Accelerating Networked Systems with Programmable and Tightly-Coupled NICs

Henry Schuh

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

Doctor of Philosophy

University of Washington
2023

Reading Committee:
Arvind Krishnamurthy, Chair
Thomas Anderson, Chair
Simon Peter

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Accelerating Networked Systems with Programmable and Tightly-Coupled NICs

Henry Schuh

Co-Chairs of the Supervisory Committee:
Arvind Krishnamurthy
Computer Science & Engineering

Thomas Anderson
Computer Science & Engineering

As datacenter networks evolve to support higher bandwidths and lower latency, delivering this performance to applications is an increasing challenge. Software packet-processing overheads and high-cost hardware data paths are main contributors to this challenge. Offloading packet-processing work from host CPU cores to the network interface controller (NIC) is one compelling solution. While existing hardware offloads are limited in flexibility, SmartNIC devices with an integrated system-on-chip (SoC) are fully programmable. However, achieving efficiency improvements using SmartNICs depends on specific architectural features, and efficient interfaces which only some devices provide. Additionally, SoC resources are inherently limited, and host-NIC communication still relies on high-overhead PCI Express transfers. This suggests that distributed systems must be carefully integrated with SmartNIC resources and interfaces, in order to benefit from these devices. Looking forward, new standards for cache-coherent interconnects have the potential to simplify host-NIC communication. These interconnects bring devices into the CPU's coherence domain, promising high performance and simpler sharing semantics.

This thesis makes the argument for tight integration of datacenter systems and NICs at multiple levels, with the goal of increasing performance and efficiency. First, we conduct a measurement characterization of the SoC SmartNIC design space, with the goal of understanding opportunities for performance and efficiency gains. Second, we present Xenic, a case study of integrating SmartNIC resources with the design of a distributed transaction processing system. By considering the NIC resources and interfaces in its system design, Xenic accelerates performance and minimizes the use of high-cost data paths. Finally, we present CC-NIC, a NIC interface design optimized for emerging cache-coherent interconnects. CC-NIC demonstrates that integrating the NIC interface into the CPU's coherence domain offers the potential to improve throughput, latency, and CPU packet-handling efficiency.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables v

Chapter 1: Introduction 1
1.1 Thesis Statement and Contributions 3
1.2 Published Works 5
1.3 Thesis Outline 5

Chapter 2: Background 6
2.1 Reducing Software Overheads 6
2.2 Optimizing Hardware Communication Paths 7
2.3 Pushing Packet-Handling Functionality to the NIC 8
2.4 SmartNICs 9

Chapter 3: SmartNIC Bake-off: Understanding Performance Opportunities 10
3.1 Survey of SmartNIC Platforms 11
3.2 Experimental Setup 12
3.3 SmartNIC Packet-Processing Performance 16
3.4 Deep Dive into SoC Datapaths 22
3.5 Conclusions 27

Chapter 4: Xenic: SmartNIC-Accelerated Distributed Transactions 28
4.1 Background & Related Work 30
4.2 SmartNIC Performance Analysis 34
4.3 Design 39
4.4 Evaluation 54

4.5 Conclusion . . . 63

Chapter 5: CC-NIC: a Cache-Coherent Interface to the NIC . . . 64
5.1 Dissecting the PCIe Host-NIC Interface . . . 66
5.2 System Design for Coherent Interconnects . . . 73
5.3 CC-NIC Implementation . . . 85
5.4 Evaluation . . . 86
5.5 Discussion . . . 101
5.6 Related Work . . . 102
5.7 Conclusion . . . 103

Chapter 6: Conclusion . . . 104
6.1 Future Work . . . 105

Bibliography . . . 107

LIST OF FIGURES

Figure Number Page

1.1 Data paths between host server, NIC, and SmartNIC SoC. 4

3.1 Coremark benchmark performance for SmartNIC Arm SoCs. 13
3.2 SmartNIC on-board memory throughput. 14
3.3 On-board memory access latency. 15
3.4 Latency of SmartNIC datapaths 17
3.5 Per-thread packet rate with 64B packets and a wire loopback. 19
3.6 Arm-initiated DMA read and write throughput. 21
3.7 Cycles per packet TX for Bluefield and Octeon. 23
3.8 Throughput and cycles per descriptor submission, comparing Bluefield and Octeon. 24
3.9 Throughput of the Octeon hardware buffer pool and DPDK software pool. 26

4.1 Roundtrip latency for LiquidIO and CX5 remote access operations. 35
4.2 Remote memory write throughput with and without batching enabled. 36
4.3 DMA engine throughput and latency. 38
4.4 Diagram of Xenic data store. 41
4.5 Xenic design overview, showing one server. 47
4.6 Commit messages for a transaction writing to local and remote shards. 51
4.7 Throughput and latency for TPC-C, Retwis, and Smallbank benchmarks. 56
4.8 Throughput and latency impacts of Xenic design features. 61

5.1 PCIe and UPI transfer paths. 68
5.2 Single-threaded write throughput for WC MMIO, WC DRAM, and WB DRAM. . . 70
5.3 MMIO store latency versus iteration count. 71
5.4 Overview of CC-NIC design features. 73
5.5 TX path accesses for the Intel E810 and CC-NIC. 74
5.6 The core CC-NIC data plane interface. 74
5.7 Signaling communication with registers versus inlined signals. 77

5.8 Local and cross-UPI access latency for Sapphire Rapids and Ice Lake hosts. 78
5.9 UPI pingpong latency with varying memory placement. 79
5.10 Stream transfer throughput comparing caching and nontemporal accesses. 81
5.11 CC-NIC and PCIe NIC buffer management designs. 84
5.12 Throughput and latency overview, comparing CC-NIC and PCIe NIC interfaces. . 89
5.13 Throughput-latency curves for CC-NIC and CX6, with ICX host. 90
5.14 Throughput-latency curves for CC-NIC and with SPR host. 90
5.15 Performance impacts of inlined signaling and descriptor layout features. 92
5.16 Performance impacts of buffer management features. 93
5.17 Throughput effect of TX and RX batching factors. 94
5.18 Interconnect accesses per TX-RX loopback. 95
5.19 CC-NIC performance with NIC and host threads deployed on the same CPU. . . . 97
5.20 Key-value store throughput versus thread count, comparing CC-NIC and CX6. . . 98
5.21 Impact of hardware prefetching on packet throughput. 100
5.22 Performance with reduced UPI throughput and latency. 100

LIST OF TABLES

Table Number Page

3.1 Set of measured SmartNIC devices, and their hardware specifications. 11
3.2 Sequence of TX and RX operations for Bluefield and Octeon. 22
4.1 Coremark benchmark results for the LiqudIO Arm and host Xeon CPUs. 38
4.2 Objects accessed and network roundtrips per hash lookup. 45
4.3 Normalized thread count, for Xenic, DrTM+H, and FaSST. 60
5.1 Comparison of PCIe, CXL, and UPI bandwidth. 85
5.2 Peak throughput and core count for KV Store and TCP Echo RPC applications. . . 97

GLOSSARY

- **CLX**: Intel Cascade Lake server CPU platform.
- **CXL**: The Compute Express Link interconnect standard, encompassing CXL.cache, a protocol for cache-coherent host-device interaction.
- **Descriptor**: One work unit submitted to the NIC, representing a packet transmit or receive operation.
- **Doorbell**: The host's signal to the NIC, indicating the presence of new packets to transmit.
- **ICX**: Intel Ice Lake server CPU platform.
- **Loopback**: A network configuration in which packets are routed from transmit to receive queues on a single NIC, either via internal NIC settings or a wire connection between ports.
- **NIC**: Network interface controller.
- **OCC**: Optimistic concurrency control, a protocol providing distributed transaction semantics.
- **One-sided RDMA**: Remote direct memory access operations allowing remote memory reads and writes to be handled by the target server's NIC (see *RDMA*).
- **Packet buffer**: A pre-allocated memory region which contains packet data sent or received by the NIC.

- **PCIe (PCI Express)**: Peripheral Component Interconnect Express, the host-device peripheral interconnect utilized by today's NICs and SmartNICs.
- **RDMA**: Remote Direct Memory Access, a NIC hardware functionality which allows the NIC to serve memory access requests without CPU involvement.
- **RPC**: Remote Procedure Call.
- **RX**: Receive.
- **SoC**: System on Chip.
- **SPR**: Intel Sapphire Rapids server CPU platform.
- **TCP**: Transmission Control Protocol, a standard reliable transport network communication protocol.
- **Two-sided RDMA**: A send-receive message passing interface, with network transport and packetization performed by the NIC (see *RDMA*).
- **TX**: Transmit.
- **UPI**: Ultra Path Interconnect, a cache-coherent interconnect typically applied between Intel CPUs in multi-socket servers.

ACKNOWLEDGMENTS

Thank you to my advisors, Arvind and Tom. Even before starting at UW, Arvind and Tom got me excited about systems research. Early on, Tom motivated me to explore new research topics and seek out collaboration beyond UW. Arvind was a consistent source of positivity throughout my work. I am extremely grateful for the level of attention and support I received from Arvind, despite his crazy calendar and wide-ranging projects. I sincerely enjoyed working with Arvind and Tom over the past five years.

Thank you to Simon, Waleed, and the rest of the Assise group. This project introduced me to both graduate school and systems research, not to mention the conference review process and effective remote collaboration (which turned out to be hugely important in 2020).

Thank you to Brent, Samira, David, Hank, Kanthi, and Luigi. Joining SRG ended up being a pivotal experience both in graduate school, and in defining my future after graduation.

Thank you to the entire Systems Lab, who made graduate school enjoyable for me. In particular, Samantha, Jialin, Katie, and Priyal, who were consistently around in the lab to hang out and commiserate. I am extremely grateful to have been part of a fun and supportive lab group.

Finally, thanks to my parents Amy and Richard, to Adrian, and to my friends. Vinitha, thank you for helping me stay sane in graduate school. Ari and Sol, thank you for your friendship over the past thirteen years. Brittany, thank you for your support ever since our Geometry independent study in middle school!

Chapter 1

INTRODUCTION

Datacenter systems are evolving rapidly, to meet the growing demands of cloud workloads. At each datacenter server, network interface controller (NIC) bandwidths are increasing, doubling Ethernet link rates with each new NIC hardware generation [68, 67, 64, 63, 67, 65]. Switch latencies are decreasing, with the potential for sub-microsecond communication between servers within a datacenter rack. However, delivering this level of performance to applications is a growing challenge. Server CPU core performance has not kept up with network link rates, and communication between the server CPU and NIC has become the largest factor contributing to network latency within a rack [77]. These trends suggest a need to revisit datacenter network communication in order to reduce costs at the end-host.

A wide range of existing efforts aims to reduce datacenter packet-processing costs. These include optimizations to hardware data paths, such as allowing the NIC to directly populate CPU caches [36]. Other optimizations target host-NIC metadata, applying alternate data structure layouts for low-latency, low-throughput scenarios [68, 67]. Packet-processing costs may be further decreased, by offloading tasks traditionally performed by the host CPU to the NIC itself: for instance, splitting large buffers into TCP segments and routing flows to specific queues. A significant development to this end is remote direct memory access (RDMA) [68, 70], a NIC functionality which exposes remote read, write and message-passing primitives to the application. All packet handling is performed in NIC hardware, and at the target of a remote read or write, the NIC itself executes the memory access without CPU involvement.

While these offloads and optimizations make progress towards packet-processing cost reduction, they are limited in scope and flexibility. Optimizations to host-NIC data paths are fundamentally constrained by the underlying PCI Express (PCIe) interconnect. While PCIe bandwidth

has increased alongside network bandwidth, the protocol's host and device access mechanisms have not evolved, and latency remains high. Meanwhile, added offloading capabilities make host-device interactions increasingly complex. Likewise, transport, RDMA, and network function offloads are valuable in certain use cases, but their fixed functionalities do not fit all applications' packet-processing needs. It is often difficult to express distributed systems protocols in terms of the limited RDMA primitives; the underlying network transport applied by RDMA hardware is also inflexible. Similarly, while hardware offloads such as TCP segmentation reduce packet-processing cycles, their benefits are limited to specific cases, e.g. large transmit payloads.

Acceleration with Programmable SmartNICs Emerging SmartNIC devices are a compelling solution, offering customizable, efficient packet processing, while also reducing host CPU burden. These devices integrate programmable compute units with the NIC, applying a mulitcore system-on-chip (SoC) [67, 8, 64, 63, 85, 76]. Typically, current SmartNICs apply the Arm (formerly, ARM) CPU architecture. Adding a programmable SoC to the NIC brings multiple benefits. First, the tight integration of the SmartNIC SoC cores and the Ethernet interface increase efficiency relative to interfacing with a conventional PCIe-attached NIC. Second, SmartNIC SoCs apply specialized hardware units to accelerate common packet-handling work. Third, in the context of cloud service providers, host CPUs are a precious resource; shifting network functionality onto SoC cores frees host cores for customer workloads.

SmartNICs also pose challenges, given their inherent limited resources and added complexity. Offloading work to a SmartNIC SoC requires careful consideration of the SoC's small memory and compute capacities relative to the host server. SmartNICs increase complexity, with host-to-SoC and SoC-to-NIC data paths in addition to the traditional host-NIC interface. Host-SoC communication across PCIe comes at a high cost; PCIe latency and its limited transfer protocol are primary challenges to offloading any tasks that share resources between the host and SoC. While SmartNICs offer the potential for significant gains, applying them requires software systems which navigate this added complexity. Specifically, distributed systems must be designed with awareness of the NIC SoC and its data paths. Data structures and protocols must maximize use of the

tightly-integrated SoC and NIC resources, while minimizing host-SoC and host-NIC traversals.

Tight Coupling with Coherent Interconnects Emerging coherent interconnect standards have the potential to make host-NIC and host-SmartNIC data paths more efficient. These include Compute Express Link (CXL) [12], UCIe [109], and UPI [89]; all allow external devices to participate in the host CPU's cache coherence domain. These interconnects represent a step towards the tight integration of compute and network interface, much like the integrated SmartNIC SoC and network controller. Coherent interconnects have different performance characteristics and semantics relative to PCIe. The high performance of these new interconnect standards is compelling as a means of improving packet rates and host-NIC latency. Additionally, coherence between the host CPU and peripheral devices has the potential to address a critical difficulty of applying SmartNICs–sharing data between the host and SoC. However, today's host-NIC interfaces are designed around the properties and mechanisms of PCIe, which are entirely different from sharing data across coherent caches. This suggests that in order to effectively apply coherent interconnects to NICs and SmartNICs, it is necessary to revisit the interface design, namely shared metadata structures, signaling, and packet memory management. A redesigned interface enables us to attain benefits from a tightly integrated host and NIC, by optimizing for the high-performance interconnect and its new forms of cache interactions.

1.1 Thesis Statement and Contributions

In this thesis, we demonstrate increasing performance and efficiency by applying programmable SmartNICs, and by leveraging tight coupling between compute cores and the network interface. Figure 1.1 shows the datapaths and scope considered in this thesis.

First, we conduct an evaluation of the varied design space of SmartNICs, to understand the implications of current SoC-NIC integration approaches and host-SoC data paths. We identify opportunities for SmartNICs to deliver a net performance and efficiency gain, and measure the hardware features which provide these gains. This study demonstrates that some, but not all, SmartNIC architectures create potential to accelerate networked systems. Additionally, specific

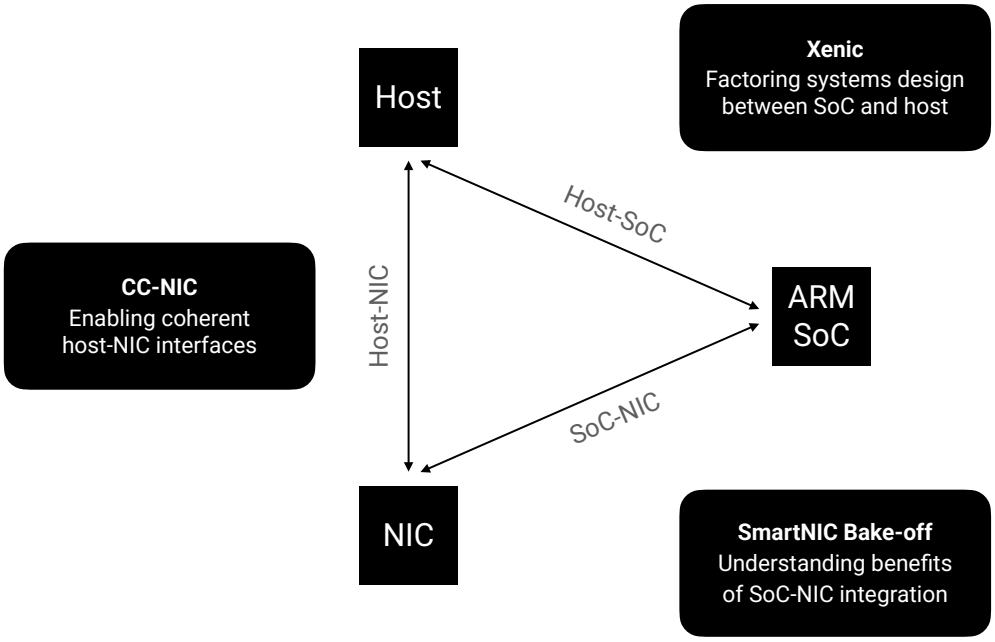

Figure 1.1: Data paths between host server, NIC, and SmartNIC SoC, along with three contributions of this thesis.

SmartNIC features serve as an example to be learned from, for future host-NIC interfaces. In particular, eliminating PCIe communication, in favor of tightly coupled core-to-NIC datapaths and efficient metadata management, enables higher-performance packet-transfer interfaces.

Given these results, we next focus on SmartNIC SoCs, and the integration of SoC resources and data paths with the design of a distributed system. We present Xenic, a distributed transaction processing system designed to leverage SmartNIC resources for increased throughput, lower latency, and core savings. Xenic is a case study of applying specialized data structures, protocols, and selective offloading, to attain these benefits. Xenic's design overcomes the challenges of SmartNIC integration, including limited SoC resources and high-cost PCIe communication.

Third, we explore the additional benefits to be achieved by replacing the PCIe host-NIC interface, to achieve tighter cache integration of the host-NIC data path. We present CC-NIC, a NIC interface design for cache-coherent interconnects. CC-NIC shows throughput and latency improvements, as well as reduced CPU utilization, over today's PCIe interfaces. CC-NIC shows

the benefits of tighter integration between the host and NIC for typical packet queue interactions.

1.2 Published Works

- Henry Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. **Xenic: SmartNIC-Accelerated Distributed Transactions.** In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21).*

- Henry Schuh, Arvind Krishnamurthy, David Culler, Hank Levy, Luigi Rizzo, Samira Khan, Brent Stephens. **CC-NIC: a Cache-Coherent Interrface to the NIC.** In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24).*

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 discusses the current state of the art for datacenter NICs, as well as existing approaches to acceleration in hardware and software. Chapter 3 is a measurement study of current SoC SmartNICs, with the goals of identifying opportunities for accelerating systems, and understanding which hardware features are responsible for increased performance and efficiency. In Chapter 4, we present Xenic, a distributed transaction system designed to leverage SmartNICs for performance and efficiency. In Chapter 5, we present CC-NIC, a host-NIC interface design for coherent interconnects, which achieves high throughput, low latency, and core savings relative to existing PCIe NIC interfaces. Finally, Chapter 6 describes conclusions and future work.

Chapter 2

BACKGROUND

Datacenter network speeds, together with application demands, have increased rapidly over recent years. This is evident in the doubling of Ethernet bandwidth with between NIC generations from major hardware vendors [68, 64, 63, 67, 65]. The bandwidth of the PCI Express (PCIe) interconnect has also doubled with each subsequent protocol version [84], tracking network bandwidth. However, the PCIe protocol itself has remained largely similar across these generations. Recent work evaluating PCIe host-NIC communications [77] finds that host-to-NIC latency is a high proportion of overall network latency within a datacenter rack. Thus, addressing PCIe inefficiencies has potential to substantially improve datacenter network latency. This evaluation also finds that PCIe may impose a bottleneck on packet rates, even when PCIe bandwidth is overprovisioned relative to the Ethernet link. In addition to these latency and throughput challenges, the flatter growth of CPU core performance [96] results in a critical need to reduce the CPU cost of network-intensive applications. The following sections discuss the range of hardware and software efforts to address these challenges.

2.1 Reducing Software Overheads

Kernel-bypass software frameworks such as the Data Plane Development Kit (DPDK) [14] and `ibverbs` [82] aim to reduce overheads by providing user-space applications with direct, poll-mode access to NIC queues. This avoids the latency cost of interrupts and user-kernel crossings to handle network communication. Instead, the user-space process maps NIC queue memory and device registers. These frameworks provide a consistent programming interface atop a range of vendor-specific NIC driver implementations. With DPDK, applications are entirely responsible for constructing packets and must implement any transport protocol they use. Unlike the traditional

Linux sockets interface, DPDK does not include any transport protocol or packet construction functionality. This adds complexity to application development, but brings the benefit of delivering raw hardware performance of the NIC to applications. In particular, DPDK applications may operate on packet data without the overheads of copying data between kernel and user space, or interrupt-based signaling. This approach has enabled new transport protocol designs optimized specifically for kernel-bypass application interfaces, such as mTCP [40], TAS [46], Snap [62], and Homa [75], among others [94, 98, 30, 57]. mTCP and TAS provide a kernel-bypass implementation of TCP (Transmission Control Protocol), a standard protocol for reliable, in-order communication. Other designs, such as Homa, replace TCP with alternate designs to further reduce communication overheads, or replace the operating system's sockets abstraction with specialized application interfaces.

Another line of work aims to reduce the bookkeeping work involved in handling per-packet NIC metadata [19, 88] to minimize cycles spent on each packet. These provide a simplified or optimized software interface at the driver level, maintaining the same NIC hardware. For example, PacketMill [19] optimizes the translation of vendor-specific metadata formats written by the NIC into generic structures utilized by DPDK. TinyNF [88] provides a simplified software interface for request-response protocols, which avoids buffer management bookkeeping work. These systems achieve performance improvements by eliminating software overheads at the driver level, but they remain limited by the PCIe NIC hardware-software interface. These systems do not change the interconnect communication which occurs between the host server and the NIC.

2.2 Optimizing Hardware Communication Paths

Efforts to optimize host-NIC communication paths include Intel's Data-Direct IO [36], which allows NIC-initiated PCIe transactions to be served by cache instead of DRAM. Research prototypes call for tighter [72, 31, 32], and more flexible [116, 20], cache integration with NIC data paths. These systems demonstrate performance benefits to providing direct data paths between the CPU cache and NIC. However, they stop short of bringing the NIC into the CPU's cache coherence domain. These designs instead still apply PCIe transfer primitives and thus, they do not

enable new forms of host-device interaction, or improve upon PCIe performance.

NICs such as the Nvidia Connect-X series enable alternative data transfer mechanisms to minimize packet transmission latency [70, 68]. This latency optimization, known as Blueflame in the context of Connect-X NICs, only applies to low-throughput scenarios. NIQ [22], a research prototype NIC interface, demonstrates a similar approach, which involves consolidating and inlining signaling, metadata, and packet data to reduce interconnect communication. As with Blueflame, these strategies improve latency. However, they are limited to specific scenarios, such as low-throughput small packet workloads.

2.3 Pushing Packet-Handling Functionality to the NIC

NIC hardware vendors, as well as wide-ranging research efforts, have proposed mechanisms for offloading packet-handling work to the NIC. This may involve accelerating existing transport protocols, such as TCP transmit segmentation offloads and TCP large receive offloads [64, 8, 67], or implementing packet-parsing engines in NIC hardware to steer flows to queues [7, 66, 64]. Remote direct memory access (RDMA) [68, 27, 43] pushes all transport and packet-handling work to the NIC, exposing memory read and write primitives; the target-side NIC performs the reads and writes, consuming no CPU cycles. Numerous distributed system designs [16, 44, 11, 113, 42, 74, 41, 2] leverage RDMA interfaces to attain CPU and latency savings.

In general, both RDMA and transport offloads require the transport protocol itself to be implemented in NIC hardware. This restricts flexibility, as transport and packetization are no longer software-defined. Further, RDMA provides a fundamentally different programming model of remote memory reads and writes, as opposed to packet transmit and receive. Consequently, applications and their data structures must be designed to use these remote memory access operations, in order to achieve CPU bypass benefits. RDMA-optimized data structures [15, 16, 113, 74] often require extraneous data to be transferred over the network for simple remote accesses, and may also increase the complexity of local accesses. In some cases, these costs of supporting RDMA result in lower overall performance [44]. Overall, offloading remote accesses to the NIC via RDMA can increase performance and save CPU cycles. However, the fixed hardware implementation of

RDMA operations, typically supporting only simple read and write accesses, limits their applicability.

2.4 SmartNICs

Programmable NICs, known as SmartNICs or DPUs, place a programmable SoC [67, 8, 64, 63, 85, 76] or FPGA [21, 69, 1, 115, 18, 89] on the NIC itself. The onboard compute serves as a target for offloading work, such as network functions or other infrastructure tasks, from the host CPU. Unlike fixed-function hardware offloads, SmartNICs inherently provide the ability to define customized work in software or FPGA logic. The SmartNIC's programmable compute raises the question of when it is beneficial to perform packet-processing work on NIC cores as opposed to host CPU cores. In some scenarios, such as cloud service providers, moving network infrastructure tasks off of the host CPU is fundamentally valuable because it frees CPU cores for paid customer workloads. This is a key motivation behind existing work in the SmartNIC space, in particular systems which offload network functions to SmartNICs. ClickNP [52], Catapult [21], Panic [56], and others [26, 3, 91, 102, 101, 50, 33, 71, 99] apply SmartNIC resources to perform various packet-processing functions.

Beyond shifting work from host cores to NIC cores, SmartNICs also have the potential to improve system performance and compute efficiency. Handling packets at the NIC avoids traversing the host-NIC interconnect, and specialized functionality at the NIC, such as packet-processing accelerators, may also be beneficial. A range of recent work targets these benefits, with both generalized interfaces for offloading and application-specific designs. FlexNIC [45], Floem [87], and iPipe [59] consider SmartNICs as a target for application-level offloading, providing frameworks for applications to utilize SmartNIC resources. Additional research works explore applying SmartNICs to specific applications, such as key-value stores [51, 60, 45] and distributed file systems [48]. Our work builds upon these ideas. We consider the potential for performance and efficiency gains enabled by SoC SmartNICs in Chapter 3. Given the specific SmartNIC hardware interfaces which show potential for performance gains, in Chapter 4, we pursue application-level offloading to maximize performance and efficiency via SmartNIC resources.

Chapter 3

SMARTNIC BAKE-OFF: UNDERSTANDING PERFORMANCE OPPORTUNITIES

First, we explore the potential for SmartNICs to increase the performance and efficiency of networked systems. Given the wide range of SmartNIC designs currently available, we aim to identify the specific architectural features which present opportunities and challenges to systems acceleration. SoC SmartNICs support the ability to offload arbitrary work to cores on the NIC. The latest devices generally provide an operating system abstraction to do so, by running embedded Linux on the SmartNIC's Arm SoC. While this software environment is similar across most current SmartNICs, the interfaces and datapaths differ.

Shifting work off of host cores and onto NIC cores can provide several benefits: for instance, in the cloud environment, offloading frees host cores for customer use instead of managerial networking tasks. But, placing work on the SoC cores does not inherently improve performance or the net CPU efficiency of a system. The same work could also be handled by increasing the host CPU core count, rather than adding complexity at the NIC. Furthermore, offloading may introduce new overheads, arising from the SoC's lower-power cores, its limited on-board memory, and the PCIe communication required to transfer data to and from the host.

Achieving acceleration through offloading depends on specific properties of the SmartNIC. In particular, some SmartNICs have the potential to increase overall system performance and efficiency by placing compute closer to the Ethernet interface. In terms of latency, this means eliminating communication over the host-to-NIC PCIe interconnect. In terms of throughput efficiency, some devices implement specialized accelerators and network TX/RX interfaces on the NIC SoC, which can support packet-handling at a lower CPU cost. However, not all SmartNICs deliver these benefits; they are the product of efficient interfaces at the NIC SoC, the design of

SmartNIC	Ethernet	PCIe	SoC CPU	On-board DRAM
Marvell **Octeon** CN10K	1×200G, 2×50G	5.0 ×8	24×ARMv9 2.2GHz	48GB DDR5
Marvell LiquidIO 3 (**LIO3**) CN9130	2×50G	4.0 ×8	24×ARMv8.2 2.2GHz	16GB DDR4
Nvidia Bluefield 3 (**BF3**) B3220	2×200G	5.0 ×16	16×ARMv8.2+ 2.2GHz	32GB DDR5
Nvidia Bluefield 2 (**BF2**) MT42822	2×100G	4.0 ×16	8×ARMv8 2.0GHz	16GB DDR4
Broadcom **Stingray** PS1100R	1×100G	3.0 ×16	8×ARMv8 3.0GHz	8GB DDR4

Table 3.1: Set of measured SmartNIC devices, and their hardware specifications. We refer to each device by its bolded name in the following experiments.

which varies considerably between platforms.

Thus, we begin with a measurement study of current SmartNIC SoCs to understand the extent to which these devices provide opportunities for performance and efficiency gains. In Section 3.1, we survey a set of current-generation SmartNIC platforms, and characterize SoC compute and on-board memory performance. Then, in Section 3.3, we measure the potential for performance and efficiency gains across platforms. We find that the potential for gains depends on tightly-integrated interfaces at the Arm SoC, in particular, the core-to-Ethernet TX-RX interface. This provides the opportunity to accelerate systems, by eliminating PCIe communication and making use of these efficient interfaces. To understand the design features leading to performance improvement, we perform a deep dive into SoC datapaths in Section 3.4, comparing Ethernet interfaces across SmartNICs. SmartNIC datapaths show benefits to tighter core-to-NIC coupling, relative to host PCIe datapaths. In Section 3.5, we conclude with insights which motivate the work in subsequent chapters of this thesis.

