core1* or *ToR1* detects a malfunction on one of *Aggregation4*'s links, they will instead use the secondary path through *Aggregation3* (the dark/blue path). However, given the global registration table at *Core1* (Figure 3.6-b), *Key1* will not be cached at *Aggregation3*. IncBox routes to the next designated caching location using network-layer ECMP, utilizing any of the paths connecting the IncBox to the next designated caching location, and skipping over the intermediate IncBox units.

We see two benefits of our design. First, the use of a designated path ensures that coherence messages traverse the same paths as the original cache request. It also increases the effectiveness of caching as the same key is not cached at multiple IncBox units at the same level of the network topology (other than ToR); this better utilizes total cache space and reduces invalidation overheads. Second, the ability to label designated caching locations provides a flexible policy mechanism that can take advantage of deployment and workload properties of the application. For example, Facebook deploys Memcached servers within a cluster and the clients in a different cluster. In such a setting, it might be more beneficial to perform caching only inside the client cluster as opposed to doing it also in the server cluster. The designated caching locations mechanism helps address

such deployment concerns.

Now, we discuss in detail how to handle different requests:

GET requests and replies: As Figure 3.6-c shows, GET requests and replies behave similarly as in the single-path case. There are three differences. First, both IncBoxes and end servers calculate the next designated caching location based on the global registration table. Second, since the original packet flow has been transformed as discussed above, the IncBox unit at the first ToR switch (which connects to the client) will "expand" the packet to include a copy of the original client source address to be used in forming the reply. This address is removed from the packet (the packet is "shrunk") before sending the reply to the client. (3) For a GET reply, an IncBox will cache the data only if it is one of the designated caching locations.

SET/DEL requests and ACKs: As Figure 3.6-d shows, for a SET/DEL request, the IncBox unit will (1) interact with the global registration table and (2) perform packet expansion/shrink operations, which are the same as in GET requests and replies. Invalidation requests are generated by the home server. When a switch receives one, if the associated IncBox is not the destination, the packet will be forwarded; otherwise, it will be sent to its network processor, which will (1) invalidate the data, (2) calculate the next IncBox to be invalidated using the global registration table, and (3) modify the packet header and send it out. The IncBoxes at client-side ToRs respond to the home server with invalidation ACKs, and the home server sends a SET/DEL ACK back to the requester when the coherence operation is complete (which is detected by counting the number of invalidation ACKs).

3.5.5 Optimization Between Switch and Network Processor

As described previously, the switch in an IncBox by default forwards any in-network caching requests to its network processor for further processing. For GET or DEL requests, if the key is not cached at the IncBox, this default operation will waste (1) one round trip between a switch and its network processor and (2) some processing time within the network processor. We observe that for a 1KB packet, these missed GET/DEL requests on average add 2.5us forwarding time at

the switch and 2.7–5.0us computation time at the IncBox. This becomes even worse if it happens multiple times along one request path. Therefore, we propose an optimization that uses the switch's Forwarding Database (FDB) and associated management handlers to mitigate this overhead.

Our proposal works in the following way. First, we program the switch parser to extract the key hash field as well as MAC source and destination addresses. Second, we modify the behavior of the layer 2 processing stage of the switch's forwarding pipeline for certain in-network cache packets. For GET requests, instead of using the MAC destination address to perform FDB lookup, we use an artificial MAC address formed by concatenating a locally administered MAC prefix [85] with the key hash. If it's a hit, the packet is forwarded to the attached IncBox network processor that caches the key. Otherwise, the switch forwards it to the end server. For GET replies, after the IncBox network processor finishes cache operation execution and responds to the switch, we trigger the switch's MAC learning handler on its management CPU, using it to insert the key hash into the FDB. For invalidations, the ACK from the IncBox's network processor triggers another handler that removes the key hash from the FDB. To guarantee consistency, these switch FDB operations are triggered only after the IncBox network processor's work is complete. This approach adds no additional latency to the switch's forwarding pipeline, and all non-INC packet types are processed exactly as before.

This approach has two limitations. First, two keys may have the same hash, so requests for keys not cached in an IncBox could be forwarded to its network processor. In this case the request is forwarded back towards the end server as it would be without this optimization. Two keys with the same hash may also be cached in the same IncBox; in this case, reference counts in the hash table can be used to ensure the FDB entry is not removed until both keys are evicted. Second, the FDB has limited size due to the switch's resource constraints (16K entries in our testbed). To cache more than this many keys in an IncBox, a smaller, more collision-prone hash may be used in the FDB. Alternatively, heavy-hitter detection techniques may be used to identify a subset of frequently-accessed keys that can benefit from this optimization, while other keys would follow the normal, slower path.

3.5.6 Fault Tolerance

In this section, we discuss how to handle failures. Generally, a failure must be one of the following: a client failure, a non designated switch/IncBox failure, a designated switch/IncBox failure, or a server failure. We categorize link failures as that of a switch/IncBox that is reached through that link.

Client failures could happen at three points in time: (1) before sending a request; (2) after sending a request but before receiving an ACK or data response packet; or (3) after receiving an ACK. None of these have an impact on the behavior of other clients and servers. Non-designated switch failure will result in path disconnection, in which case the preceding switch/IncBox would choose another path using ECMP. For a designated switch/IncBox failure, a request's whole communication path will be disconnected, and a backup path without caching will be employed until the global controller recomputes a new global registration table. Messages sent by a server (e.g., invalidations) could be lost, in which case the messages are retransmitted after a timeout. The server itself could fail either before or after sending the ACK or data result back. In the latter case, the data in the IncCache will be consistent but not available for updates. In the former case, the data will be inconsistent, cannot be updated, but the operation would not have been deemed complete, so the application can perform appropriate exception handling.

3.5.7 Compute Primitives and Client Side APIs

IncCache's flexible packet format supports multiple compute primitives. While this work focuses primarily on the cache, we implemented two simple compute operations as a case study: conditional put and increment. They work as follows. When an IncBox or home server receives a compute request, it (1) reads the value from the cache; (2) if it is a miss, the request is forwarded; (3) if it is a hit, the IncBox or server directly performs the operation on the cached value (e.g., increments the value); (4) coherence operations are then performed; and (5) the response is sent back to the client. As should be apparent, there are many design trade-offs about how the compute layer should interact with the caching layer. For example, when there is a key miss, it is also possible for

the IncBox to issue a fetch request and perform the operation locally (as in a write-allocate cache). For this case study, we left our cache behavior unchanged. A more complete API and exploration of design trade-offs is left to future work.

The client-side API includes three classes of operations: (1) initialization/teardown calls, (2) IncCache accesses, and (3) compute case-study operations. **`inc_init()`** can be used to enable/disable the IncCache layer and configure the coherence protocol; **`inc_close()`** tears it down. **`inc_{get(), set(), del()}`** read, write, and delete keys from the IncCache and block to guarantee data coherence. **`inc_increment()`** adds its integer value argument to a specified key. **`inc_cond_put()`** sets a key to its value argument if the cached value is smaller or unset.

3.5.8 IncCache Implementation

IncCache is implemented across end-host servers, switches, and network accelerators. The server-side implementation is built following the model of Memcached and can support native Memcached traffic with modifications to the packet format. We do not take advantage of fast packet processing libraries (e.g., DPDK [59]) at present because most GET requests are served by IncBox units. Moreover, our server-side could be easily replaced with a high-performance key-value store, like MICA [193] or Masstree [200], with modifications to perform coherence operations.

We program the Cavium XPliant switch with its SDK, which includes C++ APIs to access hardware features from the management processor, as well as a proprietary language to program the forwarding pipeline. The implementation of the network accelerator component is based on the Cavium Development Kit [79] and makes heavy use of the hardware features provided by the OCTEON-II network processor. We use a fixed-length header for INC packets to accelerate parsing. For hashtable entries, we pre-allocate a big memory space ("arena" in the CDK) during the kernel loading phase and then use a private memory allocator (dlmalloc2) to allocate from that arena. There are multiple memory spaces in the processor, and we found that careful use is required to avoid considerable overhead. We also use the hardware-accelerated hash computation

engine and the fetch and add accelerator (FAU) to hash keys and access lock-free data structures. As packet headers need frequent re-encapsulation, we use fast packet processing APIs to read and write the header. When generating a packet inside the accelerator, we take advantage of the DMA scatter and gather functions provided by the transmit units.

3.6 Evaluation

In this section, we demonstrate how IncBricks impacts a latency-critical data center application—an in-memory key-value store similar to Memcached [147]. Similar architectures are used as a building block for other data center workloads [191, 126]. We evaluate this with a real system prototype, answering the following questions:

- How does IncBricks impact the latency and throughput of a key-value store cluster under a typical workload setup?
- What factors will impact IncBricks performance?
- What are the benefits of doing computation in the cache?

3.6.1 Experimental Setup

Platform. Our testbed is realized with a small-scale cluster that has 24 servers, 2 switches, and 4 network accelerators. Detailed information of each component is shown in Table 3.1. In our experiments, we set up a two-layer network topology where 4 emulated ToR switches connect to 2 emulated root switches; each ToR connects to both root switches. We emulate two ToR switches and one root switch in each Cavium switch by dividing the physical ports using VLANs. The IncBox at each ToR and Root switch is equipped with 4096 and 16384 entries, respectively, except for the cache size experiment.

Workload. For the throughput-latency experiment, we set up a 4-server key-value store cluster and use the other 20 servers as clients to generate requests. We use our custom key-value request generator which is similar to YCSB [128]. Based on previous studies [110, 213], the workload is configured with (1) a Zipfian distribution [99] of skewness 0.99; (2) 95%/5% read/write ratio; (3)

Component	Description
Server	Dual-socket. A total of two Intel Xeon 6-core X5650 processors (running at 2.67GHz) and 24GB memory. The server is equipped with a Mellanox MT26448 10Gbit NIC.
Switch	Cavium/XPliant CN88XX-based switch with 32 100Gbit ports, configured to support 10Gbit links between emulated ToR switches and clients, and 40Gbit between emulated ToR and Root switches.
Network Accelerators	Two Cavium LiquidIO boards, each with one CN6640 processor (8 cnMIPS64 cores running at 0.8GHz) and 2GB memory. Two OCTEON-II EEB68 evaluation boards, with one CN6880 processor (32 cnMIPS64 cores running at 1.2GHz) and 2GB memory.

Table 3.1: Testbed details. Our IncBox comprises a Cavium switch and a network accelerator (OCTEON or LiquidIO).

1 million keys with 1024B value size; (4) two deployments: within-cluster and cross-cluster. The compute performance evaluation uses the full cluster with a uniform random load generator.

3.6.2 Throughput and Latency

In this experiment, we compare the throughput and latency of a vanilla key-value cluster with that of one augmented with IncBricks. Figures 3.7 and 3.8 report the average latency versus throughput under two cluster deployments. This comparison shows:

- At low to medium request rates (less than 1 million ops/s), IncBricks provides up to 10.4 us lower latency for the within-cluster case. The cross-cluster deployment provides an additional 8.6 us reduction because of the additional latency at Root and ToR switches. For high request rates (at 1.2 million ops/s), IncBricks saves even more: 85.8 us and 123.5 us for the within-cluster and cross-cluster cases, respectively.

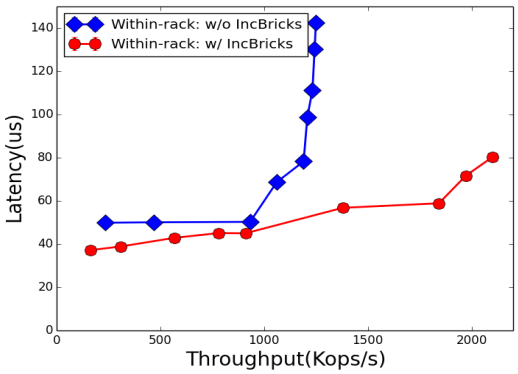


Figure 3.7: Average latency versus throughput in the within-cluster scenario.

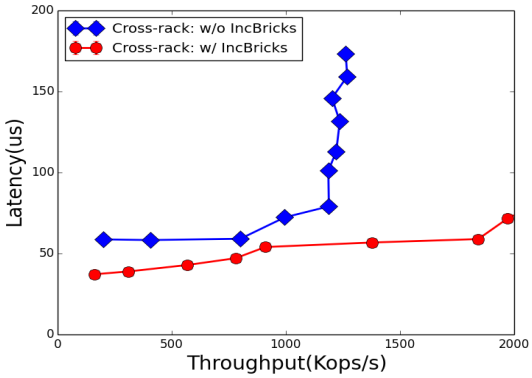


Figure 3.8: Average latency versus throughput in the cross-cluster scenario.

- IncBricks is able to sustain higher throughput than the vanilla key-value cluster for all scenarios. For example, IncBricks provides 857.0K ops/s and 458.4K ops/s more throughput for the within-cluster and cross-cluster scenarios, respectively. Furthermore, our measured maximum throughput is limited by the number of client nodes in our cluster and the rate at which they generate requests.
- This experiment does not include switch optimization. If it did, our measurements suggest that we could save up to an additional 7.5 us in each IncBox, by reducing both (1) communication time between the switch and accelerator and (2) execution time inside the accelerator. This would provide significant benefits in the cross-cluster scenario since it has more hops.

SET request latency IncBricks SET requests have higher latency than the vanilla case due to cache coherence overhead. In this section, we break down SET processing latency into four components: server processing, end-host OS network stack overhead, switch forwarding, and IncBox processing. Figure 3.9 shows this for both the within-cluster and cross-cluster scenarios.

First, server processing and the client and server OS network stack take the same amount of time (20 us) for both scenarios because the end-hosts behave the same in each. To reduce these components, we could apply kernel bypass techniques[59] or use a higher-performance key-value

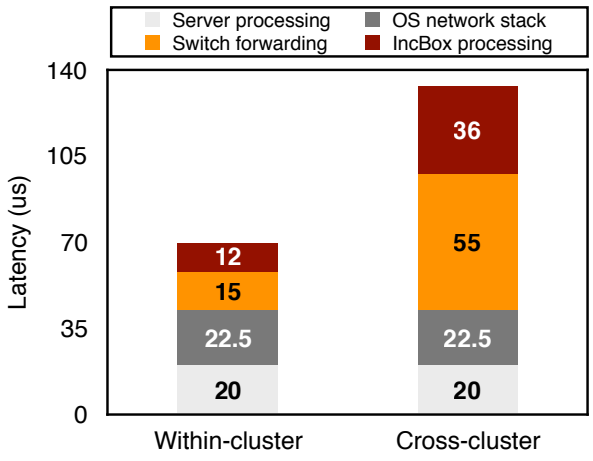


Figure 3.9: SET request latency breakdown for within-cluster and cross-cluster cases.

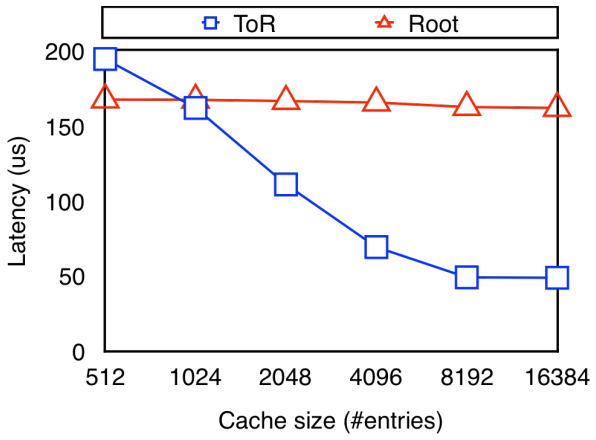


Figure 3.10: The impact of cache size at ToR and Root IncBoxes. y-axis is the average request latency.

store instead (e.g., MICA[193] or Masstree[200]). Second, the switch forwarding latency increases from 21.5% of the total latency in the within-cluster case to 41.2% of the total latency in the cross-cluster case. This is due to the larger number of hops between clients and servers, which both requests and replies as well as invalidation traffic must traverse. Third, IncBox processing latency is higher (36 us) in the cross-cluster scenario compared with the within-cluster one (12 us) for the same reason. The optimization from Section 3.5.5 can help reduce this.

3.6.3 Performance Factors of IncBricks

We explored two parameters that impact IncBricks performance: cache size and workload skewness.

Cache size Figure 3.10 shows the result of varying the number of entries in the IncCache for both ToR and Root IncBoxes. We find that ToR cache size has a significant impact. As we increase the number of available cache entries from 512 to 16384, the average request latency decreases from 194.2 us to 48.9 us. Since we use a Zipfian distribution with skewness 0.99, 4096 entries covers

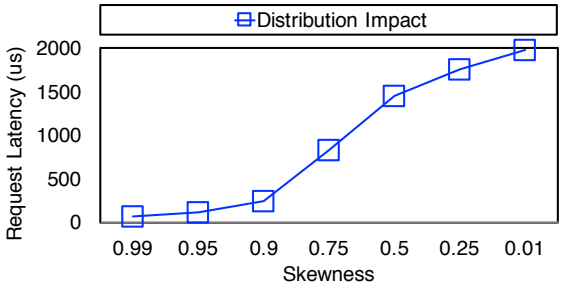


Figure 3.11: The impact of skewness on IncBricks for the cross-cluster case. y-axis is the average request latency.

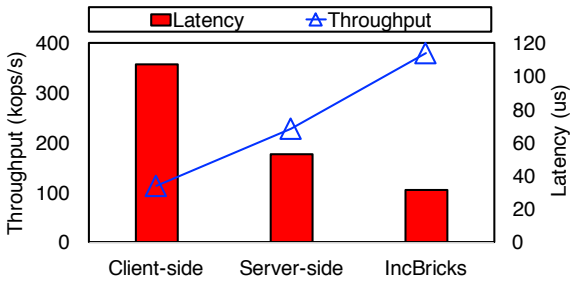


Figure 3.12: Performance comparison of the conditional put command among client-side, server-side, and IncBricks

60% of the accesses, and 16384 entries covers 70%. That is why larger cache sizes provide less performance benefits. We also find that Root cache size has little impact on performance. This is due to our cache being inclusive; it is hard for requests that miss in the ToR IncBox to hit in a Root IncBox. This suggests that a ToR-only IncCache might provide the bulk of the benefit of the approach, or that moving to an exclusive cache model might be necessary to best exploit IncCaches at root switches.

Workload skewness We vary the skewness from 0.99 to 0.01 and find that the average request latency at the client ToR IncBox increases significantly (at most 2 ms). Since our workload generates 1M keys and there are 4096 entries in the ToR IncCache, there will be many cache miss events as the workload access pattern becomes more uniform. These misses will fetch data from end-servers but also go through the IncBricks caching layer, adding latency. This may have two other consequences: the additional misses may cause network congestion on the key path, and the high miss rate may lead to high cache eviction overhead in the memory management layer of each IncBox unit. This suggests that a mechanism to detect "heavy-hitter" keys and bias the cache toward holding these hot keys could be beneficial. We plan to explore this in future work.

3.6.4 Case Study: Conditional Put and Increment

This experiment demonstrates the benefit of performing compute operations in the network. As described above, we use conditional puts to demonstrate this capability. We compare three implementations: (1) client-side, where clients first fetch the key from the key-value servers, perform a computation, and then write the result back to the server, (2) server-side, where clients send requests to the server and the server performs the computation locally and replies, and (3) IncBricks, where computation is done in the network. We use a simple request generator for this experiment that generates uniform random operands that are applied to one of the keys, selected uniformly at random.

Figure 3.12 reports both latency and throughput for these three cases. The client-side implementation must perform both a GET and a SET for each operation, leading to 106.9 us latency and 113.3K ops/s. The server-side implementation only sends one message per remote operation and thus obtains better performance: 52.8 us and 228.1K ops/s. IncBricks performs the best since the operation can be performed directly by the IncBox, which results in 31.7 us latency and 379.5K ops/s. We also conducted the same experiment for increment operations, and the results were essentially identical and thus are omitted.

3.7 Related Work

3.7.1 In-network Aggregation

Motivated by the partition/aggregation model [104] used by many data center applications (e.g., [257, 162, 135]), in-network aggregation techniques have been developed to mitigate bandwidth constraints during the aggregation phase of these applications. Camdoop [131] exploits a low-radix directly-connected network topology to enable aggregation during the MapReduce shuffle phase. Servers are responsible for forwarding packets through the network, making computation on data traversing the network straightforward. However, this network is very different from the high-radix indirect topologies commonly seen in data centers today. NETAGG [199] targets similar

aggregation opportunities, but on common data center network topologies, by attaching software middleboxes to network switches with high-capacity links. A set of shims and overlays route requests to middleboxes to be aggregated on their way to their final destination. In contrast to these largely-stateless approaches, IncBricks supports stateful applications with its coherent key-value cache. Data aggregation operations could be implemented as applications on top of IncBricks.

3.7.2 In-network Monitoring

The limited compute capability found in switches today is largely intended for collecting network performance statistics (queue occupancy, port utilization, etc.) in order to support tasks such as congestion control, network measurement, troubleshooting, and verification. Minions [165] proposes embedding tiny processors in the switch data plane to run small, restricted programs embedded in packets to query and manipulate the state of the network. Smart Packets [233] explores a similar idea but in the switch control plane, with fewer restrictions on what could be executed, making it harder to sustain line rate. In contrast, the vision of IncBricks is to repurpose this in-switch compute capability (which was originally intended for these administrator-level tasks) and network accelerators, for user-level applications.

3.7.3 Network Co-design for Key-value Stores

Dynamic load balancing is a key technique to ensure that scale-out storage systems meet their performance goals without considerable over-provisioning. SwitchKV [192] is a key-value store that uses OpenFlow-capable switch hardware to steer requests between high-performance cache nodes and resource-constrained backend storage nodes based on the content of the requests and the hotness of their keys. IncBricks also does content-based routing of key-value requests, but by dispersing its cache throughout the network it can reduce communication in the core of the network, avoiding the hot spots SwitchKV seeks to mitigate.

3.8 Summary and Learned Lessons

This chapter presents IncBricks, a hardware-software co-designed in-network caching fabric with basic computing primitives for data center networks. IncBricks comprises two components: (1) IncBox, a programmable middlebox combining a reconfigurable switch and a network accelerator, and (2) IncCache, a distributed key-value store built on IncBoxes. We prototype IncBricks in a real system using Cavium XPliant switches and OCTEON network accelerators. Our prototype lowers request latency by over 30% and doubles throughput for 1024 byte values in a common cluster configuration. When doing computations on cached values, IncBricks provides 3 times more throughput and a third of the latency of client-side computation. These results demonstrate the promise of in-network computation.

We learned a couple of lessons from building IncBricks. First, approximation is the key technique to cope with resource constraints on a programmable switch. For example, one should take advantage of the probabilistic data structures (like count-min sketch, bloom filter) to save states due to the storage limit. One should tailor the application logic that can be fit into one ALU unit and spread computation across several stages. Second, co-design the switch and accelerator logic is helpful. Switches can do certain things very effectively, such as application-level monitoring and traffic control over all terabits of traffic. It can forward the interesting subset of traffic to the network accelerator. Accelerators can maintain interesting data structures with a reasonable amount of memory. It can also handle the modest amounts of traffic forwarded to it by the switch. Finally, IncBricks is one kind of disaggregation designs that fully explore in-network computing capabilities. Recently, there is some existing work [167, 166] that tries to fit the application into the switch to take advantage of its high computational bandwidth. We instead take a different approach that combines the programmable switch with an accelerator. This allows for the use of different types of accelerators as needed. It also allows for the independent scaling of accelerators and offloading processing to customized hardware.

4 iPipe: Offloading Distributed Actors onto SmartNICs

Emerging SmartNICs, enclosing rich computing resources (e.g., a multi-core processor or FPGAs, reconfigurable match-action tables, onboard SRAM/DRAM, accelerators, programmable DMA engines), hold the potential to offload generic data center server tasks. However, it is unclear how to use a SmartNIC efficiently and maximize the offloading benefits, especially for distributed applications. In this chapter, we firstly characterize commodity SmartNICs and summarize the offloading performance implications from four perspectives: traffic control, computing capability, onboard memory, and host communication. Next, based on our characterization, we build the second PNF-enabled system (called iPipe), an actor-based framework for offloading distributed applications onto SmartNICs. At the core of iPipe is a hybrid scheduler, combining FCFS and DRR-based processor sharing, which can tolerate tasks with variable execution costs and maximize NIC compute utilization. We built three applications (i.e., a real-time data analytics engine, a distributed transaction system, and a replicated key-value store) using iPipe, and evaluate them on our prototyped systems.

This chapter is organized as follows. Section 4.1 describes the background and motivation for SmartNIC offloading. Section 4.2 highlights the overall design of the iPipe framework. Section 4.3 presents the characterization results and summarizes the design implications. We describe the design and implementation of iPipe framework and applications in Sections 4.4 and 4.5. Section 4.6 reports our evaluation results. We review relevant studies in Section 4.7. Section 4.8 concludes this chapter and discusses our learned lessons.

4.1 Background

SmartNICs have emerged for the data center, aiming to mitigate the gap between increasing network bandwidth and stagnating CPU computing power [2, 3, 10]. In the last two years, major network hardware vendors have released different SmartNIC products, such as Mellanox's Bluefield [68], Broadcom's Stingray [42], Marvell(Cavium)'s LiquidIO [43], Huawei's IN5500 [16], Netronome's Agilio [76], Mellanox Innova [69], and Alpha Data [27]. They not only target acceleration of traditional networking/storage protocol processing (e.g., OVS/RoCE/iWARP/TCP offloading, traffic monitoring, firewall, etc.), but also bring a new computing substrate into the data center to expand the server computing capacity at a lower cost: SmartNICs usually enclose simple microarchitecture computing cores or reconfigurable hardware logics that are cheap and cost-effective.

