OPTIMIZING REMOTE PROCEDURE CALLS IN DATACENTERS USING HARDWARE/SOFTWARE CO-DESIGN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Stephen Ibanez
March 2021

Abstract

In this dissertation, we present the nanoPU, a new NIC-CPU co-design that provides ultra low and predictable remote procedure call (RPC) response time and thus accelerates datacenter applications. The nanoPU achieves its goal by providing a *fast path* between the network and applications. This fast path has the following three characteristics: (1) it moves key resource scheduling decisions from software to hardware (reliable network transport & congestion control, RPC load balancing across cores, thread scheduling) allowing them to operate much more efficiently, (2) it provides a path directly between the network and applications which bypasses the cache and memory hierarchy placing arriving messages directly into the CPU register file, and (3) it supports a unique thread scheduling feature to *bound* the tail response time experienced by certain high-priority applications.

We built an FPGA prototype of the nanoPU fast path by modifying an open-source RISC-V CPU, and evaluated its performance using cycle-accurate simulations on AWS FPGAs. The wire-to-wire time for nanoPU to receive an incoming message and initiate transmission of a response (*response time*) is just 69ns, an order of magnitude quicker than the best-of-breed, low latency, commercial NICs. Our hardware implementation of the NDP transport protocol adds less than 10ns to the wire-to-wire response time. We demonstrate that the hardware thread scheduler is able to lower (and potentially bound) RPC tail response time by about 5× while enabling the system to sustain 20% higher load, relative to traditional thread scheduling techniques. Furthermore, the nanoPU's core selection policy in hardware is able to efficiently distribute RPCs across cores eliminating hot spots and reducing tail response time. We implement and evaluate a suite of applications on the nanoPU, including MICA, Raft, and set algebra for document retrieval.

Acknowledgments

The PhD has been an incredible journey and I have matured significantly as both a researcher and as a teacher. I was fortunate to have had the opportunity to work with an incredible advisor, Nick McKeown. Nick taught me how to conduct interesting and impactful research. He showed me how to focus on the important questions and to always keep the big picture in mind. His persistent reminders to question conventional wisdom and to always push the envelope will stick with me for the rest of my career. Nick's never ending intellectual curiosity demonstrates that learning does not stop once we complete the PhD, so I look forward to lifetime of learning.

Gordon Brebner was another individual that has significantly shaped me as a researcher. I consider him to be my informal Ph.D. co-advisor. Gordon taught me to love clean abstractions when building programmable systems and he was always able to offer an interesting perspective on challenging problems. I am grateful for the years of mentorship and guidance that he has given me.

John Ousterhout is the third member of my dissertation reading committee. John's passion for building real systems and his persistence to do it well is incredibly inspiring. I appreciate all of the useful feedback that he has given me during my time at Stanford.

Over the course of my Ph.D. I was very fortunate to be able to work with incredible people in both industry and academia. Chris Neely and Robert Halstead from Xilinx Labs, as well as Noa Zilberman from the University of Cambridge, were instrumental in helping to develop the P4→NetFPGA workflow [38]. S. Mohan (Xilinx Labs) also provided much appreciated advice and assistance over the years. I learned a great deal of practical FPGA development skills from Anthony Dalleggio at NYU. Gianni Antichi was a great friend and an enthusiastic collaborator. I enjoyed our numerous brainstorming sessions as well as sharing an office with him during the Summer of 2017 at the University of Cambridge.

I would also like to acknowledge my wonderful Stanford collaborators. Alex Mallery, Muhammad Shahbaz, Serhat Arslan, Theo Jepsen, and Changhoon Kim all contributed to the work presented in this dissertation. Lavanya Jose was one of my mentors during the early years of my PhD. Sarah Tollman helped me to develop CS344 and was a fantastic TA. To the entire McKeown Group, both past and present, thank you for all the stimulating discussions and helpful feedback over the years. I count myself lucky to be associated with such a talented and kind group of individuals. I would like to thank the Stanford Platform Lab, as well as Xilinx and Google for funding my Ph.D.

The P4 community provided an ideal setting for me to develop and improve my research ideas. In particular, Nate Foster and Robert Soulé were always willing to discuss ideas and teach P4 tutorials with me.