3.1 Survey of SmartNIC Platforms

Our measurement characterization considers a set of recent SoC SmartNICs released from a range of vendors, listed in Table 3.1. We focus on current-generation devices with at least 100Gbps of Ethernet bandwidth. These devices cover the range of SoC integration approaches available today. Specifically, the Bluefield and Stingray SmartNICs apply a switch-based *off-path* [59] model, while the Octeon and LiquidIO devices apply a *on-path* architecture, where the SoC interposes on all

inbound and outbound traffic.[1] In the following sections, we compare these approaches, as well as the hardware performance of the SmartNICs, with a series of benchmark experiments.

3.2 Experimental Setup

For all measurements in this chapter, we apply the following server setup.

Server specifications. We perform measurements on the Intel Ice Lake server platform. Each server has dual 3.1GHz Xeon Gold 6346 CPUs (32 hyperthreads and 36MB LLC per socket). Each CPU has 12×16GB DDR4 DRAM. The CPU provides PCIe 4.0×16 peripheral interfaces, to which the NICs are attached. This PCIe interface provides 256Gbps of link bandwidth. For compute benchmarks, we provide the additional CPU comparison point of an Intel Xeon Gold 5218. This CPU has 32 hyperthreads at 2.3GHz, and 22MB LLC per socket, in a server with 6×8GB DDR4 DRAM per socket.

Software environment. Each server runs Ubuntu 18.04. For each SmartNIC SoC, we deploy the respective vendor-provided Linux operating system release. The Bluefield SmartNICs run Ubuntu by default; Marvell and Broadcom devices apply customized compiler toolchains and variants of embedded Linux. We use the DPDK 23.03 framework for all network experiments, to demonstrate kernel-bypass networking performance at both the host and SmartNICs. DPDK includes updated, open-source driver implementations for each NIC. For packet-processing benchmarks, we use included DPDK tools such as the `testpmd` performance-testing utility. Additionally, we implement a customized workload generator based on `testpmd`. This application transmits timestamped UDP packets, with configurable batching and rate-limiting, and measures roundtrip echo time for each received packet. Unlike the built-in DPDK tools, this enables latency profiling.

[1] The LiquidIO 3 and Octeon CN10K devices both contain an Octeon SoC, part numbers CN9130 and CN10624 respectively. For simplicity, we refer to the devices by their product names, LiquidIO and Octeon.

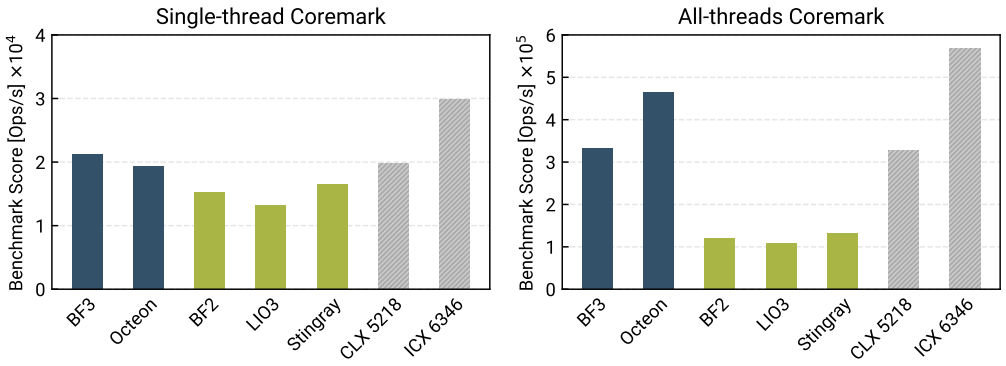

Figure 3.1: SmartNIC CPU performance measured using the Coremark benchmark in (a) single-threaded and (b) all-threads configurations (note different y-axis scales). Blue bars show the two newest SmartNIC platforms; gray bars denote the Intel server CPUs for comparison.

Network setup. Since our measurements focus on internal NIC datapaths, we aim to minimize network impacts on latency or packet rates. Thus, we perform network measurements with a direct-attach copper QSFP cable between the two Ethernet ports of each NIC. For the Octeon, which has a singular QSFP port, we instead connect a loopback module. This has the same effect of directing transmitted traffic to the NIC's receive path. We observe comparable TX-RX latency with the loopback module, and with a cable between the Octeon's dual 50Gbps SFP ports. The loopback module allows us to test the device at its maximum Ethernet bandwidth.

3.2.1 SoC Compute and Memory Performance

To provide context for the SoC capabilities of each SmartNIC, we begin with benchmarks of core and memory performance. These SoC benchmarks focus on compute capability and do not involve network communication. We compare the SmartNICs listed in Table 3.1, as well as two recent mid-range Intel server CPUs described in Section 3.2, Cascade Lake Xeon Gold 5218 (32 hyperthreads at 2.3GHz, 22MB LLC per socket) and Ice Lake Xeon Gold 6346 (32 hyperthreads at 3.1GHz, 36MB LLC per socket). To measure compute performance, we use the standard Coremark benchmark [17], reporting average throughput score using default runtime settings. This

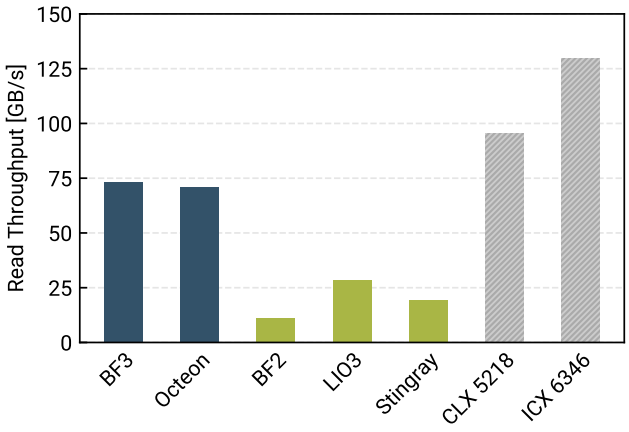

Figure 3.2: On-board memory throughput of sequential reads, with 1GB per-thread working set. Blue bars show the two newest SmartNIC platforms; gray bars denote the Intel server CPUs for comparison.

benchmark demonstrates performance of synthetic operations relating to typical network stack computation, such as state machine processing, checksum calculation, and list access.

Figure 3.1 shows Coremark benchmark throughput for (a) single-threaded runs and (b) multi-threaded runs utilizing all threads of each CPU. The two most recent SmartNIC platforms, Octeon and Bluefield 3, approach the single-threaded and multi-threaded performance of the Intel server CPUs. This is a notable improvement over the previous generation of platforms. Relative to the server CPUs, the lower Bluefield 2 and Stingray core count (8 for both devices) result in lower multi-threaded performance. The LiquidIO 3 has 24 cores, but throttles when all cores are active, which also results in lower multi-core performance.

As with compute performance, The Bluefield 3 and Octeon platforms also demonstrate on-board memory bandwidth competitive with server-class CPUs. Using the `pmbw` memory access benchmark [6], we measure the sequential read throughput with all threads of each device accessing private 1GB arrays. Figure 3.2 shows these results. The Bluefield 3 and Octeon platforms again overcome a major limitation of the earlier-generation SmartNICs, by providing DRAM bandwidth approaching that of the server CPU.

In Figure 3.3, we use `pmbw` to measure single-threaded sequential memory access latency with

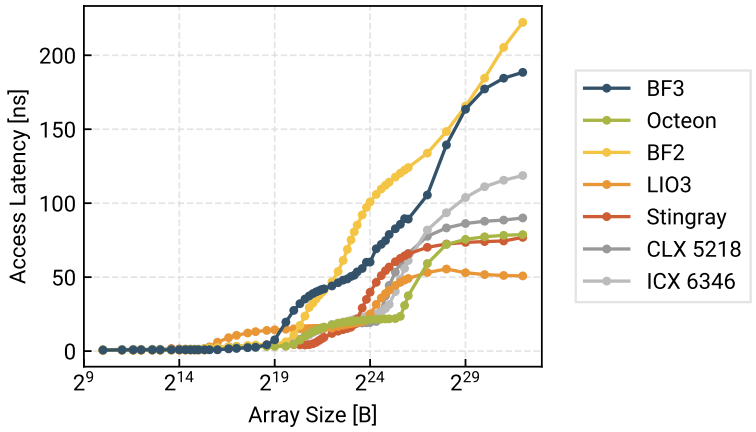

Figure 3.3: On-board memory access latency, with a single-thread sequential read pattern over varied array sizes, showing cache hierarchy performance.

varied array sizes. This benchmark performs simple access patterns, such as sequential and random reads, to measure bandwidth and latency. We use standard 64b load instructions (as opposed to vector loads), implemented in an unrolled loop to avoid any compiler optimization impacts. Our sequential access workload shows the latency profile of each device's cache hierarchy. As access sizes exceed the size of each cache level, accesses are served by the next cache level or DRAM.

The results suggest that access latencies of on-board DRAM and SoC cache are comparable to the host, with the significant exception of the Bluefield 2 and 3 platforms. These devices have smaller cache capacities, and show higher latency than all other platforms, for random access patterns above cache capacity. Both the Bluefield 2, with DDR4 memory, and the Bluefield 3 with DDR5, show higher latency.

This suggests that offloading tasks to the Bluefield SoC may result in increased latency, relative to other SmartNICs and the server CPUs. Even tasks entirely self-contained to the SoC resources may be penalized by high DRAM access latency. On the other hand, the LiquidIO, Octeon, and Stingray devices show similar or lower latencies relative to the server CPUs, across the range of access sizes.

3.3 SmartNIC Packet-Processing Performance

In the following sections, we focus specifically on the Octeon and Bluefield 3 platforms. These are the most recent devices, and they show leading performance, in terms of compute, memory bandwidth, and Ethernet link rates (§3.1). Importantly, these devices represent two opposing areas of the SoC SmartNIC design space. The Bluefield is an *off-path* SmartNIC, with distinct data paths and network traffic flows from the *on-path* Octeon platform [59].

The off-path Bluefield architecture centers around an internal switch, which directs traffic between the device's Ethernet TX/RX hardware, the host PCIe interface, and the on-board SoC's Arm PCIe interface. This allows traffic to be directed to either the host or Arm, e.g., by MAC address. However, in both cases, the packets cross a PCIe NIC interface; either between the NIC and host, or the NIC and the Arm SoC. While the SoC's PCIe link is internal to the SmartNIC, it applies the same software interface and PCIe semantics as the host. This suggests from a core utilization perspective, the Arm interface has no inherent efficiency benefit. In Section 3.3.2, we compare the core-to-NIC interfaces of the Bluefield Arm and host, as well as the Octeon interface discussed below. Another challenge of the off-path design is communication between the host and SoC. This may involve packet TX/RX or RDMA operations, which are routed through the internal SmartNIC switch, or reads and writes via a PCIe DMA engine exposed to the SoC. We measure these datapaths in Sections 3.3.1 and 3.3.3.

The on-path Octeon architecture applies specialized integration between the SoC cores and Ethernet MAC. The SoC directly interfaces with the hardware Ethernet TX/RX queues, as well as specialized hardware modules for flow steering and packet buffer management. The interface between the SoC and Ethernet is not a PCIe link, but instead a specialized interface leveraging FIFO instructions, shared memory, and hardware registers for communication. A similar interface applies for PCIe packet transfers; the SoC utilizes a PCIe DMA engine and hardware-defined packet queue interface. This architecture requires that each packet is handled by SoC cores. Even for traffic unrelated to offloaded functionality, the SoC must transfer packet buffers between the Ethernet and PCIe queue interfaces. This results in software overhead on all traffic passing through

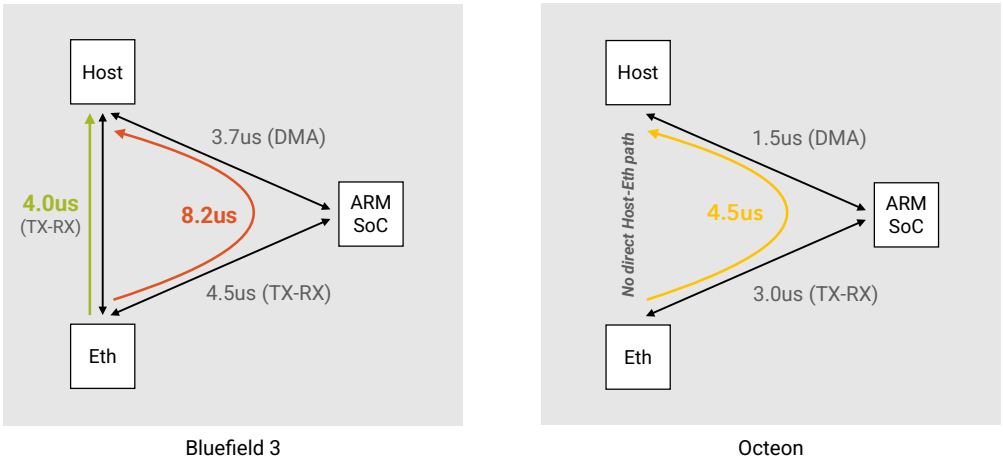

Figure 3.4: Latency of datapaths from Ethernet to host memory, through the Arm SoC and bypassing it. We compare on-path Bluefield and off-path Octeon platforms.

the NIC. However, unlike off-path NICs, this design creates the potential for efficiency improvement by moving work to the SoC: by eliminating PCIe packet transfer and handling work on the NIC itself, or replacing host network stack traversal with simple SoC-initiated DMA accesses. In Sections 3.3.1 and 3.3.3, we measure the performance potential of these offloaded operations. Also, the SoC's hardware-accelerated TX/RX interface creates the potential for higher-efficiency packet handling relative to the host's PCIe NIC interface. In Section 3.3.2, we measure the efficiency of this specialized Ethernet interface at the SoC.

3.3.1 SmartNIC Latency Reduction Potential

In order to achieve latency reduction through SmartNIC offloading, there must be a latency benefit to handling packets at the SoC instead of the host. Specifically, latency must be lower between the SoC cores and Ethernet interface, than between the host cores and Ethernet. This includes hardware data transfer latency of the core-to-NIC interconnects, in addition to the latency of signaling and metadata communication required to send and receive traffic. For traffic which is not fully offloaded to the SmartNIC SoC, and instead requires host processing or memory access, the latency of SoC-to-host and Ethernet-to-host data paths is also important to consider. Given

the architectural differences between on- and off-path SmartNICs, latency savings potential varies considerably between SmartNICs.

In Figure 3.4, we compare latency for the Bluefield and Octeon devices. We measure roundtrip latency of individual datapaths between the host, SoC, and Ethernet controller. We use the setup described in 3.2 to measure host-Ethernet and SoC-Ethernet latency with DPDK traffic, with both NICs configured for an Ethernet TX-RX loopback. For both devices, we also show cumulative latency of Ethernet-to-host traffic which passes through the SoC. For SoC-to-host communication, we measure latency of SoC-initiated reads of host memory performed using each SoC's DMA engine. DMA read accesses are a single PCIe transfer, as opposed to the multiple data and metadata transfers required for packet TX-RX. This operation represents a scenario where an offloaded operation requires access to host memory, but not a full host network stack traversal.

The Bluefield results show that its off-path design does not have potential for latency reduction: direct traffic to the host (4.0us roundtrip) has lower latency than traffic to the SoC (4.5us). In contrast, the Octeon SoC shows lowest overall core-to-NIC latency with Ethernet traffic targeting the SoC (3.0us). This suggests that with the on-path Octeon architecture, handling network traffic on the SoC can reduce latency. However, the on-path architecture requires all traffic to be processed by the SoC; the minimum Ethernet-to-host latency is the sum of the Ethernet-to-SoC and SoC-to-Host latencies. With the Octeon, this cumulative latency is 0.5us higher than the Bluefield's direct Ethernet-to-Host path. The Octeon SoC's potential to reduce latency comes at the cost of increased latency and SoC utilization for non-offloaded traffic.

The resource limitations of SmartNICs, namely their small on-board memory capacity, also pose a challenge to potential performance gains. Tasks exceeding the SmartNIC memory capacity, or those which require data sharing with host-side threads, inherently require SoC access to host memory. As such, the latency of SoC-to-host datapaths is important. SmartNICs typically implement a PCIe DMA engine for the SoC to perform transfers to and from host memory. To avoid a latency penalty for offloaded operations requiring host memory access, this DMA engine must support low-latency transfers (and line-rate throughput, discussed in §3.3.3). Not all SmartNICs achieve this; the Bluefield in particular demonstrates high overhead for SoC-to-host

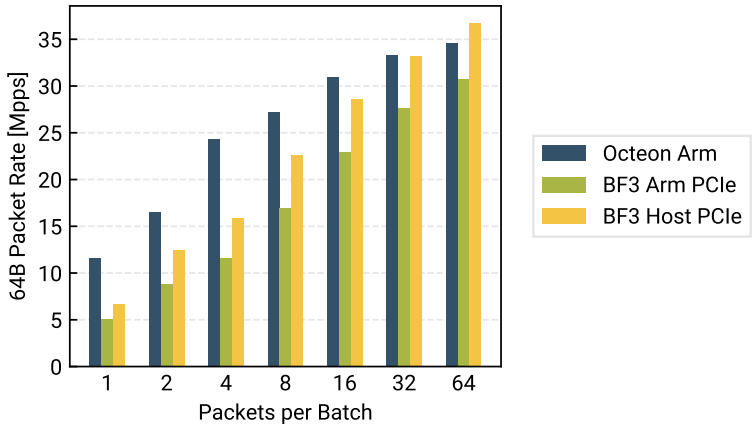

Figure 3.5: Per-thread packet rate with 64B packets and a wire loopback, varying TX/RX batching factor, for three interfaces: Octeon Arm, Bluefield 3 Arm, and Bluefield 3 Host.

DMA transfers. As shown in Figure 3.4, Bluefield SoC-to-host DMA accesses incur similar latency (3.7us for 64B memory read) to direct-to-host Ethernet traffic (4.0us for 64B packet TX-RX). This suggests that offloaded work which performs a single host memory access has at least 8.2us of cumulative latency, more than double the direct host-to-Ethernet latency. On the other hand, the Octeon SoC performs PCIe DMA read accesses to host memory with 1.5us minimum latency, closer to the physical interconnect latency of PCIe.

Overall, attaining latency reduction benefits is nontrivial. For the Octeon platform, the latency of host TX/RX is higher than with the direct Ethernet-to-host path of the Bluefield, and the SoC cores impose on all inbound and outbound traffic. To achieve an overall latency improvement, a significant proportion of traffic must therefore be handled on the SmartNIC SoC.

3.3.2 SmartNIC Ethernet TX/RX Efficiency

Beyond latency reduction, the integration of the SoC and Ethernet interface creates the potential for greater packet-handling efficiency relative to the host's PCIe interface. Since the PCIe interface has greater bandwidth than the NIC's Ethernet line rate, this does not necessarily enable higher overall throughput. But, an efficient interface increases per-core throughput, CPU cycles per TX/RX operation, and reduces the need for batching. In Figure 3.5, we compare maximum

per-thread TX-RX packet rates for the cases in Figure 3.7. We use the DPDK `testpmd` workload in `flowgen` mode, sending packets to a thread-local TX queue and receiving them at a corresponding RX queue via wire loopback. In a polling loop, each `flowgen` thread receives a batch of packets from a private RX queue, immediately releasing the buffers, and transmits a batch of packets, created by allocating packet buffers and copying headers from a packet template. By focusing on small packets, we demonstrate the efficiency of the signaling and descriptor metadata for each packet transfer, as opposed to the bulk data transfer (which, in this workload, does not consume CPU cycles).

We measure a range of TX/RX batching factors to understand the impact of both per-packet and per-batch costs. Typical PCIe NIC interfaces (such as that of the Bluefield SoC and host) use a *doorbell* to signal each transmit batch. The Octeon uses specialized instructions to submit descriptors, without doorbell signaling, as well as hardware instructions to allocate and release packet buffers.

Our results show that the Octeon SoC's Ethernet interface is more efficient than that of the Bluefield SoC, achieving greater per-thread packet rates across the range of batching factors: 2.4× without batching, and 1.1× with a batching factor of 64. The Octeon also outperforms the host's Ethernet interface by 1.8× without batching, although the performance gap closes at batching factors of 32 and above. These results suggest that with the Octeon, offloading creates not only the potential for latency reduction, by avoiding PCIe traversals, but also potential to reduce the CPU cycles and batch sizes for a given packet rate. In Section 3.4, we perform further measurements of the Octeon descriptor and buffer management interfaces, to identify the source of these efficiency gains.

3.3.3 Line-rate SoC to Host Communication

SoC-to-host access performance is another important factor influencing a SmartNIC's capability for performance gains. Apart from completely self-contained tasks, fully offloaded to the SoC with no host data sharing, all other offloading requires SoC-to-host communication. Additionally, for on-path platforms, the SoC handles all Ethernet traffic. This means that SoC-to-host transfer

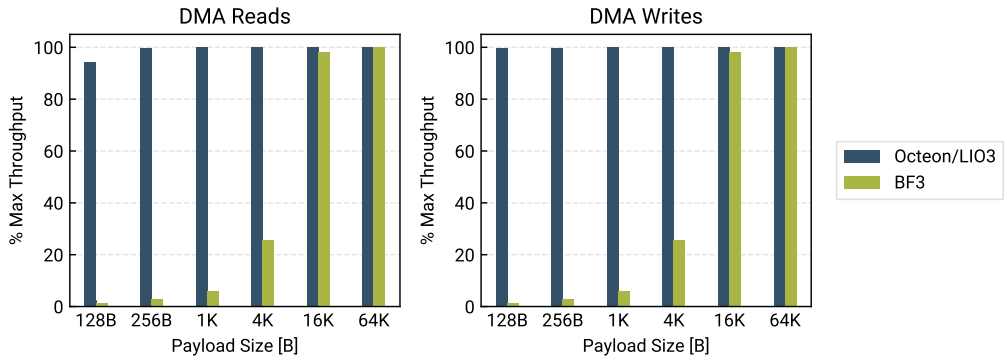

Figure 3.6: Arm-initiated DMA read (a) and write (b) throughput relative to payload size, for Octeon/LiquidIO and Bluefield 3 platforms.

paths are critical to overall performance, and these interfaces must be able to sustain network line rates. Figure 3.6 compares the DMA engine performance of Bluefield and Octeon/LiquidIO platforms, with a range of packet-sized payloads. While both devices can saturate PCIe bandwidth under some conditions, e.g., large 64KB reads and writes, there is a substantial discrepancy in performance at lower sizes typical of packet payloads. For this experiment only, we measure the LiquidIO SmartNIC's Octeon CN9130 SoC, instead of the newer Octeon CN10K SoC. The two SoCs share the same DMA engine hardware; we measure the LiquidIO because it is a PCIe card form factor which can be installed into a host server, unlike the Octeon CN10K development board.

The Octeon DMA engine achieves at least 97.9% of its maximum throughput with all sizes 128B and above. In contrast, the Bluefield 3 DMA engines achieve only 1.4% (writes) and 0.9% (reads) of peak throughput with 128B sizes, 8.5% (writes) and 6.0% (reads) with 1KB sizes. Achieving peak throughput requires individual transfer sizes to be 16KB or larger. Since 16KB is significantly greater than typical network MTUs, this suggests offloaded applications will be bottlenecked by the DMA engine if even a small proportion of packets trigger SoC accesses to host memory. This severely limits the ability to share data between the SoC and host on a per-request or per-packet basis. On the other hand, the Octeon DMA engine can keep up with network line rate across the range of packet sizes. Offloaded functionality on the Octeon SoC may therefore perform per-

Operation	Bluefield Implementation	Octeon Implementation
TX 1. Allocate buffer	`rte_mempool` concurrent dequeue	Load address from NPA HW register
TX 2. Write descriptor	Store to ring memory + barrier	Store to SoC scratchpad
TX 3. Signal device	PCIe MMIO store ring tail index	`LMTST` instr. copy scratchpad to HW register
TX 4. Device completion	PCIe DMA write ring head index	Increment TX count register
TX 5. Await completion	Poll ring head index	Poll TX count register
TX 6. Free buffer	`rte_mempool` concurrent enqueue	HW frees to NPA automatically after TX
RX 1. Allocate buffer	`rte_mempool` concurrent dequeue	Eth interface allocates automatically from NPA
RX 2. Post blank buffer	Store to ring memory + barrier	–
RX 3. Signal device	PCIe MMIO store ring head index	–
RX 4. Device completion	PCIe DMA write descriptor	–
RX 5. Read descriptor	Poll next descriptor at ring tail	Load descriptor from HW register
RX 6. Free buffer	`rte_mempool` concurrent enqueue	Store address to NPA HW register

Table 3.2: The sequence of operations for DPDK packet TX and RX, and the associated communication on the Octeon and Bluefield Ethernet interfaces.

packet host memory accesses, without becoming throughput-bottlenecked by the PCIe interface.

3.4 Deep Dive into SoC Datapaths

In this section, we perform additional measurements with the goal of quantifying the CPU efficiency gains of the Octeon's hardware-accelerated descriptor and buffer management. Table 3.2 summarizes the sequence of operations which take place in a basic scenario of transmitting and receiving a packet. We compare Bluefield (which is representative of typical PCIe NIC interfaces) with the Octeon's interface.

At a high level, PCIe NICs handle buffer management with a software data structure, while the Octeon provides a *NPA* (Network Pool Allocator) hardware-managed buffer pool. The Octeon interface simplifies buffer management at the software level; each allocate and release is a load or store instruction to the NPA's corresponding hardware register, instead of a concurrent data structure update. This reduces CPU data structure management costs associated with the buffer pool, avoiding software synchronization between cores. The Octeon Ethernet interface automatically frees completed TX buffers, and allocates blank RX buffers, via the NPA pool. Similarly, PCIe NICs apply a descriptor ring structure to handle submit work to the device and handle completions. The Octeon interface avoids the PCIe descriptor ring, including the associated head and

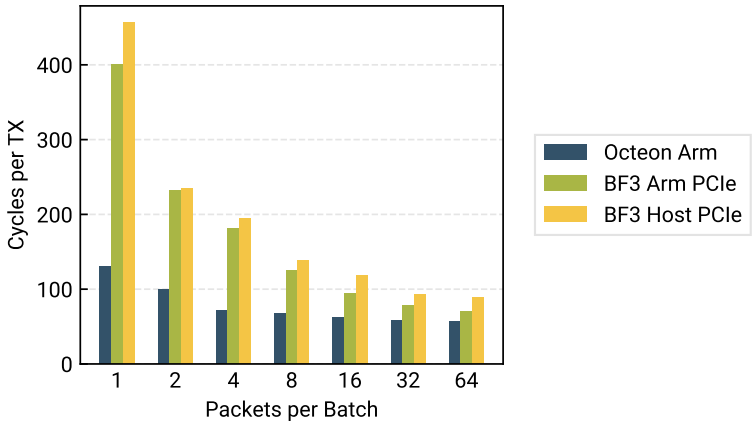

Figure 3.7: Cycles per packet TX operation, with 64B payloads and varied batching factors, for three interfaces: Octeon Arm, Bluefield 3 Arm, and Bluefield 3 Host.

tail registers. Instead, descriptors are buffered in SoC scratchpad and pushed to the Ethernet interface with the `LMTST` 128B store instruction. SoC cores poll for the next RX packet by loading an SoC register managed by the scheduler component of the Ethernet interface.

In the following subsections, we perform experiments comparing these designs. First, we evaluate overall CPU cost of TX workloads (§3.4.1). This demonstrates the cumulative gains from hardware buffer and descriptor management, and from the integration of these features (i.e., direct NIC access to the buffer pool). Then, we measure descriptor handling (§3.4.2) and buffer management (§3.4.3), individually.

3.4.1 SoC Descriptor Submission

To compare the Octeon and Bluefield designs, we measure a transmit-only workload, measuring cycles per packet across a range of batching factors. As in §3.3.2, we use 64B packet sizes to focus on the metadata efficiency of the interface. Figure 3.7 shows these results for the Bluefield SoC, Bluefield host, and Octeon interfaces.

Our measurements show that the Octeon descriptor interface achieves the lowest cycle count per transmit operation. This is true across batching factors and core counts; the Octeon interface shows the greatest CPU cycle savings with low batching factors and high core counts. The Blue-

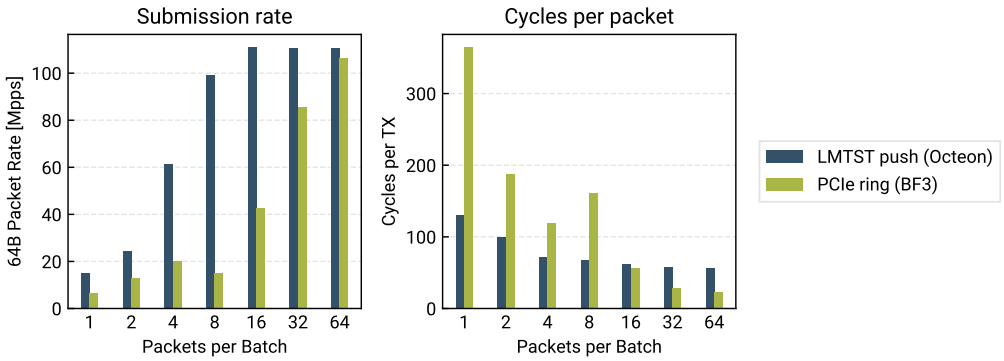

Figure 3.8: Throughput and per-packet cycles comparison of Octeon and Bluefield descriptor submission, varying doorbell batching factor, with a single TX queue.

field SoC interface requires up to 3.0× more cycles per transmit in the unbatched scenario. Comparing cycles per TX in batched versus unbatched cases for each platform, the Bluefield interface shows 5.7× cycle overhead for unbatched TX packets, compared to 2.3× with the Octeon interface. Overall, these results demonstrate CPU efficiency gains arising from tighter coupling between the cores and NIC. The Octeon's direct interface to manage descriptors and packet buffers, relative to traditional PCIe descriptor rings, achieve greater packet handling efficiency at a lower CPU cost. In the next sections, we compare the performance of descriptor and buffer handling, in isolation, between the two platforms.