Generally, these SmartNICs comprise a multi-core, possibly wimpy, processor (i.e., MIPS/ARM ISA), FPGAs, onboard SRAM/DRAM, packet processing/domain-specific accelerators, and programmable DMA engines. Different architectural components are connected by high-bandwidth coherent memory buses, or high-performance interconnects. Today, most of these SmartNICs are equipped with one or two 10/25GbE ports, and 100GbE is on the horizon. These rich computing resources allows hosts to offload generic computations (with complex algorithms and data structures) without sacrificing performance (i.e., latency/throughput) and program generality. ***The key question we ask in this project is how to use these SmartNICs efficiently to maximize such benefits for distributed applications?***

There have been some recent research efforts that offload networking functions onto FPGA-based SmartNICs (e.g., ClickNP [189], AzureCloud [146]). They take a conventional domain-specific acceleration approach that consolidates as much of the application logic onto FPGA programmable logic blocks. This approach is applicable to a specific class of applications that exhibit sufficient parallelism, deterministic program logic, and regular data structures that can be synthesized efficiently on FPGAs. Our focus, on the other hand, is to target distributed applications with complex data structures and algorithms that cannot be realized efficiently on FPGA-based

SmartNICs.

In this chapter, we'll describe how to efficiently use SmartNICs diverse computing resources to offload distributed applications. We build a programming framework along with a runtime to achieve this.

4.2 Solution Overview

We first perform a detailed performance characterization of four commodity SmartNICs (i.e., LiquidIOII CN2350, LiquidIOII CN2360, Bluefield 1M332A, and Stingray PS225), shown in Table 2.1. We categorize the Multi-core SoC SmartNIC into four architectural components – traffic control, computing units, onboard memory, host communication – and use microbenchmarks to understand their performance implications. The experiments identify the resource constraints that we have to be cognizant of, illustrate the utility of a SmartNIC's hardware acceleration units, and provide guidance on how to efficiently utilize the SmartNIC resources.

We design and implement the iPipe framework based on our characterization observations. iPipe introduces an actor programming model for distributed application development. Each actor has self-contained private states and communicates with other actors via messages. Our framework provides a distributed memory object abstraction and enables actor migration, responding to dynamic workload changes and ensuring the delivery of line rate traffic. A central piece of iPipe is the actor scheduler that combines the use of FCFS and DRR-based processor sharing, in order to tolerate tasks with variable execution costs and maximize the SmartNIC computing resource utilization. iPipe allows multiple actors to coexist safely on the SmartNIC, protecting against actor state corruption and denial-of-service attacks.

We prototype iPipe and build three applications (i.e., a real-time data analytics engine, a distributed transaction processing system, and a replicated key-value store) using commodity 10GbE/25GbE SmartNICs. We evaluate the system using an 8-node testbed and compare the performance against DPDK-based implementations. Our experimental results show that we can significantly reduce the host load for three real-world distributed applications; iPipe saves up to 3.1/2.2 beefy Intel cores

used to process 25/10Gbps of application bandwidth, along with up to 23.0μs and 28.0μs savings in request processing latency.

4.3 Understanding the SmartNIC

This section characterizes the SmartNIC from four perspectives (i.e., traffic control, computing units, onboard memory, host communication), and summarizes implications that guide the design of iPipe.

4.3.1 Experiment Setup

We use Supermicro 1U boxes as host servers for both the client and server and an Arista DCS-7050S/Cavium XP70 ToR switch for 10/25GbE network. The client is equipped with a dumb NIC (i.e., Intel XL710 for 10GbE and Intel XXV710-DA2 for 25GbE). We insert the SmartNIC on one of the PCIe 3.0 $\times$8 slots on the server side. The server box has a 12-core E5-2680 v3 Xeon CPU running at 2.5GHz with hyperthreading enabled, 64GB DDR3 DRAM, and 1TB Seagate HDD. When evaluating Bluefield and Stingray cards, we use a 2U Supermicro server with two 8-core Intel E5-2620 v4 processors at 2.1GHz, 128GB memory, and 7 Gen3 PCIe slots.

We take the DPDK pkt-gen as the workload generator and augment it with the capability to generate different application layer packet formats at the desired packet interval. We report end-to-end performance metrics (e.g., latency/throughput), as well as microarchitectural counters (e.g., IPC, L2 cache misses per kilo instruction or MPKI).

4.3.2 Traffic Control

As described above, traffic control is responsible for two tasks: packet forwarding (through TX/RX ports) and packet feeding to the NIC computing cores. Here, we use an ECHO server that entirely runs on a SmartNIC to understand: (1) how many NIC cores are sufficient to saturate the link speed for different packet sizes and how much computing capacity is left for other "offloaded

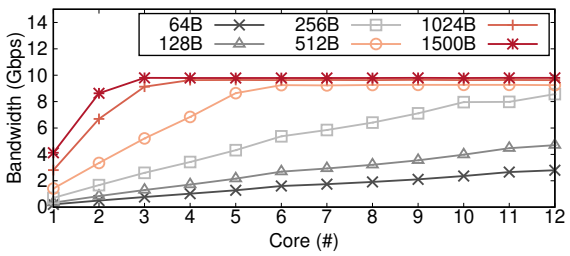


Figure 4.1: SmartNIC bandwidth varying the number of NIC cores for the 10GbE LiquidIOII CN2350.

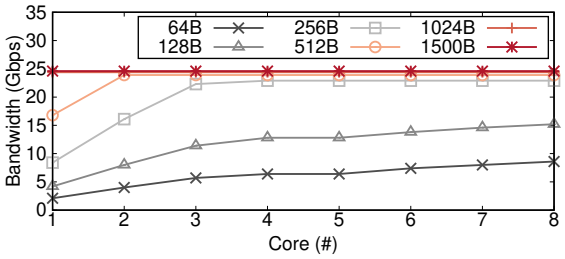


Figure 4.2: SmartNIC bandwidth varying the number of NIC cores for the 25GbE Stingray PS225.

applications"; (2) what are the synchronization overheads in supplying packets to multiple NIC cores.

Figures 4.1 and 4.2 present experimental data for 10GbE LiquidIOII CN2350 and 25GbE Stingray PS225. When packet size is 64B/128B, neither NICs can achieve full link speed even if all NIC cores are used. However, when packet size is 256B/512B/1024B/1500B(MTU), the LiquidIOII requires 10/6/4/3 cores to achieve line rate, while Stingray takes 3/2/1/1 cores. Stingray uses fewer cores due to its much higher core frequency (3.0GHz v.s. 1.20GHz). This indicates that packet forwarding is not free, which is the default execution tax of a SmartNIC. Figure 4.3 reports the average and tail (p99) latency when achieving the maximum throughput for four different packet sizes using 6 and 12 cores. Interestingly, the latencies don't increase as we increase the core count; compared to the 6 core case, the 12 core experiments only add 4.1%/3.4% average/p99 latency on average across the four scenarios. Such results can also be observed from the Stingray card. This means that the hardware traffic manager is effective at providing a shared queue abstraction with little synchronization overhead for the packet buffer management.

Design implications: ***I1:*** Not only is packet forwarding not free, but the packet size distribution of incoming traffic significantly impacts the availability of computing cycles on a Multi-core SmartNIC. One should monitor the packet (request) sizes and adaptively make the offloading decisions. ***I2:*** Hardware support reduces synchronization overheads and enables scheduling paradigms

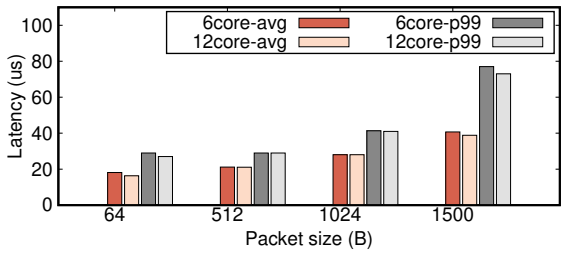


Figure 4.3: Average/p99 latency when achieving the max throughput on the 10GbE LiquidIOII CN2350.

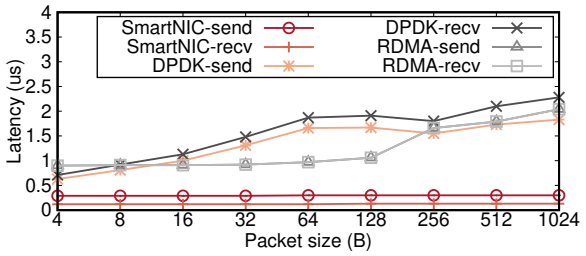


Figure 4.4: Send/Recv latency on the 10GbE LiquidIOII CN2350, compared with RDMA/DPDK ones.

that involve multiple workers pulling incoming traffic from a shared queue.

4.3.3 Computing Units

To explore the execution behavior of the computing units, we use the following: (1) a microbenchmark suite comprising of representative in-network offloaded workloads from recent literature; (2) low-level primitives to trigger the domain-specific accelerators. We conduct experiments on the 10GbE LiquidIOII CN2350 and report both system and microarchitecture results in Table 4.1. We observe the following results. First, the execution times of the offloaded tasks vary significantly from 1.9/2.0us (replication and load balancer) to 34.0/71.0us (ranker/classifier). Second, low IPC[1] or high MPKI are indicators of high computing cost, as in the case of the rate limiter, packet scheduler, and classifier. Tasks with high MPKI are memory-bound tasks that are less likely to benefit from the complex microarchitecture on the host, and they might be ideal candidates for offloading. Third, SmartNIC accelerators provide fast domain-specific processing appropriate for networking/storage tasks. For example, the MD5/AES engine is 7.0X/2.5X faster than the one on the host server (even using the Intel AES-NI instructions). However, invoking an accelerator is not free since the NIC core has to wait for the execution completion and also incurs cache misses

[1]Note that the cnMIPS OCTEON [44] is a 2-way processor and the ideal IPC is 2.

Applications	Comp.	DS	Exe. Lat.(us)	IPC	MPKI	Acl.	IPC	MPKI	Exe. Lat.(us)		
									bsz=1	bsz=8	bsz=32
Baseline (echo)	N/A	N/A	1.87	1.4	0.6	**CRC**	1.2	2.8	2.6	0.7	0.3
Flow monitor [235]	Count-min sketch	2-D array	3.2	1.4	0.8	**MD5**	0.7	2.6	5.0	3.1	3.0
KV cache [188]	key/value Rr/Wr/Del	Hashtable	3.7	1.2	0.9	**SHA-1**	0.9	2.6	3.5	1.2	0.9
Top ranker [218]	Quick sort	1-D array	34.0	1.7	0.1	**3DES**	0.8	0.9	3.4	1.3	1.1
Rate limiter [189]	Leaky bucket	FIFO	8.2	0.7	4.4	**AES**	1.1	0.9	2.7	1.0	0.8
Firewall [189]	Wildcard match	TCAM	3.7	1.3	1.6	**KASUMI**	1.0	0.9	2.7	1.1	0.9
Router [176]	LPM lookup	Trie	2.2	1.3	0.6	**SMS4**	0.8	0.9	3.5	1.4	1.2
Load balancer [140]	Maglev LB	Permut. table	2.0	1.3	1.3	**SNOW 3G**	1.4	0.5	2.3	0.9	0.8
Packet scheduler [105]	pFabric scheduler	BST tree	12.6	0.5	4.9	**FAU**	1.4	0.6	1.9	1.4	1.0
Flow classifier [195]	Naive Bayes	2-D array	71.0	0.5	15.2	**ZIP**	1.0	0.2	190.9	N/A	N/A
Packet replication [175]	Chain replication	Linklist	1.9	1.4	0.6	**DFA**	1.3	0.2	9.2	7.5	7.3

Table 4.1: Performance comparison among generic offloaded applications and accelerators for the 10GbE LiquidIOII CN2350. Request size is 1KB for all cases. We report both per-request execution time as well as microarchitectural counters. DS=Data structure. IPC=Instruction per cycle. MPKI=L2 cache misses per kilo-instructions. Acl=Accelerator. bsz=Batch size. DFA=Deterministic Finite Automation.

(i.e., higher MPKI) in feeding data to the accelerator. Batching can amortize invocation costs but result in tying up the NIC core for longer periods. Other SmartNICs (e.g., BlueField and Stingray)

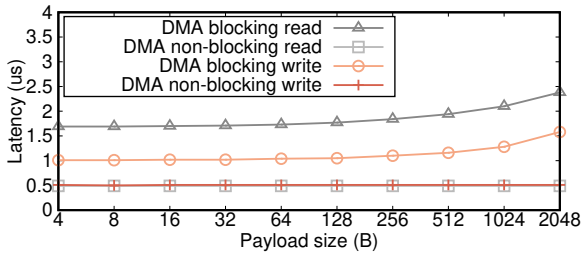


Figure 4.5: Per-core blocking/non-blocking DMA read/write latency when increasing payload size.

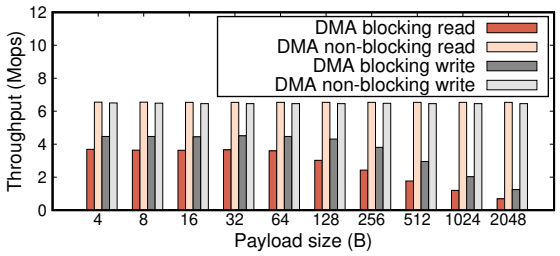


Figure 4.6: Per-core blocking/non-blocking DMA read/write throughput when increasing payload size.

display similar characteristics.

SmartNICs also usually enclose specialized accelerators for packet processing. Take the LiquidIOII ones (CN2350/CN2360) for example. It has packet input (PKI) and packet output units (PKO) for moving data between MAC and packet buffer, and a hardware managed packet buffer along with fast packet indexing. When compared with two host-side kernel-bypass networking stacks (DPDK/RDMA), even with the polling model, for SEND, it shows 4.6X/4.2X speedups on average across all cases (shown in Figure 4.4), respectively.

Design implications: ***I3:*** The offloading framework should be able to handle tasks with a wide range of execution latencies and simultaneously ensure that the NIC's packet forwarding is not adversely impacted. ***I4:*** One should take advantage of the available accelerators on the SmartNIC and perform batched execution for domain-specific ones if necessary (at the risk of increasing queueing for incoming traffic).

4.3.4 Onboard Memory

Generally, a SmartNIC has five onboard memory resources in its hierarchy: (1) Scratchpad/L1 cache is per-core local memory. It has limited size (e.g., LiquidIO has 54 cache lines of scratchpad) with fast access speed. (2) Packet buffer. This is onboard SRAM along with fast indexing. A SmartNIC (like LiquidIOII) usually has hardware-based packet buffer management. Some Smart-

	L1(ns)	L2(ns)	L3(ns)	DRAM(ns)
LiquidIOII CNXX	8.3	55.8	N/A	115.0
BlueField 1M332A	5.0	25.6	N/A	132.0
Stingray PS225	1.3	25.1	N/A	85.3
Host Intel Server	1.2	6.0	22.4	62.2

Table 4.2: Access latency of 1 cacheline to different memory hierarchies on 4 SmartNICs and the Intel server. The cacheline for LiquidIOII ones is 128B while the rest is 64B. The performance of LiquidIOII CN2350 and CN2360 is similar.

NICs (like Bluefield and Stingray) don't have a dedicated packet buffer region. (3) L2 cache, which is shared across all NIC cores. (4) NIC local DRAM, which is accessed via the onboard high-bandwidth coherent memory bus. Note that a SmartNIC can also read/write the host memory using its DMA engine (as evaluated in the next section).

We use a pointer chasing microbenchmark (with random stride distance) to characterize the access latency for different memory hierarchies for 4 SmartNICs and compare it with the host server. The results in Table 4.2 illustrate that there is significant diversity in memory subsystem performance across SmartNICs. Also, the memory performance of many of the SmartNICs is worse than the host server (e.g., the access latency of SmartNIC L2 cache is comparable to the L3 cache on the host server), but the well-provisioned Stingray has a performance comparable to the host.

Design implications: ***I5:*** When the application working set exceeds the L2 cache size of the SmartNIC, executing memory-intensive workloads on the SmartNIC might result in a performance loss than running on the host.

4.3.5 Host communication

A SmartNIC communicates with host processors using DMA engines through the PCIe bus. PCIe is a packet-switched network with 500ns-2us latency and 7.87 GB/s theoretical bandwidth per

Gen3 x8 endpoint (which is the one used by all of our SmartNICs). Its performance is usually impacted by many runtime factors. With respect to latency, DMA engine queueing delay, PCIe request size and its response ordering, PCIe completion word delivery, and host DRAM access costs will all slow down PCIe packet delivery [151, 171, 211]. With respect to throughput, PCIe is limited by transaction layer packet (TLP) overheads (i.e., 20-28 bytes for header and addressing), the maximum number of credits used for flow control, the queue size in DMA engines, and PCIe tags used for identifying unique DMA reads.

Generally, a DMA engine provides two kinds of primitives: blocking accesses, which wait for the DMA completion word from the engine, and non-blocking ones, which allow the processing core to continue executing after sending the DMA commands into the command queue. Figures 4.5 and 4.6 show our performance characterizations of the 10GbE LiquidIO CN2350. Non-blocking operations insert a DMA instruction word into the queue and don't wait for completion. Hence, its read/write latency and throughput are independent of packet size, which outperform the blocking primitives. For blocking DMA reads/writes, a large message can fully utilize the PCIe bandwidth. For example, with 2KB payloads, one can achieve 2.1/1.4 GB/s per-core PCIe write/read bandwidth, outperforming the 64B case by 8.7X/6.0X. This indicates that one should take advantage of the DMA scatter and gather technique.

Some SmartNICs (like BlueField and Stingray) expose RDMA verbs instead of native DMA primitives. We characterize the one-sided RDMA read/write latency from a SmartNIC to its host using the Mellanox Bluefield card, which resembles the DMA blocking operations. We observe similar results as the LiquidIOII ones, but it requires much larger payloads to amortize the verbs software processing cost.

SmartNICs can assist in intelligent packet steering. Multiqueue is a de-facto standard for today's modern NICs. By providing multiple transmit and receive queues, different host CPU cores can process packets simultaneously without coordination, achieving higher throughput. Normal NICs usually apply a hashing function to map network flows into different queues, referred as RSS [87] (receiver side scaling) or Intel flow director [61]. SmartNICs can push this benefit even further – by looking into the application header, one can perform application-level dispatching.

The performance benefits of this feature over host-side dispatching can be significant.

Design implications: ***I6:*** There are significant performance benefits to using non-blocking DMA and aggregating transfers into large PCIe messages (via DMA scatter and gather). In addition, SmartNICs can assist host processing by performing intelligent application-layer dispatching.

4.4 iPipe Framework

This section describes the design and implementation of our iPipe framework. We use the insights from our characterization experiments to address the following challenges.

- **Programmability:** A commodity server equipped with a SmartNIC is a non-cache-coherent heterogeneous computing platform with asymmetric computing power. We desire simple programming abstractions that can be used for developing general distributed applications.
- **Computation efficiency:** There are substantial computing resources on a SmartNIC (e.g., a multi-core processor, modest L2/DRAM, plenty of accelerators, etc.), but one should use them in an efficient way. Inappropriate offloading could cause NIC core overloading, bandwidth loss, and wasteful execution stalls.
- **Isolation:** A SmartNIC can hold multiple applications simultaneously. One should guarantee that different applications cannot touch each others' state, that there is no performance interference between applications; and tail latency increases, if any, are modest.

4.4.1 Actor Programming Model and APIs

iPipe applies an actor programming model [159, 103, 240] for application development. iPipe uses the actor-based model, instead of saying data-flow or thread-based models for the following reasons. First, unlike a thread-based model, actors interact with each other not through shared memory but with explicit messages. Given the communication latencies that we observe between the NIC and the host, explicit messaging is more appropriate in our setting. Second, the actor model is able to support computing heterogeneity and hardware parallelism automatically. While data-flow models also provide such support, the actor-based model allows for non-deterministic

and irregular communication patterns that arise in complex distributed applications. Finally, actors have well-defined associated states and can be migrated between the NIC and the host dynamically. This allows us to have dynamic control over the use of SmartNIC computing capabilities as we adapt to traffic characteristics (which is necessary given our characterization experiments).

An actor is a computation agent that performs two kinds of operations based on incoming type of messages: (1) trigger its execution handlers and manipulate its private state; (2) interact with other actors by sending messages asynchronously. Actors don't share memory. In our system, every actor has an associated structure with the following fields: (1) *init_handler* and *exec_handler* for state initialization and message execution; (2) *private_state*, which can use different data types (described below); (3) *mailbox* is a multi-producer multi-consumer concurrent FIFO queue, which is used to store incoming asynchronous messages; (4) *exec_lock*, used to decide whether an actor can be executed on multiple cores; (5) some runtime information, such as *port*, *actor_id*, and a reference to the *actor_tbl*, which contains the communication address for all actors. The structure outlined above represents a streamlined and light-weight actor design.

The iPipe runtime enables the actor-based model by providing support for actor allocation/destruction, runtime scheduling of actor handlers, and transparent migration of actors and its associated state for actor development. Specifically, iPipe has three key system components: (1) an actor scheduler that works across both SmartNIC and host cores and uses a hybrid FCFS/DRR scheduling discipline to enable execution of actor handlers with diverse execution costs; (2) a distributed memory object (DMO) abstraction that enables flexible actor migration, as well as support for a software-managed cache to mitigate the cost of SmartNIC to host communications; (3) a security isolation mechanism that protects actor state from corruption and denial-of-service attacks. We describe them in the following sections.

Table 4.3 presents the major APIs. Specifically, the actor management APIs are used by our runtime. We provide five calls for managing DMOs. When creating an object on the NIC, iPipe first allocates a local memory region using the *dlmalloc2* allocator and then inserts an entry (i.e., object ID, actor ID, start address, size) into the NIC object table. Upon *dmo_free*, iPipe frees the space allocated for the object and deletes the entry from the object table. *dmo_memset, dmo_memcpy,*

dmo_memmove resemble memset/memcpy/memmove APIs in glibc, except that it uses the object ID instead of a pointer. For the networking stack, iPipe takes advantage of packet processing accelerators to build a shim networking stack for the SmartNIC. This stack performs Layer2/Layer3 protocol processing, such as packet encapsulating/decapsulation, fragmentation, checksum verification, etc. When building a packet, it uses the DMA scatter-gather technique to combine the header and payload if they are not colocated. This helps improve the bandwidth utilization, as shown in our characterization (Section 4.3.5).

4.4.2 iPipe Actor Scheduler

iPipe schedules the actor execution among SmartNIC/host cores. The scheduler allocates actor execution tasks to computing cores and specifies a custom scheduling discipline for different tasks. In designing the scheduler, we not only want to maximize the computing resource utilization on the SmartNIC but also ensure that the computing efficiency does not come at the cost of compromising the NIC's primary task of conveying traffic. Recall that all traffic is conveyed through SmartNIC cores, so executing actor handlers could adversely impact the latency and throughput of other traffic.

Problem formulation and background. The runtime system executes on both the host and the SmartNIC, determines on which side an actor executes, and schedules the invocation of actor handlers. There are two critical decisions in the design of the scheduler: (a) whether the scheduling system is modeled as a centralized, single queue model or as a decentralized, multi-queue model, and (b) the scheduling discipline used for determining the order of task execution. We consider each of these issues below.

It is well-understood that the decentralized multi-queue model can be implemented without synchronization but would suffer from temporary load imbalances, thus leading to worse tail latencies. Fortunately, hardware traffic managers on SmartNICs provide support for a shared queue abstraction with low synchronization overhead (see Section 4.3). We therefore resort to using a centralized queue model on the SmartNIC and a decentralized multi-queue model on the host side,

	API	Explanation
Actor	actor_create (*)	create an actor
	actor_register (*)	register an actor into the runtime
	actor_init (*)	initialize an actor private state
	actor_delete (*)	delete the actor from the runtime
	actor_migrate (*)	migrate an actor to host
DMO	dmo_malloc	allocate a dmo obj.
	dmo_free	free a dmo obj.
	dmo_mmset	set space in a dmo with value.
	dmo_mmcpy	copy data from a dmo to a dmo.
	dmo_mmmove	move data from a dmo to a dmo.
	dmo_migrate	migrate a dmo to the other side.
MSG	msg_init	initialize a remoge message I/O ring
	msg_read (*)	read new messages form the ring
	msg_write	write messages into the ring
Nstack	nstack_new_wqe	create a new WQE
	nstack_hdr_cap	build the packet header
	nstack_send	send a packet to the TX
	nstack_get_wqe	get the WQE based on the packet
	nstack_recv(*)	receive a packet from the RX

Table 4.3: iPipe major APIs. There are four categories: actor management (Actor), distributed memory object (DMO), message passing (MSG), and networking stack (Nstack). The Nstack has additional methods for packet manipulation. APIs with * are mainly used by the runtime as opposed to actor code.

along with NIC-side support for flow steering.

We next consider the question of what scheduling discipline to use and how that impacts the average and tail response times for scheduled operations (i.e., both actor handlers and message forwarding operations). Note that the response time or sojourn time is the total time spent including queueing delay and request processing time. If our goal is to optimize for mean response time,

then Shortest Remaining Processing Time (SRPT) and its non-preemptive counterpart, Shortest Job First (SJF), are considered optimal regardless of the task size and interarrival time distributions [232]. However, in our setting, we also care about the tail response time; even if the application can tolerate it, a high response latency in our setting means that the NIC isn't performing its basic duty of forwarding traffic in a timely manner. If we were to consider minimizing the tail response time, then First Come First Served (FCFS) is considered optimal when task size distribution has low variance [242] but has been shown to perform poorly when the task size distribution has high dispersion or is heavy-tailed [106]. In contrast, Processor Sharing is considered optimal for high variance distributions [256].

In addition to the issues described above, the overall setting of our problem is unique. Our runtime manages the scheduling on both the SmartNIC and the host with the flexibility to move actors between the two computing zones. Crucially, we want to increase the occupancy on the SmartNIC, without overloading it or causing tail latency spikes, and can shed load to the host if necessary. Furthermore, given that the offloaded tasks will likely have different cost distributions (as we saw in our characterization experiments), we desire a solution that is suitable for a broad class of tasks.

Scheduling algorithm. We propose a hybrid scheduler that: (1) combines FCFS and DRR (deficit round robin) service disciplines; (2) migrates actors between SmartNIC and host processors when necessary. Essentially, the scheduler takes advantage of FCFS for tasks that have low dispersion in their service times and delegates tasks with a greater variance in service time to a DRR scheduler. The scheduler uses DRR for high variance tasks as DRR is an efficient approximation of Processor Sharing in a non-preemptible setting [238]. Further, the scheduler places as much computation as possible on the SmartNIC and migrates actors when the NIC cannot promptly handle incoming bandwidth. To assist in these transitions, the scheduler collects statistics regarding the average and the tail execution latencies, actor-specific execution latencies, and queueing delays. We mainly describe the NIC-side scheduler below and then briefly describe how the host-side scheduler differs from it.