To my parents, Carlos and Elaine, my brother Nick and sister Anna, thank you for the continuous love and support. And finally, my wife Michelle Ibanez has been by my side throughout the entire Ph.D. I am grateful for her unwavering support and her honest advice during difficult times.

My Ph.D. was filled with incredible memories that I will cherish for the rest of my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Modern datacenter applications are implemented as a collection of self-contained software modules (services) that communicate with one another using remote procedure calls (RPCs). A single user query can, in turn, generate a sequence of RPC fanouts that span thousands of services across thousands of servers.

One of the reasons that applications are implemented using this approach is to reduce the end-to-end runtime by harnessing many servers in parallel. In fact, researchers have demonstrated that by harnessing thousands of cores in parallel, the runtime of highly parallelizable applications such as video encoding [31], object classification [5], software compilation [30], and map-reduce style analytics [43, 53, 81] can be reduced from hours to minutes. For example, ExCamera [31] demonstrated that by harnessing almost 4000 cores in parallel, video encoding time can be reduced from two hours on a single machine to just two minutes.

This speedup, though impressive, begs the question: why use only 4000 cores? Why not 400,000 cores? Many applications would benefit from being able to harness hundreds of thousands (or even millions) of cores, and if tasks are divided up with such fine granularity such that the working set of each task is cache-resident, then we can expect significant reductions in application completion times. However, in order to make this degree of scalability practical, we need to solve a major problem: RPC tail response time. Tail response time refers to the response time of the slowest RPC requests—that is, the response time at the tail of the distribution. The reason tail response time matters so much is because applications need to wait for the slowest request to complete in order for the job to complete. As a result, the application completion time is very sensitive to tail response time.

To better understand the impact of RPC tail response time on application runtime, let us examine an example from Luiz Barroso's book "The Datacenter as a Computer" [10]. Consider a hypothetical

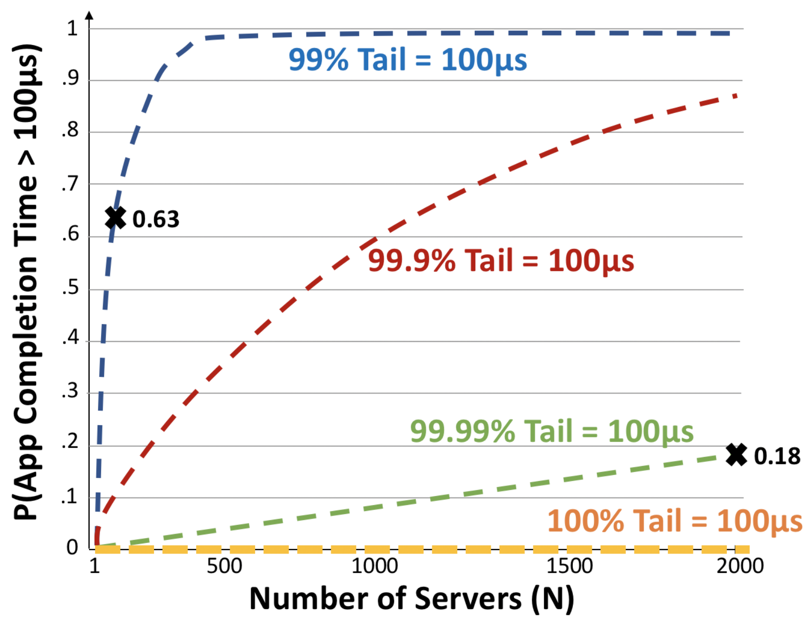

Figure 1.1: A plot depicting how the probability of application completion time is affected by both the number of servers harnessed in parallel and the tail response time.

system where each server usually responds in 1μs but has a 99% tail response time of 100μs. To run an application, RPCs are issued across N servers in parallel. If we run the application using just 1 server ($N = 1$), then 1 in 100 times it will be slow, taking 100μs or longer. What happens to the application completion time as we try to harness the computing power of more servers in parallel? The blue line in Figure 1.1 depicts the probability that the application will take longer than 100μs to complete as it scales to harness more servers. It is clear that if the application tries to use only 100 servers in parallel, there will be a 63% chance that it will take longer than 100μs to complete. If we can reduce tail response time, then the application can harness more servers in parallel without increasing completion time. However, note that even if we are able to reduce the 99.99% tail response time to 100μs, there is still almost a 20% chance that the application will be slow when scaling to just 2000 servers. Clearly, enabling applications to scale to hundreds of thousands of servers will require drastic reductions in tail response time.