3.4.2 Descriptor Submission

PCIe NIC interfaces, such as the Bluefield's SoC and host interfaces, instead apply a descriptor ring with doorbell registers for signaling. The host first writes a batch of TX descriptors into a ring buffer in local writeback memory. Then, it signals the availability of new descriptors to the device by writing the updated ring tail index to the device's *doorbell* register via PCIe memory-mapped IO (MMIO). This MMIO store triggers the device to read new entries from the host's descriptor ring in a subsequent PCIe transfer. The device indicates transmit completion by incrementing the descriptor ring's head index, allowing the host to release the buffers to the software-managed pool data structure. The Octeon implements a proprietary instruction, `LMTST`, to submit up to

4 descriptors (128B total) to an Ethernet transmit queue. Each individual descriptor contains the packet address and metadata such as data length and checksum offload flags, with a total size of 32B. After the transmission is completed, the buffer is returned to its hardware-managed packet pool (discussed in §3.4.3). Relative to PCIe interfaces, the Octeon's descriptor submission mechanism reduces interconnect communication, by eliminating the need for separate doorbell and ring accesses. It also reduces software bookkeeping work to handle completions, and avoids the PCIe MMIO datapath and its uncacheable and write-combining memory modes.

To compare these designs, we measure a transmit-only workload, measuring cycles per packet across a range of batching factors. As in previous experiments, we use 64B packet sizes to focus on the metadata efficiency of the interface. To isolate descriptor handling performance, we pre-allocate a set of packet buffers, equal to the configured batching factor, and repeatedly submit the same packet buffers. We disable freeing of completed TX buffers by the NIC driver. This eliminates buffer management and payload accesses from the critical path, isolating TX descriptor submission and completion work. Figure 3.8 shows these results for the Bluefield and Octeon interfaces, measuring both single-threaded TX packet rates and cycles per TX.

These results show benefits to the Octeon descriptor push mechanism, especially in scenarios with low batching. The Octeon descriptor submission performance approaches its maximum with batching factors 8 and above. The Bluefield's PCIe descriptor ring requires higher batching, up to 128 packets per doorbell, to achieve its peak. This suggests that the PCIe doorbell and other per-batch work is expensive, but can be amortized with high enough batching factors. With batching of 64 and above, the Octeon and Bluefield show similar single-thread packet rates, and the Bluefield consumes fewer cycles per packet. However, the Octeon's interface is substantially more efficient than the Bluefield at lower batching factors: with 8 packets per burst, the Octeon achieves a 6.6× higher packet rate (89% of its maximum), with just 87% fewer cycles per packet relative to the Bluefield.

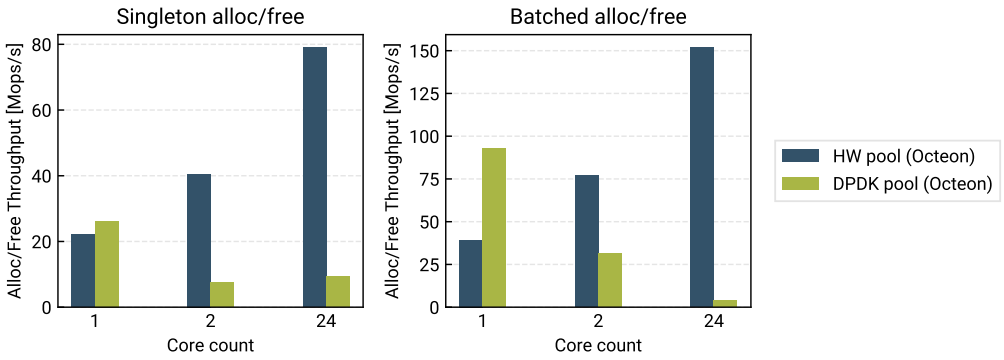

Figure 3.9: Throughput of the Octeon NPA hardware buffer pool and the DPDK `rte_mempool` software implementation.

3.4.3 Hardware and Software Packet Buffer Management

Next, we compare the Octeon NPA hardware-managed buffer pool with the DPDK `rte_mempool` software buffer pool structure utilized by the Bluefield and all other NICs within the DPDK framework. Like descriptor submission, the Octeon buffer pool makes use of a proprietary instruction to allocate or release a batch of up to 128 buffers at once. The `rte_mempool` software pool is a concurrent queue data structure, and a thread-local cache to reduce sharing.

Since it is not possible to configure each interface to change buffer management features, we measure buffer management in isolation using the DPDK `mempool_perf` performance test. This is a multi-threaded workload, where each thread repeatedly allocates, holds, and then releases, a batch of buffers. This is representative of a typical network application, where each thread has a working set of packets being processed at a given time. On the Octeon SoC, we repeat the benchmark with the hardware and software pool implementations. Figure 3.9 shows pool throughput for each case.

In the single-threaded scenario, software pool operations are local data structure manipulations, but with multi-core accesses, atomic operations and synchronization between cores are necessary. The software pool performs well with one thread, showing higher throughput than the hardware pool. However, total throughput of the software pool drops substantially when multiple threads are introduced, due to the synchronization required for correct allocation and freeing

behavior. Since multiple threads are required to saturate the Octeon and Bluefield Ethernet interfaces, these multi-threaded software overheads apply to any throughput-oriented workloads. With the hardware pool, sharing is handled transparently by the underlying buffer management hardware, and throughput scales with core count. With all threads performing buffer management operations, hardware pool throughput is up to 40.2× (batch size 4) and 8.3× (batch size 32) greater. These results indicates that, as with descriptor management, hardware acceleration of NIC buffer management creates the potential for more-efficient NIC packet handling, relative to software buffer pools designed for today's PCIe NIC interfaces.

An additional benefit of hardware buffer management, not captured in this microbenchmark, is enabling the NIC to directly allocate RX buffers, and release TX buffers, instead of relying on driver software to do so. This behavior is demonstrated in the overall packet-handling performance measurements (§3.3.2, §3.4.1).

3.5 Conclusions

Together, these experiments motivate the remainder of the thesis in two ways:

1. In Section 3.3, we find that SmartNICs can provide an opportunity for accelerating systems performance, and reducing CPU load, beyond simply shifting work to the NIC cores. This opportunity includes latency reduction by eliminating PCIe communication, and making use of the tightly integrated SoC packet-handling data paths. We pursue this idea in Chapter 4.

2. As measured in Section 3.4, the Octeon SoC-NIC interface suggests that there remains substantial room to improve PCIe host-NIC interfaces. It is possible to achieve greater packet handling performance and efficiency, through tighter integration of the core-to-NIC datapaths. We extend this tight integration to the host's peripheral interfaces in Chapter 5.

Chapter 4

XENIC: SMARTNIC-ACCELERATED DISTRIBUTED TRANSACTIONS

Distributed transactions, though a valuable programming model, are a challenging workload in the datacenter environment. Providing replication and serializability requires coordination between multiple shards of data with multiple replicas of each shard. Together, these guarantees incur a high communication cost, making the practicality of the distributed transaction model contingent on the performance of datacenter networks [16, 44, 114].

Recent developments in hardware and software acceleration have increased the performance of distributed transaction systems. Kernel-bypass networking reduces both the latency of network transfers and the end-host processing overhead. RDMA further cuts server processing costs by offloading simple memory operations from the server CPU to the NIC itself. One-sided RDMA enables reads, writes, and atomic operations on a remote server's memory without involving the remote server's CPU and with lower latency than operations that traverse the host networking stack. By expressing the transaction commit and replication protocol in terms of one-sided RDMA operations, the server-side computation involved in performing transactions can be eliminated, avoiding software network stack overheads [15, 16, 114].

A critical limitation of current RDMA NICs, however, is their small set of memory access primitives: read, write, fetch-and-add, and compare-and-swap. Applying these RDMA primitives to a distributed system typically requires significant design trade-offs in terms of data structures and protocol logic. For instance, a remote hash lookup using one-sided RDMA reads necessitates multiple network roundtrips for a hash miss; this can be mitigated by reading multiple buckets at once, but we then waste bandwidth to improve latency [15]. Likewise, one-sided RDMA supports only a request/response message pattern, limiting options for protocol communication. Often, these compromises negate the benefits of hardware offloading. In the context of distributed

transactions, using RPCs for some [113] or all [44] remote operations can lead to higher performance than fully applying one-sided RDMA. Ultimately, the limited applicability of one-sided RDMA limits the potential for performance benefit.

SmartNICs provide a path forward. These devices integrate compute cores and memory into the network interface, plus accelerators for common packet-processing functions. In particular, on-path ("bump in the wire") SmartNICs offer flexible compute cores on the packet data path, suggesting a new approach to hardware-accelerated distributed transactions. The SmartNIC's cores enable remote data structure accesses without network or RPC overhead. The on-board DRAM allows for maintenance of metadata state on the NIC itself, eliminating unnecessary PCIe memory accesses. When PCIe DMAs are required, the NIC can batch these operations to reduce overhead. Finally, the NIC can handle arbitrary protocol logic at both the source and remote target of a request, implementing flexible, multi-hop, network communication.

Given these potential benefits, we conduct a performance characterization of SmartNIC packet processing to identify the challenges and opportunities of using SmartNICs to accelerate distributed systems protocols. We find that the SmartNIC's software-based packet processing comes at a performance cost relative to hardware-supported RDMA. Therefore, a SmartNIC solution would only be effective if the NIC programmability can be used to significantly optimize communications over the wire and PCIe. Further, the NIC's limited resources pose a challenge. The NIC cores have low computational power relative to host cores, and the on-board memory is small. Careful placement of state and logic is critical to benefit from the NIC's limited resources. Operations must be interleaved and aggregated to effectively utilize NIC compute, PCIe, and network bandwidth.

Armed with these insights, we design Xenic, a SmartNIC-accelerated transaction processing system. Xenic adapts the protocol of prior designs to benefit from a stateful, asynchronous SmartNIC execution model. First, Xenic employs a co-designed data store that resides in host and SmartNIC DRAM, conforms to the SmartNIC's restrictions, and provides fast access to host-based data via indexing hints on the SmartNIC. Second, Xenic maintains temporary synchronization state on the SmartNIC to optimize concurrency control mechanisms. Third, Xenic takes advantage of

the SmartNIC's flexible communication primitives and a function shipping interface [13, 16] to implement multi-hop (i.e., non-request-response) distributed concurrency control optimizations that lead to lower latencies and higher throughputs. Finally, Xenic achieves communication efficiency by asynchronously aggregating work at all inputs and outputs of the SmartNIC. The batched, asynchronous execution model enables high utilization of network bandwidth and the PCIe DMA engine.

We implement Xenic using Marvell LiquidIO SmartNICs [64] and compare it to well-optimized RDMA- and RPC-based designs using Mellanox CX5 RDMA NICs [68]. Our evaluation focuses on the TPC-C, Retwis, and Smallbank transaction benchmarks. On a 100Gbps network, Xenic demonstrates a 2.42×, 2.07×, and 2.21× peak throughput increase relative to the best-performing RDMA and RPC alternatives, for the three respective benchmarks, with 59%, 42%, and 22% improvements in median latency, while saving 2.3, 8.1, and 10.1 threads per server.

4.1 Background & Related Work

4.1.1 RDMA NICs

Modern datacenter NICs commonly implement a hardware-accelerated remote memory interface known as RDMA. An application uses RDMA by registering regions of host DRAM with the local NIC to enable remote access. There are two categories of RDMA operations:

One-sided RDMA operations are simple memory manipulations that are handled fully by the RDMA NIC. The target server's NIC parses the request, issues a PCIe DMA to read or write the host memory region, and sends a response over the network. One-sided verbs utilize connection-based transport. Three one-sided RDMA verbs are supported by mainstream RDMA NICs: (a) READ a remote memory address, copying the requested data over the network, (b) WRITE data from a local buffer to a remote address, returning a completion ack, and (c) ATOMIC compare-and-swap or fetch-and-add a remote buffer, returning the result.

Two-sided RDMA provides an efficient send/receive interface for message passing. Two-sided operations involve the host CPU on both sending and receiving ends. On the receiving end,

the host CPU must poll for received messages, handle the buffer contents, and release the buffers to receive later messages. Two-sided RDMA offers a lightweight message-passing abstraction to support an RPC-based system but does not provide the CPU-bypass properties of one-sided RDMA.

4.1.2 Distributed Transactions

We target serializable distributed transactions over a replicated key-value store. The keyspace is partitioned, with designated primary and backup replicas for each partition. Recent work in this space assume persistent memory or battery-backed DRAM for fault tolerance and a separate service off the critical path to handle reconfiguration [16, 44, 113].

Commit Protocol

Recent research systems share a similar commit protocol design, extending optimistic concurrency control (OCC) [108] with primary-backup replication for availability [16, 44, 11, 113]. Each transaction consists of a set of keys to read and a set of key-value pairs to write. The coordinator issues a series of operations for each transaction:

1. In the EXECUTE phase, the coordinator reads all read-set objects from the objects' primary replicas. Writes are buffered locally at the coordinator, and the coordinator contacts the primary for each write-set key to lock the object. If a lock is already held, the transaction aborts.

2. In the VALIDATE phase, the coordinator again reads each read-set value from its primary. If any value has changed after being read in the execution phase (determined using version counters), or its lock is held, the transaction aborts. Otherwise, the transaction will commit.

3. In the LOG phase, the transaction record is written to a log on each write-set backup replica. The write-set updates are applied to the backup shards in the background.

4. In the COMMIT phase, the coordinator applies the new write-set values to the primary replicas, increments the objects' version counters, and unlocks the objects.

RDMA-assisted Distributed Transactions

Recent systems use RDMA to accelerate the commit protocol, both by using one-sided verbs to reduce host involvement and two-sided verbs to implement low-latency RPCs.

FaRM [16]: FaRM's design prioritizes the use of one-sided RDMA. In particular, FaRM applies a Hopscotch hash data structure, enabling remote key lookups with a single one-sided READ. The Hopscotch structure incurs a high bandwidth overhead, as a neighborhood of multiple objects must be read for a single lookup. Further, key insertion also requires displacement of existing objects, which cannot be done efficiently using one-sided RDMA primitives. Thus, FaRM can fully offload execution and validation phase reads using one-sided RDMA, but it consumes remote CPU cycles for all other operations. To acquire write locks, to log transaction records, and to commit write objects, FaRM applies an RPC protocol based on one-sided WRITEs to pairwise message logs on each server. The servers poll logs and handle requests, sending back responses using the same mechanism.

FaSST [44]: Instead of pursuing the CPU and latency savings of one-sided RDMA, FaSST implements a lightweight two-sided RPC protocol for all remote operations. With an RPC model, no specialized data structure is required since lookups and insertions occur locally at the RPC handler. This avoids the read amplification and insertion complexity of FaRM's hash structure. FaSST also consolidates multiple operations into a single RPC: one RPC can lock a write-set object and retrieve a read-set value, providing performance benefits at the cost of host core usage.

DrTM+R [11]: DrTM+R aims to handle all remote operations with one-sided RDMA. This is accomplished with separate locking schemes for local and distributed transactions: remote locking uses one-sided ATOMIC operations, and local locking uses hardware transactional memory (HTM). DrTM+R addresses the incompatibility of RDMA ATOMICs with host CPU atomic instructions by applying an HTM procedure for each local key operation, and instead of optimistically reading and performing a validation check, the coordinator locks all keys in a transaction.

DrTM+H [113]: DrTM+H uses both one-sided RDMA and two-sided RPCs, performing a phase-by-phase selection of one-sided versus two-sided options to maximize performance. Like FaRM, DrTM+H uses one-sided RDMA to read remote records during the execution/validation phases and to write backup log entries. Writes during the execution and commit phases are done via RPCs. The RPC protocol makes use of two-sided RDMA, like FaSST, instead of FaRM's one-sided RDMA message logs. DrTM+H stores objects in a standard open hash table and achieves one-sided lookups in a single roundtrip by storing at each coordinator the remote memory address of each key, incurring a memory cost. Ultimately, DrTM+H exploits the CPU savings of one-sided RDMA to a limited extent while using two-sided RPCs for all other work. This selective use of one-sided RDMA shows a performance benefit relative to the purely two-sided alternative.

4.1.3 SmartNIC-based Systems

Emerging programmable NICs, or SmartNICs, represent another promising approach to reducing host processing overheads. By offloading computations onto a NIC-side multi-core processor [67, 8, 64, 85, 76] or an FPGA [21, 69, 1, 115, 18], we can not only save server CPU cores but also achieve lower request latency and higher overall energy efficiency.

A SmartNIC has become a cost-effective computing unit for stateful packet processing, as the programmable components require only a modest amount of chip logic. For example, the Pensando Elba chip devotes less than 30% of its die for sixteen 2.8GHz ARM cores and the accompanying memory controller [65, 86], with the majority of the chip logic consumed by flow processing engines (e.g., Broadcom's TruFlow [7], Mellanox's ASAP2 [66], and Pensando's P4 [86]). Moreover, the enclosed processors and ASIC-based accelerators consume much less power than a Xeon-based solution when performing line-rate traffic processing (e.g., the Pensando NIC consumes less than 25W [65]).

Thus, there is a growing body of research on SmartNICs, with a significant focus on the offloading of network functions [52, 21, 56, 26, 3, 91, 102, 101, 50, 33, 71]. There is also work on generic frameworks for offloading [45, 87, 61, 59] and individual case studies focused on accelerating specific applications (e.g., key-value storage offloads [51, 60]). Our work is along these

lines and examines the utility of SmartNICs in accelerating distributed transactions, an application that has traditionally been optimized using RDMA. Crucially, unlike prior efforts that focus on network functions or complete offloads of applications, we pursue a design where the NIC and the host are closely coupled, with shared data structures and a fine-grained division of application logic.

4.2 SmartNIC Performance Analysis

We perform an experimental characterization to identify the opportunities and challenges of using SmartNICs for distributed transactions. Building upon the experimental results in Chapter 3, we focus on SmartNICs that enclose a system-on-chip (SoC) multi-core processor. These SmartNICs offer the potential for hardware acceleration while, unlike RDMA, delivering a flexible, programmable interface. We provide a performance evaluation of the 2x50GbE Marvell LiquidIO 3 CN3380 SmartNIC. The LiquidIO has 24 ARMv8 cores running at 2.2GHz, 16GB of on-board DDR4 DRAM, and an 8-lane PCIe 3.0 interface. We compare the LiquidIO to the 100GbE Mellanox CX5 (MCX516A-CCAT) RDMA NIC. We detail our server specifications in §4.4.

The following sections, build upon the opportunities for performance gains discussed in Chapter 3. We focus specifically on applying LiquidIO 3 SmartNICs as an alternative to CX5 RDMA NICs. In Section 4.2.1, we compare latency of CX5 hardware-handled RDMA accesses, and software remote operations, targeting SmartNIC and host memory, implemented on the LiquidIO SmartNIC. In Sections 4.2.2 and 4.2.3, we compare RPC throughput between the LiquidIO cores and the host cores via the CX5 NIC, and evaluate the performance gains enabled by batching. Finally, in Sections 4.2.4, we measure the LiquidIO DMA engine performance, to understand its capability of facilitating offloaded host memory accesses.

4.2.1 NIC and Host Access Latency

In Figure 4.1, we present a roundtrip latency comparison of remote operations for the LiquidIO and CX5 NICs. For the LiquidIO, we measure operations initiated on the source host server via DPDK and operations initiated on the source NIC cores. For both sources, we measure the end-to-

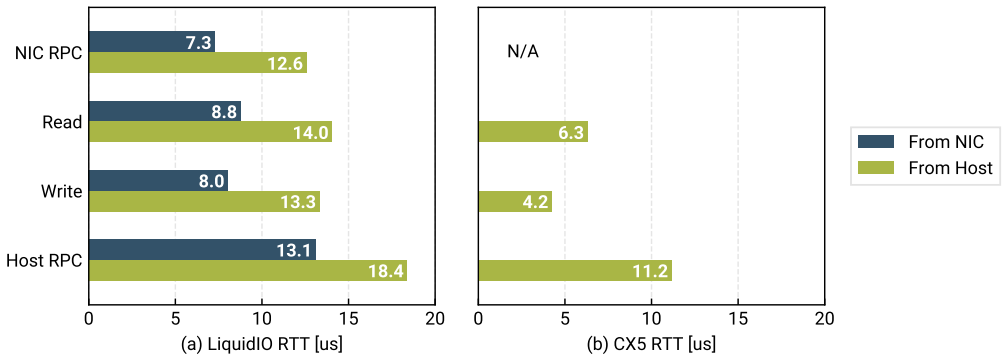

Figure 4.1: Roundtrip latency for LiquidIO SmartNIC remote operations originating from the host and from the NIC (a), and for CX5 RDMA (b). 256B data buffer used for all measurements.

end latency of operations executed on the remote NIC, such as a NOP (NIC RPC case), DMA reads and writes to host memory (Read/Write cases), as well as two-sided operations handled by DPDK on the target-side server (Host RPC). For RDMA, we perform comparable experiments with the CX5 NIC, demonstrating READ and WRITE verbs, as well as two-sided RPCs using SEND/RECV verbs with the RPC framework from DrTM+H [113]. All cases use a 256B data payload; latency is similar for smaller sizes.

Our measurements show a latency penalty for the SmartNIC software packet pipeline. RDMA operations, which leverage specialized hardware, demonstrate lower latency than the equivalent operations implemented in target-side LiquidIO and initiated from the host CPU. While the SmartNIC's software overhead poses a challenge, the ability to reduce costly PCIe operations presents an opportunity for performance improvement. Both devices show a significant cost for PCIe operations, with two-sided host RPCs incurring the highest latency. For the LiquidIO, operations local to the NIC cores (e.g., NIC RPC with NIC source) outperform all operations involving PCIe accesses. Operations initiated from the NIC, avoiding the latency cost of a host-side DPDK roundtrip, outperform two-sided RDMA RPCs. These observations suggest the LiquidIO has the potential for latency improvement over two-sided RDMA without sacrificing the software flexibility of RPCs.

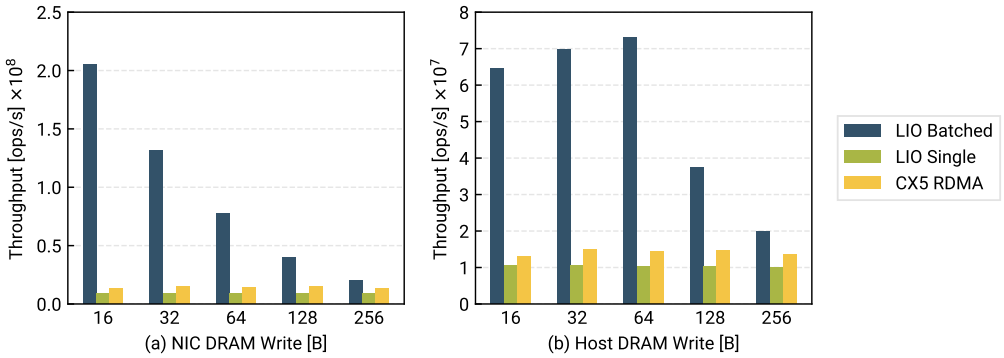

Figure 4.2: Remote memory write throughput, targeting SmartNIC DRAM (a) and host DRAM (b), with and without batching enabled. CX5 RDMA WRITE throughput is also shown for comparison.

4.2.2 NIC and Host RPC Throughput

To compare packet-handling throughput between NIC and host cores, we implement a minimal echo RPC handler in a host DPDK application and in the LiquidIO firmware. For this experiment, RPC requests and responses consist of 80B UDP packets. We send requests from 5 remote servers to the target server and measure total response throughput. First, we deploy the host RPC handler on the target server, using 16 threads with dedicated RX/TX queues (enough to reach maximum throughput) and a packet burst size of 64. Second, we repeat the experiment with the same configuration but instead deploying the RPC handler on 16 NIC threads. In both cases, further increasing the thread count did not increase throughput. We measure an average host RPC throughput of 23.0Mops/s and an average NIC RPC throughput of 71.8Mops/s. This suggests that the NIC cores, though wimpier in computational performance (see §4.2.5), demonstrate higher packet-handling efficiency than the host-side alternative. Handling RPCs on the NIC cores, therefore, creates the potential for throughput improvement, in addition to latency reduction, relative to host RPCs.

4.2.3 Batching Optimizations

Using read and write microbenchmarks, we consider the potential of aggregation and batching at all stages of the packet pipeline. We apply a software batching layer at the NIC's PCIe TX/RX

queues, Ethernet packet output, and DMA engine. We measure throughput for remote DMA writes to host memory and remote writes to the LiquidIO's on-board memory, with and without batching. To compare against the CX5, we measure RDMA WRITE throughput, applying doorbell batching [70] of up to 64 requests (more batching did not increase throughput). Figure 4.2 shows throughput for remote memory writes at a range of 16-256B buffer sizes, with and without batching. Reads demonstrate similar performance. We find that batching enables efficient bandwidth utilization for small remote memory operations. With batching disabled, throughput is consistent across the range of buffer sizes, 9.0-10.4Mops/s for both NIC and host memory targets. Batching network and PCIe transfers results in a throughput increase of up to 22.2× for NIC memory writes and 7.0× for host memory writes. For operations on remote NIC memory, throughput scales to the usable network bandwidth for all write sizes. For operations on host memory, throughput is limited by the DMA engine for requests smaller than 64B; larger requests saturate the usable network bandwidth. For CX5 RDMA, we observe 13.5-15.0Mops/s across the range of buffer sizes, lower than the respective batched LiquidIO operations. 16-256B RDMA writes do not saturate network bandwidth, even with extensive doorbell batching. This indicates that application-level doorbell batching is insufficient to achieve high throughput with small RDMA operations.

4.2.4 DMA Performance

To understand the performance characteristics of the LiquidIO's DMA hardware, we measure DMA throughput (Figure 4.3a) and latency (Figure 4.3b) for singular and vectored host memory accesses at a range of sizes. The DMA engine provides 8 hardware request queues; we initiate DMAs on 8 NIC cores, with each core assigned a dedicated queue. The DMA engine supports vectors of up to 15 reads or writes; we measure with individual requests and full vectors.

Our throughput measurements indicate that using vectored submission to batch DMAs improves throughput for the range of request sizes, up to the hardware maximum of 8.7Mops/s. Full vectors do not increase submission or completion latency relative to single-buffer requests. Instead, vectored operations may amortize the request submission time, up to 190ns, across up to 15 memory operations. Finally, we observe that the significant DMA completion latency, typically

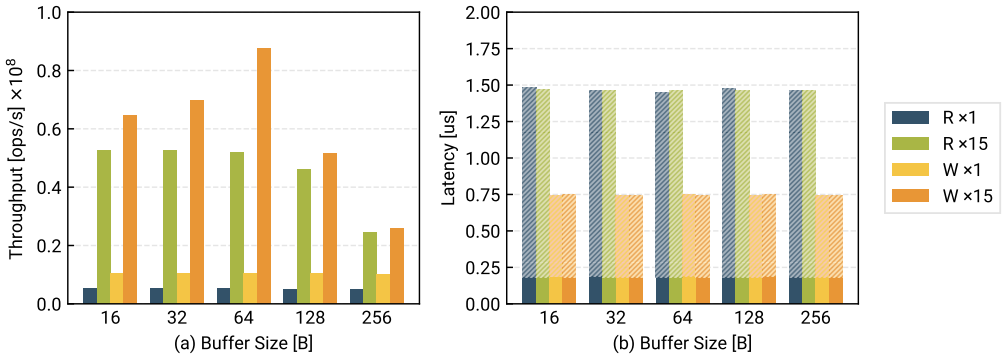

Figure 4.3: DMA engine throughput (a), and latency (b), with individual requests and with full 15-element vectors. For latency, solid bars denote submission time and hatched bars denote completion time.

Benchmark	Cores	ARM	Xeon	×
Coremark	multi	4530	14771	3.3
DPDK `hash_perf`	multi	349.8s	108.1s	3.2
DPDK `readwrite_lf_perf`	multi	179.6s	52.5s	3.4
Coremark	single	14294	29193	2.0
DPDK `memcpy_perf`	single	325.8s	174.4s	2.0
DPDK `rand_perf`	single	7.5s	2.9s	2.6
DPDK `hash_perf`	single	186.5s	84.0s	2.2

Table 4.1: Benchmark results for the NIC ARM and host Xeon cores, with relative the per-thread performance for the Xeon versus ARM.

up to 1295ns for reads and 570ns for writes, must be hidden to efficiently utilize the NIC cores.

4.2.5 SmartNIC Core Performance

We compare the performance of the ARM and Intel Xeon Gold 5218 CPU cores using the Coremark benchmark [17] and relevant performance tests in DPDK's test suite. We measure single-threaded performance and per-thread performance for workloads utilizing all cores. Table 4.1 shows the results. For the LiquidIO's 2.2GHz 24-thread ARM CPU, we measure a Coremark throughput score of 108724 or 4530 per thread. The host-side 2.3GHz, 32-thread Xeon's score is 472691, with per-thread throughput 3.26× higher than that of the LiquidIO. While the Xeon's

throughput scales with its 32 threads, the LiquidIO's per-thread Coremark throughput is substantially lower with all cores active. Running Coremark on one thread of each CPU, we observe higher relative throughput on the ARM, with a smaller 2.04× difference. The DPDK tests, demonstrating hash table, random number, and memcpy workloads, show a similar single-threaded (1.99× to 2.60×) and multi-threaded (3.24× to 3.42×) performance difference.