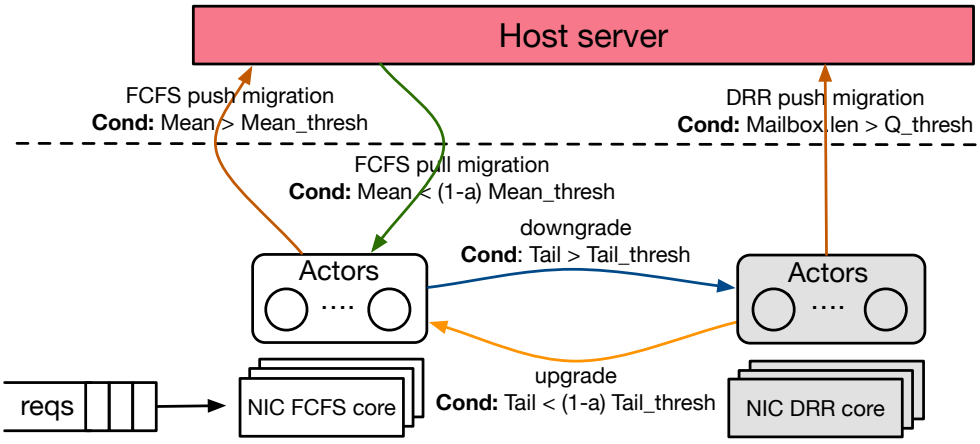


Figure 4.7: An overview of iPipe scheduler on the SmartNIC. **Cond** is the operation triggered condition.

The scheduler works as follows. Initially, all scheduling cores start in FCFS mode, where they fetch packet requests from the shared incoming queue, dispatch requests to the target actor based on their flow information, and perform run-to-completion execution (see lines 5-6, 11-12 of ALG 1). When the measured tail latency of operations in the FCFS core is greater than *tail_thresh*, the scheduler downgrades the actor with the highest dispersion (a measure that we describe later) by pushing the actor into a DRR runnable queue and spawns a DRR scheduling core if necessary (lines 13-16 ALG 1). All DRR cores share one runnable queue to take advantage of the execution parallelism.

We next consider the DRR cores (see ALG 2). These cores scan all actors in the DRR runnable queue in a round-robin way. When the deficit counter of an actor is larger than its estimated latency, the core pops a request from the actor's mailbox and conducts its execution. The DRR quantum value for an actor, which is added to the counter in each round, is the maximum tolerated forwarding latency for the actor's average request size (obtained from the measurements in Section 4.3). When the measured tail latency of operations performed by FCFS is less than $(1-\alpha)$*tail_thresh* (where α is a hysteresis factor), the actor with the lowest dispersion in the DRR runnable queue is pushed back to the FCFS group (lines 10-12 of ALG 2).

Finally, when the scheduler detects that the mean request latency for FCFS jobs is larger than *mean_thresh*, it indicates a queue build-up at the SmartNIC and one should migrate the actor that contributes the most to the overloading to the host processor (lines 17-23 ALG 1). Similarly, when the mean request latency of the FCFS core group is lower than $(1-\alpha)$*mean_thresh* and if there is sufficient CPU headroom in the FCFS cores, the scheduler issues a pull request to the host server to migrate the actor that will incur the least load back to the SmartNIC. We use a dedicated core of the FCFS group (core 0) for the migration tasks.

Bookkeeping execution statistics. Our runtime monitors the following statistics to assist the scheduler: (1) Request execution latency distribution of all actors: We measure μ, the execution latency of each request (including its queueing delay) using microarchitectural time stamp counters. To efficiently approximate the tail of the distribution, we also track the standard deviation of the request latency σ and use $\mu+3\sigma$ as a tail latency measure. Note that this is close to the P99 measure for normal distributions. All of these estimates are updated using exponentially weighted moving averages (EWMA). (2) Per-actor execution cost and dispersion statistics. For each actor i, we track its request latency μ_i, the standard deviation of the latency σ_i, request sizes, and the request frequency. We use $\mu_i+3\sigma_i$ as a measure of the dispersion of the actor's request latency. Again, we use EWMA to update these measures. (3) Per-core/per-group CPU utilization. We monitor the per-core CPU usage for the recent past and also use its EWMA to estimate its current utilization. The CPU group utilization (for FCFS or DRR) is the average among all cores' CPU usage. Finally, we use measurements from our characterization study to set the thresholds *mean_thresh* and *tail_thresh*. We consider the MTU packet size at which the SmartNIC is able to sustain line rate and use the average and P99 tail latencies experienced by traffic forwarded through the SmartNIC as the corresponding thresholds (Section 4.3). This means that we provide the same level of service with offloaded computations as when we have full line rate processing of moderately sized packets.

FCFS and DRR core auto-scaling. All cores start in FCFS mode. When an actor is pushed into the DRR runnable queue, the scheduler spawns a core for DRR execution. When all cores in the

Algorithm 1 iPipe FCFS scheduler algorithm

```
1: wqe : contains packet data and metadata
2: DRR_queue : the runnable queue for the DRR scheduler
3: procedure FCFS_SCHED                                   ▷ on each FCFS core
4:     while true do
5:         wqe = iPipe_nstack_recv()
6:         actor = iPipe_dispatcher(wqe)
7:         if actor.is_DRR then
8:             actor.mailbox_push(wqe)
9:             Continue
10:        end if
11:        actor.actor_exe(wqe)
12:        actor.bookeeping()                             ▷ Update execution statistics
13:        if T_tail > Tail_thresh then                   ▷ Downgrade
14:            actor.is_DRR = 1
15:            DRR_queue.push(actor)
16:        end if
17:        if core_id is 0 then                           ▷ Management core
18:            if T_mean > Mean_thresh then               ▷ Migration
19:                iPipe_actor_migrate(actor_chosen)
20:            end if
21:            if T_mean < (1-α)Mean_thresh then          ▷ Migration
22:                iPipe_actor_pull()
23:            end if
24:        end if
25:    end while
26: end procedure
```

Algorithm 2 iPipe DRR scheduler algorithm

```
1:  procedure DRR_SCHED                                          ▷ On each DRR core
2:      while true do
3:          for actor in all DRR_queue do
4:              if actor.mailbox is not empty then
5:                  actor.update_deficit_val()
6:                  if actor.deficit > actor.exe_lat then
7:                      wqe = actor.mailbox_pop()
8:                      actor.actor_exe(wqe)
9:                      actor.bookeeping()
10:                     if T_tail < (1-α)Tail_thresh then            ▷ Upgrade
11:                         actor.is_DRR = 0
12:                         DRR_queue.remove(actor)
13:                     end if
14:                 end if
15:                 if actor.mailbox is empty then
16:                     actor.deficit = 0
17:                 end if
18:                 if actor.mailbox.len > Q_thresh then             ▷ Migration
19:                     iPipe_actor_migrate(actor)
20:                 end if
21:             end if
22:         end for
23:     end while
24: end procedure
```

DRR group is nearly fully used ($CPU_{DRR} \geq 95\%$ and the CPU usage of the FCFS group is less than $\frac{100\times(FCFS_{Core\#}-1)}{FCFS_{Core\#}}\%$, FCFS is able to spare one core for DRR, and the scheduler will migrate a core to DRR. A similar condition is used for moving a core back to the FCFS group.

SmartNIC push/pull migration. We only allow the SmartNIC to initiate the migration operation since it is much more sensitive than the host processor in case of overloading. As described before, when there is persistent queueing and the mean response time is above a threshold, the scheduler will move an actor to the host side. We pick the actor with the highest load (i.e., average execution latency scaled by frequency of invocation) for migration. We perform a cold migration mechanism in four phases as follows:

- Phase 1: The actor transitions into the *Prepare* state and removes itself from the runtime dispatcher. An actor in the DRR group is also removed from the DRR runnable queue. The actor stops receiving incoming requests and buffers them in the iPipe runtime.
- Phase 2: The actor finishes the execution of its current tasks and changes to the *Ready* state. Note that, for an actor in the DRR group, it finishes executing all the requests in its mailbox.
- Phase 3: The scheduler moves the distributed objects of an actor to the host runtime, starts the host actor, and marks the NIC actor state as *Gone*.
- Phase 4: The scheduler forwards the buffered requests from the NIC to the host and rewrites their destination addresses. We will label the NIC actor as *Clean*.

When migrating an actor to the host, as shown in Figure 4.8, our runtime (1) collects all objects that belong to the actor; (2) sends the object data to the host side using messages and DMA primitives; (3) creates new objects on the host side and then inserts entries into the host-side object table; (4) deletes related entries from the NIC-side object table upon deleting the actor. The host-side DMO works similarly, except that it uses the glibc memory allocator.

SmartNICs with no hardware traffic managers. We now consider SmartNICs that do not provide a shared queue abstraction to the NIC processor (especially off-path ones like BlueField and Stingray). There are two possible ways to overcome this limitation. One is to apply a dedicated

kernel-bypass component (such as the IOKernel module in Shenango [214]) that processes all incoming traffic and exposes FCFS cores a single queue abstraction. This module will run on one or several NIC cores exclusively, depending on the traffic load. Another way is to add an intermediate shuffle layer across FCFS cores. Essentially, this shuffle queue is a single-producer, multiple-consumer one. This approach could cause load imbalances due to flow steering. Similar to ZygOS [221], one should also allow a FCFS core to steal other cores' requests when it becomes idle with no pending requests in its local queue. Note that both approaches bring in performance overheads, so future on-path/off-path SmartNICs could benefit from adding a hardware traffic manager to simplify NIC-side computation scheduling.

Summary: The scheduler manages the execution of actor requests on both the SmartNIC and the host. We use a hybrid scheme that combines FCFS and DRR on both sides. With the scheme outline above, light-weight tasks with low dispersion are executed on the SmartNIC's FCFS cores, light-weight tasks with high dispersion are executed on the SmartNIC's DRR cores, and heavy-weight tasks are migrated to the host. These decisions are performed dynamically to meet the desired average and tail response times.

4.4.3 Distributed Memory Objects and Others

iPipe provides a distributed memory object (DMO) abstraction to enable flexible actor migration. Actors allocate and de-allocate DMOs as needed, and a DMO is associated with the actor that allocated it; there is no sharing of DMOs across actors. iPipe maintains an object table (Figure 4.8-a) on both sides and utilizes the local memory manager to allocate/deallocate copies. At any given time, a DMO has only one copy, either on the host or on the NIC. We also do not allow an actor to perform reads/writes on objects across the PCIe because remote memory accesses are 10x slower than local ones (as shown in Section 4.3). Instead, iPipe would automatically move DMOs along with the actor and all DMO read/write/copy/move operations are performed locally.

When using DMOs to design a data structure, one has to use the object ID for indexing instead of pointers. This provides a level of indirection so that we can change the actual location of the

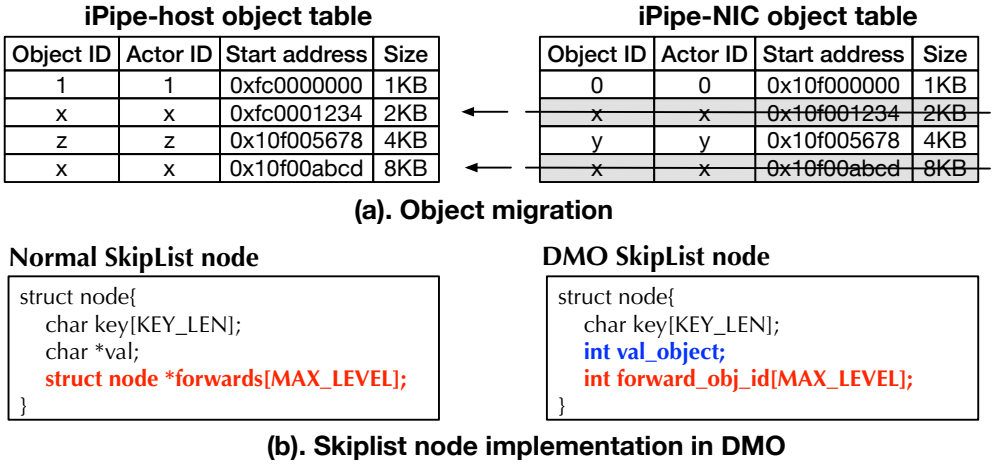


Figure 4.8: iPipe distributed memory objects.

object (say during migration to/from the host) without impacting an actor's local state regarding DMOs. As an example, in our replicated key-value store application (discussed later), we built the skiplist based memtable via DMO. As shown in Figure 4.8-b, a traditional skiplist node includes a key string, a value string, and a set of forwarding pointers. With DMO, the key field is the same. Value and forwarding pointers are replaced by object IDs. When traversing, one will use the object ID to get the start address of the object, cast the type, and then read/write its contents.

Scratchpad. Our characterization experiments (Section 4.3) have shown that the scratchpad memory provide the fastest performance but has very limited resources. Instead of exposing this to applications and managing them, we decide to keep this memory resource internally and use it for storing the iPipe bookkeeping information.

4.4.4 Security Isolation

iPipe allows multiple actors to concurrently execute on a SmartNIC. There are two attacks that iPipe should protect against: (1) actor state corruption, where a malicious actor manipulates other actors' states; (2) denial-of-service, where an actor hangs on the SmartNIC core and violates the service availability of other actors. We primarily describe how to protect against these attacks on

the Cavium LiquidIOII, as one can apply similar techniques to other SmartNICs.

Actor state corruption Since iPipe provides the distributed memory object abstraction to use the onboard memory DRAM, we rely on the processor paging mechanism to secure the object accesses. LiquidIOII CN2350/CN2360 SmartNICs employ a MIPS processor (which has a software managed TLB) and a light-weight firmware for memory management. In this case, we use a physical partition approach. During the initialization phase, iPipe creates large equal size chunks of memory regions for each registered actor (where its size is $\frac{Mem_{total}-Mem_{fimrware}-Mem_{runtime}}{Num_{actor}}$). iPipe runtime maintains the mapping between actor ID, its base address, and size. During execution, an actor can only allocate/reclaim/access its objects within its region. Invalid reads/writes from an actor causes a TLB miss and will trap into the iPipe runtime. If the address is not in the region, access is not granted.

Denial-of-service. A malicious actor might occupy a NIC core forever (e.g., executing an infinite loop), violating actor availability. We apply a timeout mechanism to address this issue. LiquidIOII CN2350/CN2360 SmartNICs include a hardware timer with 16 timer rings. We give each core a dedicated timer. When an actor is executed, it clears out the timer and initializes the time interval. The timeout unit will traverse all timer rings and notify the NIC core when there is a timeout event. If a NIC receives the timeout notification, iPipe deregisters the actor, removes it from the dispatch table/runnable queue (if it is in the DRR group), and frees the actor resource.

4.4.5 Host/NIC Communication

We use a message-passing mechanism to communicate between host and the SmartNIC. iPipe creates a set of I/O channels, and each one includes two circular buffers for sending and receiving. A buffer is unidirectional and stored in the host memory. NIC cores write into the receive buffer, and a host core polls it to detect new messages. The send buffer works in reverse. We use a lazy-update mechanism to synchronize the header pointer between the host and the NIC, wherein the host notifies the SmartNIC when it has processed half of the buffer via a dedicated message. We use batched non-blocking DMA reads/writes for the implementation. In order to avoid the case of a

DMA engine not writing the message contents in a monotonic sequence (unlike RDMA NICs), we add a 4B checksum into the message header to verify the integrity of the whole message. Table 4.3 shows the messaging API list.

4.5 Applications Built with iPipe

We implement three distributed applications using iPipe: a replicated key-value store, a distributed transaction system, and a real-time analytics engine.

4.5.1 Replicated Key-value Store

Replicated key-value store (RKV) is a critical data center service, comprising of two key system components: a *consensus protocol*, and a *key-value data store*. We use the traditional Multi-Paxos algorithm [183] to achieve consensus among multiple replicas. Each replica maintains an ordered log for every Paxos instance. There is a distinguished leader that receives client requests and performs consensus coordination using Paxos prepare/accept/learning messages. In the common case, consensus for a log instance can be achieved with a single round of accept messages, and the consensus value can be disseminated using an additional round (learning phase). Each node of a replicated state machine can then execute the sequence of commands in the ordered log to implement the desired replicated service. When the leader fails, replicas will run a two-phase Paxos leader election (which determines the next leader), choose the next available log instance, and learn accepted value from other replicas if its own log has gaps. Typically, the Multi-Paxos protocol can be expressed as a sequence of messages that are generated and processed based on the state of the RSM log.

For the key-value store, we take the log-structure merge tree (LSM) that is widely used for many KV systems (such as Google's Bigtable [124], LevelDB [12], Cassandra [8]). An LSM tree accumulates recent updates in memory and serves reads of recently updates values from in-memory data structure, flushes the updates to the disk sequentially in batches, and merges long-lived on-disk persistent data to reduce disk seek costs. There are two key system components: *memtable*, a

sorted data structure (i.e., SkipList) and *SSTables*, collections of data items sorted by their keys and organized into a series of levels. Each level has a size limit on its SSTables, and this limit grows at an exponential rate with the level number. Low-level SSTables are merged into high-level ones via minor/major compact operations. Deletions are a special case of insertions wherein a deletion marker is added. Data retrieval might require multiple lookups on the Memtable and the SSTables (starting with level 0 and moving to high levels) until a matching key is found.

We implement the application with four kinds of actors: (1) consensus actor, receives application requests and triggers the Multi-Paxos logic; (2) LSM memtable actor, accumulates incoming writes/deletes and serves fast reads; (3) LSM SSTable read actor, serves SSTable read requests when requests are missing in the Memtable; (4) LSM compaction actor, performs minor/major compactions. The consensus actor sends a message to the LSM memtable one during the commit phase. When requests miss in the Memtable actor, they are forwarded to the SSTable read actor. Upon a minor compaction, the Memtable actor migrates its Memtable object to the host and issues a message to the compaction actor. Our system has multiple shards, based on the NIC DRAM capacity. The two SSTable related actors are stationary on the host because they have to interact with persistent storage.

4.5.2 Distributed Transactions

We build a distributed transactions system that uses optimistic concurrency control and two-phase commit for distributed atomic commit, following the design used by other systems [259]. Note that we choose to not add a replication layer as we try to eliminate the application function overlap with our replicated key-value store. The application includes a coordinator and participants that run a transaction protocol. Given a read set (R) and a write set (W), the protocol works as follows: Phase 1 (read and lock): the coordinator reads values for the keys in R and locks the keys in W. If any key in R or W is already locked, the coordinator aborts the transaction and replies with the failure status; Phase 2 (validation): after locking the write set, the coordinator checks the version of keys in its read set by issuing a second read. If any key is locked or its version has changed

after the first phase, the coordinator aborts the transaction; Phase 3 (log): the coordinator logs the key/value/version information into its coordinator log and then sends a reply to the client with the result; Phase 4 (commit): the coordinator sends commit messages to nodes that store the *W* set. After receiving this message, the participant will update the key/value/version, as well as unlock the key.

In iPipe, we implement the coordinator and participant as actors running on the NIC. The key storage abstractions required to implement the protocol are the coordinator log [132] and the data store, which we realize using a traditional extensible hashtable [95]. Both of these are realized using distributed shared objects. We also cache responses from outstanding transactions. There is also a logging actor pinned to the host since it requires persistent storage access. When the coordinator log reaches a storage limit, the coordinator migrates its log object to the host side and sends a checkpointing message to the logging actor.

4.5.3 Real-time Analytics

Data processing pipelines use a real-time analytics engine to gain instantaneous insights into vast and frequently changing datasets. We acquired the implementation of FlexStorm [174] and extended its functionality. All data tuples are passed through three workers: *filter*, *counter*, and *ranker*. The filter applies a pattern matching module to discard uninteresting data tuples. The counter uses a sliding window and periodically emits a tuple to the ranker. Ranking workers sort incoming tuples based on count and then emit the *top-n* data to an aggregated ranker. Each worker uses a topology mapping table to determine the next worker to which the result should be forwarded.

To implement this application in iPipe, we build three workers as actors. Filter actor is a stateless one. Counter uses the software-managed cache for statistics. Ranker is implemented using a distributed shared object, and we consolidate all top-n data tuples into one object. Among them, ranker performs quicksort to order tuples, which could impact the NIC's ability to receive new data tuples when the network load is high. In such cases, iPipe will migrate the actor to the

host side.

4.6 Evaluation

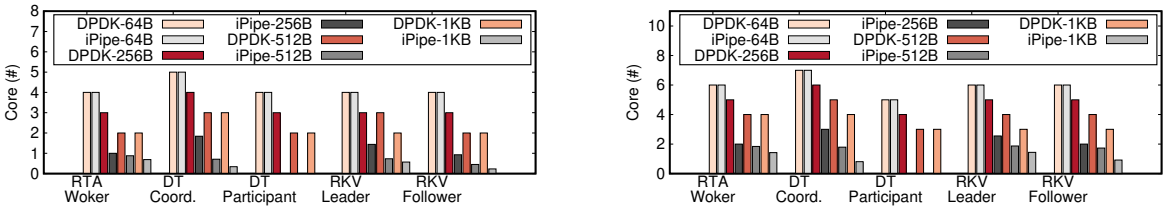
Our evaluations aim to answer the following questions:

- What are host CPU core savings when offloading computations using iPipe? (§4.6.2)
- What are the latency savings with iPipe? (§4.6.3)
- How effective is the iPipe actor scheduler? (§4.6.4)
- What is the iPipe migration performance? ($4.6.5)
- How effective when using a SmartNIC as a smart dispatcher? ($4.6.6)
- When compared with another SmartNIC programming system (i.e., Floem [218]), what are the design trade-offs in terms of performance and programmability? (§4.6.7)
- Can we use iPipe to build other applications (e.g., network functions)? How does it perform? (§4.6.8)

4.6.1 Experimental Methodology

We use the same testbed as our characterization experiments Section 4.3.1. For evaluating our application case studies, we mainly use the LiquidIOII CN2350/CN2360 (10/25 GbE) as we had a sufficient number of cards to build a small distributed testbed. We built iPipe into the LiquidIOII firmware using the Cavium Development Kit [80]. On the host side, we use pthreads for iPipe execution and allocate 1GB pinned hugepages for the message ring. Each runtime thread periodically polls requests from the channel and performs actor execution. It transits to idle C-states when there is no more work. The iPipe runtime spreads across the NIC firmware and host system with 10683 LOCs and 4497 LOCs, respectively. To show the effectiveness of the actor scheduler, we also present results for the Stingray card.

Programmers use the C langugage to build applications (which are compiled with SmartNIC/host GNU tool chains). Our three workloads, real-time analytics (RTA), distributed transactions (DT), replicated key-value store (RKV), built with iPipe have 1583 LOCs, 2225 LOCs,



(a) 10GbE w/ LiquidIOII CN2350.

(b) 25GbE w/ LiquidIOII CN2360.

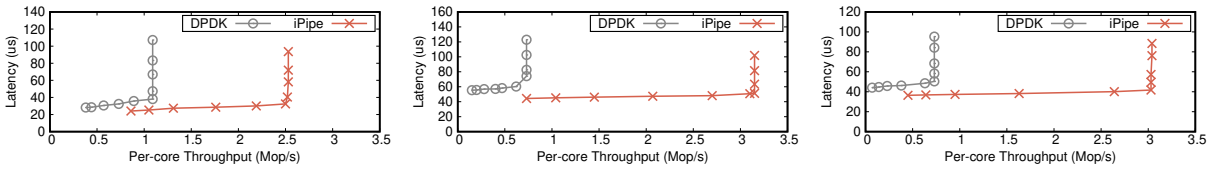
Figure 4.9: Host used CPU cores compared between DPDK and iPipe on three different applications varying packet sizes for a 10GbE/25GbE network.

and 2133 LOCs, respectively, and we compare them with similar implementations that use DPDK. Our workload generator is implemented using DPDK and invokes operations in a closed-loop manner. For RTA, we generate the request based on the Twitter tweets [187]. The number of data tuples in each request vary based on the packet size. For DT, each request is a multi-key read-write transaction including 2 reads and 1 writes (used in previous work [170]). For RKV, we generate the <key,value> pair in each packet, with the following characteristics: 16B key, 95% read/5%write, zipf distribution (skewness=0.99), and 1 million keys (used in previous work[213, 193]). For both DT and RKV, the value size increases with the packet size.

We deploy each of the applications on three servers, equipped with SmartNICs in the case of iPipe and normal Intel NICs in the case of DPDK. The RTA application runs a RTA worker on each server, the DT application runs coordinator logic on one server and participant logic on two servers, and the RKV application involves a leader node and two follower nodes.

4.6.2 Host Core Savings

We find that we can achieve significant host core savings by offloading computations to the SmartNIC. Figure 4.9 reports the average host server CPU usage of three applications when achieving the maximum throughput for different packet sizes under 10/25GbE networks. First, when packet size is small (i.e., 64B), iPipe will use all NIC cores for packet forwarding, leaving no room for



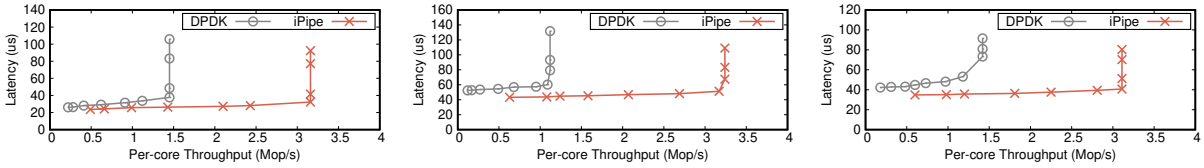
(a) RTA. (b) DT. (c) RKV.