What if we could actually *bound* the response time of each server to 100μs? That would mean, for this hypothetical application, that the completion time would be bounded; the application could use as many servers in parallel as it desires and would never take longer than 100μs to complete. Is this possible? *Can we bound the response time of each server?* Conventional wisdom says it is not possible [10]; however, I believe that good research requires us to question conventional wisdom.

Furthermore, as applications scale out into fine-grained software modules, the communication to computation ratio increases; more RPCs are generated, and the time required to process each RPC (i.e., service time) decreases. It is already common for RPC service times to last only a few microseconds [44, 72, 45], and it is reasonable to expect nanosecond scale service times in the future as applications continue to scale out for performance gains. In light of this trend, it is becoming increasingly important to handle RPCs efficiently; future datacenter infrastructure must aim to minimize overheads associated with network communication in addition to RPC tail response time.

1.2 Primary Contributions

Our key finding is that both high RPC tail response times and high communication overheads stem from the fact that the network interface is treated as a second-class citizen of the CPU architecture. Modern CPU architectures are only optimized to load data from memory, compute, and store the result back in memory; they are not optimized to perform network IO. There was a time when it made sense to treat network IO like disk IO, back when the network was slow and applications used it sparingly. However, this is no longer true. In a modern datacenter, the latency through the network fabric is closer to a memory access (hundreds of nanoseconds) than a disk access (tens of milliseconds), and virtually all applications make extensive use of the network. As a result, we argue that it is time to rethink the traditional CPU architecture.

My thesis is as follows: *it is possible to achieve extremely efficient RPCs with low and predictable (potentially even bounded) response times by offloading the entire RPC stack into hardware.* To demonstrate this claim, we design and build the nanoPU, a novel CPU-NIC architecture, which is based upon our proposal for a fast-path to the CPU [39]. The challenge here is to develop a simple and elegant hardware architecture that is performant but also flexible enough to cope with the diverse requirements of modern networks. At the same time, the networking hardware must have a simple software interface that enables low overhead network communication. Before describing the key characteristics of the nanoPU, we will explain, in more detail, the problems with modern RPC processing systems.

1.3 Modern RPC Processing

In general, high tail latency in queueing systems is caused by poor resource-scheduling decisions. In the context of RPC processing, the relevant resources are network fabric resources, CPU cores, host memory bandwidth, and cache space. Modern systems are ill-equipped to efficiently schedule access to these resources for network-intensive workloads. To explain why, we will walk through the steps involved in processing an RPC request on modern systems.

When a client sends an RPC request to a remote server, the transport logic splits the request into one or more network packets and then schedules those packets on the network fabric resources.

Suboptimal congestion control policies, such as TCP, are in widespread use today, and TCP is very ill-suited for low-latency RPCs for a number of reasons. For instance, it explicitly tries to shove more and more data into the network until a packet is dropped. Dropped packets need to be detected and then retransmitted, which significantly increases the latency through the network, especially for small messages. Furthermore, the large queues caused by TCP increase the latency for all packets through the network, even the ones that have not been dropped. In fact, it is common for modern switches to have enough buffering to add up to somewhere between 2 and 5 ms of latency. We are aiming for hundreds of nanoseconds; thus, this is a massive slow down—4 orders of magnitude! Another reason TCP is ill-suited for low latency RPCs is because it is a streaming-based protocol where communication between applications often uses the same long-lived connection, which means short messages can get queued behind long messages, increasing tail latency through the network.

Researchers have recognized these problems and have proposed a number of new and improved transport protocols, such as NDP [33], Homa [66], and HPCC [59], which are much more well-suited to provide low latency through the network. We aim to enable efficient implementations of these protocols by providing a programmable platform that will allow network operators to deploy any of them in line rate in hardware.