4.2.6 Opportunities

Our measurements suggest that the SmartNIC's software-based packet processing comes at a latency cost relative to RDMA. Despite this, we identify three optimization opportunities. The first is using the SmartNIC cores for stateful remote operations without host RPC overhead or one-sided RDMA limitations. NIC cores can handle protocol logic with the flexibility of an RPC design. NIC cores are also a valuable target for function shipping; logic can be pushed to NIC cores to eliminate PCIe roundtrips, exploit low-latency NIC-to-NIC communication (§4.2.1), and efficient packet-handling (§4.2.2). The second is using the SmartNIC memory to serve remote operations without PCIe overhead. With co-designed data structures spread across the host and NIC, we can use NIC memory to avoid PCIe latency. We can use PCIe DMAs to access host memory with lower latency than RPCs (§4.2.1), and high throughput potential relative to one-sided RDMA (§4.2.3). The third is leveraging the SmartNIC's efficient hardware interfaces, which show high throughput with software-defined asynchronous (§4.2.4), batched (§4.2.3) operations.

4.3 Design

Xenic provides a distributed, replicated database in server DRAM with a transactional interface. Each node acts as a transaction coordinator, a primary replica of one database shard, and a backup replica for f other shards, if we use a replication factor of $f + 1$. A coordinator application, running on each node, initiates transactions. The commit protocol utilizes the coordinator's local ("coordinator-side") SmartNIC, and the ("server-side") SmartNICs at remote primary and backup nodes. Xenic is designed to benefit from SmartNICs in the following ways:

- *Stateful offloads:* Xenic implements its transaction commit protocol as a set of stateful operations on the coordinator- and server-side SmartNICs. By storing temporary transaction state, e.g., locks, in SmartNIC memory, Xenic avoids PCIe roundtrips and host RPCs on the critical path.

- *Co-designed data store:* Xenic's data structures are spread across the host and SmartNIC memory. All key-value objects are stored in server DRAM, supporting local memory access at the server. For remote access, Xenic utilizes server-side SmartNIC memory to avoid PCIe reads for hot objects. By storing lightweight location metadata on the distribution of objects in host memory, Xenic can minimize the latency and size of DMAs for cold objects.

- *Distributed multi-hop OCC protocol:* Like prior systems [15, 16], Xenic uses function shipping [13]. But, Xenic can target SmartNICs and employ non-request-response protocols, unlike RDMA-based systems. This not only reduces PCIe operations but also enables flexible OCC communication protocols that reduce network communication.

- *Runtime support for asynchronous and batched communication:* Xenic performs all work asynchronously and aggregates operations at all inputs and outputs. By implementing an asynchronous, batched operation model, Xenic efficiently utilizes the limited SmartNIC cores. By batching work across PCIe DMAs, packet IO, and Ethernet transmissions, Xenic achieves high bandwidth utilization.

4.3.1 Co-designed Data Store

Xenic's data store is a co-designed hash structure residing in host and SmartNIC DRAM. All key-value objects are stored in host memory. The following factors drive this design choice. First, the host application may retrieve objects via local memory access, not requiring communication with the SmartNIC. Second, the host memory size is typically much larger than that of the SmartNIC memory. Finally, the host's memory can be battery-backed to provide durability (as is the case in

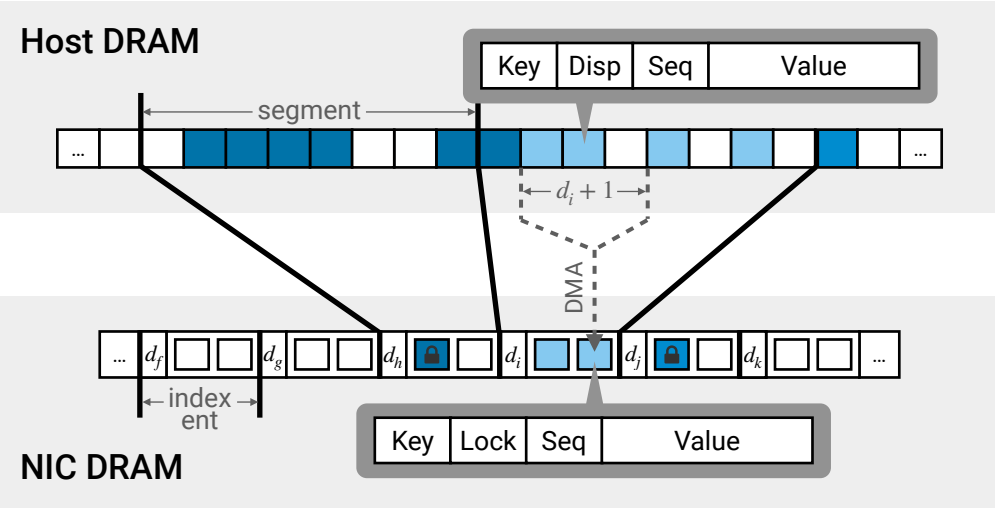

Figure 4.4: Overview of the Xenic data store, showing data and metadata placement. Overflow is omitted for simplicity.

FaRM [16]). We optimize the host-side structure for efficient lookups and reads from the SmartNIC via PCIe DMA. Local transactions' lookups, insertions, and writes are performed on the host via local memory access.

The SmartNIC hash structure serves as a caching index of the host data store. It maintains fine-grained distribution metadata for regions of the host hash structure, enabling low-cost lookups via PCIe DMA. Given the latency of PCIe, we target lookups with a common-case single DMA read and low bandwidth overhead. In addition to storing lookup metadata, the SmartNIC structure maintains transaction metadata for key-value objects accessed by ongoing transactions.

This design leads to three possible cases for lookups: first, the host can perform lookups in its local memory; second, the SmartNIC can serve remote lookups of hot objects via its cache; third, upon a cache miss, the SmartNIC can retrieve objects in host memory via a low-overhead DMA read.

Data Structures

The Xenic data store applies three structures. The host-side hash table (see §4.3.1) contains all key-value objects. The NIC hash table (see §4.3.1) caches hot objects and stores metadata. It is not a complete index of the host hash structure but instead a cache with location hints to facilitate efficient DMA lookups on the host-side hash table. The NIC table also stores transaction commit metadata, e.g., lock state, for ongoing transactions. Placing this metadata in NIC memory brings it closer, in terms of latency, to inbound remote accesses, while the on-path NIC architecture keeps it on the data path of outbound local requests. Figure 4.4 shows the layout of the host and SmartNIC hash structures. Finally, a host memory log (see §4.3.2) stores recently committed transactions. The NIC efficiently appends transactions' write sets to the log, and the host applies the updates to the host-side structure off the critical path.

Robinhood Hash Table

Xenic's host-side hash structure is a closed hash table adopting the Robinhood hash table design [10], with several modifications to achieve efficient operations in the SmartNIC context. The Robinhood design is a form of closed hash table applying linear probing, which aims to reduce the cost of lookup probing by displacing existing objects as new ones are inserted. The insertion procedure attempts to even out the *displacement* of objects in the table: the distance of each object from its initial hash position. Objects with a comparatively low displacement are moved further as later insertions take place. The insertion probing function accomplishes this by checking the displacement of each element it reaches in the table. If the existing element's displacement is less than the current displacement of the element to be inserted, it swaps the existing element with the one to be inserted; thus, it steals the "displacement wealth" from well-placed elements and hands it out to other elements. Probing continues until reaching an empty slot, where the element to be inserted is placed.

This swapping procedure results in low probing distance variance throughout the hash table, even at high occupancy. While the insertion cost is higher than simply probing for an empty

slot, the uniformity of probing distance improves lookup efficiency. This is important for lookups in the context of high-latency, throughput-limited memory access, i.e., PCIe DMA. Unlike other swapping designs, such as Cuckoo hashing, the Robinhood design prioritizes the locality of objects mapping to the same hash position. As a result, typical lookups read a single, contiguous region of memory instead of multiple disjoint buckets. This is crucial in the context of remote memory access, where the initiation cost of reading disjoint addresses is higher than that of a single buffer.

Xenic imposes a global limit on *maximum displacement*, D_m. Xenic divides the table memory into fixed-size *segments*, and for each segment, a linked overflow bucket may be allocated if necessary. If displacement reaches D_m during insertion, the object to be inserted is instead appended to the overflow bucket corresponding to its initial hash position. This allows Xenic to limit the cost of insertion and deletion. While prior Robinhood implementations typically apply tombstones to ensure an erased entry does not prematurely end probing, Xenic uses a simpler approach. Xenic simply swaps an overflow element over the deleted element, if one exists. If no overflow element exists, Xenic performs a backward shift; size is limited by D_m.

DMA-Consistent Swapping When Robinhood insertion swaps a table element, the existing element is replaced with the object to be inserted and buffered until the next swap occurs. A concurrent DMA read could therefore miss the existing element. Xenic addresses this by building a copy list and performing swaps starting from the last (free) element. This ensures an existing object is never removed from the table. Xenic must also guarantee consistent DMA reads for objects spanning multiple host cache lines. In this case, Xenic surrounds swaps with transactional memory instructions (`XBEGIN`, `XEND`), causing the swap to abort and retry if there is a concurrent DMA. Because the NIC caches objects returned in a DMA read, the host retries an aborted swap without continued contention. Xenic stores large objects above 256B outside the host hash table to avoid swapping large object payloads and reduce DMA lookup cost. Instead, the hash table contains pointers, which the NIC can use to retrieve the value via a single-object DMA read.

SmartNIC Caching Index

Xenic uses NIC memory to maintain lookup metadata for the host-side hash table, as well as transaction metadata for objects actively involved in transactions. For each segment of the host-side table, Xenic allocates a NIC *index entry*. An index entry contains a cache of objects that map to the corresponding host-side segment, transaction metadata (lock, version number) for those objects, and a *known displacement* value, d_i, for objects mapping to the corresponding host-side segment. Xenic implements a fixed-size set of cache positions for each index entry, with chained overflow pages allocated as necessary.

The NIC index also enables efficient host memory lookups when a cache miss occurs. Each NIC index entry maintains the highest known displacement d_i of objects mapping to the entry's host-side segment as well as an overflow address if objects in the segment have reached the displacement limit. Lookups require reading a region of the table in host memory, from the key's initial hash position to its actual displacement. While this actual displacement value is unknown until reaching the key, d_i serves as an effective location hint for locating a key with a single DMA read.

The NIC's d_i values may be invalidated by concurrent insertions performed in host memory: inserting one object may move another object beyond the corresponding d_i maintained at the NIC. To address this, the NIC reads $d_i + k$ additional elements beyond its known displacement, up to the limit D_m. While insertions will invalidate d_i values somewhat regularly (e.g., 6% of insertions at 90% occupancy), d_i is rarely increased by more than one (e.g., only 0.2% of insertions at 90% occupancy); therefore, we set $k = 1$ based on experimentation. If the NIC does not read the item within $d_i + k$ entries, the NIC performs a second, adjacent DMA read up to the limit, D_m. If d_i is already equal to D_m, the NIC instead reads the segment's overflow page.

Insertions, deletions, and cache eviction make use of the transaction protocol and its metadata to ensure consistency between the SmartNIC index and the host structure (§4.3.2).

Data Structure	Objects Read	Roundtrips
Xenic Robinhood, $D_m = 8$	3.43	1.07
Xenic Robinhood, $D_m = 16$	4.13	1.04
Xenic Robinhood, $D_m = 32$	4.84	1.02
Xenic Robinhood, no limit	6.39	1
FaRM Hopscotch, $H = 8$ [15]	> 8	1.04
DrTM+H Chained, $B = 4$	4.65	1.16
DrTM+H Chained, $B = 8$	8.81	1.10
DrTM+H Chained, $B = 16$	16.96	1.06

Table 4.2: Average number of objects read and number of roundtrips per lookup, at 90% occupancy.

Lookup Efficiency

We compare the efficiency of lookup operations to the hash table designs of FaRM and DrTM+H. The three designs share similar priorities; each is optimized for remote hash lookups via remote memory access. All three designs perform updates using local memory operations at the target, and their insertion procedures prioritize placing objects mapping to the same hash value within a small, contiguous area of memory. This enables common-case remote lookups with one remote memory read at the cost of reading multiple objects per lookup. FaRM applies a Hopscotch hash table; like Robinhood, the Hopscotch table is a variant of linear probing. This design ensures that any element must be located within a fixed neighborhood size H, with $H = 8$ in FaRM's published results. A remote lookup first reads the neighborhood of H elements, and if the object is not found, issues a second read of the corresponding overflow bucket, resulting in an additional roundtrip. DrTM+H applies a simpler hash design, with a closed array of B-element fixed-size buckets and additional linked buckets allocated as necessary. A remote lookup traverses bucket links until finding the object.

We measure remote lookup performance at 90% table occupancy, comparing to FaRM's published results at the same occupancy. Table 4.2 shows the mean number of objects read and mean roundtrips per lookup for 8 million uniform-random keys. FaRM's design reads $H = 8$ objects per lookup in the common case, with a second roundtrip necessary for 4% of keys; average overflow

read size is not published. DrTM+H reads at least B keys for each lookup, and due to its chained placement policy, often incurs multiple roundtrips traverse the chain. Xenic dynamically bounds the size of lookup reads based on hints stored in the NIC index. At high occupancy, and with a similar 4% overflow utilization to FaRM, Xenic's average read size is 48% lower than that of FaRM. If we disable overflow buckets and do not limit displacement, Xenic still achieves 20% fewer object reads per lookup than FaRM while also eliminating the overflow roundtrip.

FaRM and DrTM+H both perform remote hash lookups across the network via RDMA. Xenic instead performs remote lookups at the target-side NIC. RDMA lookups impose an end-to-end bandwidth cost and a full network roundtrip penalty if multiple reads are needed. Xenic's lookups, in contrast, consume only PCIe bandwidth and incur only PCIe access latency. Likewise, Xenic's remote lookups are always performed at the target NIC rather than at any remote client. This creates the opportunity for the NIC to cache objects and metadata for the host-side structure. While DrTM+H also applies index caching, it must store remote object addresses for each remote primary at each coordinator. DrTM+H's approach is limited in scalability, given its memory overhead, and lacks an efficient mechanism for cache invalidation.

4.3.2 Transaction Protocol

We first describe Xenic's distributed transaction commit procedure, then we detail special transaction cases and Xenic's respective optimizations. Xenic applies function shipping to offload execution logic from the host server to the coordinator-side SmartNIC (§4.3.2). Further, Xenic extends OCC with distributed, multi-hop protocol variants to increase communication efficiency based on a transaction's access pattern (§4.3.2). While Xenic targets distributed transaction workloads, where common-case transactions involve remote data shards, we ensure the SmartNIC optimizations do not add communication complexity to purely local transactions. §4.3.2 outlines Xenic's fast path for local transactions.

Figure 4.5 shows the components of each Xenic node. The coordinator application, running on each host, initiates all transactions and handles commits and aborts. We summarize the execution of a distributed, read-write transaction:

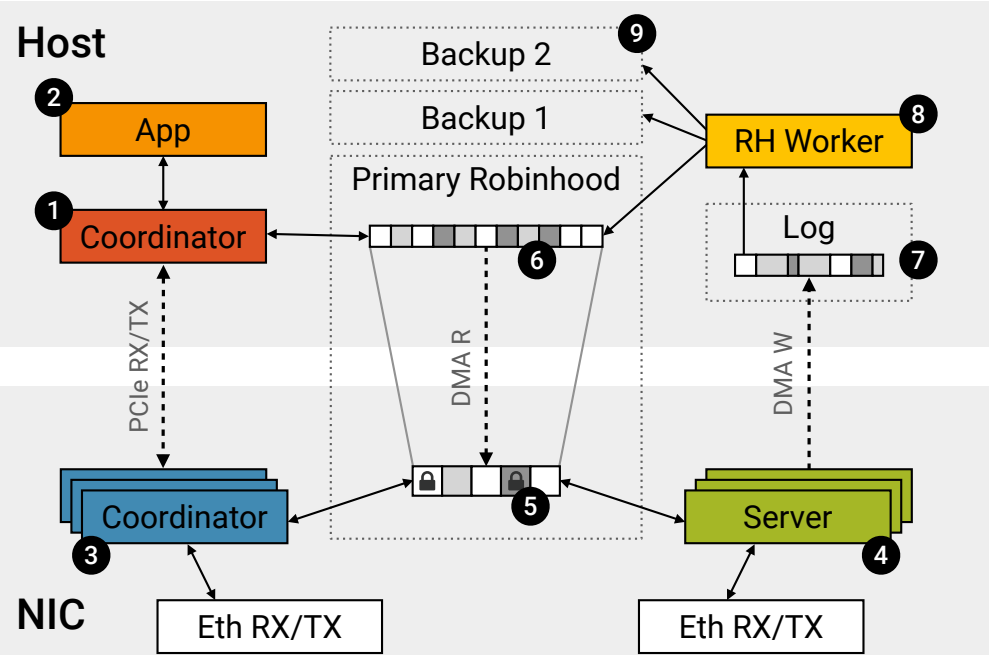

Figure 4.5: Xenic design overview, showing one server. Solid lines indicate local memory access; dashed lines indicate PCIe transfers.

1. A host coordinator thread (A) initiates the transaction, determining the initial read-set and write-set objects. The coordinator may either generate the transaction or poll a request queue from an external application thread (B). It assigns a transaction ID (node index and sequence number), then sends the transaction state, including its read-set and write-set objects, to its local SmartNIC.

2. A coordinator-side SmartNIC thread (C) then issues remote EXECUTE requests to each primary involved, specifying the shard's read-set and write-set keys. The request is received by a server-side SmartNIC thread (D), which performs a lookup for each read- and write-set key in its local-memory index (i). If any key exists and is locked, the NIC returns an *abort* response. Otherwise, the NIC allocates an index entry, if necessary, and acquires a lock for each write-set key. The NIC then retrieves the values and version numbers of the read-set keys. As described in §4.3.1, cached values are retrieved from SmartNIC memory

i, and cache-miss reads retrieve the value from host memory ii via PCIe DMA. Finally, the coordinator sends a response containing the read-set values and versions.

3. After receiving successful EXECUTE responses from all primaries, the coordinator SmartNIC updates its transaction state with the returned read-set values and sends the transaction state via PCIe to the host. The host coordinator performs an application-level function to generate write-set values given the read-set values and sends these writes to its NIC. For a multi-shot transaction, the coordinator may issue subsequent execute requests to read and/or lock additional keys until execution is finished.

4. The coordinator-side SmartNIC issues a VALIDATE request to the primary of each read-set key, except for those locked for writing. The request includes the version number for each key obtained by EXECUTE. The primary NIC retrieves the current version for each key in its index i, and returns *commit* if the version numbers match and no keys are locked. Otherwise, the NIC returns *abort*, which is propagated to the application and other primaries.

5. After receiving successful VALIDATE responses, the transaction completes if it is read-only. For read-write transactions, the coordinator replicates the write set to each shard's backup replicas using a LOG request with the shard's key-values and version numbers. The NIC for each backup handles the LOG request by appending it to a hugepage of host memory reserved for logging iii via DMA write. The NIC responds after the DMA completes.

6. After receiving all LOG responses, the coordinator-side NIC reports a *Committed* outcome to the host, and issues a COMMIT request to each primary, with write-set key-values and version numbers. The primary NIC appends the COMMIT request to the host-memory log iii. Then, it applies the new values to cached entries in the index and updates the write version numbers. Once the DMA completes, the NIC releases the write-set locks and sends an ack response. The write-set objects are pinned in the NIC's index cache and cannot yet

be evicted. This ensures NIC lookups will not read a stale object before the host applies the COMMIT writes to its hash structure.

7. The host-side Robinhood worker threads (E) poll the log for entries written by the NIC. The host threads asynchronously handle requests by applying LOG write sets to the backup shards (iv) in host memory and COMMIT write sets to the primary shard (ii). The host application appends a log ack to traffic between the host and the NIC, allowing the NIC to reclaim log space and unpin cache entries for committed writes.

Fault Tolerance

Xenic applies the reconfiguration and recovery design of FaRM without additional requirements. To do so, we ensure that (a) lock state is maintained in only one location (SmartNIC memory) and rebuilt upon recovery, (b) Xenic's host-side hash table maintains the same set of objects as that of a static hash table, and (c) operations are executed in the same sequence across the coordinator, primaries, and backups as in FaRM's protocol, with log records written to host memory before a LOG operation or COMMIT operation returns an acknowledgement.

Given these similarities, FaRM's recovery protocol applies to Xenic as follows. Xenic uses a typical Zookeeper-based cluster manager to determine membership. Each node holds a lease with the cluster manager, and lease expiration triggers reconfiguration. Only primaries maintain lock state, so when a primary fails, a backup is promoted to become the new primary, and the lock state is reconstructed. While other shards may proceed, each node of the recovering shard scans its log for transactions that have not yet been acknowledged as committed to the primary. These recovering transactions' write-set keys are communicated to the new primary, which acquires locks on each object. Once all locks are set, the shard can serve new transactions. Meanwhile, the replicas communicate to ensure each recovering transaction is either aborted or fully applied to all replicas before its associated locks are finally released.

SmartNIC Function Shipping

Offloading execution logic from the host to the coordinator-side NIC provides an opportunity for latency reduction, eliminating all but one coordinator PCIe roundtrip. We apply function shipping [13, 15, 16]; while FaRM used it between hosts, we use it to move execution from the host to the NIC. Xenic implements function shipping by adding an optional, application-defined data field to each transaction state entry maintained on the NIC. This data consists of the application's *external state*, if any, required for a transaction's execution. Second, Xenic provides an abstract interface for execution logic. This interface exposes the transaction's read and write sets and the external state associated with the transaction. When a transaction request is initially sent from the host to its coordinator-side NIC, any external state data is attached to the request and buffered at the NIC. When the transaction's EXECUTE responses are received by the coordinator-side NIC, the execution logic is invoked, transforming the transaction's read and write sets based on the current objects and external state. If execution adds keys to the transaction, coordinator issues EXECUTE requests for the new keys, collects responses, and repeats the execution function. Otherwise, the coordinator proceeds to commit the transaction. Offloading execution requires performing execution logic on the NIC and sending associated application state to the NIC, potentially incurring additional NIC CPU load and PCIe bandwidth utilization. Offloading execution is feasible only when the object manipulation is not computationally intensive and the application state is small, i.e., when it does not introduce the NIC cores or PCIe bandwidth as a performance bottleneck.

Multi-Hop OCC Communication

Xenic additionally applies function shipping to reduce commit protocol network communication. By leveraging point-to-point operations between NICs, in addition to the standard coordinator-server pattern, Xenic can reduce commit messages and message delays based on a transaction's access pattern. When the coordinator-side NIC receives a transaction request, it determines the optimal execution node based on any remote accesses in the read and write sets. If commit com-

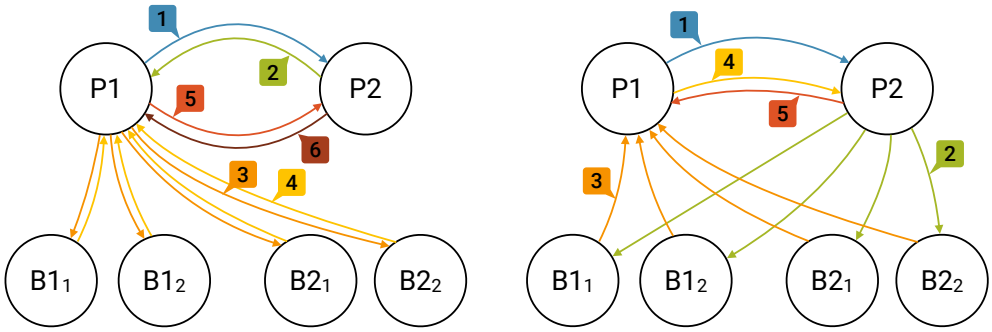

Figure 4.6: Commit messages for a transaction writing to local and remote shards, with execution (a) performed at the coordinator *P1*, and (b) shipped to remote *P2* to minimize communication.

munication can be simplified by performing execution at a remote primary NIC, Xenic applies function shipping to invoke execution remotely and uses multi-hop requests to reduce communication. For instance, transactions writing to the local shard and one remote shard are executed at the remote primary NIC. Figure 4.6 shows communication with coordinator execution and only request-response operations (a) and the optimized communication pattern enabled by shipping execution to the remote primary NIC (b). In the optimized case, the *P2* NIC performs execution, then issues Log requests to the backup NICs, and the backups send Log responses to the coordinator-side *P1* NIC. By shipping execution to the remote primary NIC, Xenic eliminates a network message delay from the commit protocol. Xenic handles remote execution with the same function shipping mechanism as in the coordinator offload optimization. We limit remote execution to transactions involving a single execution round, where all keys are specified in the initial request. We implement multi-hop commit operations for all single-shard transactions and transactions involving the local shard and one remote shard.

Local Transactions

Local write transactions execute optimistically on the host, accessing objects in the host-side hash structure. After execution, the host sends the transaction state to its coordinator-side SmartNIC for replication. Before issuing Log requests, the coordinator-side NIC acquires write-set locks

in its index and aborts if any lock is already held. Otherwise, the NIC proceeds with the commit protocol. This adds no network or PCIe overhead for committed transactions. Local read transactions require no PCIe communication, performing reads and VALIDATE logic locally at the host-side hash table.

4.3.3 SmartNIC Operations Framework

We apply our performance analysis to design an efficient framework for Xenic's SmartNIC commit operations. First, our results show that PCIe DMAs have significant submission and completion latency (§4.2.4). To achieve high core utilization, NIC cores must perform work while awaiting DMA completion. Second, we find substantial throughput opportunities in batching DMA submissions (§4.2.4). Submitting full vectors to the DMA engine amortizes submission cost, without adding completion latency, and increases maximum throughput. Combining batched DMA submission with batched Ethernet transmission (§4.2.3) results in high network utilization, with potential for improved throughput over one-sided RDMA. Individual commit operations, however, typically do not fill a 15-buffer DMA vector or an Ethernet MTU and do not have work to perform while awaiting DMA completion. For this reason, we must interleave commit operations and aggregate work at the point of DMA submissions, NIC-to-NIC, and NIC-to-host communications.

Asynchronous Operations

Xenic implements continuation-passing, asynchronous operations to interleave work, and to minimize blocking for DMA completions. Each NIC core maintains two vectors for pending read and write DMAs, respectively. Transaction operations insert entries (NIC/host addresses, size) into the read and write vectors, along with a callback function to be executed upon DMA completion. This callback may produce a network output, e.g., a LOG acknowledgement, or further manipulate NIC state, e.g., unlocking objects after COMMIT writes are transferred to host memory. When a NIC core is idle, or when the DMA vector fills, it is submitted to the core's assigned DMA engine. The DMA engine writes a completion status byte once it has performed the DMA. Each

core tracks in-flight DMAs using a core-local ring buffer, mapping completion byte addresses to the associated batch of callback work.

Opportunistic Batching

Xenic runs a burst-oriented polling loop on each NIC core, applying the NIC's hardware flow engine to route flows to cores. Each loop iteration handles a burst of Ethernet traffic and a burst of DMA completions, accumulating DMA requests and their callbacks in the pending read/write vectors. After handling the burst, the NIC submits any DMA requests and collects all outbound NIC-to-host and NIC-to-NIC packet transmissions. The NIC core uses a gather-list for each destination and performs an aggregated Ethernet or PCIe packet transmission. This allows Xenic to combine as many outputs as possible into each packet.

Xenic's batching approach allows the SmartNIC to aggregate communication whenever there is sufficient traffic between two nodes. PCIe communications are batched separately, and do not always achieve full batches; for instance, a read-heavy workload largely served by the SmartNIC cache results in few PCIe accesses. However, this scenario does not result in lower performance because cache hits to SmartNIC memory are lower-cost than DMA lookups.

Limited SmartNIC Resources

The SmartNIC's compute and memory capacities are small relative to the host server. Xenic is designed with this in mind, allowing workloads to appropriately utilize the SmartNIC. First, Xenic selectively applies function shipping to execute transactions on NIC cores. This is applied on a per-transaction basis via a user annotation. Doing so allows the NIC to execute latency-critical transactions, reducing PCIe crossings, while the host executes compute-heavy or predominantly local transactions. Second, Xenic uses SmartNIC memory to cache objects, adapting to available capacity. When caching is ineffective, due to the access pattern or cache eviction policy, the need for DMA lookups increases. These misses incur PCIe bandwidth overhead (§4.3.1), potentially becoming a bottleneck. Decreasing probing distance sufficiently, say by expanding the host-side hash memory, reduces PCIe overhead and allows lookups to reach network throughput.

Other SmartNIC Platforms

Xenic relies on SmartNIC hardware characteristics to reduce latency. First, handling a remote request on the SmartNIC must be lower latency than doing so with a host RPC. Some off-path SmartNICs demonstrated higher latency when directing traffic to the SmartNIC cores versus sending requests directly to the host with an RDMA NIC. Second, the SmartNIC must have an efficient mechanism for host memory access. SmartNICs that rely on an RDMA interface between the NIC cores and host memory showed prohibitively high latency, precluding Xenic's latency reduction goal. If the SmartNIC hardware does not show latency reduction potential, using SmartNICs may not be justifiable over a host-only design. In contrast, with a platform meeting these requirements, Xenic can improve both latency and throughput.