Figure 4.10: Latency versus per-core throughput for three applications under 10GbE, compared between DPDK and iPipe cases. Packet size is 512B.

actor execution. In this case, one will not save host CPU cores. Second, host CPU usage reduction is related to both packet size and bandwidth. Higher link bandwidth and smaller packet size bring in more packet level parallelism. When the SmartNIC is able to absorb enough requests for execution, one can reduce host CPU loads significantly. For example, applications built on iPipe save 3.1, 2.6, and 2.5 host cores for 256/512/1KB cases, on average across three applications using the 25GbE CN2360 cards. Such savings are marginally reduced with the 10GbE CN2350 ones (i.e, 2.2, 1.8, 1.8 core savings). Among these three applications, DT participant saves the most since it is able to run all its actors on the SmartNIC, followed by the DT coordinator, RTA worker, RKV follower, and RKV leader.

4.6.3 Latency versus Throughput

We next examine the latency reduction and per-core throughput increase provided by iPipe and find that SmartNIC offloading provides considerable benefits. Figures 4.10 and 4.11 report the results comparing DPDK and iPipe versions of the applications, when we configure the system to achieve the highest possible throughput with the minimal number of cores. When calculating the per-core throughput of three applications, we use the CPU usage of RTA worker, DT coordinator, and RKV leader to account for fraction core usage. First, under 10GbE SmarNICs, applications (RTA, DT, and RKV) built with iPipe outperform the DPDK ones by 2.3X, 4.3X, and 4.2X, respectively, as



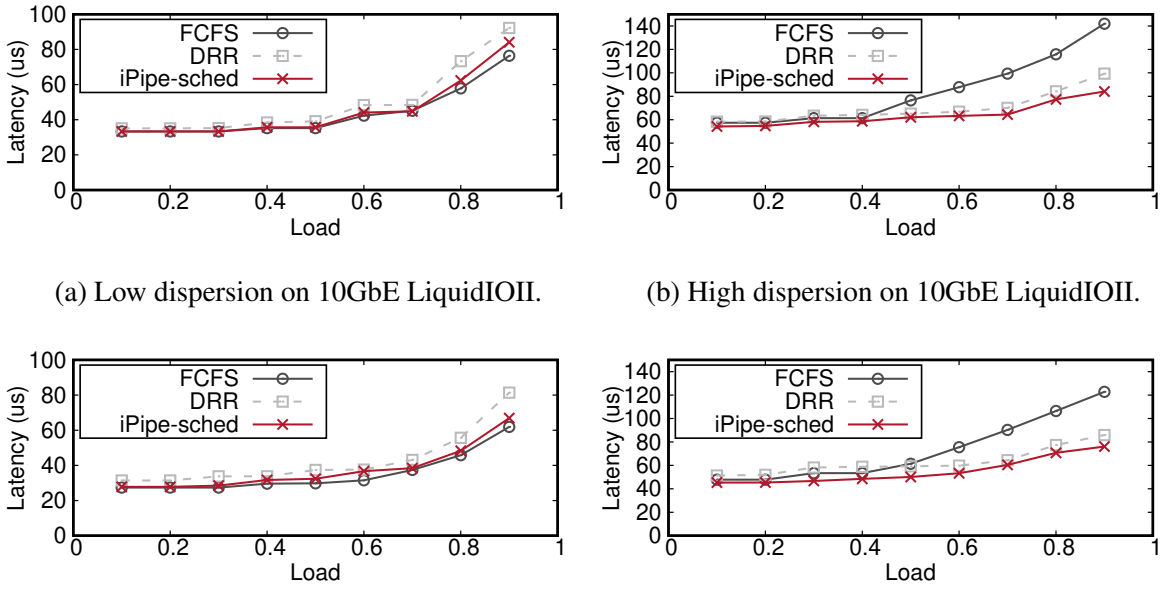
(a) RTA. (b) DT. (c) RKV.

Figure 4.11: Latency versus per-core throughput for three applications under 25GbE, compared between DPDK and iPipe cases. Packet size is 512B.

iPipe allows applications to use a fewer number of host CPU cores. The benefits diminish a little under the 25GbE setup (with 2.2X, 2.9X, and 2.2X improvements) since actors running on the host CPU receive more requests and require more CPU power.

Second, at low to medium request rates, NIC-side offloading reduces request execution latency by 5.7μs, 23.0μs, 8.7μs for 10GbE and 5.4μs, 28.0μs, 12.5μs for 25GbE, respectively. Even though the SmartNIC has only a wimpy processor, the iPipe scheduler keeps the light-weight fast path tasks on the NIC and moves the heavy-weight slow ones to the host. As a result, PCIe transaction savings, fast networking primitives, and hardware-accelerated buffer management can help reduce the fast path execution latency. DT benefits the most as both the coordinator and the participants mainly run on the SmartNIC processor and the host CPU is only involved for the logging activity.

P99 tail latency. We measured the tail latency (P99) when achieving 90% of the maximum throughput for the two link speeds. For the three applications, iPipe reduces tail latency by 7.3μs, 11.6μs, 7.5μs for 10GbE and by 3.4μs, 10.9μs, 12.8μs for 25GbE. This is not only due to fast packet processing (discussed above), but also because iPipe's NIC-side runtime guarantees that there is no significant queue build up.



(a) Low dispersion on 10GbE LiquidIOII.

(b) High dispersion on 10GbE LiquidIOII.

(c) Low dispersion on 25GbE Stingray.

(d) High dispersion on 25GbE Stingray.

Figure 4.12: P99 tail latency varies with the networking load for low and high dispersion request distribution for 10GbE LiquidIOII CN2350 and 25GbE Stingray cards.

4.6.4 iPipe Actor Scheduler

We evaluate the effectiveness of iPipe's scheduler, comparing it with standalone FCFS and DRR schedulers under two different request cost distributions: one is exponential with low dispersion; the other one is bimodal-2 with high dispersion. We choose two SmartNICs (i.e., 10GbE LiquidIOII CN2350 and 25GbE Stingray) representing the cases where the actor scheduler runs as firmware hardware threads and OS pthreads, respectively. The workload generator is built using packet traces obtained from our three real world applications and issues requests assuming a Poisson process. We measure the latency from the client side. The mean service times of the exponential distribution on the two SmartNICs (i.e., LiquidIOII and Stingray) is 32μs and 27μs, while b1/b2 of the bimodal-2 distribution is 35μs/60μs and 25μs/55μs.

Figure 4.12 shows the P99 tail latency as we increase the network load for four different cases.

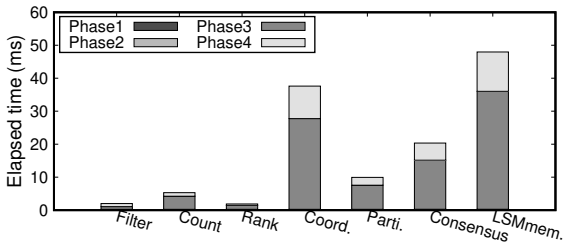


Figure 4.13: Migration elapsed time breakdown of 8 actors from three applications evaluated with 10GbE CN2350 cards.

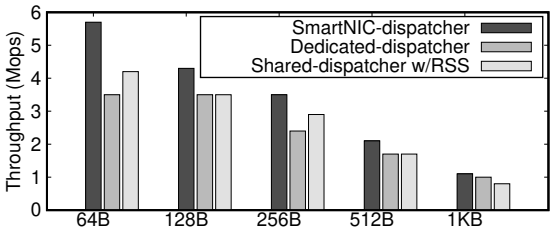


Figure 4.14: Throughput comparison varying packet size (on 10GbE), compared among SmartNIC, dedicated, and shared dispatchers.

For the low dispersion one, iPipe's scheduler behaves similar to FCFS, but outperforms DRR. Under 0.9 networking load, iPipe can reduce 9.6% and 21.7% of DRR's tail latency for LiquidIOII and Stingray, respectively. For the high dispersion one, iPipe scheduler is able to tolerate the request execution variation and serve short tasks in time, outperforming the other two. For example, when the networking load is 0.9, iPipe can reduce 68.7%(61.4%) and 10.9% (12.9%) of the tail latency for FCFS and DRR cases on LiquidIOII (Stingray).

4.6.5 iPipe Migration

We estimate the migration cost (SmartNIC-pushed) by breaking down the time elapsed of four phases 4.6.4. We choose 8 actors from three applications. Our experiments are conducted under 90% networking load and we force the actor migration after the warm up (5s). Figure 4.13 presents our results. First, phase 3 dominates the migration cost (i.e., 67.8% on average of 8 actors) since it requires to move the distributed objects to the host side. For example, the LSM memtable actor has around 32MB object and consumes 35.8ms. Phase 4 ranks the second (i.e., 27.2%) as it pushes buffered requests to the host. Also, it varies based on the networking load. Phase 1 and Phase 2 are two light-weight parts because they only introduce the iPipe runtime locking/unlocking and state manipulation overheads.

4.6.6 SmartNIC as a Dispatcher

SmartNICs allows application-specific flow steering into multiple transmit/receive NIC queues. To demonstrate this, we use a sharded key-value store (which uses the Memtable implementation of the RKV) along with our workload generator and compare with three different dispatcher designs. Specifically, we configure the system with 4 host CPU cores (where each core runs one shard), 4 NIC queues, and use the 10GbE LiquidIO CN2350. We compare three scenarios: (1) *SmartNIC dispatcher*, where NIC cores obtain the shard information from the request application header and push request into the appropriate queue. (2) *dedicated dispatcher*, where we use one dedicated host core to poll incoming requests from all NIC queues, and push them to the other three cores appropriately. We divide the keyspace into three shards in this case. (3) *shared dispatcher*, where we run the dispatcher on all 4 cores and co-locate it with the key-value store. In this case, we also enable RSS. Note that the dedicated and shared dispatchers have a lockless FIFO command queue to buffer requests from other cores.

Figure 4.14 reports the measured throughput for different packet sizes. On average, the SmartNIC dispatcher outperforms the dedicated and shared dispatchers by 32.9% and 25.8%, respectively. This is because the SmartNIC steering eliminates the inter-core request transfer and command queue operation overheads.

4.6.7 Comparison with Floem

Floem [218] is a programming system aimed at easing the programming effort for SmartNIC offloading. It applies a data-flow language to express packet processing and proposes a couple of programming abstractions, such as logic queue, per-packet state, etc. iPipe also has similar designs (like message rings, packet metadata). However, compared with iPipe, the key difference is that the language runtime of Floem doesn't use the SmartNIC computing power in an efficient way. First, the offloaded elements (computation) on Floem is stationary, no matter what the incoming traffic is. However, we have shown that, when the packet size is small and networking load is high (Section 4.3), such Multi-core SoC SmartNICs have no room for application computation. In iP-

ipe, we will migrate the computation to the host side. Second, the common computation elements of Floem mainly comprise of simple tasks (like hashing, steering, or bypassing). Complex ones (even though they can be expressed) are performed on the host side. In iPipe, we have shown that complex operations can also be offloaded, and our runtime will dynamically schedule them in the right place.

We take the real-time analytics (RTA) workload, and compare its Floem and iPipe implementations. With the same experimental setup, Floem-RTA achieves at most 1.6Gbps/core (in the best case), while iPipe-RTA can achieve 2.9Gbps. As described above, this is because iPipe can offload the entire actor computation while Floem utilizes a NIC-side bypass queue to mitigate the multiplexing overhead. For the small packet size case (i.e., 64B), iPipe-RTA delivers 0.6Gbps/core, outperforming Floem by 88.3%, since iPipe moves all the actors to the host side and uses all NIC cores for packet forwarding, while Floem still uses the NIC-side for offloading. In sum, we believe iPipe can be an efficient backend for Floem.

4.6.8 Network Functions on iPipe

The focus of iPipe is to accelerate distributed applications with significant complexity in program logic and maintained state. For network functions with easily expressed states (or even stateless ones) that have sufficient parallelism, FPGA-based SmartNICs are an appropriate fit. We now consider how well iPipe running on multi-core SmartNICs can approximate FPGA-based SmartNICs for such workloads. We built two network functions with iPipe (i.e., Firewall and IPSec gateway) and evaluated them on the 10/25GbE LiquidIOII cards. For the firewall, we use a software based TCAM implementation matching wildcard rules. Under 8K rules and 1KB packet size, the average packet processing latency ranges from 3.65μs to 19.41μs as we increase the networking load. However, a FPGA based solution achieves 1.23$\sim$1.6μs. For the IPsec gateway, we take advantage of the crypto engines to accelerate packet processing. For 1KB packets, iPipe achieves 8.6Gbps and 22.9Gbps bandwidth on the 10/25 GbE SmartNIC cards, respectively. Such results are comparable to the ClickNP ones (i.e., 37.8Gbps under 40GbE link speed). In other words, if one can use the

accelerators on a Multi-core SoC SmartNIC, one can achieve comparable performance as FPGA based ones for network functions.

4.7 Related Work

4.7.1 SmartNIC Acceleration

In addition to Floem [218], ClickNP [189] is another framework using FPGA-based SmartNICs for network functions. It uses the Click [179] data-flow programming model and statically allocates a regular data-flow graph model during configuration, whereas iPipe is able to move computations based on runtime workload (e.g., request execution latency, incoming traffic). There are a few other studies that use SmartNICs for application acceleration. KV-Direct [188] is an in-memory key-value store system, which runs key-value operations on the FPGA and uses the host memory as a storage pool. HotCocoa [108] proposes a set of hardware abstractions to offload the entire congestion control algorithm to a SmartNIC.

4.7.2 In-network Computations

Recent RMT switches [117] and SmartNICs enable programmability along the packet data plane. Researchers have proposed the use of in-network computation, where one can offload compute operations from end-hosts into these network devices in order to reduce data center traffic and improve application performance. For example, IncBricks [248] is an in-network caching fabric with some basic computing primitives. NetCache [167] is another in-network caching design, which uses a packet-processing pipeline on a Barefoot Tofino switch to detect, index, store, invalidate, and serve key-value items. DAIET [230] conducts data aggregation (for MapReduce and TensorFlow) along the network path using programmable switches.

4.7.3 RDMA-based Data center Applications

Recent years have seen growing use of RDMA in data center environments due to its low-latency, high-bandwidth, and low CPU utilization benefits. These applications include key-value store system [207, 138, 169], DSM system [210, 138], database and transactional system [125, 139, 254]. Generally, RDMA provides fast data access capabilities but limited opportunities to reduce the host CPU computing load. While one-sided RDMA operations allow applications to bypass remote server CPUs, they are hardly used in general distributed systems given the narrow set of remote memory access primitives associated with them. In contrast, *iPipe* provides a framework to offload simple but general computations onto SmartNICs. It does however borrow some techniques approaches from related RDMA projects (e.g., lazy updates for the send/receive rings in FaRM [138]).

4.8 Summary and Learned Lessons

This chapter presents a framework to offload distributed application onto a SmartNIC. We firstly conduct a detailed performance characterization across different commodity SmartNICs, and then build the iPipe system based on the experimental observations. We then develop three applications using iPipe and prototype them on these SmartNICs. Our evaluations show that by offloading computation on a SmartNIC, one can achieve considerable host CPU and latency savings.

We learned three lessons by building iPipe. First, the design of iPipe framework is driven by our experimental observations. When using a new kind of SmartNIC, one should perform an empirical analysis first. However, our characterization methodology (that views the SmartNIC from four angles: traffic control, computing units, onboard memory, and host communication) can be applied to most SmartNICs. Second, there are different structural design (on-path v.s. off-path) as well as interesting innovations in hardware acceleration, which would impact the way we offload. Finally, when moving computations to a SmartNIC, it has to be performed with care given small computing headrooms, sensitivity to traffic workloads, and the need to provide isolation.

5 E3: Running Microservices on the SmartNIC

In this chapter, we investigate the use of SmartNIC-accelerated servers to execute microservice-based applications in the data center. By offloading suitable microservices to the SmartNIC's low-power processor, we can improve server energy-efficiency without latency loss. However, as a heterogeneous computing substrate in the data path of the host, SmartNICs bring several challenges to a microservice platform: network traffic routing and load balancing, microservice placement on heterogeneous hardware, and contention on shared SmartNIC resources. We present the third PNF-enabled system – E3, a microservice execution platform for SmartNIC-accelerated servers. E3 follows the design philosophies of the Azure Service Fabric microservice platform and extends key system components to a SmartNIC to address the above-mentioned challenges. E3 employs three key techniques: ECMP-based load balancing via SmartNICs to the host, network topology-aware microservice placement, and a data-plane orchestrator that can detect SmartNIC overload. Our E3 prototype using Cavium LiquidIO SmartNICs shows that SmartNIC offload can achieve great energy-efficiency improvements.

The remainder of this chapter is organized as follows. Sections 5.1 and 5.2 provide the background and solution overview of E3. We then discuss the benefits and challenges of running microservices on SmartNICs. Sections 5.4 and 5.5 describe the design and implementation of the E3 microservice execution platform. Section 5.6 reports our evaluation results regarding latency, the effectiveness of proposed techniques, and energy/cost efficiency. We review related studies in Section 5.7 and conclude this chapter in Section 5.8.

5.1 Background

Energy-efficiency has become a major factor in data center design [252]. U.S. data centers consume an estimated 70 billion kilowatt-hours of energy per year (about 2% of total U.S. energy consumption) and as much as 57% of this energy is used by servers [6, 142]. Improving server energy-efficiency is thus imperative [122]. A recent option is the integration of low-power processors in server network interface cards (NICs). Table 2.1 lists some examples, such as Netronome Agilio-CX [76], Mellanox BlueField [67, 68], Broadcom Stingray [42], and Cavium LiquidIO [43], which rely on ARM/MIPS-based processors and on-board memory. These SmartNICs can process microsecond-scale client requests but consume much less energy than server CPUs. By sharing idle power and the chassis with host servers, SmartNICs also promise to be more energy and cost-efficient than other heterogeneous or low-power clusters. However, SmartNICs are not powerful enough to run large, monolithic cloud applications, preventing their offload.

Today, cloud applications are increasingly built as microservices, prompting us to revisit SmartNIC offload in the cloud. A microservice-based workload comprises loosely coupled processes, whose interaction is described via a data-flow graph. Microservices often have a small enough memory footprint for SmartNIC offload and their programming model efficiently supports transparent execution on heterogeneous platforms. Microservices are deployed via a microservice platform [168, 13, 15, 20] on shared data center infrastructure. These platforms abstract and allocate physical data center computing nodes, provide a reliable and available execution environment, and interact with deployed microservices through a set of common runtime APIs. Large-scale web services already use microservices on hundreds of thousands of servers [172, 168].

In this chapter, we'll describe how to run such Microservice-based applications on SmartNICs to improve system energy-efficiency. Essentially, we show how to integrate multiple SmartNICs per server into a microservice platform with the goal of achieving better energy efficiency at minimum latency cost.

5.2 Solution Overview

It is non-trivial to transparently integrate SmartNICs into microservice platforms. Unlike traditional heterogeneous clusters, SmartNICs are collocated with their host servers, raising a number of issues. First, SmartNICs and hosts share the same MAC address. We require an efficient mechanism to route and load-balance traffic to hosts and SmartNICs. Second, SmartNICs sit in the host's data path and microservices running on a SmartNIC can interfere with microservices on the host. Microservices need to be appropriately placed to balance network-to-compute bandwidth. Finally, microservices can contend on shared SmartNIC resources, causing overload. We need to efficiently detect and prevent such situations.

We build E3, a microservice execution platform for SmartNIC-accelerated servers that addresses these issues. E3 follows the design philosophies of the Azure Service Fabric microservice platform [168] and extends key system components to allow transparent offload of microservices to a SmartNIC. To balance network request traffic among SmartNICs and the host, E3 employs equal-cost multipath (ECMP) load balancing at the top-of-rack (ToR) switch and provides high-performance PCIe communication mechanisms between host and SmartNICs. To balance computation demands, we introduce *HCM*, a hierarchical, communication-aware microservice placement algorithm, combined with a data-plane orchestrator that can detect and eliminate SmartNIC overload via microservice migration. This allows E3 to optimize server energy efficiency with minimal impact on client request latency. We prototype E3 within a cluster of Xeon-based servers with up to 4 Cavium LiquidIO-based SmartNICs per server. We evaluate energy and cost-efficiency, as well as client-observed request latency and throughput for common microservices, such as a real-time analytics framework, an IoT hub, and various virtual network functions, across various homogeneous and heterogeneous cluster configurations . Our results show that offload of microservices to multiple SmartNICs per server with E3 improves cluster energy-efficiency up to 3$\times$ and cost efficiency up to 1.9$\times$ at up to 4% client-observed latency cost versus all other cluster configurations.

5.3 Benefits and Challenges of Microservices Offloading

Microservices simplify distributed application development and are a good match for low-power SmartNIC offload. Together, they are a promising avenue for improving server energy efficiency. We discuss this rationale, quantify the potential benefits, and outline the challenges of microservice offload to SmartNICs in this section.

5.3.1 Microservices

Microservices have become a critical component of today's data center infrastructure with a considerable and diverse workload footprint. Microsoft reports running microservices 24/7 on over 160K machines across the globe, including Azure SQL DB, Skype, Cortana, and IoT suite [168]. Google reports that Google Search, Ads, Gmail, video processing, flight search, and more, are deployed as microservices [172]. These microservices include large and small data and code footprints, long and short running times, billed by run-time and by remote procedure call (RPC) [149]. What unifies these services is their software engineering philosophy.

Microservices use a modular design pattern, which simplifies distributed application design and deployment. Microservices are loosely-coupled, communicating through a set of common APIs, invoked via RPCs [88], and maintain state via *reliable collections* [168]. As a result, developers can take advantage of languages and libraries of their choice, while not having to worry about microservice placement, communication mechanisms, fault tolerance, or availability.

Microservices are also attractive to data center operators as they provide a way to improve server utilization. Microservices execute as light-weight processes that are easier to scale and migrate compared with a monolithic development approach. They can be activated upon incoming client requests, execute to request completion, and then swapped out.

A microservice platform, such as Azure Service Fabric [168], Amazon Lambda [13], Google Application Engine [15], or Nirmata [20], is a distributed system manager that enables isolated microservice execution on shared data center infrastructure. To do so, microservice platforms include the following components (cf. [168]): 1. *federation subsystem*, abstracting and grouping servers

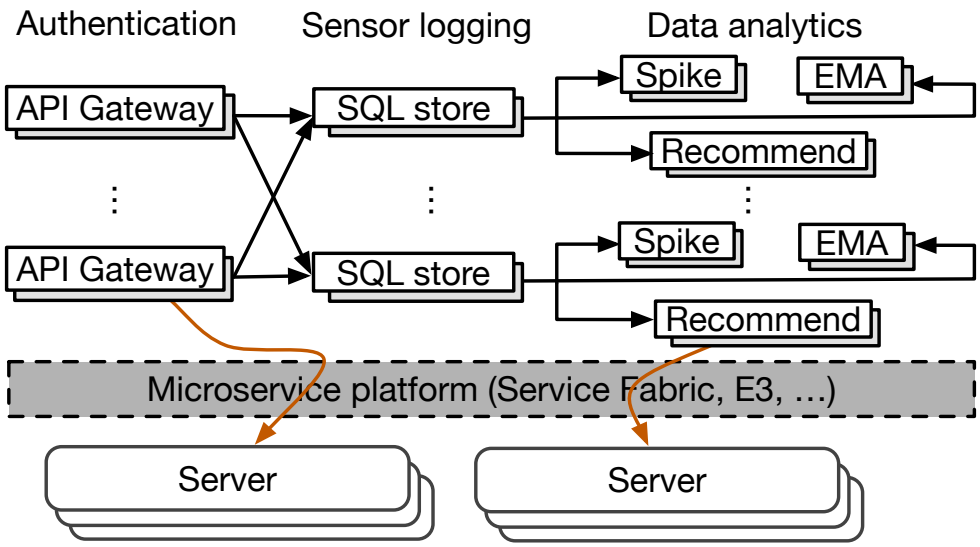


Figure 5.1: Thermostat analytics as DAG of microservices. The platform maps each DAG node to a physical computing node.

into a unified cluster that holds deployed applications; 2. *resource manager*, allocating computation resources to individual microservices based on their execution requirements; 3. *orchestrator*, dynamically scheduling and migrating microservices within the cluster based on node health information, microservice execution statistics, and service-level agreements (SLAs); 4. *transport subsystem*, providing (secure) point-to-point communication among various microservices; 5. *failover manager*, guaranteeing high availability/reliability through replication; 6. troubleshooting utilities, which assist developers with performance profiling/debugging and understanding microservice co-execution interference.