4.4 Evaluation

We implement Xenic using LiquidIO 3 SmartNICs. We extend the generic NIC firmware to add transaction-processing logic, written in C using DPDK and the LiquidIO hardware interfaces. The host-side coordinator also uses DPDK. Our testbed consists of 6 servers, each with Intel Xeon Gold 5218 CPUs (16 cores, 32 hyperthreads, 2.3GHz) and 96GB DDR4 DRAM. Each server contains a 2x50GbE Marvell LiquidIO 3 (CN3380), with 24 2.2GHz ARM cores, 16GB DDR4 DRAM, and a PCIe 3.0 x8 interface. We utilize both links of the NIC for a per-server total network bandwidth of 100Gbps. Each server also contains a 100GbE Mellanox CX5 (MCX516A-CCAT) NIC with a PCIe 3.0 x16 interface for comparison.

4.4.1 Comparisons

We evaluate Xenic with case studies of the TPC-C [107], Retwis [92, 117, 104], and Smallbank [28] benchmarks. We use each benchmark to compare performance against recent work in hardware-accelerated distributed transactions, measuring per-server average throughput and median latency. We focus on versions of DrTM+H [113] for this comparison, a well-optimized research system applying a hybrid of one-sided RDMA and two-sided RPCs to maximize performance. In

addition to its hybrid design, DrTM+H provides additional versions representing alternate decisions in the RDMA design space. We compare the following configurations:

- *DrTM+H* is the best-case combination of one-sided and two-sided operations for each protocol phase. One-sided operations are typically used for execution reads, validation, and logging. DrTM+H avoids remote data structure traversals by caching addresses for remote objects.

- *DrTM+H with no remote caching (NC)* matches DrTM+H but disables the coordinator's remote address cache. This configuration demonstrates the impact of RDMA hash traversal for Execute reads.

- *FaSST* involves two-sided RPC operations exclusively, emulating the design by Kalia *et al.* [44]. This version performs remote data structure lookups via host RPC, and where possible, consolidates multiple operations (e.g., reading and locking) into individual RPCs.

- *DrTM+R.* This configuration emulates DrTM+R's use of one-sided RDMA, retaining the OCC protocol of DrTM+H [113].

Of the open-source related work, only DrTM+H implements the TPC-C benchmark. However, DrTM+H's support is limited to a simplified version of the TPC-C workload, consisting of *new order* transactions, instead of the typical mix of five types, and using a customized access pattern. We evaluate Xenic using this workload for comparison with the DrTM+H configurations (§4.4.2). Xenic supports the full TPC-C workload, which we evaluate separately (§4.4.3). DrTM+H provides a Smallbank implementation; we migrate their code to Xenic and implement Retwis on both systems. For the three benchmarks, we discuss host and NIC resource utilization (§4.4.6). Finally, we evaluate key aspects of Xenic's design and their contributions to throughput and latency (§4.4.7).

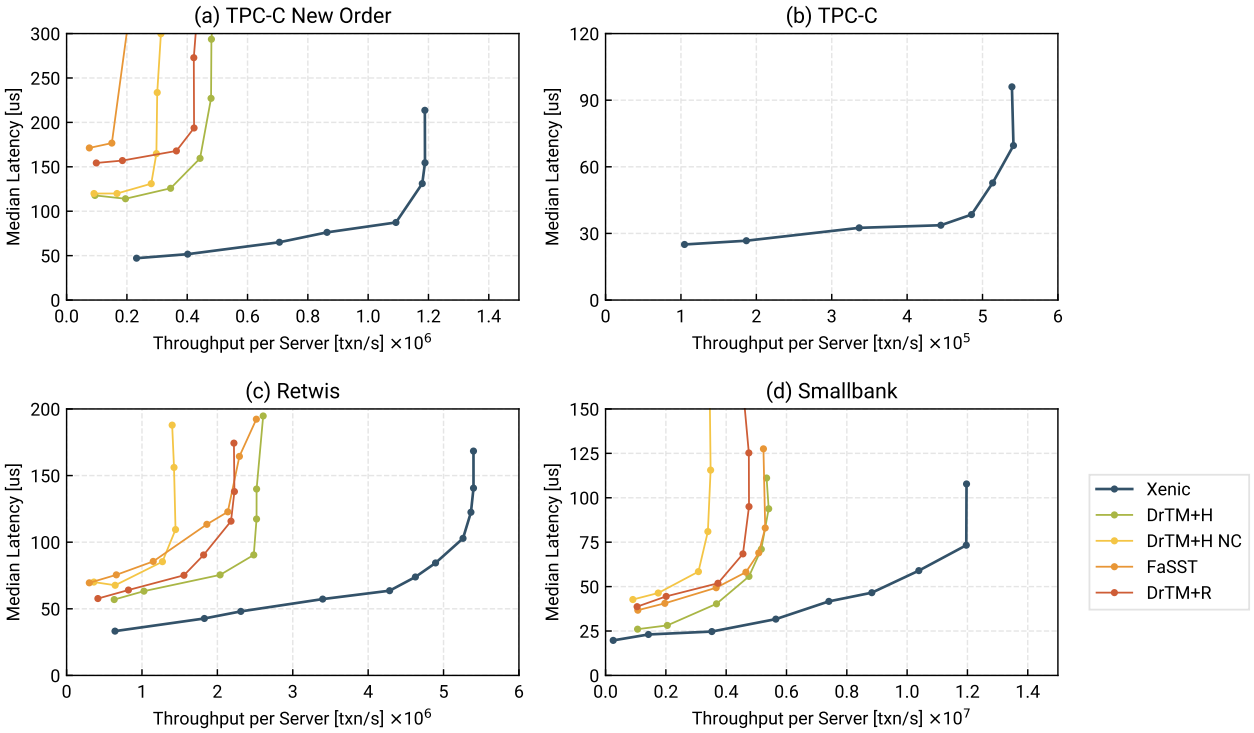

Figure 4.7: Throughput per server and median latency for (a) TPC-C New Order, (b) TPC-C, (c) Retwis, and (d) Smallbank benchmarks.

4.4.2 Case Study: TPC-C New Order

The TPC-C benchmark simulates a warehouse order processing system with nine tables and a range of object sizes up to 660B. We first evaluate the performance of TPC-C's *new order* transaction, the predominant transaction of the five types in the TPC-C specification. Because DrTM+H only supports the *new order* transaction, not the full workload, we use this benchmark to compare performance with DrTM+H and evaluate the full workload mix in §4.4.3. Each *new order* selects 5-15 items, updates stock counts, and writes order line-item records. The coordinator picks items from partitions chosen uniformly at random; this matches the DrTM+H authors' evaluation, creating a strenuous remote access pattern. Three of the tables are accessed by transactions across the cluster, while the others are B+ trees local to their respective coordinators; all tables are replicated. We deploy TPC-C on the 6-server testbed with a replication factor of 3 (2 backups for each primary) at the scale of 72 warehouses per server. Figure 4.7a shows the results.

Xenic achieves an average peak throughput of 1.19M txn/s per server, a 2.42× improvement over DrTM+H, the best alternative. While both systems saturate network bandwidth, DrTM+H requires multiple network operations for each TPC-C stock object to retrieve the value, then lock and validate. Xenic can lock and read a remote object in one remote operation, reducing bandwidth consumption and latency. Xenic effectively aggregates work at the SmartNIC, further allowing throughput to scale. Xenic's throughput is 3.81× greater than DrTM+H with coordinator-side caching disabled, showing the overhead of DrTM+H's remote lookups. Although RPCs avoid these one-sided RDMA inefficiencies, handling all operations with host RPCs limits FaSST's throughput to 232k txn/s, even when utilizing all host threads.

At low load, Xenic's median latency is 59% below that of DrTM+H, the lowest-latency alternative. While DrTM+H applies one-sided RDMA for reads, this requires separate remote operations to read, lock, and validate a remote object, limiting latency savings. Xenic can perform these functions with a single remote request while reducing latency relative to a host RPC. The latency penalty of FaSST's RPC approach is high for this benchmark since FaSST handles RPCs on the same threads performing compute-intensive B+ tree operations. At 95% of peak throughput,

FaSST shows high latency: 2.2× that of DrTM+H and 4.0× that of Xenic.

4.4.3 Case Study: TPC-C

The full TPC-C workload consists of five transaction types, including *new order*. We deploy the full workload mix at the same scale as §4.4.2, configured to match the standard benchmark specification. Like prior implementations [114], we chop the long-running local transaction logic into multiple database transactions. Xenic ships execution of the *new order* and *payment* transactions to the NIC; other transactions execute on the host. Per the specification, we measure throughput as the rate of *new order* transactions per second within the full workload mix; this is approximately 45% of the overall transaction throughput. Figure 4.7b shows the result.

Xenic achieves peak throughput of 541k new orders per second per server, saturating the network. With *new order* transactions comprising 45% of the workload, the other transactions consume bandwidth and limit throughput to approximately half that of the *new order* workload in §4.4.2. In the standard TPC-C configuration, only ~10% of *new order* and 15% of *payment* transactions access a remote warehouse's objects; this results in a median latency of 25μs at low load, below that of the modified *new order* workload.

Of the related work, DrTM+R and FaRM implement the full TPC-C workload. While neither system is open-source, DrTM+R's authors provide a throughput evaluation at the same scale as our testbed: 6 servers with 3-way replication [11]. With a 56Gbps network, DrTM+R's evaluation reports 150k new orders per second per server, fully utilizing the network bandwidth (a higher per-server throughput than FaRM). Because DrTM+R's throughput is limited by network bandwidth, we deploy Xenic with a similar network configuration to compare throughput with this published result. For Xenic, we use one 50Gbps link per server, instead of two, and run TPC-C at a scale of 384 warehouses to match DrTM+R. In this experiment, Xenic achieves a peak throughput of 322k new orders per second per server, 2.1× higher than DrTM+R. This is a smaller increase than Xenic's 2.7× improvement for the modified *new order* workload. The full TPC-C workload involves a higher frequency of local transactions, which only utilize the network for replication (Log operations). Xenic does not improve efficiency for these transactions relative to DrTM+R.

4.4.4 *Case Study: Retwis*

We evaluate the Retwis benchmark [92, 117], representing a Twitter-like application. The benchmark includes a mix of transaction types, with 50% read-only transactions and 1-10 keys per transaction. Unlike TPC-C, minimal coordinator-side computation is involved in performing transactions. Relative to Smallbank, objects are moderately larger (64B versus 4B values), accessed with a Zipf distribution, $\alpha = 0.5$, with a higher proportion of read-only transactions. We deploy Retwis with a replication factor of 3 and 1 million keys per server. Figure 4.7c shows the results.

Xenic shows a 2.07× peak throughput increase relative to DrTM+H and 42% lower median latency at low load. As with TPC-C, both systems fully utilize network bandwidth, while Xenic achieves higher efficiency. DrTM+H's hybrid design improves the performance of Retwis' read-only transactions, but its use of one-sided RDMA multiplies the number of requests for read-write transactions. This imposes a throughput and latency cost; we evaluate this impact on Retwis throughput in §4.4.7. Given the minimal computation involved in the benchmark, FaSST nears the peak throughput of DrTM+H without fully utilizing the host CPU. However, its RPC design results in consistently higher latency, with a minimum median latency 2.12× higher than that of Xenic.

4.4.5 *Case Study: Smallbank*

The Smallbank benchmark represents simple transactions on a database of account balances, with small 12B objects. 15% of transactions are read-only, and the remainder involves additions and subtractions of balances, with up to 3 keys per transaction. 90% of transactions access 4% of keys, resulting in relatively low contention. We deploy Smallbank at a comparable scale to our related work: 2.4M accounts per server, with a replication factor of 3. Figure 4.7d shows the results.

We observe a peak throughput of 12.0M txn/s per server with Xenic, 2.21× the maximum throughput of DrTM+H. Both systems saturate network bandwidth at peak throughput. Xenic delivers throughput improvement through protocol and communication efficiency. Smallbank's

Benchmark	Xenic Norm. (Host, NIC)	DrTM+H	FaSST
TPC-C NO	21.7 (18, 12)	24	32
Retwis	9.9 (5, 16)	18	24
Smallbank	9.9 (5, 16)	20	28

Table 4.3: Normalized thread count, for Xenic, DrTM+H, and FaSST. NIC thread count is scaled by NIC/host Coremark score ratio.

workload of 12B key-value objects presents a significant opportunity for batching. Given the small object sizes, minimizing the metadata overhead of each remote request is especially critical for bandwidth efficiency. The software flexibility of Xenic's commit operations enables higher bandwidth utilization, and its aggregation of remote requests enables aggressive batching.

However, Smallbank's small remote operations also demonstrate the best-case latency potential of one-sided RDMA, and DrTM+H performs optimal one-sided READs due to its pointer cache. Xenic shows 21.5% lower minimum median latency than DrTM+H, achieving competitive performance by eliminating PCIe accesses, utilizing NIC memory for transaction metadata, and caching hot objects. As in the other benchmarks, Xenic's commit protocol requires fewer remote operations per key than that of DrTM+H. For most Smallbank transactions, Xenic reduces communication via function shipping; we evaluate this optimization in §4.4.7.

4.4.6 SmartNIC Resource Utilization

To study utilization, we measure the minimum number of cores to run each benchmark at peak throughput, with Xenic, DrTM+H, and FaSST. We run each benchmark and decrease thread count until throughput drops below 95% of its maximum. For Xenic, we repeat this analysis with NIC cores. Table 4.3 shows the result. We find that Xenic requires few host threads for Retwis and Smallbank: 2 application threads to initiate transactions and handle completions, and 3 worker threads to apply writes to the primary and backup tables. TPC-C, however, requires 18 host threads due to its compute-intensive local B+ tree manipulations, which are performed on both host application threads and worker threads (to apply updates to each backup). Smallbank and

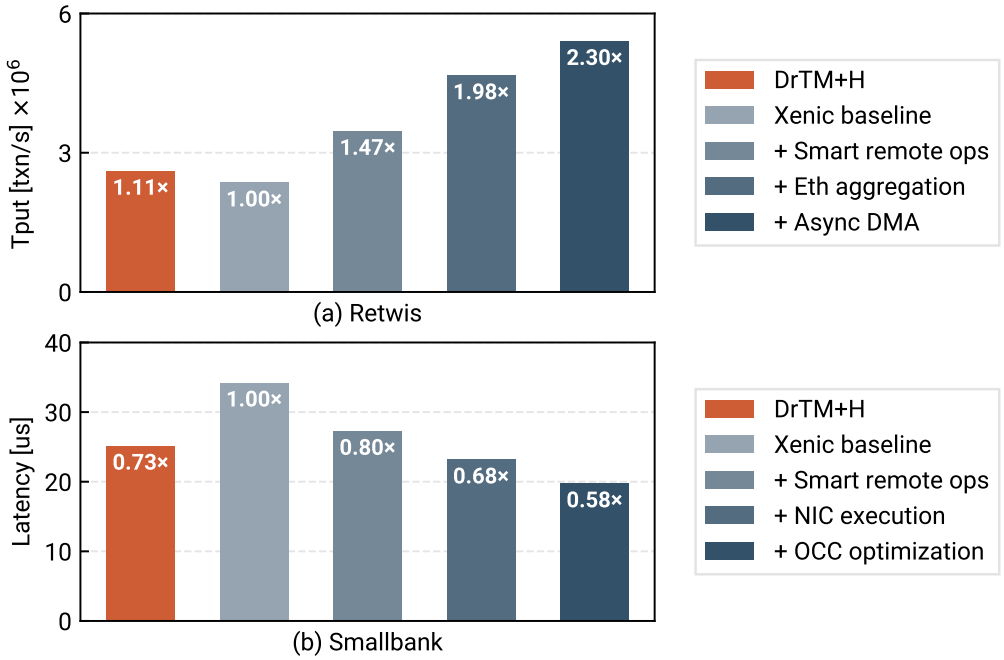

Figure 4.8: Retwis per-server throughput (a) and Smallbank median latency (b), sequentially enabling key aspects of Xenic's design.

Retwis offload all execution to the NIC, resulting in higher NIC utilization; TPC-C instead shows higher host utilization.

To compare cumulative utilization across host and NIC processors, we use the Coremark benchmark to normalize computation power. We use the ratio of the NIC's per-thread Coremark score to that of the host: 0.31×. This clearly is an approximation as the relative power is workload-dependent (§4.2.5). With this approximation, we report that relative to DrTM+H, Xenic saves 2.3 threads for TPC-C, 8.1 threads for Retwis, and 10.1 threads for Smallbank. In all cases, Xenic achieves higher throughput and core savings relative to FaSST and DrTM+H. Xenic's lower utilization suggests that exploiting wimpy NIC cores close to the NIC's hardware interfaces enables higher overall computation efficiency.

4.4.7 *Impact of Optimizations*

To evaluate how Xenic's design features contribute to improvements in throughput and latency, we begin with a baseline design and sequentially enable features. The Xenic baseline resembles DrTM+H, implementing the same set of remote operations. We impose the same restrictions that arise from DrTM+H's use of one-sided RDMA; in particular, we use separate requests to read, lock, and validate objects.

In Figure 4.8a, we enable a series of throughput-oriented optimizations and measure their impact on Retwis' throughput relative to the baseline and to DrTM+H. Despite their similar protocol, the Xenic baseline shows 10% lower throughput than DrTM+H. The NIC cores are saturated, with the NIC's software packet processing limiting throughput. Adding Xenic's optimized remote commit operations reduces the number of remote requests; this increases throughput by 1.47×. Adding aggregated Ethernet transmissions facilitates higher bandwidth utilization, for an overall 1.98× increase. Finally, we enable asynchronous NIC execution, batching DMAs across multiple operations, to amortize overhead and minimize blocking time. This results in a cumulative 2.30× peak throughput increase, 2.07× relative to DrTM+H.

Next, we evaluate latency-oriented optimizations and their impact on Smallbank's median latency. Figure 4.8b shows these measurements. Relative to DrTM+H, the Xenic baseline latency is 1.37× higher. As in §4.2.1, the LiquidIO demonstrates consistently higher latency than the CX5 for comparable remote memory accesses, explaining this latency difference. Enabling Xenic's optimized commit operations, reducing the number of requests involved per transaction, improves latency by 20%. By shipping execution to the coordinator SmartNIC, Xenic eliminates intermediate coordinator-side PCIe traversals during each transaction, further reducing latency, 32% below the baseline. Smallbank's workload of 1-2 shard transactions presents the opportunity to further reduce latency by shipping execution to remote SmartNICs and applying optimized communication patterns. This achieves a 42% latency reduction over the baseline, 22% below DrTM+H.

4.5 Conclusion

We argue that SmartNICs offer an opportunity for high-performance, hardware-accelerated distributed transactions, without the trade-offs that define RDMA systems. Using measured performance characteristics to inform our design, we build Xenic, a transaction processing system leveraging on-path SmartNICs. Xenic employs a co-designed data store spread across the NIC and the host, an asynchronous and batched execution model, and flexible communications to improve efficiency. With three benchmarks comprising a range of workloads, we compare Xenic against RDMA-based systems. Our results show that despite software overheads relative to RDMA, Xenic effectively applies the SmartNIC to increase throughput and reduce latency.

Chapter 5

CC-NIC: A CACHE-COHERENT INTERFACE TO THE NIC

A wide range of new interconnects is emerging for accelerators, disaggregated memory, and multi-GPU systems. PCI Express (PCIe) [84] has long been the standard interconnect between a server and peripheral devices, such as the network interface controller (NIC). While PCIe bandwidth has increased substantially over the seven protocol generations, its interface for host-device communication has remained consistent. Now, new interconnect specifications [12, 109, 9, 81, 110, 80] propose to either replace or build upon the PCIe physical layer, while providing fundamentally different data paths and communication abstractions between the host and the peripheral.

A key attribute of these interconnects is allowing the host and devices to participate in coherence protocols. Hosts can access devices through the processor's highly optimized cache hierarchy, and devices can participate in the CPU's cache coherence protocol while accessing memory. These interconnects enable devices to be integrated into the host processor's coherence domain in different settings. For instance, Compute Express Link (CXL) [12] targets devices housed on expansion cards, Ultra Path Interconnect (UPI) [24, 34] is an inter-socket interconnect that also allows for the integration of hardware devices (e.g., Intel Agilex FPGA [89, 49]), and Cache Coherence Interconnect for Accelerators (CCIX) [9] proposes a coherent interface for chiplet-based systems.

Coherent device access to shared memory is a powerful programming model for data sharing, providing semantics not available with the typical read/write primitives of PCIe transactions. PCIe uses specialized data paths for transfers between the CPU and the device: CPUs access devices using memory-mapped I/O (MMIO) transfers that bypass the cache. Devices access host state using direct memory access (DMA). DMAs traditionally target data in host DRAM and place DRAM on the critical path for device access, although newer platforms add a limited form of cache

interactions [36]. In contrast, coherent interconnects integrate with the CPU's existing, highly-optimized memory data path. UPI, CCIX, and CXL directly interface with the coherence protocol and the L2 cache state, handling cache ownership and data transfers to a peripheral. This not only results in a shorter data path to the CPU core (as the device can target L2 instead of LLC or DRAM) but also enables the device to poll locally on cache-coherent state. Likewise, CPU accesses to the device can utilize the caching memory path instead of MMIO.

Coherent interconnects thus represent a fundamental change in host-device communication, offering new benefits and posing challenges. This paper aims to understand the value of coherent interconnects in the context of NICs. While many emerging interconnect specifications are in flux, we apply existing UPI hardware as a means to explore cache-coherent host-NIC interface designs and develop principles that could apply across a range of interconnects.

We first study today's PCIe-based NICs, identifying the unique tradeoffs that PCIe imposes on NIC interfaces. PCIe limits the use of shared data structures and imposes CPU overheads for host-initiated interconnect operations. Today's NIC designs, therefore, aim to minimize host PCIe overheads at the expense of transmission latency by introducing additional signaling trips and batching. The impact on packet latency is significant: the host-NIC loopback latency on a Mellanox CX6 NIC is 2.1us at low load and 6.0us at 80% load, almost an order of magnitude higher than switch traversal.

The streamlined datapaths of coherent interconnects can improve latency for existing NIC interface designs. But, we observe performance is highly sensitive to the access pattern on both sides. The producer-consumer patterns typical of existing NIC interfaces incur significant overheads without an optimized combination of access instructions, data ownership, and cache-line layout decisions. Achieving optimal communication requires data structures specifically designed for coherence. Finally, caching must be carefully managed; data and metadata may be retained in remote caches longer than needed, triggering expensive remote communication upon a future local access. Thus, redesigning the host-NIC interface is required to fully take advantage of coherent interconnects, and doing so allows us to benefit from the new signaling and sharing interactions made possible by coherence.

We present CC-NIC, a host-NIC interface optimized for coherent interconnects. We redesign all aspects of the host-NIC interface (namely, data structures, layouts, and signaling) to take advantage of the new data paths and cache interactions supported by coherent interconnects. To design CC-NIC, we consider the space of access type, layout, homing, and prefetching decisions, for each element of the interface. Our redesign not only offers improved latency but also delegates certain buffer management tasks to the NIC, thus reducing host-side costs.

We demonstrate the performance of CC-NIC over UPI on Intel's Ice Lake and Sapphire Rapids server platforms. CC-NIC demonstrates a packet rate of 1.5Gpps and a minimum TX-RX latency of 494ns. Latency under 80% load is 716ns, an even greater reduction relative to PCIe NICs. Compared to an interface matching a current PCIe NIC, on the same UPI link, our proposed design achieves a 3.3× throughput improvement and 52% minimum latency reduction, in addition to decreased latency under load and terabit bandwidth saturation. We evaluate key-value store and RPC workloads; both show that CC-NIC saturates network bandwidth with up to 50% fewer application threads versus PCIe NICs.

We present CC-NIC as a case study of optimizing coherent host-device interactions. The design of CC-NIC can be applied to other coherent interconnects. Our evaluation shows that CC-NIC's design benefits hold, maintaining consistent relative improvement, across varied interconnect performance characteristics.

5.1 Dissecting the PCIe Host-NIC Interface

We first analyze the host-device interface of today's PCIe NICs. Our goal is to understand how the characteristics of PCIe drive the interface design of existing NICs. In §5.1.1, we describe the packet queue interface and its data structures in the context of PCIe. Then, in §5.1.2, we measure the performance of PCIe access mechanisms. We discuss how these performance characteristics lead to tradeoffs in §5.1.3–the tradeoffs ultimately dictating the design of PCIe NIC metadata structures, data transfer, and buffer management.

5.1.1 The Host-NIC Interface

In this section, we focus on packet transmit (TX) and receive (RX) queues, the main transfer interface between host and NIC. This interface is consistent across a wide range of NICs and consists of the following components:

Packet buffers store the data payloads transmitted to or received by the NIC. Buffers are pre-allocated memory chunks. A pointer to each buffer is inserted into a *buffer pool* data structure, typically a queue. Allocating and releasing packet buffers involve dequeuing or enqueuing a pointer from the buffer pool. Packet buffers also include application-level metadata, e.g., length of the packet's headers, etc. Although the driver may use this metadata, it is not transferred to the NIC. The address communicated to the device is the start of the packet payload, which is typically cache-aligned and placed after any application metadata.

Descriptors represent the work requests. Each descriptor contains the address of a corresponding packet buffer. Typical descriptors are 16B, including 8B of tightly-packed metadata, such as the data length. They are organized into a ring buffer implemented as a circular array. On the TX path, the driver writes a descriptor for each submitted TX packet. The descriptor is derived from the driver's configuration state and the per-packet metadata in the TX buffer. The address and packet length are critical components of the TX descriptor; the NIC must have these values before it can access the packet from the host. On the RX path, the host first posts RX descriptors to provide the NIC with blank packet buffers. After the NIC writes a received packet into the RX buffer, it overwrites the descriptor with RX metadata (e.g., completion status).

Head and tail registers serve as signals to coordinate the producer-consumer relationship between the host and NIC. Registers are typically 32 or 64-bit values representing the producer and consumer positions of the ring array. After writing a TX descriptor, the host makes the descriptor available to the NIC by incrementing the TX tail register. When the NIC receives the tail value, it reads and handles packet descriptors up to the new tail index. After the NIC transmits a packet, it signals completion to the host by incrementing the TX head. The host handles transmit completions by returning the freed packets to the buffer pool, reclaiming descriptor ring space.

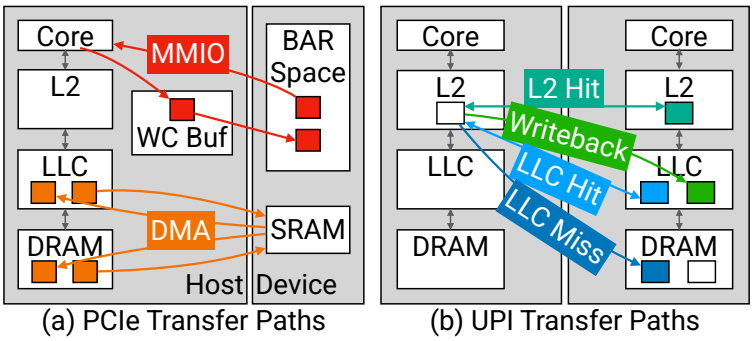

Figure 5.1: PCIe (a) and UPI (b) transfer paths.

For the receive path, the host allocates blank RX buffers from the pool, writes their addresses to the RX descriptor ring, and then increments RX head to signal the presence of new RX buffers. When the NIC receives packet data, it uses blank RX buffers at the RX tail index and notifies the host of RX packet availability by incrementing the RX tail. The host handles descriptors up to the RX tail index by returning the RX buffers to the application. In summary, the host writes the TX tail to submit TX packets to the NIC and writes the RX head to submit blank buffers to the NIC. The NIC writes the TX head to indicate transmit completions and writes the RX tail to indicate newly received packets.

5.1.2 PCIe Microbenchmarks

We now perform a measurement characterization of host-device accesses to understand how PCIe performance dictates host-NIC interface designs. While existing work has identified PCIe limitations [77, 116, 22, 94, 95], we aim to understand the extent to which performance limitations hold on current server platforms and NIC interfaces.

PCIe interconnect latency. PCIe presents an asymmetric interface to the device and the host. Figure 5.1a shows the mechanisms for transfers initiated by the host and by the device: *MMIO* and *DMA*, respectively.

Host-to-NIC reads and writes are performed via memory-mapped IO (MMIO). The device

exposes a memory area mapped into the host address space as an uncacheable (UC) or write-combining (WC) memory type. This allows the host to issue loads and stores to the device, which are executed as PCIe read and write transactions. The UC and WC memory types do not provide cache coherence or operate within the cache hierarchy. Instead, CPU loads always require a PCIe roundtrip, resulting in expensive accesses. On the ICX CPU platform, targeting an Intel E810 NIC (testbed described in §5.4.1), we measure a median MMIO read latency of 982ns (8B) and 1026ns (64B AVX512).

PCIe devices read and write host memory using Direct Memory Access (DMA). DMAs may be significantly larger (e.g., 4KB) than the 64B MMIO write-combining buffer size and typically access standard writeback host memory. While conventional PCIe NICs do not expose DMA latency statistics, we expect DMA roundtrip latency to be comparable to that of MMIO. SmartNICs that provide a low-level DMA controller interface, such as Marvell's LiquidIO [64], show a minimum DMA read latency of at least 1μs [59, 97].

Implication: The high latency of MMIO and DMA accesses suggests that each PCIe roundtrip contributes significantly to overall packet latency. Host-NIC data structures should also be designed to minimize high-cost CPU operations such as polling or reading across the PCIe bus (MMIO loads).

MMIO write throughput. Unlike loads, MMIO stores are posted, so a store does not incur a PCIe roundtrip delay from the host's perspective. However, MMIO writes are still expensive in the context of both UC and WC memory types. The UC memory type bypasses host caches altogether, so each MMIO access results in a PCIe transfer. To preserve ordering, only one MMIO access may be in flight between the CPU core and PCIe root, thus limiting throughput. The WC memory type offers more flexibility via write-combining store buffers, which can merge contiguous stores within a 64B-aligned region into a single PCIe transfer. This can reduce PCIe protocol overhead but complicates write ordering since writes may be buffered for an arbitrary time before being flushed. To ensure writes are flushed, it is necessary to issue a fence instruction, e.g., `sfence`, or ensure that each 64B buffer is completely filled in sequential order. These flush

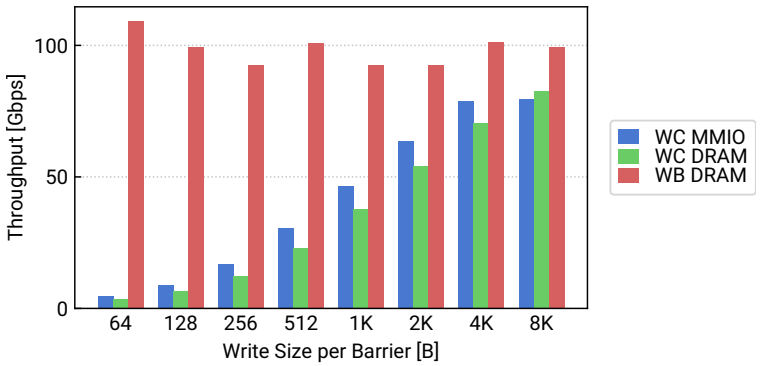

Figure 5.2: Single-threaded write throughput for WC MMIO (to E810 NIC), WC-mapped DRAM, and regular WB DRAM.

conditions make it difficult to achieve fine-grained control over PCIe write ordering.

Figure 5.2 compares the write throughput of WC MMIO accesses to a device, WC-mapped local DRAM, and writeback DRAM. We run this experiment on the ICX platform, targeting the E810 NIC for MMIO accesses using a single thread. We repeat the experiment with write sizes between 64B and 8KB, issuing an `sfence` barrier after each write.

With the WC data path, writing to both PCIe MMIO and DRAM, we find that the barriers, which may be needed to ensure ordering, impact throughput. This is not the case with WB DRAM, where throughput is consistent regardless of barrier frequency. Our results suggest that the MMIO WC data path cannot achieve high throughput without extensive batching. Near-maximum single-threaded throughput requires writing at least 4KB per barrier; with 64B packets, this means a batching factor of 64. This batched throughput is still only 76% of singleton 64B WB performance.

Implication: The WC and UC data paths are throughput-limited relative to standard write-back memory. Our measurements represent a NIC design that uses MMIO for bulk data/metadata transfer, unlike the signaling-only MMIO operations of current NICs. For streaming writes using the WC datapath, the barriers required to ensure ordering and flush to the device limit throughput.

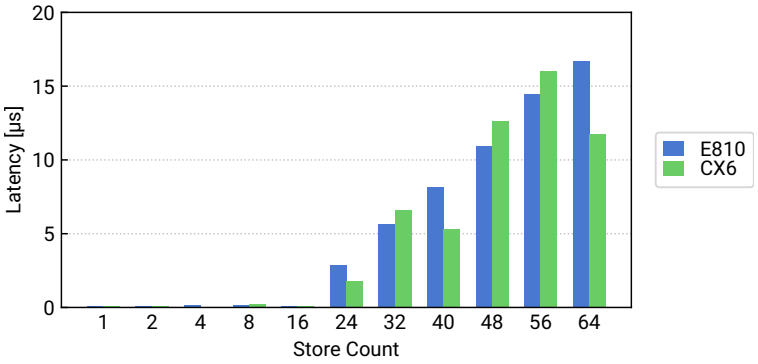

Figure 5.3: MMIO store latency versus iteration count for PCIe 4.0 ×16, from Ice Lake host to CX6 and E810 NICs.

MMIO write-combining latency. WC memory also introduces the limitation of a fixed number of store buffers. When all WC buffers are occupied, issuing a store within a 64B region not already buffered results in stalling until a buffer is flushed. Figure 5.3 shows the cumulative latency of N 32-bit MMIO stores to the E810 and CX6 NICs, up to $N = 64$. Latency remains uniform and low (< 20ns) until $N = 24$, where all WC buffers are utilized for the N stores. Beyond that, store latency is at least 15× greater and increases with N, as store buffers are flushed on the critical path.

Implication: When the MMIO data path is used for bursts of small stores, limited store buffer availability leads to expensive, high-latency accesses. This represents a NIC interface design that applies MMIO stores for metadata transfer, e.g., submitting descriptors to the device.

5.1.3 PCIe NIC Interface Design

The nature of PCIe and its performance characteristics impose constraints on the host-NIC interface. We identify three issues:

1. Since PCIe is not a coherent interconnect, local data structure updates must be communicated or signaled with explicit PCIe transactions.

2. PCIe operations incur high latency, so reducing the number of interconnect traversals is critical to achieving low-latency packet transmissions (§5.1.2).

3. Data and metadata writes over PCIe are expensive for the CPU in terms of both throughput and high-latency stalls (§5.1.2, §5.1.2).

The above issues define the performance tradeoffs imposed by the PCIe interface. An ideal design would achieve high packet throughput, low latency, and high CPU efficiency, but the PCIe prevents us from achieving all three goals simultaneously. Today's PCIe NICs prioritize CPU efficiency and throughput at the expense of latency by making the following design decisions:

Data structures are host-local, and updates are explicitly signaled. The host maintains packet buffers and descriptor rings in its local memory (as opposed to device MMIO) to reduce the CPU overheads for data structure access and updates. Requests to transmit a packet and newly received packets are signaled explicitly to the NIC. Other arrangements (e.g., host or NIC polling across the PCIe) waste PCIe bandwidth for each polling access and consequently are not used. For instance, in the transmit path, the host writes TX packets and TX descriptors into host memory and writes only a TX signal to the queue tail register maintained on the device side via MMIO. This results in a tradeoff: minimal data transfer over MMIO at the cost of extra interconnect roundtrips to read descriptors and packets from host memory.[1]

Descriptor transfer is batched. Given the host-side CPU stalls for MMIO writes to uncacheable NIC-side registers, the host may enqueue a large group of descriptors per MMIO register signal. This batching optimization again trades off latency for CPU efficiency.

The host handles all buffer management. The PCIe read-write interface, without cache coherence, limits the sharing of data structures between host and NIC. The synchronization mechanisms that support multi-core pool accesses are incompatible with PCIe reads and writes. Thus, the host performs all buffer management. This includes pre-allocating RX buffers to be used by the NIC and freeing completed TX buffers after NIC transmission. This results in additional

[1]Some NICs, e.g., the CX6, implement an alternative data path that writes descriptors over MMIO; however, this is typically enabled only for low-throughput, latency-critical workloads.

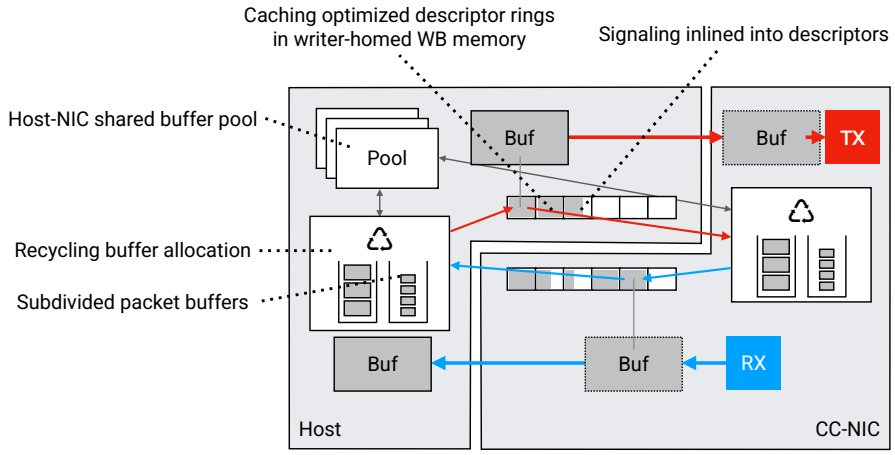

Figure 5.4: Overview of CC-NIC design features. Thick lines denote packet data transfers, and thin colored lines denote metadata accesses, such as buffer allocation.

bookkeeping communication over PCIe and also limits the NIC from performing memory optimizations based on the properties (such as size) of received packets.

In the Evaluation, §5.4.3, we provide end-to-end measurements of PCIe NIC latency, throughput, and core utilization.

5.2 System Design for Coherent Interconnects

In this section, we describe the design of CC-NIC, a host-NIC interface optimized for cache-coherent interconnects such as UPI. First, in §5.2.1, we contrast a cache-coherent interconnect with PCIe, given our analysis of today's NIC interface designs. Then, we discuss the CC-NIC design in terms of metadata structures (§5.2.2), data accesses (§5.2.3), and packet buffer management (§5.2.4).

We present the design of CC-NIC as a series of empirically-backed design decisions, with the eventual design having the following desirable properties: (1) low-latency packet transmissions through the use of cache-to-cache transfers and hardware-supported signaling, (2) high throughput for data and descriptor communications using the efficient write-back datapath, and (3) reduced CPU management overheads realized by sharing buffer management and optimizing

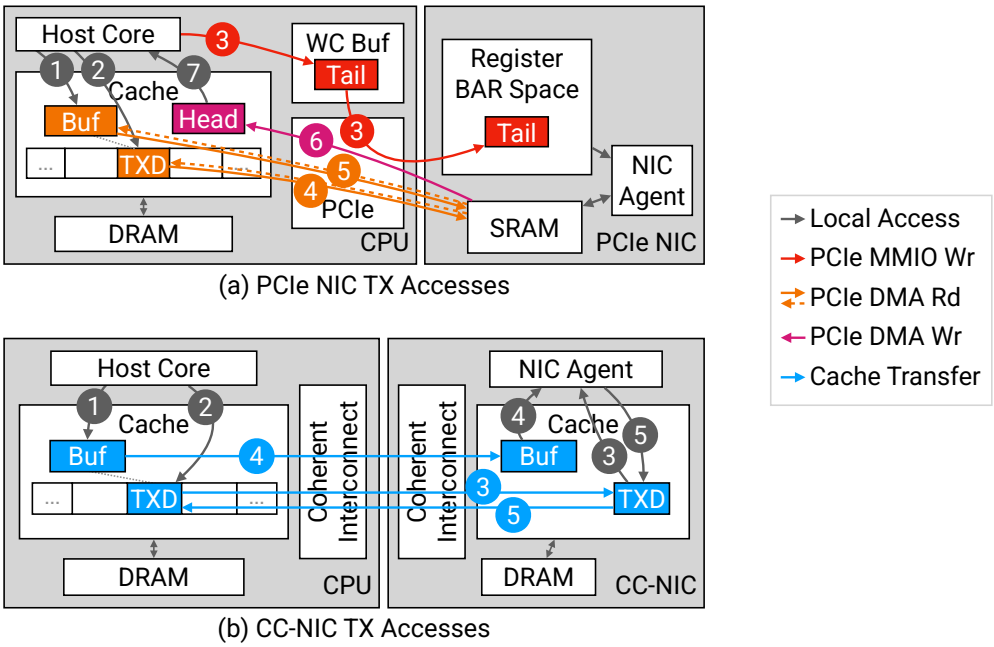

Figure 5.5: TX path accesses for (a) the Intel E810, and (b) the CC-NIC interface. Numbers denote access order.

```
int ccnic_buf_alloc(struct ccnic_pool *pool, struct ccnic_buf **bufs,
                    unsigned count);
void ccnic_buf_free(struct ccnic_pool *pool, struct ccnic_buf **bufs,
                    unsigned count);

int ccnic_tx_burst(int txq_index, struct ccnic_buf **bufs, unsigned count);
int ccnic_rx_burst(int rxq_index, struct ccnic_buf **bufs, unsigned count);
```

Figure 5.6: The core CC-NIC data plane interface, maintaining the semantics of DPDK `mempool` and `ethdev` APIs.

buffer placements for both TX and RX paths. Figure 5.4 summarizes the design features of CC-NIC, discussed in the following sections. Figure 5.5 compares the host and NIC TX path accesses for CC-NIC and the Intel E810, a typical PCIe NIC. CC-NIC provides a data plane interface analogous to DPDK's `mempool` and `ethdev` APIs, with burst semantics to enable batched TX/RX and buffer management operations. Figure 5.6 shows the core software interface.

5.2.1 Contrasting Coherent Interconnects and PCIe

Coherent interconnects, such as UPI and CXL, are tightly integrated with the CPU's memory data paths. Cross-interconnect accesses may target DRAM and caches (see Figure 5.1b). The coherence protocol manages shared cache state, transferring lines into local caches when memory is accessed. The protocol ensures a writer gains exclusive control of a cache line before writing, invalidating any copies in remote caches. The protocol allows multiple caches to share reader access to a line, and lines may be forwarded between caches.

Overall, coherent interconnects provide a fundamentally different interface from PCIe. The coherence abstraction enables new forms of signaling and data structure sharing without the constraints of the PCIe read and write interfaces. Coherent interconnects integrate with the memory datapath and cache hierarchy, unlike PCIe MMIO, and also provide a symmetric interface, avoiding the tradeoffs between MMIO and DMA operations. However, cross-interconnect transfers depend on cache presence and coherence states, in terms of latency, memory controller requests, protocol metadata overhead, and roundtrips. There are limited means of manipulating these cache line states and caching behavior in general. As a result, implementing NIC data structures in the context of a coherent interconnect leads to both opportunities and challenges. We identify three factors that call for a different design:

1. **Coherence enables interface signaling and shared data structures.** PCIe NICs typically implement TX signaling with a separate mechanism, MMIO, from data and metadata DMA transfers. This results in an extra interconnect roundtrip to retrieve TX metadata via DMA after receiving the signal. Cache coherence performs signaling in hardware: when

the remote side performs a write, the coherence protocol will invalidate any locally cached copy and fetch the new value upon subsequent access. Further, a coherent interconnect also enables the use of shared data structures between host and NIC, thus allowing for shared management of the buffer pool.

2. **CC-NIC has to choose between different data transfer mechanisms and homing options.** Coherent interconnects provide a diverse set of transfer mechanisms. For instance, CC-NIC can target write-back memory in addition to the cache-bypass data path and home data structures on either the host or the NIC, thus providing it with multiple transfer options to choose from. Cross-interconnect data transfers and cache state transitions also depend on the current cache residency of an object, possible prefetching, and the cache states caused by previous accesses. As a result, small objects, e.g., signals and descriptor metadata, are highly sensitive to layout.

3. **CC-NIC has to carefully manage caching.** The coherence protocol exchanges cache line ownership across the interconnect. Therefore, a remote access may result in additional communication when a later local access is performed. This is specifically problematic for producer-consumer workloads. For instance, the typical TX path transfers metadata and data from the host to the NIC, then the NIC performs the data transmission, and the data is not needed again by the NIC. In the coherent interconnect context, retaining the packet buffer or descriptor in the NIC-side cache is unhelpful; it adds overhead to subsequent host accesses that would have to perform remote cache invalidations. This suggests that minimizing overhead requires selectively invalidating cached data, which is unsupported on typical x86 platforms, or re-designing the data structures to avoid this access pattern.

5.2.2 Metadata Structures

In this section, we discuss the key questions that inform CC-NIC's handling of metadata, such as TX/RX descriptors.

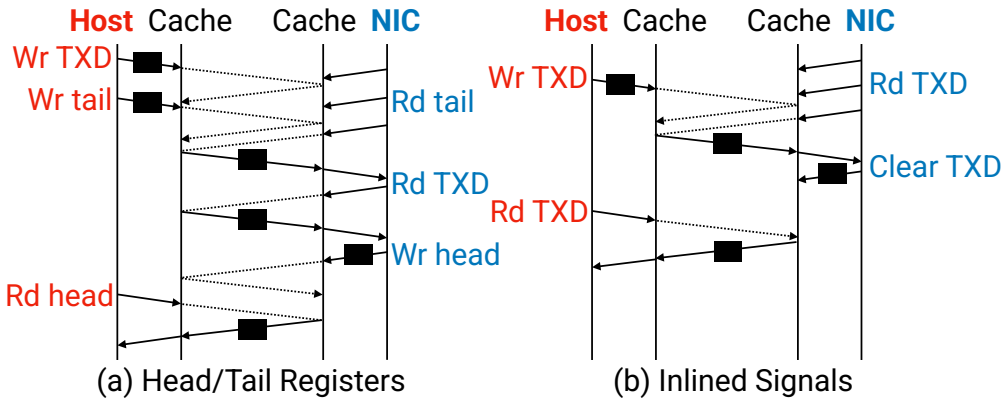

Figure 5.7: Signaling communication with (a) registers, and (b) signals inlined in the descriptor. Dotted lines represent coherence metadata.

How can we take advantage of cache coherence to reduce software overhead? Cache-coherent interconnects provide an underlying hardware mechanism to transfer and signal the availability of new data, via cache state transitions. This avoids the need for software-based signaling via head and tail index registers. To this end, CC-NIC applies an *inlined* signal in the descriptor, implemented as a flag indicating whether the descriptor is ready for consumption or free. Integrating the signal and descriptor eliminates a cache line transfer per signal and saves a cross-socket cache line access delay, as shown in Figure 5.7. For transmission, instead of polling a register containing the queue tail index, the NIC polls the next descriptor in the ring. The descriptor metadata includes a ready flag, which the host sets after other descriptor fields are written. Once the flag is set, the NIC receives the signal and the descriptor content in one access.

Event-driven implementation. A coherent NIC ASIC could further optimize signaling communication by directly handling coherence protocol messages. Instead of accessing descriptors through the cache polling abstraction, the device would directly take action in response to snoop messages received over the interconnect. Handling the coherence messages as signals avoids the scalability limitations of software-based polling in the presence of large queue counts.

What is the ideal data path for metadata transfers? Since a coherent interconnect may retrieve data from DRAM or multiple levels of the cache hierarchy, we measure the performance

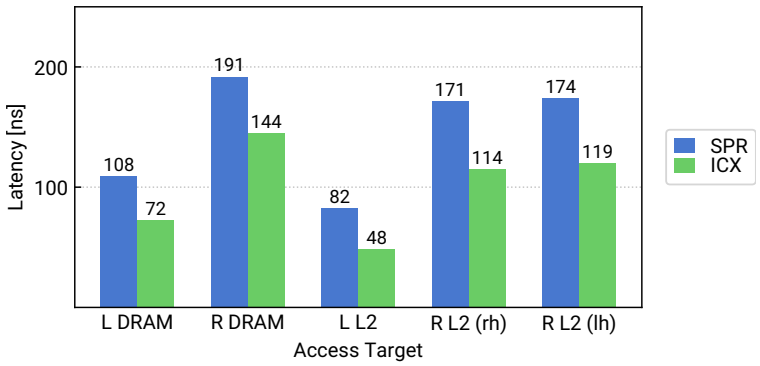

Figure 5.8: Local and cross-UPI access latency for Sapphire Rapids and Ice Lake hosts with various cache states.

of each transfer case to understand performance implications for signaling communication. Figure 5.8 shows the median access latency of a 64B-aligned object in various cache states for both local and remote (cross-UPI) memory on Ice Lake (ICX) and Sapphire Rapids (SPR) server platforms. We find that accessing remote uncached DRAM incurs approximately twice the latency of local DRAM access. Accessing data cached in remote L2 is faster: 171ns on SPR and 114ns on ICX for memory homed on the remote socket (*rh* case), and slightly higher for memory homed on the local socket (*lh*). In these cases, the remote CPU has written to and retained a line in its L2 cache in the `M` (modified) state, and then the local reader accesses the address. When an `M`-state object exists in a remote L2 cache, it cannot exist in any other L2, so there is always a local L2 miss. With reader-homed memory, the reader's L2 miss causes a speculative memory read in addition to the remote request to the writer's cache. This speculative read is unneeded and causes lower performance when an object is reader-homed, increasing bus utilization with spurious traffic. Regardless of homing, remote L2 accesses are faster than a remote DRAM access, suggesting that cache-to-cache transfers achieve the best-case latency.

CC-NIC applies these observations in its design. CC-NIC places metadata structures in writer-homed memory: the TX descriptor ring is host-homed, and the RX ring is NIC-homed. It enhances the possibility of cache-to-cache transfers by utilizing write-back memory with regular caching accesses instead of nontemporal stores that target memory. However, the working set size of the

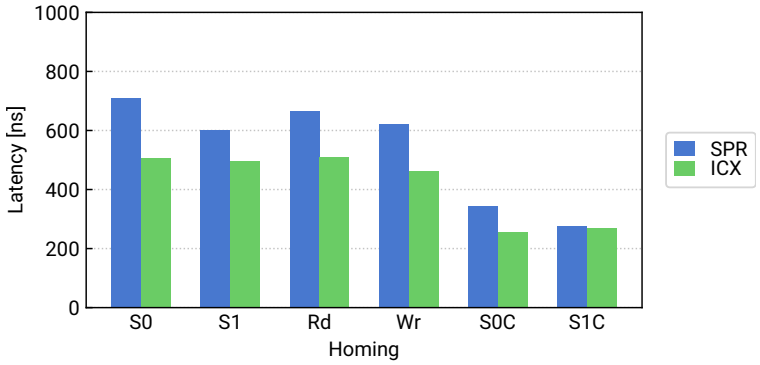

Figure 5.9: UPI pingpong experiment showing median latency with different memory layout choices.

NIC interface affects performance, as do prefetch accesses (see §5.2.3).

How does memory layout affect metadata? NIC metadata, such as descriptors and signals, exhibit a producer-consumer access pattern in which each descriptor is written by one side and read by the other. For instance, TXDs are written by the host and RXDs by the NIC. Performance depends on the access pattern of a cache line, as this determines the protocol communication necessary to ensure coherence.

We use a pingpong microbenchmark to analyze producer-consumer accesses. We run a single thread on both sockets, accessing two shared 64-bit registers. The first thread increments the first value, while the second thread polls. The polling thread increments the second register after reading the updated value. We report roundtrip time, from writing the first register to reading the same value in the second register. Figure 5.9 shows median latency with both registers allocated in separate cache lines; both homed on the same socket (*S0/S1* cases); both homed on the respective reader/writer sockets (*Rd/Wr*); and both co-located on one cache line (*S0C/S1C*). In the *S0/S1* and *Rd/Wr* cases, each register exists on a separate cache line written by one CPU and read by the other. These correspond to PCIe NIC signaling, where host-to-NIC registers exist in the MMIO address space and NIC-to-host registers in write-back memory.

With separate cache lines, we find that writer-homed memory yields the lowest latency, con-

sistent with Figure 5.8. These scenarios all show $1.7 - 2.4\times$ higher latency than when the values are on one cache line, homed on either socket. Co-locating producer and consumer structures on a single cache line achieves the best overall latency. With separate cache lines, each read transfers a cache line over the interconnect, and each write incurs another roundtrip to invalidate the reader cache. The co-located case instead uses one cache line for two-way communication. The $1.7 - 2.4\times$ latency difference shows the benefit of applying memory layouts that enable this two-way communication. Measuring *offcore response* perf counters shows a reduction of remote-socket requests from 4 to 2 per pingpong. This indicates reduced interconnect utilization in addition to improved latency.

CC-NIC applies two-way communication for signaling and descriptor transfers. Unlike the PCIe head and tail register layout, the host and NIC communicate by writing and clearing each descriptor and its inlined signal. This results in an access pattern matching the minimum pingpong latency.

How do we optimize for both latency-sensitive and high-bandwidth regimes? With inlined signaling, the host and NIC directly poll descriptor ring memory rather than separate registers. This results in the cache line thrashing between sockets when writing and polling a series of descriptors smaller than the 64B cache line. This thrashing increases latency compared to a cache-aligned case where descriptors are padded to 64B. Cache-aligned descriptors result in significant wasted space (e.g., 48 out of 64B), impacting the maximum packet rate. Furthermore, both scenarios prevent batching multiple descriptors per signal, a technique existing NICs typically rely on to maximize packet rate. To address these tradeoffs, CC-NIC implements a balanced solution: it packs bursts of up to $4\times$ 16B descriptors into a cache line, with unused entries zeroed out, and uses one signal per cache line. If the consumer reaches a blank descriptor in the middle of a group, it skips to the next cache line to poll for the subsequent descriptor group. This eliminates wasted space in the high-throughput case while avoiding thrashing in the low-throughput, un-batched case. With one signal per descriptor group, CC-NIC applies batching to utilize each descriptor cache line fully in high-throughput scenarios.

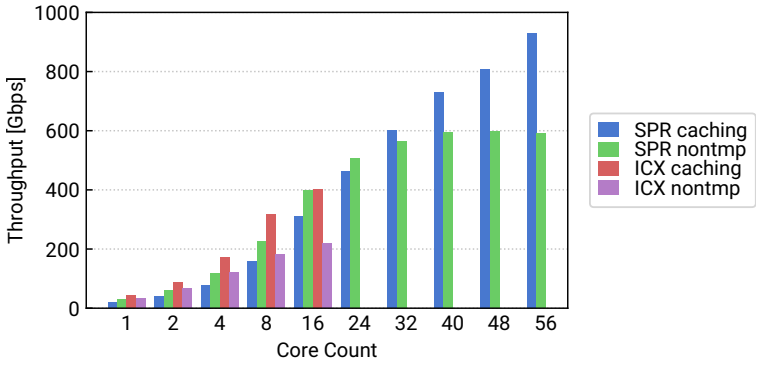

Figure 5.10: Stream transfer experiment comparing UPI throughput with caching and nontemporal accesses.

5.2.3 Data Accesses

Next, we discuss the questions guiding the design of packet data transfer in CC-NIC.

How should we write packet data? We run a streaming write microbenchmark to compare caching and nontemporal store throughput. For each case, we run a writer thread on the local CPU and a reader thread on the remote CPU. The writer thread sequentially writes data to a shared memory region, signaling the reader via a register for each 1MB written. The reader accesses 1MB per signal, copying into a thread-local buffer. We run a copy of the workload on each pair of threads, up to all 16 ICX and 56 SPR cores. Figure 5.10 shows the results using two access types. In the *caching* case, the writer applies cacheable stores, which follow the typical memory datapath and enter the cache. This results in cross-interconnect cache transfers from the writer to the reader. In the *nontemporal* case, the writer uses cache-bypassing nontemporal stores to target reader-socket DRAM. This case aligns with the PCIe MMIO datapath, where stores are submitted over the interconnect directly without entering the cache. Our results show that the cache-to-cache transfer path enables higher throughput on both platforms: 1.8× (ICX) and 1.6× (SPR).

Additionally, we measure the maximum achievable interconnect throughput on our platforms using the Intel `mlc` benchmark utility [38]. This measurement uses a read-only remote access

workload, which shows higher throughput than other patterns. On SPR and ICX, we find a maximum data throughput of 1020Gbps and 443Gbps, respectively. This suggests that the reader-writer streaming workload reaches 91% of best-case read-only throughput by using cache-to-cache transfers.

Based on this result, we apply caching stores to write packet data, both at the host application (while generating TX packets) and at the CC-NIC (while writing inbound RX packets).

How can we minimize coherence protocol overhead for data transfers? Packet buffers also demonstrate the producer-consumer access pattern described in §5.2.2; NIC's TX buffer accesses are read-only, and RX buffer accesses are write-only. Like descriptors, performance depends on underlying coherence state transitions. After the NIC completes a packet transmission, the buffer is likely to remain in the NIC's local cache. Although the buffer contents are no longer needed by the NIC, when the buffer is next allocated on the host side, writing to the buffer memory requires cross-socket access to invalidate this unnecessarily cached data.

There is no ideal instruction available to purge the consumed buffer memory out of the consumer-side cache. While `CLFLUSHOPT` does trigger cache invalidation, it is an expensive instruction that must be called on a per-cache-line basis and may incur memory accesses after the invalidation. Other cache control instructions, such as `CLWB`, do not help, as they do not result in cache invalidation.

Instead, CC-NIC implements a buffer recycling allocator to reuse the most recently freed TX buffers as RX buffers and vice versa. In both cases, the goal is to allocate buffer memory still present in the writer's cache. CC-NIC's buffer recycling provides similar application-level semantics to the TX-RX buffer reuse implemented by some PCIe NIC drivers (such as the i40e kernel driver [105]). However, these existing mechanisms are software-only driver optimizations and thus do not affect interconnect communication. CC-NIC's buffer recycling takes place at both the NIC and the host, and addresses the unique producer-consumer overheads imposed by cache coherence. We implement buffer recycling using host- and NIC-local stacks, which cache free buffer addresses from the pool of packet buffer memory. This technique is suitable for applica-

tions with a single buffer pool for TX and RX traffic, the typical design pattern among DPDK NIC drivers. But, in cases where there are multiple references to the TX buffer payload, and the host retains the TX buffer after transmission (e.g., for potential retransmission), this optimization falls back to standard buffer allocate/release behavior. The CC-NIC buffer allocator is described fully in §5.2.4.

Where should data be homed? Our measurements of remote-socket accesses (§5.2.2) demonstrate a latency benefit to homing memory on the writer socket. As a result, CC-NIC places the TX descriptor ring on the host socket and the RX ring in NIC memory. However, we allocate packet buffer memory entirely homed on the host. Since applications may arbitrarily access packet buffer data, placing it in remote memory could have unexpected application-level consequences. Applications may, for instance, submit RX buffers to a TX queue, so writer-homing does not universally apply to packet buffer memory. Instead, CC-NIC's recycling buffer allocation policy and locality-oriented optimizations, described next, aim to minimize the cost of host and NIC buffer accesses.