A microservice platform usually provides a number of programming models [86] that developers adhere to, like data-flow and actor-based. The models capture the execution requirements and describe the communication relationship among microservices. For example, the data-flow model (e.g. Amazon Datapipe [29], Google Cloudflow [55], Azure Data Factory [72]) requires programmers to assemble microservices into a directed acyclic graph (DAG): nodes contain microservices that are interconnected via flow-controlled, lossless data-flow channels. These models

MS.	Host (Linux)						Host (DPDK)							SmartNIC					
	RPS	W	C	L	99%	RPJ	RPS	W	C	L	99%	RPJ	%	RPS	W	L	99%	RPJ	×
IPsec	0.8M	117.0	12	1.8	6.6	7.0K	0.9M	112.1	12	1.7	5.2	8.1K	15.9	1.9M	23.4	0.2	0.8	79.0K	9.7
BM25	91.9K	116.4	12	40.3	205.8	0.8K	99.5K	110.0	12	30.7	155.6	0.9K	14.5	0.4M	19.2	4.1	12.4	20.6K	22.8
NIDS	1.8M	111.0	12	0.06	0.2	16.1K	1.8M	106.8	12	0.05	0.15	17.2K	7.4	2.0M	23.4	0.03	0.1	84.8K	4.9
Recom.	3.6K	109.4	12	86.6	477.0	0.03K	4.1K	111.7	12	78.7	358.6	0.04K	11.6	12.8K	18.9	21.3	123.6	0.7K	18.4
NATv4	1.9M	72.1	8	0.04	0.1	26.2K	1.9M	52.1	4	0.04	0.1	36.8K	40.4	2.0M	23.6	0.03	0.09	86.9K	2.4
Count	2.0M	68.1	6	0.07	0.1	28.8K	2.0M	48.6	4	0.03	0.1	40.3K	40.0	2.0M	21.0	0.03	0.09	96.1K	2.4
EMA	2.0M	72.7	8	0.04	0.2	27.0K	2.0M	52.1	4	0.03	0.09	38.6K	42.8	2.1M	22.0	0.03	0.08	93.5K	2.4
KVS	1.9M	48.6	8	0.04	0.1	40.0K	2.0M	33.6	2	0.04	0.1	59.6K	49.0	2.0M	21.6	0.03	0.1	97.1K	1.6
F.M.	1.9M	70.9	8	0.04	0.1	27.4K	2.0M	49.8	4	0.03	0.09	40.4K	47.4	2.0M	24.3	0.03	0.08	83.6K	2.1
DDoS	2.0M	111.2	12	0.05	0.2	17.9K	1.8M	105.7	12	0.05	0.2	17.4K	**-3.0**	2.0M	24.3	0.03	0.1	80.4K	4.6
KNN	42.2K	118.3	12	53.7	163.4	0.4K	42.4K	110.4	12	45.8	161.3	0.4K	7.5	29.9K	20.0	20.6	80.3	1.5K	3.9
Spike	91.9K	112.5	12	29.3	94.5	0.8K	0.1M	112.3	12	25.7	83.0	0.9K	13.7	73.8K	23.5	9.0	50.3	3.1K	3.4
Bayes	12.1K	113.9	12	82.0	406.5	0.1K	13.7K	112.0	12	80.6	400.5	0.1K	14.8	1.6K	19.5	41.9	164.7	0.08K	**0.7**
GW	1.5M	108.5	12	0.9	3.2	14.2K	1.6M	110.6	12	0.8	2.7	14.3K	1.1	0.1M	24.7	8.5	403.6	5.0K	**0.4**
TopR	0.7M	119.7	12	4.0	15.0	5.9K	0.8M	109.2	12	3.5	12.3	7.1K	18.9	14.8K	20.3	31.1	154.9	0.7K	**0.1**
SQL	0.5M	114.7	12	6.9	31.1	4.0K	0.5M	113.0	12	6.7	29.5	4.7K	15.7	39.5K	18.8	29.5	104.2	2.1K	**0.4**

Table 5.1: Microservice comparison among host (Linux and DPDK) and SmartNIC. RPS = Throughput (requests/s), W = Active power (W), C = Number of active cores, L = Average latency (ms), 99% = 99th percentile latency, RPJ = Energy efficiency (requests/Joule).

bring attractive benefits for a heterogeneous platform since they explicitly express concurrency and communication, enabling the platform to transparently map it to the available hardware [226, 227]. Figure 5.1 shows an IoT thermostat analytics application [57] consisting of microservices arranged in 3 stages: 1. Thermostat sensor updates are authenticated by the API gateway; 2. Updates are logged into a SQL store sharded by a thermostat identifier; 3. SQL store updates trigger data analytic tasks (e.g, spike detection, moving average, and recommendation) based on thresholds. The data-flow programming model allows the SQL store sharding factor to be dynamically adjusted to scale the application with the number of thermostats reporting. Reliable collections ensure state consistency when re-sharding and the microservice platform automatically migrates and deploys DAG nodes to available hardware resources.

A microservice can be stateful or stateless. Stateless microservices have no persistent storage

and only keep state within request context. They are easy to scale, migrate, and replicate, and they usually rely on other microservices for stateful tasks (e.g., a database engine). Stateful microservices use platform APIs to access durable state, allowing the platform full control over data placement. For example, Service Fabric provides reliable collections [168], a collection of data structures that automatically persist mutations. Durable storage is typically disaggregated for microservices and accessed over the network. The use of platform APIs to maintain state allows for fast service migration compared with traditional virtual machine migration [127], as the stateful working set is directly observed by the platform. All microservices in Figure 5.1 are stateful. We describe further microservices in §5.5.

5.3.2 SmartNIC-accelerated Server

A SmartNIC-accelerated server is a commodity server with one or more SmartNICs. Host and SmartNIC processors do not share thermal, memory, or cache coherence domains, and communicate via DMA engines over PCIe. This allows them to operate as independent, heterogeneous computers, while sharing a power domain and its idle power.

SmartNICs hold promise for improving server energy-efficiency when compared to other heterogeneous computing approaches. For example, racks populated with low-power servers [107] or a heterogeneous mix of servers, suffer from high idle energy draw, as each server requires energy to power its chassis, including fans and devices, and its own ToR switch port. System-on-chip designs with asymmetric performance, such as ARM's big.LITTLE [164] and DynamIQ [49] architectures, and AMD's heterogeneous system architecture (HSA) [30], which combines a GPU with a CPU on the same die, have scalability limits due to the shared thermal design point (TDP). These architectures presently scale to a maximum of 8 cores, making them more applicable to mobile than to server applications. GPGPUs and single instruction multiple threads (SIMT) architectures, such as Intel's Xeon Phi [62] and HP Moonshot [58], are optimized for computational throughput and the extra interconnect hop prevents these accelerators from running latency-sensitive microservices efficiently [205]. SmartNICs are not encumbered by these problems and can thus be used to balance

the power draw of latency-sensitive services efficiently.

5.3.3 Benefits of SmartNIC Offload

We quantify the potential benefit of using SmartNICs for microservices on energy efficiency and request latency. To do so, we choose two identical commodity servers and equip one with a traditional 10GbE Intel X710 NIC and the other with a 10GbE Cavium LiquidIO SmartNIC. Then we evaluate 16 different microservices (detailed in §5.5) on these two servers with synthetic benchmarks of random 512B requests. We measure request throughput, wall power consumed at peak throughput (defined as the knee of the latency-throughput graph, where queueing delay is minimal) and when idle, as well as client-observed, average/tail request latency in a closed loop. We use host cores on the traditional server and SmartNIC cores on the SmartNIC server for microservice execution. We use as many identical microservice instances, CPUs, and client machines as necessary to attain peak throughput and put unused CPUs to their deepest sleep state. The SmartNIC does not support per-core low power states and always keeps all 12 cores active, diminishing SmartNIC energy efficiency results somewhat. The SmartNIC microservice runtime system uses a kernel-bypass network stack (cf. §5.5). To break out kernel overheads from the host experiments, we run all microservices on the host in two configurations: 1. Linux kernel network stack; 2. kernel-bypass network stack [217], based on Intel's DPDK [59].

Table 5.1 presents measured peak request throughput, active power (wall power at peak throughput minus idle wall power), number of active cores, (tail-)latency, and energy efficiency, averaged over 3 runs. Active power allows a direct comparison of host to SmartNIC processor power draw. Energy efficiency equals throughput divided by active power.

Kernel overhead. We first analyze the overhead of in-kernel networking on the host (Linux versus DPDK). As expected, the kernel-bypass networking stack performs better than the in-kernel one. On average, it improves energy efficiency by 21% (% column in Table 5.1) and reduces tail latency by 16%. Energy efficiency improves because (1) DPDK achieves similar throughput with fewer cores; (2) at peak server CPU utilization, DPDK delivers higher throughput.

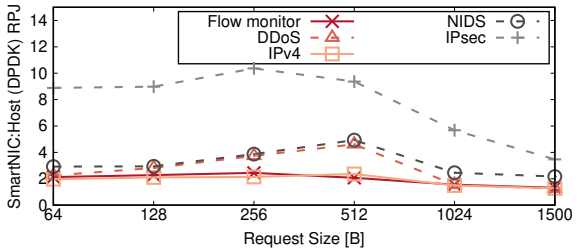


Figure 5.2: Request size impact on SmartNIC Request/Joule benefits for five different microservices.

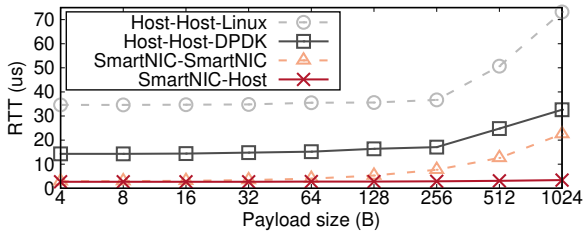


Figure 5.3: Average RTT (3 runs) of different communication mechanisms in a SmartNIC-accelerated server.

SmartNIC performance. SmartNIC execution improves the energy efficiency of 12 of the measured microservices by a geometric mean of 6.5× compared with host execution using kernel-bypass (× column in Table 5.1). The SmartNIC consumes at most 24.7W active power to execute these microservices while the host processor consumes up to 113W. IPSec, BM25, Recommend, and NIDS particularly benefit from various SmartNIC hardware accelerators (crypto coprocessor, fetch-and-add atomic units, floating-point engines, and pattern matching units). NATv4, Count, EMA, KVS, Flow monitor, and DDoS can take advantage of the computational bandwidth and fast memory interconnect of the SmartNIC. In these cases, the energy efficiency comes not just from the lower power consumed by the SmartNIC, but also from peak throughput improvements versus the host processor. KNN and Spike attain lower throughput on the SmartNIC. However, since the SmartNIC consumes less power, the overall energy efficiency is still better than the host. For all of these microservices, the SmartNIC also improves client-observed latency. This is due to the hardware-accelerated packet buffers and the elimination of PCIe bus traversals. SmartNICs can reduce average and tail latency by a geometric mean of 45.3% and 45.4% versus host execution, respectively.

The host outperforms the SmartNIC for Top ranker, Bayes classifier, SQL, and API gateway by a geometric mean of 4.1× in energy efficiency, 41.2% and 30.0% in average and tail latency reduction. These microservices are branch-heavy with large working sets that are not handled well

by the simpler cache hierarchy of the SmartNIC. Moreover, the API gateway uses double floating-point numbers for the rate limiter implementation, which the SmartNIC emulates in software.

Request size impact. SmartNIC performance depends also on request size. To demonstrate this, we vary the request size of our synthetic workload and evaluate SmartNIC energy efficiency benefits of 5 microservices versus host execution. Figure 5.2 shows that with small ($\leq$128B) requests, SmartNIC benefit of IPSec, NIDS, and DDoS is smaller. Small requests are more computation intensive and we are limited by the SmartNIC's wimpy cores. SmartNIC offload hits a sweet-spot at 256–512B request size, where the benefit almost doubles. Here, network and compute bandwidth utilization are balanced for the SmartNIC. At larger request sizes, we are network bandwidth limited, allowing us to put host CPUs to sleep and SmartNIC benefits again diminish. This can be seen in particular for IPsec, which outperforms on the SmartNIC due to hardware cryptography acceleration, but still diminishes with larger request sizes. We conclude that request size has a major impact on the benefit of SmartNIC offload. Measuring it is necessary to make good offload choices.

We conclude that SmartNIC offload can provide large energy efficiency and latency benefits for many microservices. However, it is not a panacea. Computation and memory-intensive microservices are more suitable to run on the host processor. We need an efficient method to define and monitor critical SmartNIC offload criteria for microservices.

5.3.4 Challenges of SmartNIC Offload

While there are quantifiable benefits, offloading microservices to SmartNICs brings a number of additional challenges:

- SmartNICs share the same Ethernet MAC address with the host server. Layer 2 switching is not enough to route traffic between SmartNICs and host servers. We require a different switching scheme that can balance traffic and provide fault tolerance when a server equips multiple SmartNICs.
- Microservice platforms assume uniform communication performance among all computing nodes.

However, Figure 5.3 shows that SmartNIC-Host (via PCIe) and SmartNIC-SmartNIC (via ToR switch) communication round-trip-time (RTT) is up to 83.3% and 86.2% lower than host-host (via ToR switch) kernel-bypass communication. We have to consider this topology effect to achieve good performance.

- Microservices share SmartNIC resources and contend with SmartNIC firmware for cache and memory bandwidth. This can create a head-of-line blocking problem for network packet exchange with both SmartNIC and host. Prolonged head-of-line blocking can result in denial of service to unrelated microservices and is more severe than transient sources of interference, such as network congestion. We need to sufficiently isolate SmartNIC-offloaded microservices from firmware to guarantee the quality of service.

5.4 E3 Microservice Platform

We present the E3 microservice platform for SmartNIC-accelerated servers. Our goal is to maximize microservice energy efficiency at scale. Energy efficiency is the ratio of microservice throughput and cluster power draw. Power draw is determined by our choice of SmartNIC-acceleration, while E3 focuses on maximizing microservice throughput on this heterogeneous architecture. We describe how we support microservice offload to a SmartNIC and address the request routing, microservice placement, and scheduling challenges.

E3 overview. E3 is a distributed microservice execution platform. We follow the design philosophies of Azure Service Fabric [168] but add energy efficiency as a design requirement. Figure 5.4 shows the hardware and software architecture of E3. E3 runs in a typical data center, where servers are grouped into racks, with a ToR switch per rack. Each server is equipped with one or more SmartNICs, and each SmartNIC is connected to the ToR. This creates a new topology where host processors are reachable via any of the SmartNICs (Figure 5.4-a). SmartNICs within the same server also have multiple communication options—via the ToR or PCIe (§5.4.1).

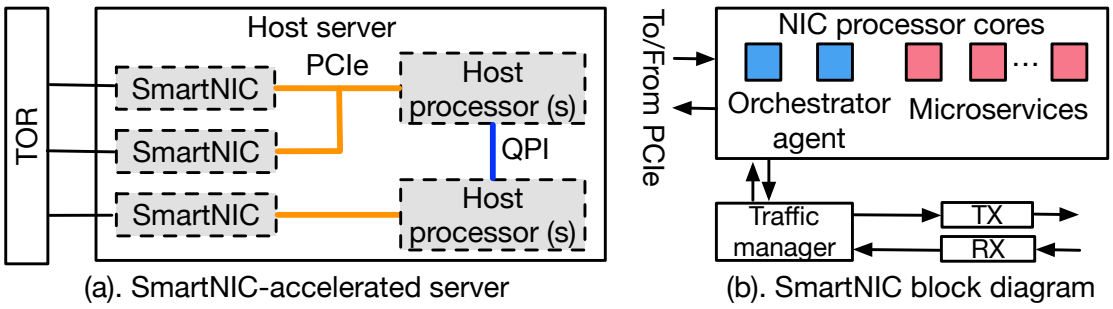


Figure 5.4: Hardware and software architecture of E3.

Programming model. E3 uses a data-flow programming model. Programmers assemble microservices into a DAG of microservice nodes interconnected via channels in the direction of RPC flow (cf. Figure 5.1). A channel provides lossless data communication between nodes. A DAG in E3 describes all RPC and execution paths of a single microservice application, but multiple DAGs may coexist and execute concurrently. E3 is responsible for mapping DAGs to computational nodes.

Software stack. E3 employs a central, replicated cluster resource controller [168] and a microservice runtime on each host and SmartNIC. The resource controller includes four components: (1) traffic control, responsible for routing and load balancing requests between different microservices; (2) control-plane manager, placing microservice instances on cluster nodes; (3) data-plane orchestrator, dynamically migrates microservices across cluster nodes; (4) failover/replication manager, providing failover and node membership management using consistent hashing [241]. The microservice runtime includes an execution engine, an orchestrator agent, and a communication subsystem, described next.

Execution engine. E3 executes each microservice as a multi-threaded process, either on the SmartNIC or on the host. The host runs Linux. The SmartNIC runs a light-weight firmware. Microservices interact only via microservice APIs, allowing E3 to abstract from the OS. SmartNIC

and host support hardware virtual memory for microservice confinement. E3 is work-conserving and runs requests to completion. It leverages a round-robin policy for steering incoming requests to cores, context switching cores if needed.

Orchestrator agent. Each node runs an orchestrator agent to periodically monitor and report runtime execution characteristics to the resource controller. The information is used by (1) the failover manager to determine cluster health and (2) the data-plane orchestrator to monitor the execution performance of each microservice and make migration decisions. On the host, the agent runs as a separate process. On the SmartNIC, the agent runs on dedicated cores (blue in Figure 5.4-b) and a traffic manager hardware block exchanges packets between the NIC MAC ports and the agent. For each packet, the agent determines the destination (network, host, or SmartNIC core).

5.4.1 Communication Subsystem

E3 leverages various communication mechanisms, depending on where communicating microservices are located.

Remote communication. When communicating among host cores across different servers, E3 uses the Linux network stack. SmartNIC remote communication uses a user-level network stack[217].

Local SmartNIC-host communication. SmartNIC and host cores on the same server communicate via PCIe. Prior work has extensively explored communication channels via PCIe [189, 194, 151, 211], and we adopt their design. High-throughput messaging for PCIe interconnects requires leveraging multiple DMA engines in parallel. E3 takes advantage of the eight DMA engines on the LiquidIO, which can concurrently issue scatter/gather requests.

Local SmartNIC-SmartNIC communication. SmartNICs in the same host can use three methods for communication. 1. Using the host to relay requests, involving two data transfers over PCIe and pointer manipulation on the host, increasing latency. 2. PCIe peer-to-peer [143], which is

supported on most SmartNICs [76, 67, 68]. However, the bandwidth of peer-to-peer PCIe communication is capped in a NUMA system when the communication passes between sockets [205]. 3. ToR switch. We take the third approach and our experiments show that this approach incurs lower latency and achieves higher bandwidth than the first two.

5.4.2 Addressing and Routing

Since SmartNICs and their host servers share Ethernet MAC addresses, we have to use an addressing/routing scheme to distinguish between these entities and load balance across them. For illustration, assume we have a server with two SmartNICs; each NIC has one MAC port. If remote microservices communicate with this server, there will be two possible paths and each might be congested.

We use equal-cost multi-path (ECMP) [1] routing on the ToR switch to route and balance load among these ports. We assign each SmartNIC and the host its own IP. We then configure the ToR switch to route to SmartNICs directly via the attached ToR switch port and an ECMP route to the host IP via any of the ports. The E3 communication subsystem on each SmartNIC differentiates by destination IP address whether an incoming packet is for the SmartNIC or the host. On the host, we take advantage of NIC teaming [19] (also known as port trunking) to bond all related SmartNIC ports into a single logical interface, and then apply the dynamic link aggregation policy (supporting IEEE 802.3ad protocol). ECMP automatically balances connections to the host over all available ports. If a link or SmartNIC fails, ECMP will automatically rebalance new connections via the remaining links, improving host availability.

5.4.3 Control-plane Manager

The control-plane manager is responsible for energy-efficient microservice placement. This is a computing-intensive operation due to the large search space with myriad constraints. Hence, it is done on the control plane. Service Fabric uses simulated annealing, a well-known approximate algorithm, to solve microservice placement. It considers three types of constraints: (1) currently

available resources of each computing node (memory, disk, CPU, network bandwidth); (2) computing node runtime statistics (aggregate outstanding microservice requests); (3) individual microservice execution behavior (average request size, request execution time and frequency, diurnal variation, etc.). Service Fabric ignores network topology and favors spreading load over multiple nodes.

E3 extends this algorithm to support bump-in-the-wire SmartNICs, considering network topology. We categorize *computing nodes* (host or SmartNIC processors) into different levels of communication distance and perform a search from the closest to the furthest. We present the HCM algorithm (Algorithm 3). HCM takes as input the microservice DAG G and source nodes V_{src}, as well as the cluster topology T, including runtime statistics for each computing node (as collected). HCM performs a breadth-first traversal of G to map microservices to cluster computing nodes (*MS_DAG_TRAVERSE*).

If not already deployed (*get_deployed_node*), HCM (via *MS_DAG_TRAVERSE*) assigns a microservice V to a computing node N via the *find_first_fit* function (lines 9-11) and deploys it via *set_deployed_node*. *find_first_fit* is a greedy algorithm that returns the first computing node that satisfies the microservice constraints (via its resource and runtime statistics) without considering communication cost. If no such node is found, it returns a node closest to the constraints. Next, for the descendant microservices of a node V (lines 12-15), HCM assigns them to computing nodes based on their communication distance to V (*MS_PLACE*). To do so, HCM first computes the *hierarchical topology representation* of computing node N via *get_hierarchical_topo*. Each level in the hierarchical topology includes computing nodes that require a similar communication mechanism, starting with the closest. For example, in a single rack there are four levels in this order: 1. The same computing node as V; 2. An adjacent computing node on the same server; 3. A SmartNIC computing node on an adjacent server; 4. A host computing node on an adjacent server. If there are multiple nodes in the same level, HCM uses *find_best_fit* to find the best fit, according to resource constraints. If no node in the hierarchical topology fits the constraints, we fall back to *find_first_fit*.

Algorithm 3 HCM microservice placement algorithm

```
1: G : microservice DAG graph
2: V_src : source microservice node(s) of the DAG
3: T : server cluster topology graph
4: procedure MS_DAG_TRAVERSE(G, V_src, T)
5:     Q.enqueue(V_src)                                   ▷ Let Q be a queue
6:     while Q is not empty do
7:         V ← Q.dequeue()
8:         N ← get_deployed_node(V)
9:         if N is NULL then
10:            N ← find_first_fit(V, T)
11:            set_deployed_node(V, N)
12:        end if
13:        for W in all direct descendants of V in G do
14:            N_W ← MS_PLACE(W, N, T)
15:            set_deployed_node(W, N_W)
16:            Q.enqueue(W)
17:        end for
18:    end while
19: end procedure
20: < V, N, T >=< microservice, computation node, cluster topology graph >
21: procedure MS_PLACE(V, N, T)
22:    Topo ← get_hierarchical_topo(N, T)
23:    for L in all Topo.Levels do
24:        N ← find_best_fit(V, Topo.node_list(L))
25:        if N is not NULL then
26:            return N
27:        end if
28:    end for
29:    return find_first_fit(V, T)                        ▷ Ignore topology
30: end procedure
```

5.4.4 Data-plane Orchestrator

The data-plane orchestrator is responsible for detecting load changes and migrating microservices in response to these changes among computational nodes at runtime. To do so, we piggyback several measurements onto the periodic node health reports made by orchestrator agents to the resource controller: This approach is light-weight and integrates well with runtime execution. We believe that our proposed methods can also be used in other microservice schedulers [216, 223, 157].

In this section, we introduce the additional techniques implemented in our data-plane orchestrator to mitigate issues of SmartNIC overload caused by compute-intensive microservices. These can interfere with the SmartNIC's traffic manager, starving the host of network packets. They can also simply execute too slowly on the SmartNIC to be able to catch up with the incoming request rate.

Host starvation. This issue is caused by head-of-line blocking of network traffic due to microservice interference with firmware on SmartNIC memory/cache. It is typically caused by a single compute-intensive microservice overloading the SmartNIC. To alleviate this problem, we monitor the incoming/outgoing network throughput and packet queue depth at the traffic manager. If network bandwidth is under-utilized, but there is a standing queue at the traffic manager, the SmartNIC is overloaded, and we need to migrate microservices.

Microservice overload. This issue is caused by microservices in aggregate requiring more computational bandwidth than the SmartNIC can offer, typically because too many microservices are placed on the same SmartNIC. To detect this problem, we periodically monitor the execution time of each microservice and compare to its exponential moving average. When the difference is negative and larger than 20%, we assume a microservice overload and trigger microservice migration. The threshold was determined empirically.

Microservice migration. For either issue, the orchestrator will migrate the microservice with the highest CPU utilization to the host. To do so, it uses a cold migration approach, similar to other microservice platforms. Specifically, when the orchestrator makes a migration decision, it will first push the microservice binary to the new destination, and then notify the runtime of the old node to (1) remove the microservice instance from the execution engine; (2) clean up and free any local resources; (3) migrate the working state, as represented by reliable collections [168], to the destination. After the orchestrator receives a confirmation from the original node, it will update connections and restart the microservice execution on the new node.

5.4.5 Failover/Replication Manager

Since SmartNICs share the same power supply as their host server, our failover manager treats all SmartNICs and the host to be in the same fault domain [168], avoiding replica placement within the same. Replication for fault tolerance is typically done across different racks of the same data center or across data centers, and there is no impact from placing SmartNICs in the same failure domain as hosts.

5.5 Implementation

Host software stack. The E3 resource controller and host runtime are implemented in 1,287 and 3,617 lines of C (LOC), respectively, on Ubuntu 16.04. Communication among co-located microservices uses per-core, multi-producer, single-consumer FIFO queues in shared memory. Our prototype uses UDP for all network communication.

SmartNIC runtime. The E3 SmartNIC runtime is built in 3,885 LOC on top of the Cavium CDK [79], with a user-level network stack. Each microservice runs on a set of non-preemptive hardware threads. Our implementation takes advantage of a number of hardware accelerator libraries. We use (1) a hardware managed memory manager to store the state of each microsevice, (2) the hardware traffic controller for Ethernet MAC packet management, and (3) atomic fetch-

Microservice	S	Description
IPsec		Authenticates (SHA-1) & encrypts (AES-CBC-128) NATv4 [173]
BM25		Search engine ranking function [81], e.g., ElasticSearch
NATv4		IPv4 network address translation using DIR-24-8-BASIC [154]
NIDS		Network intrusion detection w/ aho-corasick parallel match [26]
Count	✓	Item frequency counting based on a bitmap [173]
EMA	✓	Exponential moving average (EMA) for data streams [51]
KVS	✓	Hashtable-based in-memory key-value store [145]
Flow mon.	✓	Flow monitoring system using count-min sketch [173]
DDoS	✓	Entropy-based DDoS detection [206]
Recommend	✓	Recommendation system using collaborative filtering [46]
KNN		Classifier using the K-nearest neighbours algorithm [64]
Spike	✓	Spike detector from a data stream using Z-score [231]
Bayes		Naive Bayes classifier based on *maximum a posteriori* [195]
API gw	✓	API rate limiter and authentication gateway [32]
Top Ranker	✓	Top-K ranker using quicksort [250]
SQL	✓	In-memory SQL database [17]

Table 5.2: 16 microservices implemented on E3. S = Stateful.

and-add units to gather performance statistics. We use page protection of the cnMIPS architecture to confine microservices.

Microservices. We implemented 16 popular microservices on E3, as shown in Table 5.2, in an aggregate 6,966 LOC. Six of the services are stateless or use read-only state that is modified only via the cluster control plane. The remaining services are stateful and use reliable collections to maintain their state. When running on the SmartNIC, IPsec and API gateway can use the crypto coprocessor (105 LOC), while Recommend and NIDS can take advantage of the deterministic finite automata unit (65 LOC). For Count, EMA, KVS, and Flow monitor, our compiler automatically uses the dedicated atomic fetch-and-add units on the SmartNIC. When performing single-precision floating-point computations (EMA, KNN, Spike, Bayes), our compiler generates FPU code on the

Application	Description	N	Microservices
NFV-FIN	Flow monitoring [173, 219]	72	Flow mon., IPsec, NIDS
NFV-DIN	Intrusion detection [260, 219]	60	DDoS, NATv4, NIDS
NFV-IFID	IPsec gateway [260, 173]	84	NATv4, Flow mon., IPsec, DDoS
RTA-PTC	Twitter analytics [250]	60	Count, Top Ranker, KNN
RTA-SF	Spam filter [160]	96	Spike, Count, KVS, Bayes
RTA-SHM	Server health mon. [163]	84	Count, EMA, SQL, BM25
IOT-DH	IoT data hub [70]	108	API gw, Count, KNN, KVS, SQL
IOT-TS	Thermostat [57]	108	API,EMA,Spike,Recommend,SQL

Table 5.3: 8 microservice applications. N = # of DAG nodes.

SmartNIC. Double-precision floating-point calculations (API gateway) are software emulated. E3 reliable collections currently only support hashtables and arrays, preventing us from migrating the SQL engine. We thus constrain the control-plane manager to pin SQL instances to host processors.

Applications. Based on these microservices, we develop eight applications across three application domains: (1) Distributed real-time analytics (RTA), such as Apache Storm [250], implemented as a data-flow processing graph of workers that pass data tuples in real time to trigger computations; (2) Network function (NF) virtualization (NFV) [215], which is used to build cloud-scale network middleboxes, software switches, and enterprise IT networks, by chaining NFs; (3) An IoT hub (IOT) [36], which gathers sensor data from edge devices and generates events for further processing (e.g., spike detection, classifier) [70]. To maximize throughput, applications may shard and replicate microservices, resulting in a DAG node count larger than the involved microservice types. Table 5.3 presents the microservice types involved in each application, the deployed DAG node count, and references the workloads used for evaluation. The workloads are trace-based and synthetic benchmarks, validated against realistic scenarios. The average and maximum node fanouts among our applications are 6 and 12, respectively. Figure5.1 shows IOT-TS as an example. IOT-TS is sharded into 6×API, 12×SQL, 12×EMA, 12×Spike, and 12× recommend and each

System/Cluster	Cost [$]	BC	WC	Mem	Idle	Peak	Bw
Beefy	4,500	12	0	64	83	201	20
Wimpy	2,209	0	32	2	79	95	20
Type1-SmartNIC	4,650	12	12	68	98	222	20
Type2-SmartNIC	6,750	16	48	144	145	252	40
SuperBeefy	12,550	24	0	192	77	256	80
4×Beefy	18,000	48	0	256	332	804	80
4×Wimpy	8,836	0	128	8	316	380	80
2×B.+2×W.	13,018	24	64	132	324	592	80
2×Type2-SmartNIC	13,500	32	96	288	290	504	80
1×SuperBeefy	12,550	24	0	192	77	256	80

Table 5.4: Evaluated systems and clusters. BC = Beefy cores, WC = Wimpy cores, Mem = Memory (GB), Idle and Peak power (W), Bw = Network bandwidth (Gb/s).

microservice has one backup replica.

5.6 Evaluation

Our evaluation aims to answer the following questions:

1. What is the energy efficiency benefit of microservice SmartNIC-offload? Is it proportional to client load? What is the latency cost? (§5.6.1)
2. Does E3 overcome the challenges of SmartNIC-offload? (§5.6.2, §5.6.3, §5.6.4)
3. Do SmartNIC-accelerated servers provide better total cost of ownership than other cluster architectures? (§5.6.5)
4. How does E3 perform at scale? (§5.6.6)

Experimental setup. Our experiments run on a set of clusters (Table 5.4 presents the server and cluster configurations), attached to an Arista DCS-7050S ToR switch. Beefy is a Supermicro 1U server, with a 12-core E5-2680 v3 processor at 2.5GHz, and a dual-port 10Gbps Intel X710 NIC.

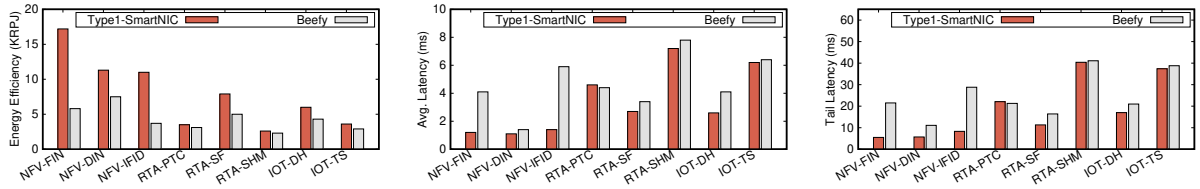
Wimpy is ThunderX-like, with a CN6880 processor (32 cnMIPS64 cores running at 1.2GHz), and a dual-port 10Gbps NIC. SuperBeefy is a Supermicro 2U machine, with a 24-core Xeon Platinum 8160 CPU at 2.1GHz, and a dual-port 40Gbps Intel XL710 NIC. Our SmartNIC is the Cavium LiquidIOII [43], with one OCTEON processor with 12 cnMIPS64 cores at 1.2GHz, 4GB memory, and two 10Gbps ports. Based on this, we build two SmartNIC servers: Type1 is Beefy, but swaps the X710 10Gbps NIC with the Cavium LiquidIOII; Type2 is a 2U server with two 8-core Intel E5-2620 processors at 2.1GHz, 128GB memory, and 4 SmartNICs. All servers have a Seagate HDD. We build the clusters such that each has the same amount of aggregate network bandwidth. This allows us to compare energy efficiency based on the compute bandwidth of the clusters, without varying network bandwidth. We also exclude the switch from our cost and energy evaluations, as each cluster uses an identical number of switch ports.

We measure server power consumption using the servers' IPMI data center management interface (DCMI), cross-checked by a Watts Up wall power meter. Throughput and average/tail latency across 3 runs are measured from clients (Beefy machines), of which we provide as many as necessary. We enable hyper-threading and use the `Intel_pstate` governor for power management. All benchmarks in this section report energy efficiency as throughput over server/cluster **wall power** (not just active power).

5.6.1 Benefit and Cost of SmartNIC-Offload

Peak utilization. We evaluate the latency and energy efficiency of using SmartNICs for microservice applications, compared to homogeneous clusters. We compare 3×Beefy to 3×Type1-SmartNIC, to ensure that microservices also communicate remotely. We focus first on peak utilization, which is desirable for energy efficiency, as it amortizes idle power draw. To do so, we deploy as many instances of each application and apply as much client load as necessary to maximize request throughput without overloading the cluster, as determined by the knee of the latency-throughput curve.

Figure 5.5 shows that Type1-SmartNIC achieves an average 2.5×, 1.3×, and 1.3× better en-



(a) Energy-efficiency (b) Average latency (c) 99th-percentile latency

Figure 5.5: Energy-efficiency, average/tail latency comparison between Type1-SmartNIC and Beefy at peak utilization.

ergy efficiency across the NFV, RTA, and IOT application classes, respectively. This goes along with 43.3%, 92.3%, and 80.4% average latency savings and 35.5%, 90.4%, 88.6% 99th percentile latency savings, respectively. NFV-FIN gains the most—3× better energy efficiency—because E3 is able to run all microservices on the SmartNICs. RTA-PTC benefits the least—12% energy efficiency improvement at 4% average and tail latency cost—as E3 only places the Count microservice on the SmartNIC and migrates the rest to the host.

Power proportionality. This experiment evaluates the power proportionality of E3 (energy efficiency at lower than peak utilization). Using 3×Type1-SmartNIC, we choose an application from each class (NFV-FIN, RTA-SHM, and IOT-TS) and vary the offered request load between idle and peak via a client-side request limiter. Figure 5.8 shows that RTA-SHM and IOT-TS are power proportional. NFV-FIN is not power proportional but also draws negligible power. NFV-FIN runs all microservices on the SmartNICs, which have low active power, but the cnMIPS architecture has no per-core sleep states.

We conclude that applications can benefit from E3's microservice offload to SmartNICs, in particular at peak cluster utilization. Peak cluster utilization is desirable for energy efficiency and microservices make it more common due to light-weight migration. However, transient periods of low load can occur and E3 draws power proportional to request load. We can apply insights from Prekas, et al. [222] to reduce polling overheads and improve power proportionality further.

5.6.2 Avoiding Host Starvation

We show that E3's data-plane orchestrator prevents host starvation by identifying head-of-line blocking of network traffic. To do so, we use 3×Type1-SmartNIC and place as many microservices on the SmartNIC as fit in memory. E3 identifies the microservices that cause interference (Top Ranker in RTA-PTC, Spike in RTA-SF, API gateway in IOT-DH and IOT-TS) and migrates them to the host. As shown in Figure 5.7, our approach achieves up to 29× better energy efficiency and up to 89% latency reduction across RTA-PTC, RTS-SF, IOT-DH, and IOT-TS. For the other applications, our traffic engine has little effect because the initial microservice assignment already put the memory-intensive microservices on the host.

5.6.3 Sharing SmartNIC and Host Bandwidth

This experiment evaluates the benefits of sharing SmartNIC network bandwidth with the host. We compare two Type2-SmartNIC configurations: 1. Sharing aggregate network bandwidth among host and SmartNICs, using ECMP to balance host traffic over SmartNIC ports; 2. Replacing one SmartNIC with an Intel X710 NIC used exclusively to route traffic to the host. To emphasize the load balancing benefits, we always place the client-facing microservices on the host server. Note that SmartNIC-offloaded microservices still exchange network traffic (when communicating remotely or among SmartNICs) and interfere with host traffic.

Figure 5.9 shows that load balancing improves application throughput up to 2.9× and cluster energy efficiency up to 2.7× (NFV-FIN). Available host network bandwidth when sharing SmartNICs can be up to 4× that of the dedicated NIC, which balances better with the host compute bandwidth. With a dedicated NIC, host processors can starve for network bandwidth. IOT-TS is compute-bound and thus benefits the least from sharing. In terms of latency, all cases behave the same since the request execution flows are the same.

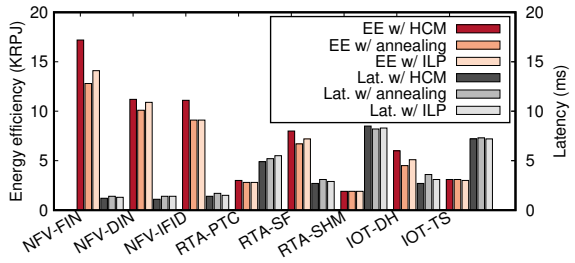


Figure 5.6: Communication-aware microservice placement.

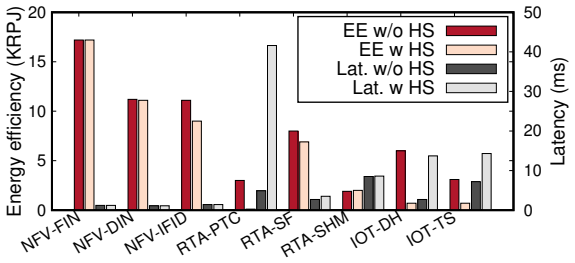


Figure 5.7: Avoiding host starvation via data-plane orchestrator.

5.6.4 Communication-aware Placement

To show the effectiveness of communication-aware microservice placement, we evaluate HCM on E3 without data-plane orchestrator. In this case, all microservices are stationary after placement. We avoid host starvation and microservice overload by constraining problematic microservices to the host.

Using 3×Type1-SmartNIC and all placement constraints of Service Fabric [168] (described in §5.4.3), we compare HCM with both simulated annealing and an integer linear program (ILP). HCM places the highest importance on minimizing microservice communication latency. Simulated annealing and ILP use a cost function with the highest weight on minimizing co-execution interference. Hence, HCM tries to co-schedule communicating microservices on proximate resources, while the others will spread them out. ILP attempts to find the best configuration, while simulated annealing approximates. Figure 5.6 shows that compared to simulated annealing and ILP, HCM improves energy efficiency by up to 35.2% and 22.0%, and reduces latency by up to 24.0% and 18.6%, respectively. HCM's short communication latency benefits outweigh interference from co-execution.

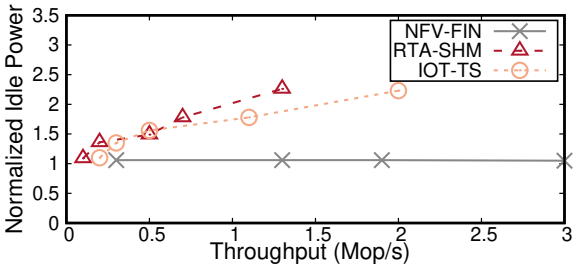


Figure 5.8: Power draw of 3 applications normalized to idle power of 3×Type1-SmartNIC, varying request load.

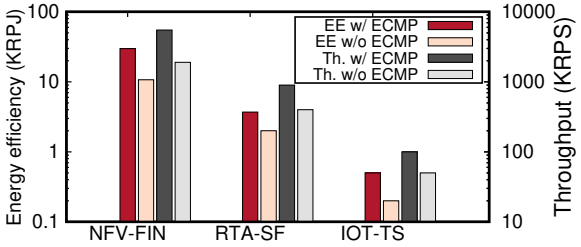


Figure 5.9: Energy-efficiency and throughput using ECMP-based SmartNIC sharing (log y scale).

5.6.5 Energy Efficiency = Cost Efficiency

While SmartNICs benefit energy efficiency and thus potentially bring cost savings, can they compete with other forms of heterogeneous clusters, especially when factoring in the capital expense to acquire the hardware? In this experiment, we evaluate the cost efficiency, in terms of request throughput over total cost and time of ownership, of using SmartNICs for microservices, compared with four other clusters (see Table 5.4). Assuming that clusters are usually at peak utilization, we use the cost efficiency metric $\frac{Throughput \times T}{CAPEX + (Power \times T \times Electricity)}$, where $Throughput$ is the measured average throughput at peak utilization for each application, as executed by E3 on each cluster, T is elapsed time, $CAPEX$ is the capital expense to purchase the cluster including all hardware components ($), $Power$ is the elapsed peak power draw of the cluster (Watts), and $Electricity$ is the price of electricity ($/Watts). The cluster cost and power data is shown in Table 5.4 and we use the average U.S. electricity price [153] of $0.0733/kWh. Figure 5.10 reports results for three applications of very different points in the workload space, extrapolated over time of ownership by our cost efficiency metric.

We make three observations. First, in the long term (>1 year of ownership), cost efficiency is increasingly dominated by energy efficiency. This highlights the importance of energy efficiency for data center design, where servers are typically replaced after several years to balance

Cluster	NFV-FIN	RTA-SHM	IOT-TS
4×Beefy	5.1	1.9	2.7
4×Wimpy	29.9	0.4	0.1
2×B.+2×W.	8.2	1.4	1.9
2×Type2-SmartNIC	29.0	4.5	6.1
1×SuperBeefy	8.8	2.9	5.0

Table 5.5: Energy efficiency across five clusters (KRPJ).

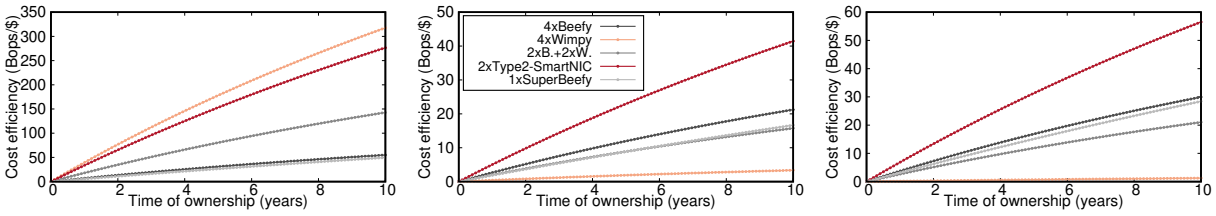
CAPEX [113]. Second, when all microservices are able to run on a low power platform (NFV-FIN), both 4×Wimpy and 2×Type2-SmartNIC clusters are the most cost efficient. After 5 years, 4×Wimpy is 14.1% more cost-efficient than 2×Type2-SmartNIC because of the lower power draw. Third, when a microservice application contains both compute and IO-intensive microservices (RTA-SHM, IOT-TS), the 2×Type2-SmartNIC cluster is up to 1.9× more cost-efficient after 5 years of ownership than the next best cluster configuration (4×Beefy in both cases).

Table 5.5 presents the measured energy-efficiency, which shows cost-efficiency in the limit (over very long time of ownership). We can see that 4×Wimpy is only 3% more energy-efficient (but has lower CAPEX) than 2×Type2-SmartNIC for NFV-FIN. 2×Type2-SmartNIC is on average 2.37× more energy-efficient (but has higher CAPEX) than 1×SuperBeefy, which is the second-best cluster in terms of energy-efficiency.

5.6.6 Performance at Scale

We evaluate and discuss the scalability of E3 along three axes: 1. Mechanism performance scalability; 2. Tail-latency; 3. Energy-efficiency.

Mechanism scalability. At scale, pressure on the control-plane manager and data-plane orchestrator increases. We evaluate the performance scalability of both mechanisms with an increasing



(a) NFV-FIN (b) RTA-SHM (c) IOT-TS

Figure 5.10: Cost efficiency of 3 applications across the cluster configurations from Table 5.4.

Servers →	**100**	**200**	**400**	**600**	**800**	**1,000**
HCM	4.85	8.31	19.83	34.32	74.39	263.46
Annealing	3.15	4.73	7.43	15.64	23.50	61.42
ILP	7.64	19.43	84.83	361.85	$\gg 1s$	$\gg 1s$

Table 5.6: Per-microservice deployment time (ms) scalability.

number of Type2-SmartNIC servers in a simulated FatTree [152] topology with 40 servers per rack. To avoid host starvation and microservice overload, E3's data-plane orchestrator receives one heartbeat message (16B) every 50ms from each SmartNIC that reports the queue length of the traffic manager and the SmartNIC's microservice execution times. The orchestrator parses the heartbeat message and makes a migration decision (§5.4.4). Figure 5.11 shows that the time taken to transmit the message and make a decision with a large number of servers stays well below the time taken to migrate the service (on the order of 10s-100s of ms) and is negligibly impacted by the number of deployed microservices. This is because the heartbeat message contributes only 1Kbps of traffic, even with 50K servers.

E3 uses HCM in the control-plane manager. We compare it to simulated annealing and ILP, deploying 10K microservices on an increasing number of servers. Table 5.6 shows that while HCM does not scale as well as simulated annealing, it can deploy new microservices in a reasonable time span ($<<$1s) at scale. ILP fails to deliver acceptable performance.

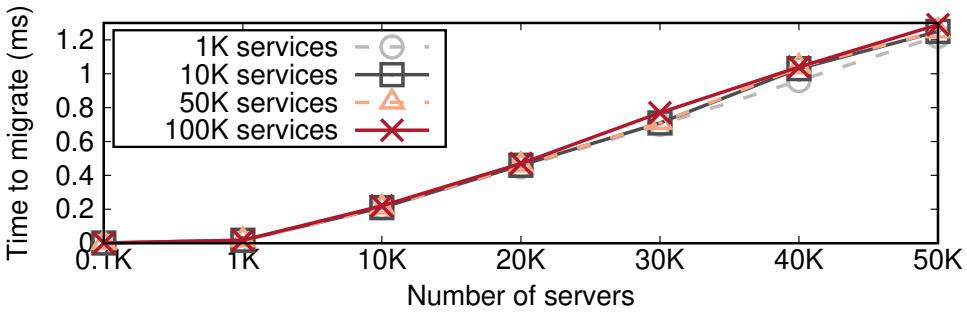


Figure 5.11: Orchestrator migration decision time scalability.

Tail latencies. At scale, tail latencies dominate [134]. While SmartNICs can introduce high tail latency for some microservices (§5.3.3), E3 places these microservices on the host to ensure that application-level tail-latency inflation is minimized (§5.6.1). The tail-latency impact of SmartNIC offload is reduced at scale, as baseline communication latency increases with increasing inter-rack distance.

Energy-efficiency and power budgets. E3's energy efficiency benefits are constant regardless of deployment size and power budgets. At scale, there is additional energy cost for core and spine switches, but these are negligible compared to racks (ToRs and servers). Within a rack, ToR switch energy consumption stays constant, as all compared systems use the same number of switch ports. Our results show that E3 achieves up to 1.9x more throughput under the same power budget. Conversely, operators can save 48% of power, offering the same bandwidth.

5.7 Related Work

5.7.1 Architecture Studies for Microservices

Prior work has explored the architectural implications of running microservices [148, 136, 251]. It shows that wimpy servers hold potential for microservices under low loads, as they have less cache utilization and network virtualization overhead compared with traditional monolithic cloud

workloads. FAWN [107] explored using only low-power processor architecture for data-intensive workloads as an energy- and cost-effective alternative to server multiprocessors. However, FAWN assumed that I/O speeds are much lower than CPU speeds and so CPUs would be left idle for data-intensive applications. With the advent of fast network technologies, server CPUs are still required. Motivated by these studies, we focus on using SmartNIC-accelerated servers for energy efficiency.

5.7.2 Heterogeneous Scheduling

A set of schedulers for performance-asymmetric architectures have been proposed. For example, Kumar et al. [180, 181] use instructions per cycle to determine the relative speedup of each thread on different types of cores. HASS [237] introduces the architectural signature concept as a scheduling indicator, which contains information about memory-boundedness, available instruction-level parallelism, etc. CAMP [229] combines both efficiency and thread-level parallelism specialization and proposes a light-weight technique to discover which threads could use fast cores more efficiently. PTask [226] provides a data-flow programming model for programmers to manage computation for GPUs. It enables sharing GPUs among multiple processes, parallelizes multiple tasks, and eliminates unnecessary data movements. These approaches target long-running computations, mostly on cache coherent architectures, rather than microsecond-scale, request-based workloads over compute nodes that do not share memory and are hence not applicable.

5.7.3 Microservice Scheduling

Wisp [243] enforces end-to-end performance objectives by globally adapting rate limiters and request schedulers based on operator policies under varying system conditions. This work is not concerned with SmartNIC heterogeneity. UNO [185] is an NFV framework that can systematically place NFs across SmartNIC and host with a resource-aware algorithm on the control plane. E3 is a microservice platform and thus goes several steps further: (1) E3 uses a data-plane orchestrator to detect node load and migrates microservices if necessary; (2) HCM considers communication

distance during the placement. With the advent of SmartNICs and programmable switches, researchers have identified the potential performance benefits of applying request processing across the communication path [120]. E3 is such a system designed for the programmable cloud and explores the energy efficiency benefits of running microservices.

5.7.4 Power Proportionality

Power proportional systems can vary energy use with the presented workload [197]. For example, Prekas, et al. propose an energy-proportional system management policy for memcached [222]. While E3 can provide energy proportionality, we are primarily interested in energy-efficiency. Geoffrey et al. [123] propose a heterogeneous power-proportional system. By carefully selecting component ensembles, it can provide an energy-efficient solution for a particular task. However, due to the high cost of ensemble transitions, we believe that this architecture is not fit for high bandwidth I/O systems. Rivoire, et al. propose a more balanced system design (for example, a low-power, mobile processor with numerous laptop disks connected via PCIe) and show that it achieves better energy efficiency for sorting large data volumes [225]. Pelican [112] presents a software storage stack on under-provisioned hardware targeted at cold storage workloads. Our proposal could be viewed as a balanced-energy approach for low-latency query-intensive server applications, rather than cold, throughput intensive ones.

5.8 Summary and Learned Lessons

This chapter presents E3, a microservice execution platform on SmartNIC-accelerated servers. E3 extends key system components (programming model, execution engine, communication subsystem, scheduling) of the Azure Service Fabric microservice platform to a SmartNIC. E3 demonstrates that SmartNIC offload can improve cluster energy-efficiency up to 3× and cost efficiency up to 1.9× at up to 4% latency cost for common microservices.

A key lesson we learned from building E3 is that when using PNFs to design an energy-efficient system, one should schedule computations to an execution engine that offers the best application

performance given the power budget. There are three perspectives regarding this. First, PNF offloading might not be an optimal design decision even though it usually encloses power-efficient computing engine. For complicated request processing such as SQL query involving compute-heavy analytical operators, the host server would be able to achieve the best requests per joule compared to any PNF hardware. Second, the interesting performance metrics that will drive the system design are throughput (i.e., requests per second) and latency SLA/SLO. Under a given power budget, one should find a solution that offers the highest throughput without violating the SLA/SLO. Third, energy-efficiency is also related to the amount of incoming traffic, which decides the request load. A line-rate energy-efficiency system solution would not be optimal for the unloaded case.

6 Exploring the PNF-based Disaggregated Storage

Storage disaggregation has become a trend for future data centers due to its flexibility and cost efficiency benefits. More recently, the significant drop of NAND flash price as well as the increasing network speeds have motivated hardware vendors to integrate PNF into disaggregated storage infrastructures to improve performance and reduce costs. In this chapter, we perform a preliminary investigation into SmartNIC-based disaggregated storage. Specifically, we take an emerging commercial-off-the-shelf (COTS) platform, analyze its hardware architecture and software stacks, and characterize its performance. Based on our observations, we then discuss open problems regarding how to use such hardware to design storage systems.

This chapter is organized as follows. Sections 6.1 and 6.2 present the background and solution overview. We then describe the SmartNIC-based disaggregated storage in detail in Section 6.3. Section 6.4 reports our characterization results. We discuss the related work in Section 6.5 and conclude this chapter by discussing open problems regarding building PNF-based disaggregated storage applications.

6.1 Background

Storage disaggregation has gained great interest recently since it allows independently scaling of compute/storage capacities and achieves high resource utilization [177, 18, 14, 9, 98]. As data center network speeds have transitioned to 40/100 Gbps and the NVMe-oF specification [78] (that enables flash-based SSDs to communicate over a network) is becoming widely adopted, a disaggregated NVMe SSD can provide microsecond-scale access latencies and millions of IOPS.