How can we maximize cache locality for packets? As measured in §5.2.2, maximum remote access performance is achieved when an object is present in remote cache. As such, it is important to maximize the caching of shared data on both the host and NIC sides. Using small packet buffer sizes for small packets reduces the overall memory footprint of the NIC interface. When supported by the application, an MTU-sized buffer, for instance, 4KB, is subdivided into 32× 128B packet buffers. When the host allocates a buffer to write a TX packet, it selects either a large or small buffer based on the packet data size, if known in advance. The RX side follows the same logic (see §5.2.4). This increases the cache efficiency of small packet transfers. Unlike PCIe NIC drivers, which may inline packet data into the descriptor ring, this approach does not require copying the packet data payload into another location.

Relative to PCIe, cache coherence brings the additional challenge of potential remote prefetching. When buffers are allocated sequentially from a contiguous region of memory, the sequential

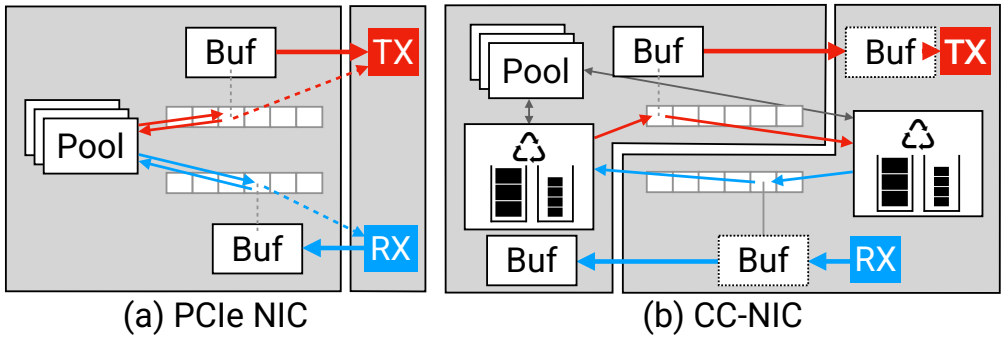

Figure 5.11: Buffer management approaches. Thin lines denote buffer allocation/release; thick lines denote buffer data transfer.

access pattern on the consumer side may result in hardware prefetching of the buffer memory just beyond the current packet buffer. These remote prefetches contend with local writes of that same packet buffer before it is submitted to the descriptor ring. This behavior occurs when the NIC handles one posted TX buffer and prefetches subsequent buffer memory while the host is writing to the next buffer allocated sequentially in memory. We avoid this contention by filling the memory pool with buffers such that repeated buffer allocations do not yield sequential memory addresses. This policy avoids unwanted cache state transitions, increasing the efficiency of producer-side packet buffer writes.

5.2.4 Buffer Management

Several of the above design features, including the recycling buffer allocator (§5.2.3), subdividing buffers for small packets (§5.2.3), and cache-aligned descriptor groups (§5.2.2), are critical to CC-NIC's efficient utilization of the interconnect. Each of these optimizations requires allocating buffers and writing RX descriptors based on the properties of the workload. In a typical PCIe NIC interface, RX buffers are allocated and posted to RX descriptors by the host prior to the actual reception of packets by the NIC. This makes it impossible to apply knowledge of the RX packet burst at the time of assigning buffers to RX descriptors. CC-NIC overcomes this by taking advantage of cache coherence to share the responsibility of buffer management with the host.

Protocol	GT/s	1 Link GB/s	Max Total GB/s
PCIe 4.0	16	2.0	31.5 (×16)
PCIe 5.0, CXL 1.0-2.0	32	3.9	63.0 (×16)
PCIe 6.0, CXL 3.0	64	7.6	121 (×16)
Ice Lake UPI	11.2	22.4	67.2 (×3)
Sapphire Rapids UPI	16	48	192 (×4)

Table 5.1: Comparison of PCIe, CXL, and UPI bandwidth.

Cache coherence allows the host and NIC to access the buffer pool data structure concurrently without the restrictions associated with simultaneous PCIe DMA and CPU accesses (e.g., lack of atomic operations). A shared buffer pool structure allows the NIC to release TX buffers to the pool after transmission. Likewise, the NIC can allocate RX buffers on demand and write their addresses into the RX descriptor ring. This results in a symmetric design that avoids extra bookkeeping passes over the queues to free completed TX packet buffers and post blank RX packet buffers. Finally, shared buffer management enables CC-NIC's buffer allocation and descriptor layout optimizations. Figure 5.11 compares CC-NIC's buffer management design to that of PCIe NICs.

5.3 CC-NIC Implementation

To demonstrate the benefits of the CC-NIC design, we implemented CC-NIC on a dual-socket server where one socket acts as a software NIC. In this implementation, all host-NIC communication occurs over the UPI interface. In addition to being a coherent interconnect that we can experiment with now, UPI also provides bandwidth higher than contemporary PCIe generations (see Table 5.1). Sapphire Rapids CPUs provide a terabit-throughput UPI interface, allowing us to model terabit NIC communication.

We designate one CPU and its local-socket memory as the host and the second socket as the NIC. NIC-socket memory represents coherent device memory, and the NIC cores represent the processing units of the NIC. The software flexibility enables experimenting with data structure designs and communication patterns since we are not restricted by the hardware interfaces of

existing NICs. We believe the software-initiated nature of NIC accesses does not change the host-NIC interactions required to transfer packets; hardware-initiated transfers would map to equivalent coherence protocol operations and show comparable interconnect performance.

To evaluate the PCIe and CC-NIC interfaces in isolation, we focus on loopback performance. Prior work finds that PCIe can contribute the majority of network TX/RX latency observed by the end-host [77]. While these experiments exclude Ethernet transmission, they demonstrate the most significant component of overall latency.

To understand end-to-end throughput and core utilization, we implement a CC-NIC Overlay interface atop a PCIe NIC. With a PCIe NIC installed on the second socket, we utilize *overlay* threads on the NIC socket to bridge between the CC-NIC UPI interface and a PCIe NIC. These threads poll both UPI TX and PCIe RX queues, copying packet data and writing descriptors between each respective pair of queues. This allows applications running on the first socket to perform network TX/RX via CC-NIC. While overlay packet forwarding adds latency and burns cores on the second CPU, it allows us to measure application throughput and core utilization.

5.4 Evaluation

Our evaluation is guided by the following questions:

1. How does CC-NIC perform relative to PCIe NICs? §5.4.2
2. What performance does CC-NIC achieve on a terabit UPI interconnect? §5.4.3
3. What are the gains from optimizing metadata structures, data accesses, and buffer management? §5.4.4
4. How does batching affect performance? §5.4.5
5. Does CC-NIC's design increase coherence communication efficiency? §5.4.6
6. Can CC-NIC save CPU cores with a key-value store application and TCP RPC stack? §5.4.7

7. How sensitive is the design to hardware prefetching, interconnect bandwidth, and latency? §5.4.8, §5.4.9

5.4.1 Evaluation Setup

We use two server platforms with the following specifications. The ICX server is a dual-CPU Intel Ice Lake Xeon Gold 6346, running at 3.1GHz, with PCIe 4.0 support and 3 × 11.2GT/s UPI links. Each ICX CPU has 16 cores (32 hyperthreads), 1.25MB per-core L2, 36MB LLC, and 12 × 16GB DDR4 at 3200MHz. This server contains two PCIe NICs, an Intel E810-2CQDA2 (E810) and Nvidia ConnectX-6 Dx MT42822 (CX6), both 2×100GbE devices. Our SPR server contains dual Intel Sapphire Rapids CPUs, running at 2.0GHz, with PCIe 5.0 support and 16GT/s UPI. Each SPR CPU has 56 cores (112 hyperthreads), a 2MB per-core L2 cache, 105MB LLC, and 8 × 64GB DDR5 at 4800MHz.

We apply these two server platforms to evaluate the following comparison points:

- **CC-NIC on ICX (UPI).** We deploy CC-NIC on the ICX server to compare UPI- and PCIe-based NIC communication on the same CPU platform.

- **CX6, E810 on ICX (PCIe).** For our PCIe NIC measurements on the ICX server, we follow the vendor-published system and driver configuration steps [39, 79] and verify that packet-forwarding performance matches these official DPDK performance reports. We enable standard platform-level optimizations such as DDIO [36].

- **CC-NIC on SPR (UPI).** To measure CC-NIC's performance across a terabit coherent interconnect, we also deploy CC-NIC on the SPR platform with the above specifications.

- **Unoptimized UPI on SPR, ICX.** To demonstrate coherent NIC performance without CC-NIC's design features, we implement the Intel E810 NIC interface over the UPI interconnect. We use writeback memory and caching accesses but maintain the E810 data structure layout and register-based signaling. This baseline scenario represents a case where future coherent NICs apply the same software interface as today's PCIe NICs.

Loopback setup. We implement a traffic generator using DPDK [14] to evaluate both CC-NIC and PCIe NICs. Each NIC serves as a loopback between pairs of TX and RX queues. Each application thread configures private queues, allocates TX buffers, and writes full, timestamped payloads for each TX packet burst; it polls RX queues and accesses each RX payload before freeing the buffer. This is more work per packet than minimal RX-TX forwarding due to payload accesses and separate TX and RX flows. We vary TX rates from one inflight packet to the maximum sustainable rate and measure median roundtrip latency and RX data throughput.

Overlay setup. We use the CC-NIC Overlay (§5.3) to evaluate per-thread application throughput with the CC-NIC interface. An application uses the CC-NIC UPI interface for TX/RX; remote-socket overlay threads transfer packets between corresponding PCIe NIC queues. As a baseline, the application interfaces directly with the same socket-local PCIe NIC. While the PCIe NIC limits throughput in both cases, this setup allows us to compare CPU utilization.

Our evaluation uses DPDK [14] for high-performance PCIe NIC TX/RX. We implement a traffic generator to evaluate both PCIe NICs and CC-NIC, with each NIC serving as a loopback between TX and RX queues. Each application thread configures private queues, allocates TX buffers, and writes full, timestamped payloads for each TX packet burst; it polls RX queues and accesses each RX payload before freeing the buffer. This is more work per packet than minimal RX-TX forwarding, due to payload accesses and separate TX/RX flows. As a baseline, we verify that an implementation without these overheads matches vendor-published DPDK performance [39, 79]. We run the loopback experiment, varying thread count and TX rates, from one inflight packet to the maximum sustainable rate. Our results show median roundtrip latency and RX data throughput. We use the CC-NIC Overlay (see §5.3) for application-level measurements, to evaluate per-thread application throughput with the CC-NIC interface. As a baseline, we run the application on the PCIe NIC's socket, using standard DPDK APIs. Then, we apply the CC-NIC UPI interface for application TX/RX; overlay threads transfer packets to and from corresponding PCIe queues.

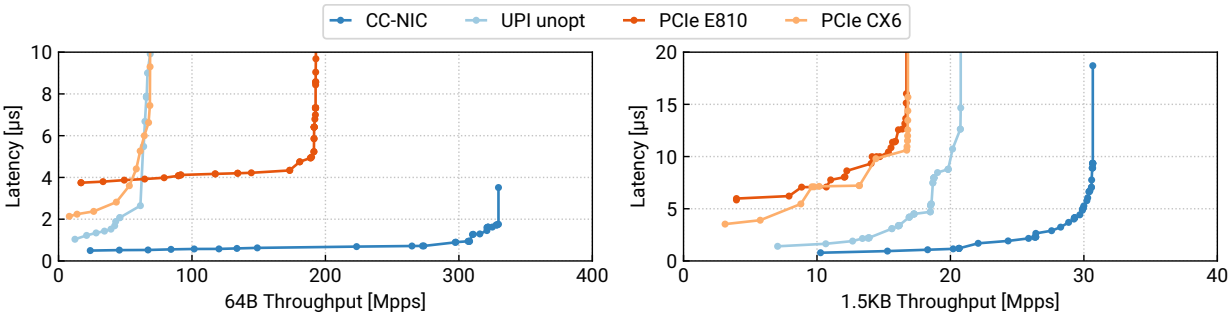

Figure 5.12: Throughput-latency curves, comparing CC-NIC and PCIe loopback performance on the ICX server. Showing 64B packets with 16 cores, and 1.5KB packets with 12 host cores.

5.4.2 Performance Comparison Overview

Figure 5.12 shows a comparison of four host-NIC interfaces on the ICX server: the CX6 and E810 PCIe NICs, a naive implementation of the E810 interface over UPI, and CC-NIC. These results show that CC-NIC provides a significant opportunity for latency improvement and higher throughput over PCIe. CC-NIC's minimum latency is 77% and 86% lower than that of the CX6 and E810. As detailed in §5.4.3, CC-NIC's latency reduction over the CX6 is more significant when considering latency under load for both large and small packets. CC-NIC also achieves a 1.7× and 4.3× higher peak packet rate than the E810 and CX6. With 1.5KB packets, we observe 1.8× higher data throughput over both PCIe NICs.

The unoptimized UPI (*unopt*) scenario shows that a coherence-optimized design is critical. This case applies the E810 interface across UPI and achieves lower 64B packet rates than the native PCIe E810 despite a higher-bandwidth interconnect. Relative to CC-NIC, this version shows 79% lower throughput and 2.1× higher minimum latency.

5.4.3 Detailed Performance Comparison

This section compares CC-NIC, CX6, and E810 results on the ICX platform. Figure 5.13 shows throughput-latency profiles for CC-NIC and CX6, with 64B and 1.5KB packet sizes. We also measure CC-NIC on the SPR platform's terabit UPI interconnect, shown in Figure 5.14.

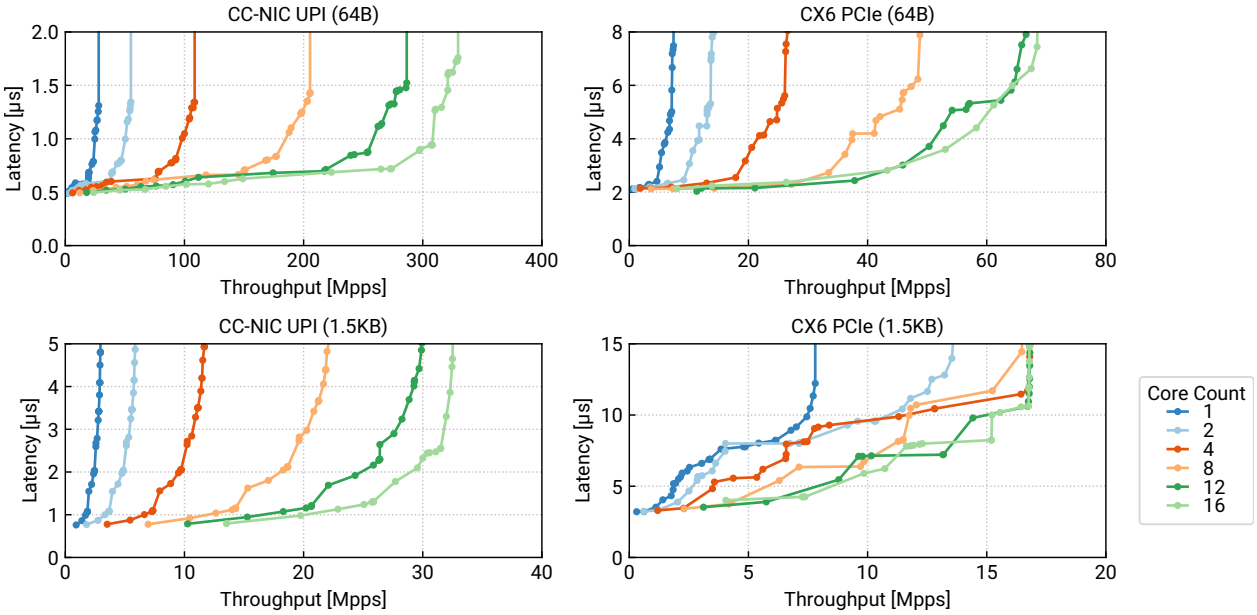

Figure 5.13: Loopback throughput-latency curves for CC-NIC (UPI) and CX6 (PCIe), with 1-16 ICX host cores, 64B and 1.5KB packet sizes.

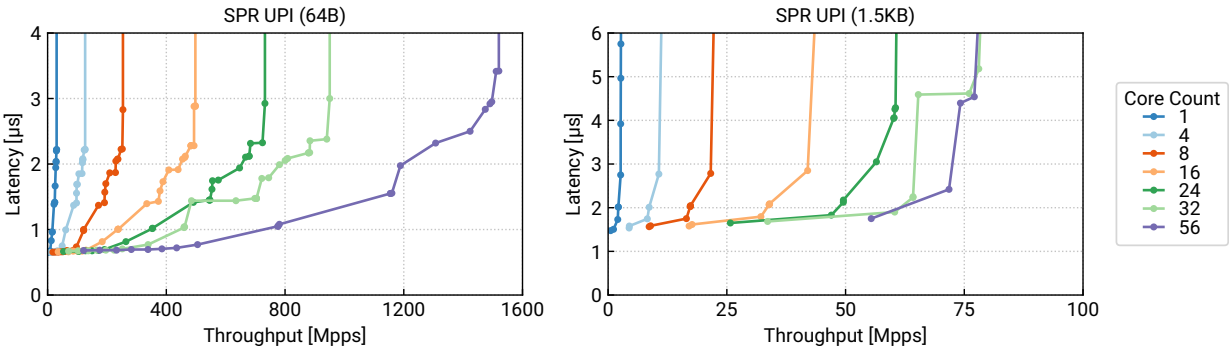

Figure 5.14: Loopback throughput-latency curves for CC-NIC on Sapphire Rapids UPI, with 64B and 1.5KB packet sizes.

Latency. CC-NIC demonstrates low minimum latency and loaded latency relative to both PCIe NICs. We measure a minimum loopback latency of 490ns (ICX) and 650ns (SPR) versus a best-case PCIe latency of 2116ns (CX6) and 3809ns (E810). The latency difference between CC-NIC and the CX6 is more significant when the load is increased. At 80% load, CC-NIC's 64B latency is 88% lower than the CX6 (85% lower than the E810). With large 1.5KB packets, minimum latency is 76% lower than the CX6 (87% lower than the E810). As with small packets, at 80% load, CC-NIC achieves a greater improvement over PCIe: 88% for both CX6 and E810.

Throughput. On ICX, CC-NIC demonstrates a maximum 64B packet rate of 330Mpps (169Gbps). This is a substantially higher packet rate than PCIe NICs on the same platform: 192Mpps (E810) and 76Mpps (CX6). For 1.5KB packets, CC-NIC reaches 403Gbps out of a maximum 443Gbps measured data throughput on the interconnect; both PCIe NICs reach their rated 200Gbps line rate on the 252Gbps PCIe link. While the ICX server core count limits CC-NIC's 64B packet rate, the SPR results demonstrate full interconnect utilization. The SPR CC-NIC loopback reaches a maximum packet rate of 1520Mpps (778Gbps) with 64B packets. Including the descriptor metadata transferred with each packet, this corresponds to 96% of the measured maximum UPI data throughput. For 1.5KB packets, we measure 986Gbps data throughput or 97% of UPI throughput.

Core count. With 64B packets, 48 of 56 (SPR) and 14 of 16 (ICX) host cores are required to reach 90% of the maximum rate. Large 4KB packets decrease the core counts to 18 (SPR) and 8 (ICX). Each core accesses full TX/RX payloads, placing a higher burden on the host cores than workloads such as forwarding with header-only accesses. We measure core utilization with application workloads in §5.4.7.

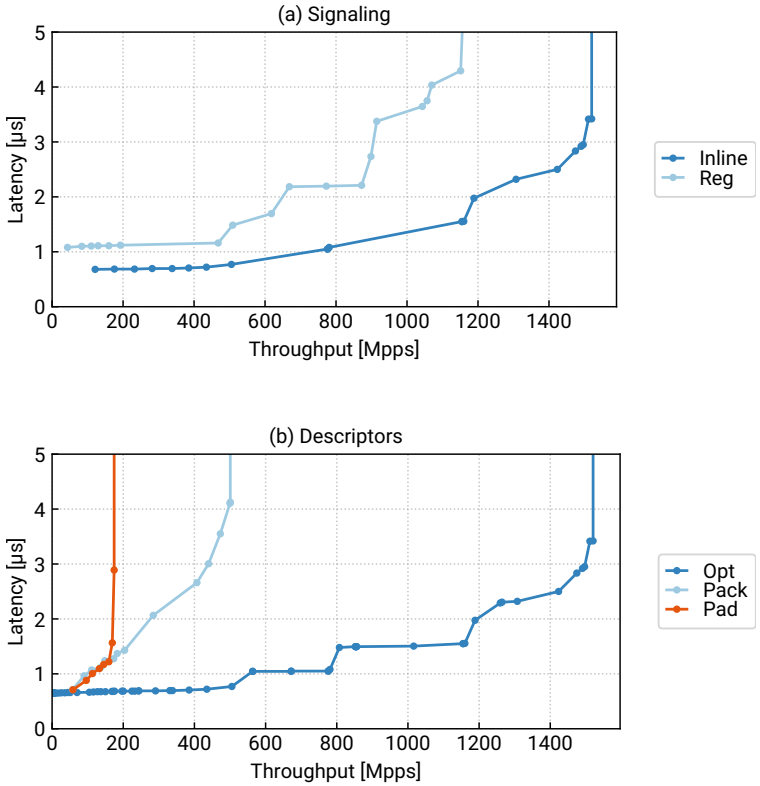

Figure 5.15: Throughput and latency varying (a) register and inline signaling options, and (b) descriptor layouts.

5.4.4 Design Feature Analysis

Next, we evaluate the impact of CC-NIC's design features. Recent work analyzing data center workloads in the context of transport protocols [75] and data stores [100] emphasize the prevalence of small packets. Thus, we examine small packet workloads and evaluate packet-handling efficiency.

Signal Inlining. Figure 5.15a shows the impact of inlining signals into the descriptor ring versus maintaining external queue tail doorbell registers. For 64B packets, inlined signals reduce minimum latency by 37% and increase maximum packet rate by 1.3×.

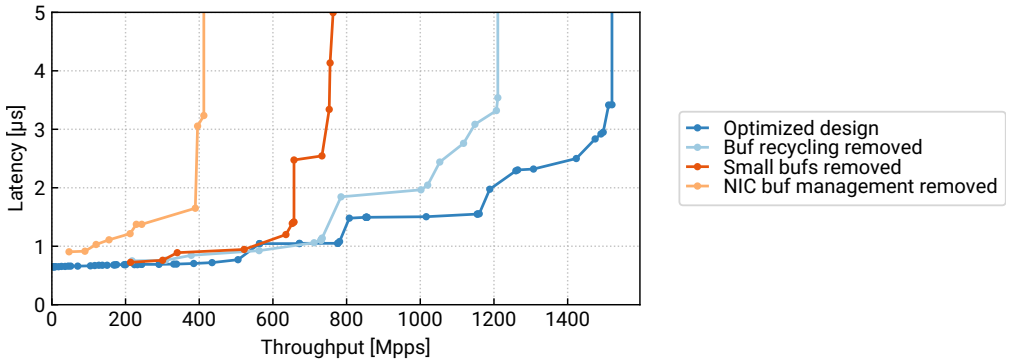

Figure 5.16: Performance impact of buffer management features.

Descriptor Layout. Using the same workload, we evaluate different descriptor layout choices: the optimized layout (§5.2.2), 16B descriptors equivalent to the E810 NIC (*pack* case), and the same format with each descriptor padded to a cache line (*pad*). Figure 5.15b shows the results. Due to the 64B granularity of UPI cache transfers and the direct descriptor polling required for inlined signals, memory layout substantially affects performance. Cache-aligning (padding) each descriptor achieves low latency by avoiding thrashing. Packing singleton 16B descriptors into a cache line improves throughput by 2.9× but causes thrashing as the host and NIC each access multiple signals and metadata fields per line. Finally, the optimized descriptor layout incorporates a single signal and a group of descriptors per cache line. This layout achieves a 3.0× throughput improvement while matching the best-case minimum latency of the padded case.

Buffer Management Optimizations. In Figure 5.16, we evaluate the performance impact of buffer management optimizations. We begin with the optimized design, removing features sequentially. All measurements use 56 SPR cores and 64B packets. First, we disable same-socket buffer reuse and nonsequential allocation (§5.2.3), so all allocations and frees access the buffer pool, not the buffer reuse cache for each core. This is representative of a workload where TX buffers are retained by the application after transmission, not returned to the buffer pool. We observe a 20% throughput reduction in this case. Second, we disable the small buffer optimization (§5.2.3), so each 64B packet is written into a separate 4KB buffer. This results in a larger shared

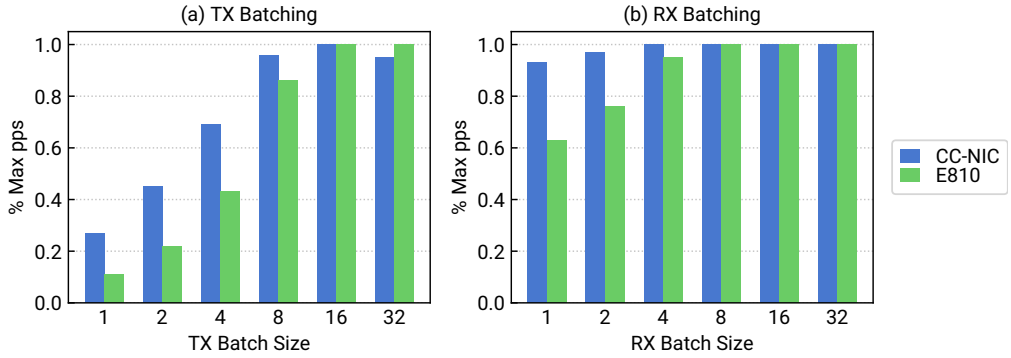

Figure 5.17: 64B packet rate relative to maximum, varying TX (a) and RX (b) batch sizes, for CC-NIC and E810 (PCIe).

memory footprint and a further 37% throughput decrease. Finally, we disable shared access to the buffer pool (§5.2.4), instead posting and freeing buffers exclusively on the host side. This change prevents the NIC from adaptively filling RX descriptors based on the available burst count and increases host bookkeeping, decreases maximum throughput by 46%, and increases latency by 1.3×. This final case is comparable to PCIe NIC buffer management.

5.4.5 Batching Effects

Batching is critical to achieving high NIC packet rates. For PCIe NICs, TX batching enables submitting multiple packets with one MMIO doorbell; larger batch sizes reduce the rate of MMIO operations. For CC-NIC, TX batching allows multiple descriptors to be transferred within a single cache line. Host-side RX batching primarily affects access patterns on the descriptor ring and buffer pool, determining whether buffers are handled individually or in bulk. Figure 5.17 shows 64B packet rate at a given host TX/RX batch size, relative to the highest achievable packet rate. We define batch size as the maximum number of buffers transmitted per polling loop iteration, i.e., the TX/RX *burst* count used in DPDK APIs. We repeat the experiment with CC-NIC and the E810 PCIe NIC, both on the ICX server. We vary the TX batch size while using a fixed RX batch size of 32, and vice versa. For TX, CC-NIC achieves higher packet rates with lower batching factors; the unbatched case shows 27% of peak throughput for CC-NIC versus 12% for the E810. Because

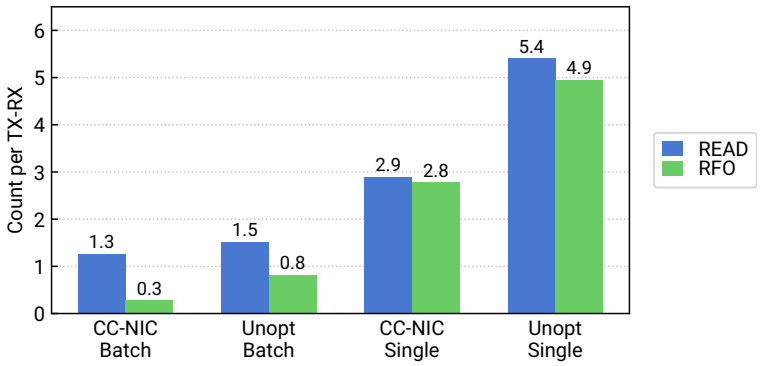

Figure 5.18: NIC remote accesses per TX-RX loopback, batched and singular descriptor cases.

CC-NIC uses lightweight per-cache-line signals instead of MMIO doorbells, the need for large batching factors is reduced; CC-NIC achieves peak packet rates when descriptor cache lines are filled. For poll-mode RX, host-side batching is less crucial to performance since the host does not perform PCIe MMIO accesses and releases RX descriptors lazily. Both NICs show significantly less sensitivity to the RX batch factor: CC-NIC maintains at least 93% of peak throughput across batch sizes, and the E810 achieves at least 63%.

5.4.6 Interconnect Communication

To measure the coherence communication required to facilitate host-NIC interactions, we measure *offcore response* PMU counters of the NIC CPU. As an additional comparison point, we deploy NIC and host threads on a single CPU. This setting eliminates UPI communication altogether, revealing the interconnect contribution to latency and bandwidth overheads.

Remote Access Counters. We measure *offcore response* PMU counters of the NIC and host CPU to quantify interconnect communication. Figure 5.18 compares the remote accesses performed by CC-NIC and the unoptimized UPI baseline per 64B TX-RX loopback operation. The figure shows NIC CPU accesses; due to the symmetric TX-RX design of CC-NIC, we observe symmetric host-side access counts. Each remote access consists of a *read* or *read for ownership*

(RFO) interconnect operation. We evaluate singleton and fully-batched (4 descriptors per cache line) cases. The batched case shows a throughput-oriented workload, processing descriptors in bursts of 8 and filling ring cache lines without wasted space. The singleton case represents a low-throughput, low-latency workload, where the host transmits one packet at a time and immediately polls for completion status. This case maximizes contention on shared cache lines, with the host and NIC accessing descriptors and signals simultaneously.

With the batched workload, CC-NIC performs one read access per packet, plus one read and one RFO per descriptor group (0.25 per packet). This suggests that CC-NIC effectively amortizes metadata cache transfers. The unoptimized UPI baseline, which uses register-based signaling, incurs one additional read and two additional RFO accesses per descriptor group. For the singular scenario, each individual packet requires full cache-line transfers for the descriptor, packet memory, and, in the unoptimized version, registers. When we compare batched and singular cases, our results show the importance of efficient descriptor cache-line layouts. Packing multiple descriptors into one cache line (§5.2.2) significantly reduces coherence communication, for both CC-NIC and the unoptimized case. Comparing optimized and unoptimized interface designs with the singleton workload, CC-NIC is able to recycle locally-cached buffer memory (§5.2.3) and avoid separate cache transfers for register signaling (§5.2.2). This reduces interconnect communication, even in the presence of contented host-NIC accesses.

Same-Socket Comparison. We deploy CC-NIC and host threads on a single NUMA node to understand the contribution of the UPI interconnect on loopback latency and per-thread throughput. This setting exhibits host-NIC interactions between local CPU cores, eliminating transfers across the UPI physical link. Figure 5.19 shows one-thread 64B loopback performance between host and CC-NIC threads on the same SPR CPU, compared to the cross-UPI deployment used for all other results. Comparing both minimum and loaded latencies shows that the interconnect accounts for approximately 40-50% of TX-RX loopback latency. The increased latency of cross-UPI accesses increases stalling for the host application, which impacts maximum per-thread throughput; the same-socket experiment shows 1.5× greater per-thread throughput.

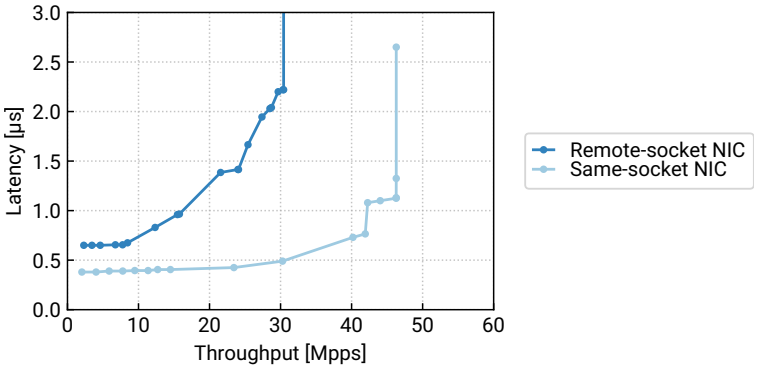

Figure 5.19: Single-thread 64B loopback performance, comparing CC-NIC thread running on local CPU versus cross-UPI remote CPU.

	PCIe Mops/s	CC-NIC Mops/s	Thread Count
KV store (ads)	37.0	42.3	16 → 8
KV store (geo)	17.8	17.9	8 → 4
TCP echo RPC	58.3	64.6	5 → 3

Table 5.2: Peak throughput and core count for KV Store and TCP Echo RPC applications, comparing CX6 and CC-NIC Overlay interfaces.

5.4.7 Application-level Performance

Table 5.2 summarizes the thread count reduction enabled by CC-NIC for the key-value store echo RPC applications discussed below. We compare the CC-NIC Overlay interface (forwarding to the CX6 PCIe NIC) to the direct interface with the CX6, both on the ICX platform.

Key-Value Store Throughput. We implement a key-value store based on the design of CliqueMap [100], with DPDK's `rte_hash` table as the index. Server threads poll NIC RX queues to handle get and set RPCs. Gets are zero-copy, applying multi-segment TX (DPDK `extbuf`) to submit the header and object payload to the NIC. This requires two buffer addresses per TX descriptor, increasing host-NIC metadata but avoiding object `memcpy`. We deploy the key-value store on one ICX server with a CX6 NIC, plus two remote clients, enough to saturate the server. We evaluate two production object distributions from Google, *Ads* and *Geo* [100], limiting sizes to a 9600B

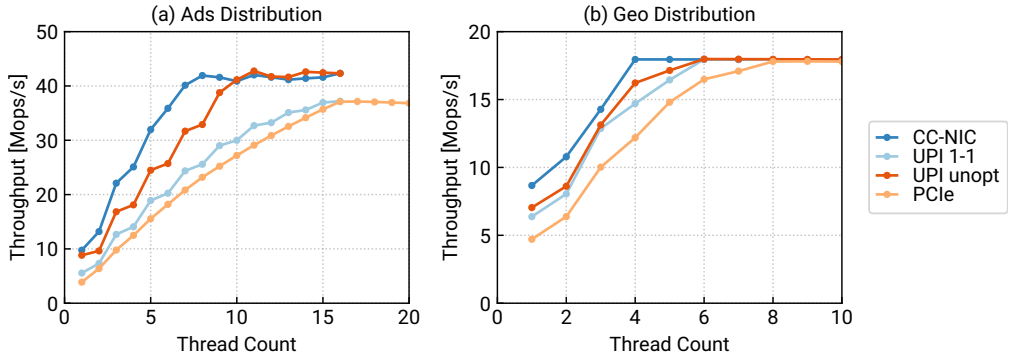

Figure 5.20: Throughput versus thread count for key-value store workloads, comparing CC-NIC Overlay and PCIe NIC interfaces.

MTU (truncating the largest 0.01% of Ads). Ads consists of smaller objects; 61% are less than 100B, compared to 13% in Geo. For both, we evaluate 95% gets, 5% sets on 1M objects, following a Zipf access pattern with a coefficient of 0.75.

Figure 5.20 shows key-value request throughput with CC-NIC and CX6 interfaces across the range of application thread counts (hyperthreading enabled). Since all scenarios perform TX/RX via the CX6 NIC, peak throughput is determined by its packet rate. However, the CC-NIC Overlay interface achieves peak throughput with fewer application threads. For Ads, 8 threads saturate throughput with the CC-NIC Overlay interface, compared to 16 with the CX6. The high rate of small objects stresses the host-NIC interface, especially with multi-segment TX. The Geo workload demonstrates a reduction of 8 to 4 threads, showing core savings with a distribution skewed towards larger objects. The *UPI 1-1* series uses one overlay thread per application thread. Relative to the direct CX6 interface, the same number of threads access PCIe NIC queues, but this work is offloaded from application threads. This increases per-thread throughput up to 31%, but the overlay thread count limits performance. Comparing CC-NIC to the unoptimized UPI (*unopt*) baseline shows the benefits of coherence-optimized buffer management: CC-NIC shows a savings of 3 (Ads) and 2 (Geo) threads at peak throughput.

TCP RPC Throughput. We evaluate an RPC server built using TAS [46], a high-performance userspace TCP service. We run the RPC server implemented by the TAS authors, a basic TCP application dynamically linked to TAS, overriding the kernel sockets interface. A set of userspace TAS *fast-path* threads to handle the TCP data plane via DPDK, achieving state-of-the-art TCP performance. We replace TAS's fast-path PCIe TX/RX with the CC-NIC Overlay to evaluate the benefits of a coherent NIC interface. We do not offload any aspects of TAS but instead deploy a drop-in replacement NIC interface.

We evaluate a workload of 64B echo RPCs, deploying one application thread and measuring the number of TAS fast-path threads required to achieve 95% peak throughput. In the PCIe baseline case, we run all application and TAS threads on the CX6's local socket CPU. For the CC-NIC Overlay case, we deploy overlay threads on the CX6 socket and all TAS and application threads on the remote CPU. On a second machine, we run the client application with all threads and a total of 96 flows, enough to saturate the server. Table 5.2 compares RPC throughput with the CC-NIC Overlay and direct CX6 interfaces. Applying the CC-NIC Overlay results in NIC saturation with 3 TAS threads versus 5 with the PCIe interface. The CX6 case shows slightly lower peak throughput due to internal TAS overheads, which increase with the fast-path thread count. Both scenarios are limited to the CX6 NIC packet rate.

5.4.8 Sensitivity to Hardware Prefetching

In Figure 5.21, we compare the impact of hardware prefetching on packet rates for CC-NIC and the unoptimized UPI baseline, on the SPR platform. We enable prefetching on the host, NIC, and both CPUs, measuring packet rate relative to the case of prefetching disabled. We find that the optimized CC-NIC interface is able to benefit from host-side prefetching for small-packet workloads: prefetching increases packet rate 1.2× for 64B packets. This gain comes from the CPU's *DCU IP Prefetcher* and affects packet buffer accesses in particular. Both designs achieve maximum throughput with prefetching enabled on the host CPU only (we use this setting for all other experiments). For the unoptimized interface design, without CC-NIC's locality-oriented buffer pool optimizations (§5.2.3), prefetching strictly decreases performance by up to 7%. These differ-

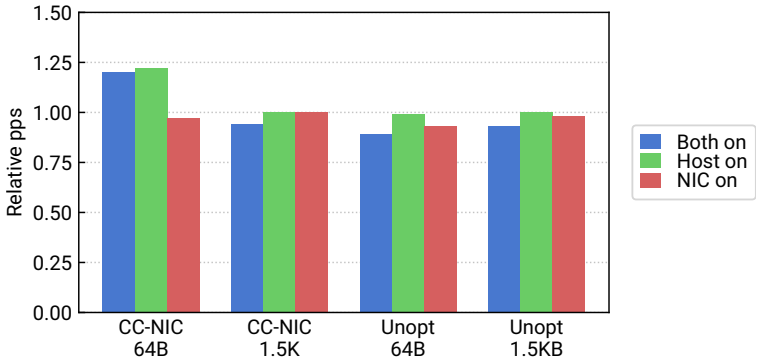

Figure 5.21: Impact of SPR hardware prefetching on 64B packet throughput, relative to the prefetching-disabled case.

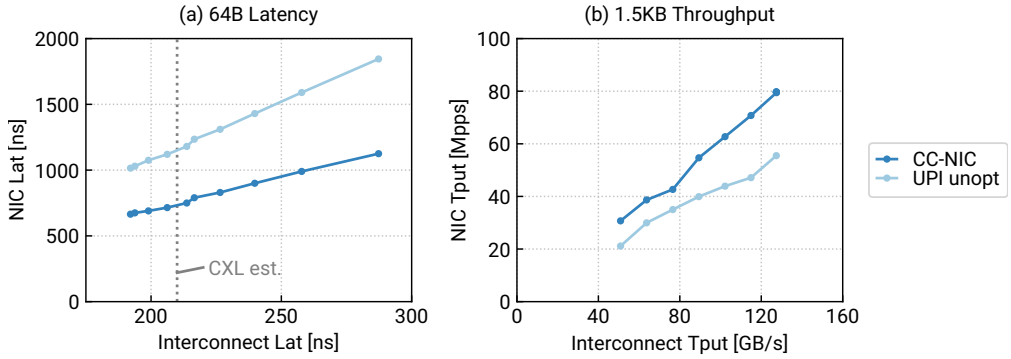

Figure 5.22: Performance with reduced UPI throughput and latency.

ences suggest that the NIC interface design dictates whether prefetching improves performance or increases interconnect overheads.

5.4.9 Sensitivity to Interconnect Performance

We analyze CC-NIC's sensitivity to interconnect bandwidth and latency by varying the NIC socket uncore frequency. This allows us to study CC-NIC under reduced interconnect performance. However, this approach results in pessimistic measurements, as downclocking the uncore impacts purely local access performance in addition to remote UPI accesses. Across the range of supported uncore frequencies (maximum is the default), we measure host-to-NIC-socket DRAM

access latency and read throughput and loopback performance with CC-NIC and the unoptimized UPI interface. Figure 5.22 shows 64B packet loopback latency relative to access latency, and 1.5KB packet throughput relative to interconnect data throughput, measured on the SPR server.

According to the CXL Consortium, the expected access latency for CXL-attached DRAM is 170-250ns [90]. This range is corroborated by research on CXL.mem prototypes, which finds that CXL.mem load latency is approximately 1.5× higher than cross-UPI remote DRAM [103]. In Figure 5.22a, we observe that CC-NIC's latency increase closely tracks the increase in host-to-NIC interconnect access latency. With a 1.11× increase in interconnect latency to 211ns (middle of the CXL range), CC-NIC loopback latency increases by 1.13×. CC-NIC maintains its relative improvement over the unoptimized UPI interface, which incurs a 1.16× latency increase. Figure 5.22b shows that performance is also stable over a range of interconnect throughputs. 1.5KB loopback throughput scales well and maintains a consistent improvement over the unoptimized case. When interconnect throughput is set to a minimum of 40%, CC-NIC throughput is 39%.

5.5 Discussion

Hardware DMA. Hardware bulk transfers, on both host and NIC sides, can potentially increase efficiency over CPU accesses. While our application-level results (§5.4.7) show that CC-NIC can reduce core utilization without DMA, efficient hardware transfers could benefit large-packet workloads. On-chip DMA engines, such as Intel's Data Streaming Accelerator [37], are one possible mechanism for CPU-initiated bulk transfers. For device-initiated DMA, a CXL-attached NIC could leverage both CXL.cache for metadata and small packet transfers, plus CXL.io DMA for bulk packet operations.

Security and Isolation. We expect coherent host-device interconnect standards to provide mechanisms for protecting and isolating host resources. We expect that current techniques to control PCIe DMA access to the host address space, e.g., IOMMU translation, apply to coherent device accesses. Likewise, the BAR space abstraction of current PCIe devices offers a means of isolation by limiting host-device coherence to a portion of the address space.

Network Function Workloads. While we studied workloads involving full packet access,

cache-coherent NICs could bring additional benefits for middlebox workloads like packet switching. Packet-switching through a PCIe NIC incurs unneeded interconnect and memory bandwidth utilization. Even if the application only operates on packet headers, the full packet payload is still transferred to and from host memory. In the case of DDIO, this may result in cache pollution. Instead, a coherent NIC may retain payloads in the NIC cache while the host operates on the header, avoiding interconnect transfers for packet data the host does not access.

5.6 Related Work

TinyNF [88], NIQ [22], PacketMill [19], and others [41, 23, 94, 46, 40, 29, 98, 62, 14] propose optimizations to the host software interface of PCIe NICs through the elimination of driver and stack overheads. Since our work maintains the packet queue model, these optimizations carry to the software stack running atop a coherent NIC interface.

NanoPU [32], Direct Cache Access [31], and Semi-Coherent DMA [72] propose new CPU-NIC data paths. Like our work, these systems demonstrate that tighter integration between the NIC and CPU caches enables higher performance. Rather than propose new data paths, our work leverages the faster paths of an existing cache-coherent interconnect.

Scale-out NUMA [78], and Dagger [49] apply cache coherence in conjunction with new communication models beyond NIC packet RX/TX. Scale-out NUMA enables remote coherent access to host memory by integrating an RDMA-like interface with the cache hierarchy. Dagger applies a UPI-attached FPGA as a target for offloaded RPCs with coherent host access. Our work focuses specifically on optimizing the producer-consumer data transfers associated with packet RX/TX. However, these systems and others [15, 114, 73, 74, 42] apply similar producer-consumer interactions, e.g., RDMA work queues. In the context of a coherent interconnect, the design we propose applies to these data structures.

Prior work on microkernel and shared-memory message passing [5, 93], as well as IO virtualization [111, 112], describes optimizations for producer-consumer accesses in the shared memory setting. With coherent host-device interconnects, these considerations (e.g., optimizing for cache-to-cache transfers) become newly important to host-device interactions. The specific context of

NIC interactions presents new opportunities for optimization and requires a specialized design. For instance, while existing message queue systems optimize for cache alignment, applying it to NIC RX queues requires broader changes to the buffer management system. CC-NIC applies a new combination of design decisions to optimize for the unique properties of both host-device coherence and NIC TX/RX descriptor communication.

Pond [53] and DirectCXL [25] explore CXL as a means of providing disaggregated memory resources. Their analysis of CXL datapath performance pertains to CXL-attached NIC interactions.

5.7 Conclusion

This work makes the case for redesigning the host-NIC software interface in the context of emerging cache-coherent interconnects. These interconnects are capable of high performance, but the interface design of current PCIe NICs performs poorly in the coherent setting. We present CC-NIC, a NIC interface designed to benefit from cache coherence. Our results, modeling CC-NIC over the coherent UPI interconnect, demonstrate high throughput, low latency, and CPU-efficient host-NIC communication.

Chapter 6

CONCLUSION

This thesis makes the case for tight integration of NIC datapaths at multiple levels, to meet the increasing demands of performance and efficiency imposed by today's workloads.

First, we perform a packet-processing measurement study of current SmartNICs. Our results show benefits to the efficient core-to-NIC data paths and hardware-accelerated packet handling functionality of some devices. These benefits create the potential for latency reduction and reduced CPU utilization by offloading work to the SoC. These efficient interfaces also demonstrate the potential for improvement relative to current PCIe host-NIC interfaces. With these insights, we propose two techniques for accelerating networked systems: coupling system design with SmartNIC resources, and bringing NIC interfaces closer to the server CPU by leveraging coherent interconnects.

Xenic serves as a case study of integrating SmartNIC resources with distributed systems design. Existing approaches to offloading and acceleration for distributed transactions come with significant tradeoffs in terms of network communication efficiency and CPU utilization. SmartNICs are a flexible alternative for accelerating distributed transactions, adding programmable compute cores and on-board memory to the network interface. The design of Xenic includes specialized data structures, transaction protocols, and selective offloading of work to the SmartNIC, to attain these benefits while avoiding challenges of SmartNIC integration: limited SoC resources and high-cost PCIe communication between the host and NIC. By designing Xenic and its data structures with careful attention to the SmartNIC's resources and data paths, we demonstrate significant improvements in throughput, latency, and core utilization.

CC-NIC, a cache-coherent host-NIC interface design, is a step towards improving upon the high-cost PCIe communication of today's server architectures. We argue that coherent inter-

connects offer a significant opportunity for performance and efficiency improvement, but the coherence abstraction is not suited to current host-NIC access patterns. We present CC-NIC, a redesigned host-NIC interface, which benefits from the new interactions coherence enables. We implement a CC-NIC prototype using Intel's coherent UPI interconnect, and demonstrate high packet rates, low latency, and core savings relative to existing PCIe NIC interfaces. Our results also demonstrate the ability to sustain terabit network traffic on currently-available coherent interconnects, which is only possible using the optimized CC-NIC interface design. CC-NIC demonstrates the potential of tighter integration between the host server and NIC, by allowing the device to participate in the server CPU's coherence domain.

6.1 Future Work

This thesis serves as a foundation for applying SmartNICs to a broader range of networked systems. Future work includes exploring SmartNIC integration of data structures and protocol logic for systems beyond distributed transactions, such as replication and caching services, as well as more sophisticated database designs, such as ordered index structures. Additionally, the methodologies applied to design Xenic could be generalized. This may take the form of a library of optimized host-SmartNIC shared data structures, integrated with an execution environment like the existing iPipe framework [59]. At the same time, emerging coherent interconnects have the potential to simplify host-SmartNIC data sharing. This may be a compelling alternative to the specialized PCIe-friendly data structures required today. A key next step arising from CC-NIC is to explore the additional benefits cache coherence provides to host-SmartNIC interactions. Likewise, CC-NIC's performance benefits, which arise from tight core-to-NIC coupling, suggest a future where the NIC is closer to the CPU than today's peripheral cards. Chiplet interconnects, such as UCIe, make this possible at a hardware level. The design of CC-NIC translates to the context of UCIe and other coherent interconnects. The SoC integration designs applied by SmartNICs may also translate to server CPUs, resulting in full hardware integration of the Ethernet controller. This changes the calculation of SmartNIC benefits, by bringing the beneficial architectural features of the SmartNIC SoC to the host server. These developments maintain

the same motivations behind the designs of Xenic and CC-NIC: namely, the performance and efficiency benefits of efficient NIC datapaths.

BIBLIOGRAPHY

[1] Alpha Data. ADM-PCIE-9V3 - High-Performance Network Accelerator, September 2021. https://www.alpha-data.com/dcp/products.php?product=adm-pcie-9v3.

[2] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020.

[3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, Santa Clara, CA, February 2020. USENIX Association.

[4] Austin Appleby. MurmurHash, September 2021. https://sites.google.com/site/murmurhash/.

[5] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, may 1991.

[6] Timo Bingmann. pmbw - parallel memory bandwidth benchmark. https://panthema.net/2013/pmbw/.

[7] Broadcom. The TruFlow Flow processing engine. https://www.broadcom.com/applications/data-center/cloud-scale-networking, 2021.

[8] Broadcom Inc. Stingray SmartNIC Adapters and IC, September 2021. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/smartnic.

[9] CCIX Consortium Inc. CCIX Base Specification 1.0. https://www.ccixconsortium.com/library/specification/.

[10] Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin Hood Hashing (Preliminary Report). In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 281–288. IEEE Computer Society, 1985.

[11] 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*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Compute Express Link Consortium Inc. CXL 3.0 Specification. https://www.computeexpresslink.org/download-the-specification.

[13] Douglas W. Cornell, Daniel M. Dias, and Philip S. Yu. On Multisystem Coupling through Function Request Shipping. *IEEE Transactions on Software Engineering*, SE-12(10):1006–1017, 1986.

[14] DPDK Project. Data Plane Development Kit. https://www.dpdk.org/.

[15] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*. USENIX – Advanced Computing Systems Association, April 2014.

[16] 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*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[17] EEMBC. Coremark. https://www.eembc.org/coremark/.

[18] Exablaze. ExaNIC V5P High Density Network Application Card, September 2021. https://exablaze.com/exanic-v5p.

[19] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-Core 100-Gbps Networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 2020.

[21] 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.

[22] Mario Flajslik and Mendel Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 333–346, San Jose, CA, June 2013. USENIX Association.

[23] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of Frameworks for High-Performance Packet IO. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, page 29–38, USA, 2015. IEEE Computer Society.

[24] Gen-Z Consortium. Gen-Z Specifications. https://genzconsortium.org/specifications/.

[25] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[26] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2020.

[27] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[28] H-Store Project. SmallBank Benchmark - H-Store, September 2021. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[29] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[30] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 135–148, Hollywood, CA, October 2012. USENIX Association.

[31] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, 2005.

[32] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[33] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. The Case for a Network Fast Path to the CPU. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 52–59, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. https://www.intel.ca/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[35] Intel Corporation. Intel Architecture Day 2021. https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf.

[36] Intel Corporation. Intel Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[37] Intel Corporation. Intel Data Streaming Accelerator Architecture Specification. https://cdrdv2-public.intel.com/671116/341204-intel-data-streaming-accelerator-spec.pdf.

[38] Intel Corporation. Intel Memory Latency Checker v3.9a. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[39] Intel DPDK Validation Team. Intel Ethernet Performance Report with DPDK 21.11. http://fast.dpdk.org/doc/perf/DPDK_21_11_Intel_NIC_performance_report.pdf.

[40] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[41] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[42] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 295–306, New York, NY, USA, 2014. Association for Computing Machinery.

[43] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[44] 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)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[45] 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.

[46] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[47] 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.

[48] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs. In

Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, page 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael Swift, and T. Lakshman. Uno: uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 506–519, 09 2017.

[51] 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.

[52] 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.

[53] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Sheng Li, Kevin Lim, Paolo Faraboschi, Jichuan Chang, Parthasarathy Ranganathan, and Norman P. Jouppi. System-Level Integrated Server Architectures for Scale-out Datacenters. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 260–271, New York, NY, USA, 2011. Association for Computing Machinery.

[55] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

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

[57] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable kernel tcp design and implementation for short-lived connections. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 339–352, New York, NY, USA, 2016. Association for Computing Machinery.

[58] Lisa Spelman. Updates on Intel's Next-Gen Data Center Platform, Sapphire Rapids. https://www.intel.com/content/www/us/en/newsroom/opinion/updates-next-gen-data-center-platform-sapphire-rapids.html.

[59] 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*, 2019.

[60] 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*, 2017.

[61] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, Renton, WA, July 2019. USENIX Association.

[62] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[63] Marvell Technology Group Ltd. LiquidIO III Solutions Brief, September 2021. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf.

[64] Marvell Technology Group Ltd. Multi-Core Processors - LiquidIO Smart NICs | Network adapter, September 2021. https://www.marvell.com/products/infrastructure-processors/multi-core-processors/liquidio-smart-nics.html.

[65] Francis Matus. Pensando: Distributed Services Architecture. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–17. IEEE Computer Society, 2020.

[66] Mellanox. Accelerated Switch and Packet Processing. http://www.mellanox.com/page/asap2?mtag=asap2, 2021.

[67] Mellanox. BlueField SmartNIC Ethernet, September 2021. https://www.mellanox.com/products/BlueField-SmartNIC-Ethernet.

[68] Mellanox. ConnectX-5 EN Single/Dual-Port Adapter, September 2021. https://www.mellanox.com/products/ethernet-adapters/connectx-5-en.

[69] Mellanox. Mellanox Innova SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic, 2021.

[70] Mellanox. OFED Documentation Rev 7.4.1.0.0.1, September 2021. https://docs.mellanox.com/display/MLNXOFEDv471001.

[71] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1–18, Santa Clara, CA, February 2020. USENIX Association.

[72] Seungwon Min, Mohammad Alian, Wen-Mei Hwu, and Nam Sung Kim. Semi-Coherent DMA: An Alternative I/O Coherency Management for Embedded Systems. *IEEE Computer Architecture Letters*, 17(2):221–224, 2018.

[73] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

[74] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 451–464, Denver, CO, June 2016. USENIX Association.

[75] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[76] Netronome. Agilio LX SmartNICs, September 2021. https://www.netronome.com/products/agilio-cx/.

[77] 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*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[78] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 3–18, New York, NY, USA, 2014. Association for Computing Machinery.

[79] NVIDIA Corporation. NVIDIA Mellanox NICs Performance Report with DPDK 21.11. http://fast.dpdk.org/doc/perf/DPDK_21_11_Mellanox_NIC_performance_report.pdf.

[80] NVIDIA Corporation. NVIDIA NVSwitch Technical Overview. https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf.

[81] OpenCAPI Consortium. OpenCAPI Specifications. https://opencapi.org/technical/specifications/.

[82] OpenFabrics Alliance. libibverbs Library, September 2021. https://downloads.openfabrics.org/libibverbs/.

[83] Fazil Osman. Distinguished Engineer, Broadcom. Private communication.

[84] PCI-SIG. PCI Express Specifications. https://pcisig.com/specifications/.

[85] Pensando. Pensando DSC-100 Distributed Services Card, September 2021. https://pensando.io/documents/pensando-dsc-100-distributed-services-card/.

[86] Pensando Floor Plan. https://www.servethehome.com/pensando-distributed-services-architecture-smartnic/, 2021.

[87] 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*, 2018.

[88] Solal Pirelli and George Candea. A Simpler and Faster NIC Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 225–241. USENIX Association, November 2020.

[89] PK Gupta. Intel Xeon+FPGA Platform for the Data Center. https://reconfigurablecomputing4themasses.net/files/2.2%20PK.pdf.

[90] Prakash Chauhan and Mahesh Wagh. CXL Memory Challenges. https://hc34.hotchips.org/assets/program/tutorials/CXL/Hot%20Chips%202022%20CXL%20MemoryChallenges.pdf.

[91] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 475–488, Seattle, WA, April 2014. USENIX Association.

[92] Redis. Retwis - Example Twitter clone based on the Redis Key-Value DB, September 2021. http://retwis.redis.io.

[93] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 927–942. USENIX Association, July 2020.

[94] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.

[95] Luigi Rizzo, Paolo Valente, Giuseppe Lettieri, and Vincenzo Maffione. PSPAT: Software packet scheduling at hardware speed. *Computer Communications*, 120, 02 2018.

[96] Karl Rupp. 42 years of microprocessor trend data. https://github.com/karlrupp/microprocessor-trend-data.

[97] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[98] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack—Highly Efficient Network Processing on Dedicated Cores. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[99] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, Renton, WA, April 2022. USENIX Association.

[100] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 93–105, New York, NY, USA, 2021. Association for Computing Machinery.

[101] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 33–46, Boston, MA, February 2019. USENIX Association.

[102] Brent Stephens, Aditya Akella, and Michael M. Swift. Your Programmable NIC Should Be a Programmable Switch. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 36–42, New York, NY, USA, 2018. Association for Computing Machinery.

[103] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices, 2023.

[104] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr. Sharma, Arvind Krishnamurthy, Dan R. K. Ports, and Irene Zhang. Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[105] The Linux Kernel Archives. Linux Base Driver for the Intel Ethernet Controller 700 Series. https://www.kernel.org/doc/html/latest/networking/device_drivers/ethernet/intel/i40e.html.

[106] Timothy Morgan. Finally, A Coherent Interconnect Strategy: CXL Absorbs Gen-Z. https://www.nextplatform.com/2021/11/23/finally-a-coherent-interconnect-strategy-cxl-absorbs-gen-z/.

[107] Transaction Processing Performance Council. TPC Benchmark C Standard Specification, Revision 5.11, September 2021. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[108] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[109] Universal Chiplet Interconnect Express. UCIe 1.0 Specification. https://www.uciexpress.org/specification.

[110] Universal Chiplet Interconnect Express. UCIe 1.0 Specification. https://www.uciexpress.org/specification.

[111] Virtio. Libvirt Virtualization API. https://wiki.libvirt.org/Virtio.html.

[112] VMWare Incorporated. Performance Evaluation of VMXNET3 Virtual Network Device. https://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf.

[113] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, Carlsbad, CA, October 2018. USENIX Association.

[114] 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*, SOSP '15, page 87–104, New York, NY, USA, 2015. Association for Computing Machinery.

[115] Xilinx. Alveo Adaptable Accelerator Cards for Data Center Workloads, September 2021. https://www.xilinx.com/products/boards-and-kits/alveo.html.

[116] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't Forget the I/O When Allocating Your LLC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021.

[117] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.*, 35(4), December 2018.