SmartNIC-based disaggregated storage solutions, such as Mellanox BlueField-2 [68] and Broad-

com Stingray PS1100R [41], have emerged and become increasingly popular because of their low deployment costs and competitive I/O performance as traditional server-based approaches. Such a storage node usually comprises a COTS high-bandwidth SmartNIC, some domain-specific accelerators (like RAID), a PCIe-switch, and many NVMe SSDs, supported by a standalone power supply. Take the Broadcom Stingray solution as an example. Compared with a conventional disaggregated server node, the Stingray PS1100R storage box costs half of its price and consumes up to 52.5W active power, but delivers 1.4/1.3 million IOPS along with 75.3/14.7μs unloaded latency for 4KB random read and 4KB sequential write, respectively.

In this chapter, we take such a SmartNIC-based solution as one kind of PNF-based disaggregated storage and perform systematic characterizations to understand its benefits and challenges. We then discuss the open problems of using it for system design.

6.2 Solution Overview

We use the Broadcom Stingray PS1100R platform [41] to understand SmartNIC-based disaggregated storage. We take the Intel server-based storage disaggregation solution as the baseline. Our characterization experiments use the fio [54] benchmark utility and mainly consider average/tail latency and throughput (in terms of IOPS) as the performance metric. Since the SmartNIC disaggregated node encloses a wimpy computing unit, our goal is to understand its performance impact and implications for system design. Specifically, we aim to answer the following questions (1) how much NIC computing capacity would be used to serve both the networking and storage I/O traffic? (2) When the node is fully loaded, what is the tail latency for different types of storage IO requests? (3) How many concurrent networking connections a node can handle without causing too much latency degradation? (4) What is the remaining processing headroom that one can add to each IO request without any bandwidth loss? We believe that such observations will guide how we structure the system.

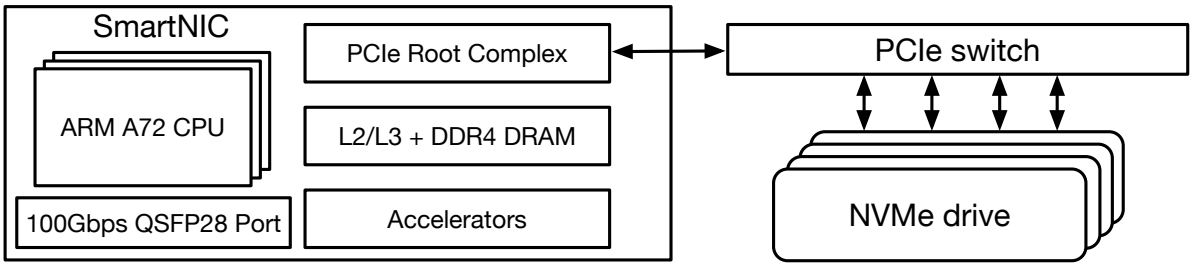


Figure 6.1: Architectural block diagram of the Stingray PS1100R SmartNIC-based disaggregated storage.

6.3 SmartNIC-based Disaggregated Storage

SmartNICs [67, 68, 69, 42, 43, 76, 27] have emerged in the data center recently, not only accelerating packet manipulations and virtual switching functionalities [146, 35], but also offloading generic distributed workloads [188, 247, 249, 141]. Generally, a SmartNIC comprises general-purpose computing substrates (such as ARM/MIPS processors or FPGAs), an array of domain-specific accelerators (e.g., crypto engine, reconfigurable match-action table, compression/hashing units), onboard memory hierarchy (including self-managed SRAM and L2/DRAM), and traffic manager (or embedded NIC switch) for packet steering. SmartNICs are low-profile PCIe devices, add incremental cost to existing data center infrastructure, and represent a promising way to expand the warehouse computing capacity.

6.3.1 Hardware Architecture

Lately, hardware vendors have combined a SmartNIC with NVMe drives as a *disaggregated computational storage node* to replace traditional server solutions for cost efficiency. Figure 6.1 presents the Broadcom Stingray solution [41], which encloses a PS1100R SmartNIC, a PCIe switch (on a standalone carrier board), and some NVMe SSDs. The SmartNIC has 8 3.0GHz ARM A72 CPU, 8GB DDR4-2400 DRAM (along with 16MB cache), FlexSPARX [40] acceleration engines,

	Idle Power (W)	Max Active Power (W)
Xeon-based	92.0	192.0
SmartNIC-based	45.5	52.5

Table 6.1: Power consumption comparison between Xeon and SmartNIC based disaggregated storage boxes. Section 6.4.1 describes the hardware configurations. We use a *Watts Up Pro* meter [96] to measure the wall power.

100Gb NetXtreme Ethernet NIC, and PCIe Gen3 root complex controllers. The PCIe switch can be configured with either 2 $\times$8 or 4 $\times$4 NVMe SSDs. A Stingray disaggregated storage box costs around 3228$ (including four Samsung 960GB DCT983 SSDs), much cheaper than a Xeon-based one with similar I/O configurations. Unsurprisingly, it also consumes lower power than a Xeon disaggregated server node. As Table 6.1 shows, the power consumption of a Stingray PS1100R based storage node is 52.5W at most, nearly one-fourth of the Xeon one.

6.3.2 Software System Stack

NVMe-over-Fabrics [78] is a remote storage protocol used for SmartNIC-based disaggregated storage. It defines a common architecture that supports the NVMe block storage protocol over a range of storage network fabrics (e.g., Ethernet, RoCE, InfiniBand, Fiber Channel). NVMe-oF extends the NVMe base specification [77] and the controller interface. A storage client (i.e., NVMe-oF initiator) first attaches to a storage server (also known as NVMe-oF target) and then issues NVMe commands to the remote controller. The NVMe-oF target comprises two main components: NVMe target core and RDMA transport. After setting up this connection with the initiator, it creates a one-to-one mapping between I/O submission queues and I/O completion queues. Each NVMe queue pair is pinned to specific CPU cores and aligned with dedicated RDMA send, receive, and completion queues. Such an end-to-end queueing design exposes the internal parallelism of NVMe SSDs to the NVMe-oF initiator.

The software stack on a SmartNIC disaggregated server node works similar to the Xeon one, except that the NVMe-oF target runs on top of the SmartNIC wimpy cores. The I/O request processing still includes three steps at the target. First, it receives an NVMe command (formatted as a capsule [78]) through a two-sided RDMA receive. The command capsule contains the NVMe submission queue entry and scatter-gather address or data (under NVMe writes). Second, the NVMe SSD controller firmware picks up commands from the submission queue, performs the execution, adds a completion notification to the completion queue, and updates the doorbell register to notify the target core. Finally, the target driver catches the completion signal, builds a response capsule (which includes the completion queue entry and data if requests are NVMe reads), and sends back to the NVMe-oF initiator. NVMe-oF is able to provide fast accesses to remote NVMe devices and consumes moderate CPU cost.

6.4 Performance Characterization and Implication

We characterize the storage I/O performance (in terms of latency and throughput) of the Stingray disaggregated node using the *fio* tool [54] and compare it with the server disaggregation case. We then summarize the implications of using SmartNIC-based disaggregated storage nodes.

6.4.1 Experiment Setup

Our experimental testbed comprises a local rack-scale RDMA-capable cluster with 7 Supermicro servers and 3 Stingray PS1100R storage nodes, connected to a 100Gbps Arista 716032-CQ switch. Each server is a 2U box, enclosing Intel Xeon Gold 5218/E5-2620 v4 processors, 96GB/128GB DDR4 memory, and a 100Gbps dual-port Mellanox ConnectX-5 MCX516A-CCAT NIC via PCIe 3.0 $\times$16. Section 6.3.1 describes the Stingray setup, and we configure it to use four NVMe SSDs. To match the same I/O capabilities, the Intel server is also configured with four NVMe SSDs when used as a disaggregated storage node. We use Samsung 960GB DCT 983 NVMe SSDs for all experiments.

We run CentOS 7.4 on Intel machines and Ubuntu 16.04 on Stingray nodes. Our experiments

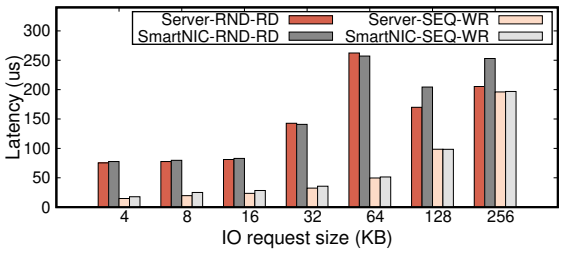


Figure 6.2: Read/write latency comparison between SmartNIC-based and server-based storage disaggregation.

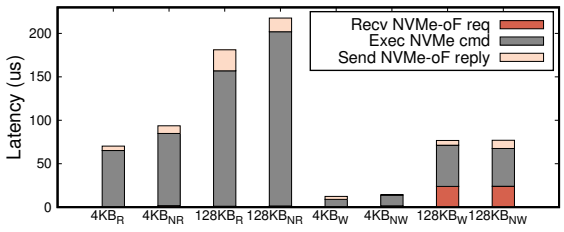


Figure 6.3: 4KB/128KB request latency breakdown at the NVMe-oF target. NR/NW = SmartNIC read/write.

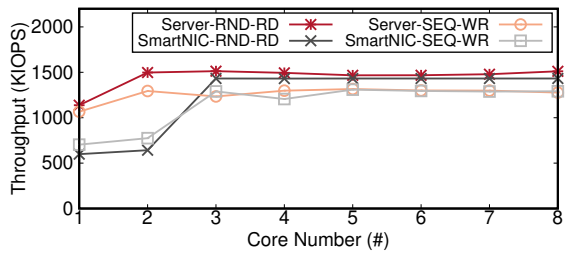


Figure 6.4: Read/write throughput as increasing the number of cores on server and SmartNIC disaggregation.

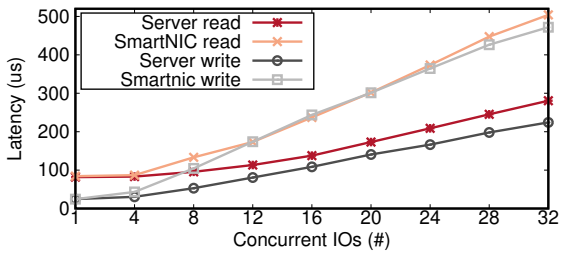


Figure 6.5: 4KB read/write request latency as increasing the number of concurrent network connections.

use the Intel SPDK [91], where its version is 19.07 and 18.04, respectively. We use the *fio* utility with the SPDK plugin and NVMe-oF SPDK target for all characterization experiments.

6.4.2 Performance Results

SmartNIC-based storage disaggregation achieves comparable read/write unloaded latency as the server disaggregation. In this experiment, we configure fio with one outstanding IO and measure the average latency while increasing the request size (Figure 6.2). In terms of serving random reads, the SmartNIC solution adds 1.0% more latency on average across five cases where

the request size is no larger than 64KB. The latency differences rise to 20.3% and 23.3% if the I/O block size is 128KB and 256KB, respectively. In the case of sequential writes, SmartNIC execution only adds 2.7μs compared with the server case on average across all cases.

We further break down the request process at the NVMe-oF target (based on the above description), and compare the server and SmartNIC cases. As Figure 6.3 shows, *NVMe command execution* is the most time-consuming part for both reads and writes, which includes writing into the submission queue, processing commands within the SSD, and handling signals from the completion queue. For a 4KB/128KB random read, it contributes to 92.4%/86.1% and 88.8%/92.2% for server and SmartNIC settings, respectively. The second expensive execution phase is data transmission, i.e., *recv NVMe-oF req* in read and *send NVMe-oF reply* in write, especially when the I/O block size is large. For example, *recv NVMe-oF req* consumes 23.9μs and 24.0μs on the server and SmartNIC, respectively.

SmartNIC-based disaggregated storage is able to saturate the storage bandwidth, but requires the use of more cores. This experiment measures the maximum 4KB random read and sequential write bandwidth as we increase the number of cores. For each fio configuration, it increases the I/O depth to maximize throughput. As depicted in Figure 6.4, the server disaggregation achieves 1513 KIOPS and 1316 KIOPS using two cores, respectively. In terms of the SmartNIC case, it is able to serve similar read and write traffic with 3 ARM cores. Under a large request size (i.e., 128KB), one core is enough to achieve the maximum bandwidth.

6.4.3 Design Implications

SmartNICs are indeed wimpy computing devices. When using it to serve 100Gbps network traffic and similar storage load, one would observe the following performance issues.

SmartNICs can only handle a small number of network connections without significant performance loss. We measure the average latency of 4KB random read and sequential write as we increase the number of concurrent workers, where each work uses its own dedicated RDMA QP. We also bound the outstanding requests of each storage stream to avoid queueing at the target

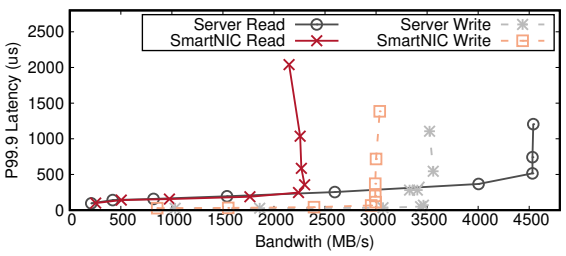


Figure 6.6: P99.9 latency v.s. bandwidth of 4KB random read and sequential write on two disaggregated cases.

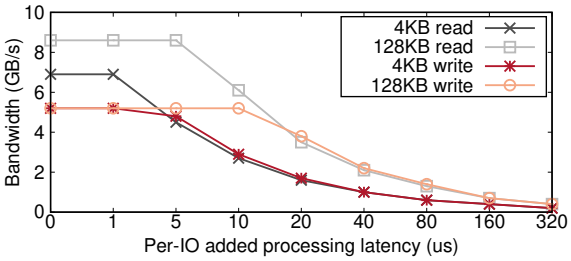


Figure 6.7: 4KB/128KB read/write bandwidth given a per-IO processing cost on the SmartNIC disaggregation.

side. When there are more than four concurrent network I/Os, because of fast polling performance on the Intel core, server disaggregation reduces the read/write request latency by 40.2%/53.2% on average across these cases when compared to the SmartNIC one. Thus, one should consider coalescing requests into fewer network connections when using SmartNIC disaggregation.

SmartNIC-based disaggregated storage achieves lower bandwidth and incurs higher tail latency, with the same number of NVMe-oF target core. This experiment measures the P99.9 latency as we increase the device load. Figure 6.6 presents the results. Compared with the SmartNIC case, server disaggregation achieves 2.1× and 1.2× higher throughput for 4KB random read and sequential write, respectively. Similarly, when offered the same load, the tail latency on SmartNIC is higher. For example, the P99.9 latency is 34/66μs on server/SmartNIC if serving ∼3000MB/s 4KB sequential writes. This indicates that one requires a careful rate-limiting mechanism for QoS control when building applications on SmartNIC-based disaggregated storage.

SmartNICs have little computing headroom for each I/O request to fully drive the storage read/write bandwidth. Figure 6.7 presents the achieved bandwidth of 4KB/128KB random read and sequential write as we add a per-I/O processing cost. We use all NIC cores in this case. The maximum tolerated latency limit is 1μs and 5μs for 4KB read and write requests, respectively. However, if the request size is 128KB, one can add at most 5μs and 10μ execution cost for reads and writes without bandwidth loss. Thus, we can offload minimal computation for each storage

I/O, and the amount of offloading depends on the storage traffic profile.

In sum, to use SmartNIC storage disaggregation in an efficient way, one should (1) keep the storage I/O data path clean and add minimal functionalities; (2) apply stringent QoS control to avoid computation contention on the NIC cores.

6.5 Related Work

6.5.1 Remote Storage I/O Stack

Researchers [177] have characterized the performance of dedicated storages using iSCSI, and proposed some optimization opportunities, such as parallel TCP protocol processing, NIC TSO offloading, jumbo frames, and interrupt affinity pinning. Reflex [178] provides a kernel-bypass dataplane to remove the NVMe storage processing stack, along with a credit-based QoS scheduler. Instead, i10 [161] is an efficient in-kernel TCP/IP remote storage stack using dedicated end-to-end I/O paths and delayed doorbell notifications. Researchers have characterized the NVMe-oF performance under server-based disaggregated storage settings [155]. Our work also uses the NVMe-oF storage protocol but in the context of the SmartNIC-based disaggregation case.

6.5.2 Disaggregated Storage Architecture

Researchers have also proposed new hardware designs to address the limitations of existing disaggregation designs. Sergey et al. [186] propose four kinds of in-rack storage disaggregation (i.e., complete, dynamic elastic, failure, and configuration disaggregation) and explores their trade-offs. Shoal [239] is a power-efficient and performant network fabric design for disaggregated racks built using fast circuit switches. LeapIO [190] provides a uniform address space across the host Intel x86 CPU and ARM-based co-processors in the runtime layer, and exposes the virtual NVMe driver to an unmodified guest VM. As a result, the ARM cores are able to run a complex cloud storage stack. We instead target emerging COTS SmartNIC disaggregated storage solutions.

6.5.3 Applications Built Using Disaggregated Storage.

There are some studies exploring how to build applications efficiently when using storage disaggregation. Decidel [208] provides the dVol abstraction for fast storage access and flexible storage management. Its runtime applies a shared-nothing architecture and allows execution whenever the hardware resource is available. Hailstore [115] addresses load skews and compaction interference of distributed LSM-based KV stores, via data placement and compaction operation scheduling. Researchers [253] have also built an elastic query execution engine on disaggregated storage to achieve flexible resource scaling and multi-tenancy.

6.6 Open Problems

Based on the system architecture of PNF-based disaggregated storage and our characterizations, we discuss a few open problems regarding how to use it to build a high-performance and cost-efficient storage application.

Remote storage management. Storage disaggregation provides a pool of persistent remote blocks. A key question on the control-plane is how to perform remote storage management. First, one needs to organize these blocks in a way that tailors to the workload requirements and simplifies the metadata management. For example, the volume abstraction would be appropriate for virtual disks, while file blobs work better for an object-store. Second, we need to figure out an efficient storage allocation mechanism that considers multiple factors, such as runtime access contention, network congestion, SSD access lifetime (or wear leveling), etc. Third, one also needs to design a naming scheme, which not only addresses node churn, but also accommodates the SSD hierarchical organization (such as controller, namespace, Zoned namespace [100]).

Performance isolation. Disaggregated storage allows multiple client tenants to access simultaneously. As a result, mitigating performance interference is a key issue of the data-plane. Along the request I/O path, there are many interference factors, such as read/write co-executing in one

SSD, concurrent storage streams sharing the NVMe-oF target core, different tenants contending for network, etc. An efficient performance isolation mechanism will help improve remote storage utilization. Essentially, such a technique should (a) be able to understand the interference among different types of I/O requests (e.g., random read v.s. sequential write); (b) enforce an end-to-end isolation path, including the network delivery and I/O execution.

Flash failures. Flash-based SSD failure is common in the data center [209, 203]. When it happens, one should minimize its impact on running applications. With a replication protocol, one would provide f replicas so that the application could have multiple choices upon reading. In terms of write requests, the protocol should guarantee the required consistency, but also take storage disaggregation into consideration. This means that it can schedule different phases of the replication protocol (e.g., commit) based on remote storage access contention. Further, the replication mechanism should ensure fast recovery by dynamically adjusting the replication granularity and various coding schemes.

Hardware diversity. In addition to SmartNIC-based solutions, there is also an Ethernet-attached disaggregated storage comprising a programmable SAN switch and micro-controller. It can convert networking requests into NVMe commands and forwards them to the storage device. Apparently, these disaggregated solutions (server-based v.s. SmartNIC-based v.s. Ethernet-attached) present different computing capabilities at the remote backend side, which will lead to different system design trade-offs. Further, there are also a range of NVMe SSDs with different features and characteristics. For example, an open-channel SSD [82] allows applications to manage its physical storage resources directly, providing multiple design choices under a disaggregated setting. Therefore, to deploy an optimal disaggregation system infrastructure, a thorough understanding of the workload characterization is necessary.

7 Conclusions and Future Work

In this chapter, we conclude the dissertation by (1) summarizing our contributions, (2) discussing limitations of current solutions, and (3) proposing several directions for future work.

7.1 Contributions

The continuing increase of data center network bandwidth and the slower improvement in CPU performance have fundamentally challenged our conventional wisdom regarding how to build high-performant and energy-efficient distributed systems that can keep up with the network speeds. Traditional server-based computing would end up using a significant amount of host CPU cores just to handle a large amount of traffic, and perform network protocol processing and application request pre-processing. As a result, there are fewer computing resources left for application processing.

Emerging programmable network fabric offers a potential solution. It is a form of domain-specific acceleration architecture that is brought into the data center by computer architects recently to accelerate packet processing tasks. A PNF hardware can be a programmable switch in the network, a SmartNIC at the networking edge, and a network accelerator along the packet communication path. These devices usually enclose wimpy computing units (like a many-core processor, domain-specific ASICs or accelerators, reconfigurable match-action tables) and diverse memory regions, which can process traffic at line rate and save states across packets. By offloading suitable computations to such a PNF device, one can reduce request serving latency, save end-host CPU cores, and enable efficient traffic control.

The main contribution of this dissertation is to bridge the gap between increasing network bandwidths and stagnating CPU computing powers by co-designing distributed systems with a

programmable network fabric. It develops principles and techniques about using in-network heterogeneous computing resources in an efficient way. Essentially, it applies approximation mechanisms, splits application logic among PNF and end-host software layers, employs new programming abstractions, and designs efficient control-/data-planes. To demonstrate the feasibility of such an approach, we build three PNF-enabled distributed systems using commodity PNF hardware and evaluate them in a real system testbed:

- **IncBricks** (Chapter 3) is an in-network caching fabric. It is a hardware-software co-designed system that supports caching in the network and provides basic computing primitives. Its hardware part combines a programmable switch (i.e., Cavium XPliant) and a network accelerator (i.e., OCTEON SoC) as a programmable middlebox. Its software part runs a distributed cache-coherent key-value store along with a topology-aware source routing protocol. IncBricks exposes programmer-friendly key-value APIs. When using it as a key-value store accelerator, our prototype lowers request latency by over 30% and doubles throughput for 1KB value. When conducting computations on cached values, IncBricks achieves 3 times more throughput and a third of the latency of client-side computation. IncBricks makes the first step towards generic in-network computations.
- **iPipe** (Chapter 4) is an actor-based framework for developing distributed applications on a SmartNIC. It is designed based on our characterization observations of a SmartNIC from four perspectives (i.e., traffic control, computing units, onboard DRAM, and host communication). The central piece of iPipe is a hybrid scheduler combining FCFS and DRR-based processor sharing that can tolerate tasks with variable execution costs and maximize NIC compute utilization. To enable flexible state migration across different execution engines, we provide the distributed memory object abstraction. Using iPipe, we build a real-time data analytics engine, a distributed transaction system, and a replicated key-value store. Based on our prototyped system with commodity SmartNICs, we show that when processing 10/25Gbps of application bandwidth, NIC-side offloading can save up to 3.1/2.2 beefy Intel cores, along with 23.0/28.0μs latency savings.
- **E3** (Chapter 5) is a microservice execution platform on SmartNIC-accelerated servers. By of-

floading suitable microservices to a SmartNIC's low-power computing units, one can improve server energy-efficiency without latency loss. E3 follows the design philosophies of the Azure Service Fabric microservice platform and extends key system components. It employs three key techniques: ECMP-based load balancing via SmartNICs to the host, network topology-aware microservice platform, and a data-plane orchestrator that can detect SmartNIC overload. Our prototyped system using Cavium LiquidIO shows that SmartNIC offload can improve cluster energy-efficiency up to 3× and cost efficiency up to 1.9× at up to 4% latency cost for common microservices, including real-time analytics, an IoT hub, and virtual network functions.

In addition to the above three systems, we also perform a preliminary investigation into PNF-based disaggregated storage (Chapter 6). Specifically, we conduct a detailed characterization study on a Broadcom Stingray PS1100R platform to understand its performance characteristics, computing capabilities when serving line-rate network/storage traffic, and computing headroom for offloading. Our gathered observations will provide some guidelines on how to build efficient applications on PNF-based disaggregated storage.

In sum, this dissertation demonstrates that co-designing distributed systems with programmable networks will be an effective approach towards building future high-performant and energy-efficient data center computing systems. We believe that our experimental observations, proposed programming models, explorations of PNF-friendly data structures and algorithms, workload-dependent control-/data-plane, real system based evaluations will provide insights on building future in-network solutions.

7.2 Limitations

This section discusses some limitations of this dissertation from different perspectives, such as applicability, experimental setup, evaluation, and system design.

Results applicability. There is a great diversity of programmable network hardware in the market. Hardware vendors are also building new products that integrate different hardware features

and further reduce costs. For example, there are already more than ten different SmartNICs in the market (shown in table 2.1). Thus, our characterization and evaluation results will not be applicable to all types of hardware. For example, the Barefoot Tofino1 [37] switch has more stateful memory than the Cavium Xpliant switch [7]. When using it to build the IncBricks count-min sketch table, it would increase the accelerator key-value cache hit accuracy and reduce latency. The LiquidIOII SmartNIC [43] used in iPipe and E3 is the second generation device along the Cavium product line. We expect its performance characteristics (such as the number of cnMIPS NIC cores to saturate the bandwidth) and power consumption will be much different compared to other SmartNICs or even the same one from the next generations. However, we do believe that our characterization methodology, analyses of hardware architecture, system design implications, and offloading trade-offs are broadly applicable.

Offloading benefits under the scale. All the evaluations in this dissertation use a small-scale testbed, which encloses a single rack of servers with a few ToR switches. When performing cross-rack experiments (such as in IncBricks), we use VLAN to configure a two-layer FatTree topology. This limits our understanding of offloading benefits under the scale. For example, under a large-scale deployment, IncBricks would see more performance degradation during the cache invalidation. Also, E3 might explore more energy-saving opportunities since there are more microservice placement opportunities. Further, a large deployment also complicates the control-plane design, such as node management, failure monitoring, and resource allocation. We leave such deployment explorations for future work.

Data-plane optimality. One of the key questions regarding PNF offloading is whether a system could maximize its computing efficiency. An ideal solution is very challenging to develop due to the dynamic networking profile, runtime workload characteristics, and non-deterministic execution behaviors. To address this, we decouple the policy from underlying mechanisms and use parameters to explore *approximately optimal* execution states. Our hybrid scheduler in iPipe combines the FCFS and DRR-based processing sharing, following this principle. Also, the data-plane orches-

trator in E3 uses thresholds to detect the SmartNIC loading factor. We believe that the proposed data-plane designs are sub-optimal. One can further improve them by adding fine-grained runtime monitoring and using them to develop new queueing/scheduling algorithms. It is also possible to use detailed architectural simulations plus packet tracing to identify an optimal schedule for each phase and compare it with the proposed solution.

Simplified transportation protocol. We apply the UDP protocol for all three PNF-based systems. This simplifies our protocol processing on the PNF hardware so that we can explore more computing headroom for application execution. We make such design decisions because more and more data center applications are using small RPC messages [213, 184] for communication, where UDP is able to satisfy workload requirements and consumes fewer CPU cycles. However, it brings us two drawbacks: one is that we need to add a reliability component in the application layer to ensure reliable message delivery; the other one is that it is unable to support streaming applications. Based on our experience using different PNF hardware, we believe that supporting a reliable transport protocol (like TCP) on the general-purpose wimpy cores of a PNF device is rather demanding. Instead, one should integrate a builtin hardware-acceleration module, and provide RDMA verbs (or active TCP message) directly to applications.

Firmware-dependent implementation. We build the runtime system directly into the PNF firmware, and assume it has abstracted the underlying hardware resources efficiently. For example, the SmartNIC firmware usually provides hardware threads, memory managers, signals/mutexes, and a set of support libraries. If such an assumption doesn't hold, the builtin runtime module will suffer form some inefficiencies. A typical example is the programmable DMA engine on a network accelerator or a SmartNIC. To check the completion of DMA request, existing firmware usually relies on a general-purpose core to poll the completion word status, consuming substantial NIC core cycles. Another related issue is system portability. Since firmware is hardware-dependent, migrating a system across different PNF devices would require substantial effort.

7.3 Future Work

In this section, we identify several research directions for future work. We try to emphasize the main challenges and possible solutions.

7.3.1 Rethinking Distributed System Abstraction

Unlike host servers with full operating system support, the software stack of a PNF device usually exposes hardware execution threads with massive parallelism, self-managed memory space, and low-level computing primitives for accelerators. As a result, a run-to-completion execution model would work better than a pipelined model. A hashtable based data structure can perform faster lookups than a tree-based one. Instead of enforcing concurrent accesses on a general-purpose core, one can use a hardware-based traffic control module to schedule different in-flight requests and avoid contention. These examples indicate that to enable better offloading, one should rethink the distribution system abstractions. We have conducted an early investigation about the replicated state machine – fRSM [196]. Its key idea is that instead of representing the whole shard as a separate replicated state machine, it presents each key-value pair as a separate RSM. As a result, one can streamline the consensus algorithm with key-value processing. This enables better use of computing and storage parallelism.

7.3.2 Multi-tenancy

In today's emerging deployments and use cases, a PNF device will be used by multiple tenants sharing the network. The results issues of multi-tenancy and the needed PNF support have received little attention from the community. Multi-tenancy brings to fore both (a) resource isolation to offer tenants minimal expected performance and also to minimize cross-tenant interference, and (b) a variety of security-related issues, such as confidentiality and integrity of tenant computation, side-channel prevention, prevention of resource exhaustion attacks, etc. We focus on discussing performance isolation here and talk about the security part below. To provide such support, one

should understand both workload characteristics and hardware capabilities. For example, in terms of a programmable switch, it is very challenging to do time multiplexing on a match-action table. Thus, one should mainly apply the space partitioning technique on the control plane, and build a packet steering module to direct the traffic. For network accelerators or SmartNIC, both time and space multiplexing would be possible. The main question will be how to architect a light-weight mechanism to enable sharing different resources (such as NIC cores, caches, interconnects, accelerators, or co-processors) for multiple application classes. We believe an efficient system will combine multiple techniques, such as flexible priority, compute/traffic isolation, DRF-based resource allocation [150], etc.

7.3.3 A Unified Programming Model

Our three PNF-enabled systems apply different programming abstractions to satisfy different workload requirements. Specifically, IncBricks exposes the key-value data store APIs that are fit for NoSQL based workloads. iPipe instead uses the actor-based programming model, which targets communication-intensive message-driven distributed applications. In E3, given that the microservice execution graph is predetermined, we use a data-flow programming model. Such ad-hoc design helps improve the offloading efficiency, but limits the system applicability. When switching to a new PNF device, one has to start from scratch and go through tedious efforts. Thus, it will be useful to build a unified programming model that satisfies all cases.

This is a non-trivial task since programming models are tied to the underlying computing resources. The hardware architecture of various PNF devices is different. For example, programmable switches have match-action tables, SRAM/TCAM, traffic manager for monitoring flow statistics. SmartNICs have wimpy many/multi-cores, programmable DMA engines, and domain accelerators. Networking accelerators are computation-specific but do require considerations as to how to feed data and catch completion signals. One possible solution is to start with a mature programming model, and see if one can extend to different hardware by building a compiler backend. For example, today's P4 language [116] provides a packet-level programming abstraction, which

aims at per-packet manipulations. One might extend it with a flow-level abstraction to broaden the application domain and then build various architecture-dependent backends.

7.3.4 In-network Security

One of the major concerns preventing cloud providers from deploying programmable networks is the capability to access raw packets and save sensitive data along the data-plane. As a result, enforcing in-network security attracts great attention. In order to do this, one has to consider three issues. The first one is the security preserving granularity. It could be at the application-session level, transportation flow level, or even between two end-host tunnels. Different granularity levels dictate how much state is to be maintained. Second, one should consider as to which networking layer should enforce the security mechanism. If doing it in the IP layer for individual packets, this is nearly stateless computation. If doing it in the transport and application layers, one has to pay the cost of rebuilding messages within the network. Given the limited computing and memory capabilities, this would add considerable overhead. Finally, due to the hardware limitations, one cannot build a homogeneous security mechanism across the communication path. As SmartNICs provide more computing power than a programmable switch, one can apply a more complicated security protocol design. We believe that a robust in-network security solution will decouple a monolithic mechanism into multiple small pieces and apply them to different entities along the data-plane.

7.3.5 Traffic control

Today's data center network has a mixed number of big and small flows. It is very challenging to achieve low latency under high-bandwidth utilization. We can also use programmable networks to enable better traffic control. With such in-network visibility, one can design more efficient packet scheduling, congestion control, and flow control mechanisms. For example, we have leveraged configurable per-packet processing and the ability to maintain mutable states of a programmable switch to provide an approximate fair queueing abstraction [235]. Similarly, we also show that it

is possible to expose a programmable calendar queue using either data-plane primitives or control-plane commands to dynamically modify the schedule status of queues [236]. Another example is to revisit the credit-based flow control protocol [182], which was proposed for ATM networks originally. It has been widely used in InfiniBand networks for high-performance computing clusters. With data-plane programmability across the packet communication path, one can build a better credit allocation, credit exchange, and credit overcommit schemes.

7.3.6 In-network Data Persistence

Commodity PNF devices enclose no persistent storage. When failures happen, all in-network intermediate computing state will be lost without logging, and applications have to reissue the commands. This adds significant overhead, especially for streaming workloads. One possible solution is to apply the typical incremental computation technique, such as delta checkpointing. To realize such a solution, there are a few issues, such as where persistent storage is located (host server or Etherent-attached SSD), what system abstractions to use (files or key-value store), and how to manage such ephemeral persistent data store. Driven by today's non-volatile memory techniques [24], it is also possible to equip some NVM on the PNF device directly. The Mellanox BlueField2 [68] will provide such support. We believe such hardware features will enable better incremental in-network computation.

7.4 Final Remarks

In this dissertation, we have presented the first solution that co-designs distributed systems with emerging programmable network fabric, including programmable switch, SmartNIC, and network accelerator. To demonstrate the feasibility and benefits of our solutions, we have designed and implemented three PNF-enabled systems: an in-network caching fabric (IncBricks), a SmartNIC offloading framework for distributed applications (iPipe), and a Microservice execution platform on SmartNIC-accelerated servers (E3). We also performed a characterization study about a PNF-based disaggregated storage and discussed potential benefits/challenges. Given the continuing

increase of data center network bandwidths and the slower improvement in CPU performance, we believe that co-designing distributed systems with programmable networks is an effective approach to building future high-performant and energy-efficient data center computing systems.

Bibliography

[1] ECMP routing protocol. https://en.wikipedia.org/wiki/Equal-cost_multi-path_routing.

[2] The New Need for Speed in the Datacenter Network. http://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-734328.pdfdf, 2015.

[3] Cisco Global Cloud Index: Forecast and Methodology, 2015-2020. http://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf, 2016.

[4] Google SyntaxNet. https://research.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html, 2016.

[5] Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning/, 2016.

[6] Trends in Server Efficiency and Power Usage in Data Centers. https://www.spec.org/events/beijing2016/slides/015-Trends_in_Server_Efficiency_and_Power_Usage_in_Data_Centers%20-%20Sanjay%20Sharma.pdf, 2016.

[7] XPliant Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html, 2016.

[8] The Apache Cassandra Database. http://cassandra.apache.org, 2017.

[9] Disaggregated or Hyperconverged, What Storage will Win the Enterprise? https://www.nextplatform.com/2017/12/04/disaggregated-hyperconverged-storage-will-win-enterprise/, 2017.

[10] Hardware-Accelerated Networks at Scale in the Cloud. https://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf, 2017.

[11] IEEE Std 802.3bs-2017 Standard. https://standards.ieee.org/standard/802_3bs-2017.html, 2017.

[12] LevelDB Key-Value Store. http://leveldb.org, 2017.

[13] Amazon Lambda Serverless Computing Platform. https://aws.amazon.com/lambda/, 2018.

[14] Free Your Flash And Disaggregate. https://www.lightbitslabs.com/blog/free-your-flash-and-disaggregate/, 2018.

[15] Google App Engine. https://cloud.google.com/appengine/, 2018.

[16] Huawei IN550 SmartNIC. https://e.huawei.com/us/news/it/201810171443, 2018.

[17] In-memory SQL database. https://www.memsql.com, 2018.

[18] Industry Outlook: NVMe and NVMe-oF For Storage. https://www.iol.unh.edu/news/2018/02/08/industry-outlook-nvme-and-nvme-storage, 2018.

[19] NIC Teaming. https://docs.microsoft.com/en-us/windows-server/networking/technologies/nic-teaming/nic-teaming, 2018.

[20] Nirmata Platform. https://www.nirmata.com/, 2018.

[21] 400GbE Takes Ethernet in a New Era. https://www.accton.com/Technology-Brief/the-new-world-of-400-gbps-ethernet/, 2019.

[22] Marvell 88SN2400. https://www.marvell.com/documents/52e9puzva8ze3hl2zjc0/, 2019.

[23] Private conversation with smartnic vendors. unpublished, 2019.

[24] 3D XPoint. https://en.wikipedia.org/wiki/3D_XPoint, 2020.

[25] Actor model. https://en.wikipedia.org/wiki/Actor_model, 2020.

[26] Aho-Corasick Algorithm. https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm, 2020.

[27] Alpha Data ADM-PCIE-9V3 SmartNIC. https://www.alpha-data.com/dcp/products.php?product=adm-pcie-9v3, 2020.

[28] Amazon. https://www.amazon.com, 2020.

[29] Amazon Data Pipeline. https://aws.amazon.com/datapipeline/, 2020.

[30] AMD HSA. https://www.amd.com/en-us/innovations/software-technologies/hsa, 2020.

[31] Apache Kafka: A Distributed Streaming Platform. https://kafka.apache.org, 2020.

[32] Api gateway. http://microservices.io/patterns/apigateway.html, 2020.

[33] Arista 7150 Series Datasheet. https://www.arista.com/assets/data/pdf/Datasheets/7150S_Datasheet.pdf, 2020.

[34] Arm Cortex-A72 Multi-core Processor. https://developer.arm.com/products/processors/cortex-a/cortex-a72, 2020.

[35] AWS Nitro System. https://aws.amazon.com/ec2/nitro/, 2020.

[36] Azure IoT hub. https://azure.microsoft.com/en-us/services/iot-hub/, 2020.

[37] Barefoot Tofino Switch. https://www.barefootnetworks.com/products/brief-tofino/, 2020.

[38] Barefoot Tofino2 Switch. https://www.barefootnetworks.com/products/brief-tofino-2/, 2020.

[39] Border Gateway Protocol. https://en.wikipedia.org/wiki/Border_Gateway_Protocol, 2020.

[40] Broadcom FlexSPARX acceleration subsystem. https://docs.broadcom.com/doc/1211168571391, 2020.

[41] Broadcom Stingray PS1100R SmartNIC. https://www.broadcom.com/products/storage/ethernet-storage-adapters-ics/ps1100r, 2020.

[42] Broadcom Stingray SmartNIC. https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225, 2020.

[43] Cavium LiquidIO SmartNICs. https://cavium.com/pdfFiles/LiquidIO_II_CN78XX_Product_Brief-Rev1.pdf, 2020.

[44] Cavium OCTEON Multi-core Processor. http://www.cavium.com/octeon-mips64.html, 2020.

[45] cnMIPS Microarchitecture. https://en.wikichip.org/wiki/cavium/microarchitectures/cnmips, 2020.

[46] Collaborative filtering. https://en.wikipedia.org/wiki/Collaborative_filtering, 2020.

[47] Command-line interface. https://en.wikipedia.org/wiki/Command-line_interface, 2020.

[48] Dataflow programming model. https://en.wikipedia.org/wiki/Dataflow, 2020.

[49] DynamIQ, 2020. https://developer.arm.com/technologies/dynamiq.

[50] eBay. https://www.ebay.com, 2020.

[51] Ema. https://en.wikipedia.org/wiki/Moving_average, 2020.

[52] Enhanced Interior Gateway Routing Protocol. https://en.wikipedia.org/wiki/Enhanced_Interior_Gateway_Routing_Protocol, 2020.

[53] Facebook. https://www.facebook.com, 2020.

[54] Flexible I/O Tester. https://github.com/axboe/fio, 2020.

[55] Google Cloud Dataflow. https://cloud.google.com/dataflow/, 2020.

[56] Google Docs. https://www.google.com/docs/about/, 2020.

[57] Honeywell Case Study. https://blogs.msdn.microsoft.com/azureservicefabric/2018/03/20/service-fabric-customer-profile-honeywell/, 2020.

[58] HPE ProLiant m800 Server Cartridge. https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04500667&sp4ts.oid=6532018, 2020.

[59] Intel DPDK. http://dpdk.org, 2020.

[60] Intel FelxPipe. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf, 2020.

[61] Intel Flow Director. https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director, 2020.

[62] Intel Xeon Phi Coprocessor 7120A. https://ark.intel.com/products/80555/Intel-Xeon-Phi-Coprocessor-7120A-16GB-1238-GHz-61-core, 2020.

[63] Intermediate System to Intermediate System Protocol. https://en.wikipedia.org/wiki/IS-IS, 2020.

[64] k-nearest neighbors algorithm. https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm, 2020.

[65] LinkedIn. https://twitter.com/?lang=en, 2020.

[66] Lyft. https://www.lyft.com, 2020.

[67] Mellanox BlueField-1 SmartNIC. http://www.mellanox.com/related-docs/npu-multicore-processors/PB_BlueField_Ref_Platform.pdf, 2020.

[68] Mellanox BlueField-2 SmartNIC. https://www.mellanox.com/files/doc-2020/pb-bluefield-2-smart-nic-eth.pdf, 2020.

[69] Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.mellanox.com/sites/default/files/doc-2020/pb-innova-2-flex.pdf, 2020.

[70] Mesh Systems. http://www.mesh-systems.com, 2020.

[71] Microprocessor Trend Data. https://github.com/karlrupp/microprocessor-trend-data, 2020.

[72] Microsoft Data Factory. https://azure.microsoft.com/en-us/services/data-factory/, 2020.

[73] Microsoft Office 365. https://www.office.com, 2020.

[74] MTP/MPO Cabling System: A Panacea for Data Centers. https://community.fs.com/blog/things-you-should-know-about-mpomtp-fiber-cable.html, 2020.

[75] Netflix. https://www.netflix.com, 2020.

[76] Netronome agilio smartnic. https://www.netronome.com/products/agilio-cx/, 2020.

[77] NVM Express Base Specification. https://nvmexpress.org/developers/nvme-specification/, 2020.

[78] NVM Express over Fabrics Specification. https://nvmexpress.org/developers/nvme-of-specification/, 2020.

[79] OCTEON Development Kits. http://www.cavium.com/octeon_software_develop_kit.html, 2020.

[80] Octeon Development Kits. http://www.cavium.com/octeon_software_develop_kit.html, 2020.

[81] Okapi BM25. https://en.wikipedia.org/wiki/Okapi_BM25, 2020.

[82] Open-channel SSD. https://en.wikipedia.org/wiki/Open-channel_SSD, 2020.

[83] Open Shortest Path First. https://en.wikipedia.org/wiki/Open_Shortest_Path_First, 2020.

[84] OpenConfig protocol. https://www.openconfig.net/, 2020.

[85] Organizationally unique identifier. https://en.wikipedia.org/wiki/Organizationally_unique_identifier, 2020.

[86] Programming Model in the Azure Service Fabric. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-choose-framework, 2020.

[87] Receiver Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, 2020.

[88] Representational State Transfer Architecture. https://en.wikipedia.org/wiki/Representational_state_transfer, 2020.

[89] SNMP protocol. https://en.wikipedia.org/wiki/Simple_Network_Management_Protocol, 2020.

[90] Spotify. https://www.spotify.com/us/, 2020.

[91] The Intel Storage Performance Development Kit (SPDK). https://spdk.io/, 2020.

[92] TikTok. https://www.tiktok.com/en/, 2020.

[93] Twitter. https://twitter.com/?lang=en, 2020.

[94] Uber. https://www.uber.com, 2020.

[95] Uthash Hashtable. https://troydhanson.github.io/uthash/, 2020.

[96] Watts up? PRO. http://www.idlboise.com/sites/default/files/WattsUp_Pro_ES.pdf, 2020.

[97] WebGUI interface. https://en.wikipedia.org/wiki/WebGUI, 2020.

[98] What is Composable Disaggregated Infrastructure? https://blog.westerndigital.com/what-is-composable-disaggregated-infrastructure/, 2020.

[99] Zipf's law. https://en.wikipedia.org/wiki/Zipf%27s_law, 2020.

[100] Zoned Namespaces (ZNS) SSDs. https://zonedstorage.io/introduction/zns/, 2020.

[101] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[102] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[103] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. 1986.

[104] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010.

[105] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[106] Venkat Anantharam. Scheduling strategies and long-range dependence. *Queueing systems*, 33(1-3):73–89, 1999.

[107] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[108] Mina Tahmasbi Arashloo, Monia Ghobadi, Jennifer Rexford, and David Walker. Hotcocoa: Hardware congestion control abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017.

[109] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[110] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[111] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011.

[112] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron. Pelican: A Building Block for Exascale Cold Data Storage. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[113] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The Datacenter as a Computer: Designing Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2018.

[114] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[115] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-Based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[116] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[117] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[118] Broadcom. The TruFlow Flow processing engine. `https://www.broadcom.com/applications/data-center/cloud-scale-networking`, 2020.

[119] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[120] Adrian Caulfield, Paolo Costa, and Monia Ghobadi. Beyond SmartNICs: Towards a fully programmable cloud. In *IEEE International Conference on High Performance Switching and Routing, ser. HPSR*, volume 18, 2018.

[121] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016.

[122] Luis Ceze, Mark D Hill, and Thomas F Wenisch. Arch2030: A vision of computer architecture research over the next 15 years. *arXiv preprint arXiv:1612.03182*, 2016.

[123] Geoffrey Challen and Mark Hempstead. The Case for Power-agile Computing. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[124] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, 2006.

[125] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[126] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[127] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, 2005.

[128] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[129] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[130] P Costa, A Donnelly, G O'shea, and A Rowstron. CamCube: a key-based data center. *Microsoft Research*, 2010.

[131] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[132] F Cristian et al. Coordinator log transaction execution protocol, 1990.

[133] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, 2016.

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

[135] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), January 2008.

[136] Christina Delimitrou. The Hardware-Software Implications of Microservices and How Big Data Can Help. 2018.

[137] Docker. Docker Container Migration. https://docs.docker.com/docker-cloud/migration/, 2018.

[138] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[139] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*, 2015.

[140] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[141] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[142] A. Shehabi et al. United States Data Center Energy Usage Report. Technical report, Lawrence Berkeley National Laboratory, 2016.

[143] Dolphin Express. Remote Peer to Peer made easy. https://www.dolphinics.com/download/WHITEPAPERS/Dolphin_Express_IX_Peer_to_Peer_whitepaper.pdf, 2020.

[144] Kevin Fall, Gianluca Iannaccone, Maziar Manesh, Sylvia Ratnasamy, Katerina Argyraki, Mihai Dobrescu, and Norbert Egi. Routebricks: Enabling general purpose network infrastructure. *SIGOPS Oper. Syst. Rev.*, 45(1), February 2011.

[145] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[146] 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 *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[147] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124), August 2004.

[148] Y. Gan and C. Delimitrou. The Architectural Implications of Cloud Microservices. *IEEE Computer Architecture Letters*, 17(2):155–158, 2018.

[149] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[150] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[151] Alex Goldhammer and John Ayer Jr. Understanding performance of pci express systems. *Xilinx WP350, Sept*, 4, 2008.

[152] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, 2009.

[153] Site Selection Group. Power in the Data Center and its Cost Across the U.S. `https://info.siteselectiongroup.com/blog/power-in-the-data-center-and-its-costs-across-the-united-states`, 2017.

[154] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1998.

[155] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017.

[156] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010.

[157] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[158] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

[159] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, 1973.

[160] Thomas Hunter II. *Advanced Microservices: A Hands-on Approach to Microservice Infrastructure and Tooling*. 2017.

[161] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[162] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[163] Rajkumar Jalan, Swaminathan Sankar, and Gurudeep Kamat. Distributed system to determine a server's health, 2018. US Patent 9,906,422.

[164] Brian Jeff. Ten Things to Know About big. LITTLE. *ARM Holdings*, 2013.

[165] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[166] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[167] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[168] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, et al. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[169] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306, 2014.

[170] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[171] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference*, 2016.

[172] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-scale Computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015.

[173] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[174] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[175] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.

[176] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (network balancing act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[177] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[178] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[179] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[180] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

[181] Rakesh Kumar, Dean M Tullsen, Parthasarathy Ranganathan, Norman P Jouppi, and Keith I Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, 2004.

[182] NT Kung and Robert Morris. Credit-based flow control for atm networks. *IEEE network*, 9(2):40–48, 1995.

[183] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[184] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[185] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. UNO: uniflying host and smart NIC offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[186] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanaël Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale Disaggregated Storage. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, 2017.

[187] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[188] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[189] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[190] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[191] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[192] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[193] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[194] Felix Xiaozhu Lin and Xu Liu. Memif: Towards Programming Heterogeneous Memory Asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[195] Jianxiao Liu, Zonglin Tian, Panbiao Liu, Jiawei Jiang, and Zhao Li. An approach of semantic web service classification based on Naive Bayes. In *Services Computing, 2016 IEEE International Conference on*, 2016.

[196] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-grained replicated state machines for a cluster storage system. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[197] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Computer Architecture, 2014 ACM/IEEE 41st International Symposium on*, 2014.

[198] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. Asic clouds: Specializing the datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[199] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NetAgg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, 2014.

[200] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[201] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2), March 2008.

[202] Mellanox. Accelerated Switch and Packet Processing. `http://www.mellanox.com/page/asap2?mtag=asap2`, 2020.

[203] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):177–190, 2015.

[204] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, 2002.

[205] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. Solros: A Data-centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[206] Jelena Mirkovic and Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.

[207] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, 2013.

[208] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[209] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. Ssd failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, 2016.

[210] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference*, 2015.

[211] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.

[212] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, 2009.

[213] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[214] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[215] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[216] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3Sigma: Distribution-based Cluster Scheduling for Runtime Uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[217] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[218] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[219] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[220] Lucian Popa, Norbert Egi, Sylvia Ratnasamy, and Ion Stoica. Building extensible networks with rule-based forwarding. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[221] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[222] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[223] Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. Decoupling the Control Plane from Program Control Flow for Flexibility and Performance in Cloud Computing. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[224] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011.

[225] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. JouleSort: A Balanced Energy-efficiency Benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, 2007.

[226] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[227] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

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

[229] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems*, 2010.

[230] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017.

[231] Felix Scholkmann, Jens Boss, and Martin Wolf. An efficient algorithm for automatic peak detection in noisy periodic and quasi-periodic signals. *Algorithms*, 5(4):588–603, 2012.

[232] Linus Schrage. Letter to the editor—a proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[233] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partridge. Smart packets: Applying active networks to network management. *ACM Trans. Comput. Syst.*, 18(1), February 2000.

[234] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3), May 2006.

[235] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[236] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020.

[237] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. HASS: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[238] Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 4(3):375–385, 1996.

[239] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[240] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *European Conference on Object-Oriented Programming*.

[241] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2001.

[242] Alexander L Stolyar and Kavita Ramanan. Largest weighted delay first scheduling: Large deviations and optimality. *Annals of Applied Probability*, pages 1–48, 2001.

[243] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[244] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[245] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. In *Proceedings DARPA Active Networks Conference and Exposition*, 2002.

[246] David L Tennenhouse and David J Wetherall. Towards an active network architecture. *ACM SIGCOMM Computer Communication Review*, 26(2):5–17, 1996.

[247] **Ming Liu**, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*, 2019.

[248] **Ming Liu**, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 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 (ASPLOS '17)*, Xi'an, China, 2017.

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

[250] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014.

[251] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*, 2016.

[252] Leendert van Doorn. Microsoft's datacenters. In *Proceedings of the 1st Workshop on Hot Topics in Data Centers*, 2016.

[253] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[254] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[255] David Wetherall. Active Network Vision and Reality: Lessons from a Capsule-Based System. In *Proceedings DARPA Active Networks Conference and Exposition*, 2002.

[256] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[257] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[258] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[259] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[260] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-NET: Effective GPU Sharing in NFV Systems. In *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.