Once the packets of an RPC request arrive at the target server two scheduling decisions are made in an attempt to make efficient use of the CPU cores: (1) a core selection policy schedules network data across cores to try to ensure an even load distribution, and (2) a thread scheduling policy multiplexes software threads on each core. Both of these decisions are implemented inefficiently on modern systems and lead to high tail response time.

First, let us examine core selection (i.e., load balancing across cores). Modern systems implement transport logic in software. This means that there are usually two core selection decisions that take place: one to balance *packets* across cores that are used to implement transport logic, and another to balance *messages* across cores that are running application logic. As network speeds continue to increase, more and more cores will be needed to implement software transport processing. To provide one concrete data point, Google Snap [63], a recent userspace networking stack, requires 9-14 cores to drive a 100Gb/s network at 80% utilization. Considering that this is about 20% of the total cores in a server, software-based transport is already very expensive.

The *packet* load balancing decision is made using a technique called Receive Side Scaling (RSS) in the NIC, which operates by hashing some packet header fields and assigning packets to cores based on the hash result. In practice, this does not always work well, and packets are occasionally sent to heavily loaded cores, resulting in high tail response time. Once the NIC decides which core to dispatch the packets to, it uses Direct Memory Access (DMA) to transfer the packets over PCIe into host memory, or it might use Direct Cache Access (DCA) to place the packets directly into the last level cache. This leads to another source of resource contention and thus tail latency: network traffic has to contend for cache space and memory bandwidth with applications. Newly arrived packets might evict either unprocessed packets or application data from the cache, leading to expensive memory accesses. When the host memory bandwidth becomes saturated, RPCs can experience significant tail latencies. This problem is exacerbated as network speeds continue to increase; as network data arrives faster, it will want to hog even more memory bandwidth and cache space. Eventually, the transport logic will read the packets from the cache or memory and produce a message for an application. Packets are usually processed in batches to amortize the cost of traversing the network stack and improve throughput; however, batching is not ideal for latency critical RPCs because it, by definition, causes some packets to wait around for other packets to arrive, which is exactly what we need to avoid to minimize response time.

The *message* load balancing decision is commonly implemented using a dedicated core. Alternatively, depending on the scalability and parallelizability of the load balancing algorithm being implemented, this logic may also be co-located with the transport cores. Throughput is a concern here; small RPC messages can arrive on a 100Gbps link at 150 million requests/second (Mrps). This means that in order to keep up with line rate, software must process each message in 5ns or less, which is less than the time for a single L2 cache miss. In practice, a single core can only process up to about 10 MRPS—only about 5% of the network capacity.

Another issue is the thread scheduling decision. To try to make efficient use of the CPU cores, a thread scheduler will try to multiplex various application threads on the cores. However, the problem is that modern software-based thread schedulers operate at very coarse granularities. In fact, even if they were to make thread scheduling decisions every 5μs, the time slice of state-of-the-art operating systems, it is still too coarse-grained when RPCs are processed in less than 1μs. Thus, today's best practice when deploying a latency critical application is to dedicate cores for use only by that one application such that it is never swapped off and is always running and ready to process messages. This is a cop-out solution, and an expensive one. It means cores are not efficiently multiplexed, and CPU resources are wasted.

Eventually, the application produces a response message that is turned back into packets, and those packets go back through the memory hierarchy—over PCIe to the NIC and then out onto the network.

In terms of performance, the minimum wire-to-wire latency that we are aware of, which measures the time from the Ethernet wire to the application and back to the wire, has been reported by eRPC [45] to be 850ns. Wire-to-wire tail latencies on modern systems can easily exceed tens or even hundreds of microseconds. In contrast, we are aiming to bring the minimum wire-to-wire latency to below 100ns and tail latency to just 1–2μs.

1.4 nanoPU Overview

Given the description of how RPCs are processed on modern systems (Section 1.3), it is clear why tail response time and communication overheads are problems: network data has to go through the NIC, PCIe, memory, cache, and up to two cores before even reaching the application. There are numerous layers of processing and queueing—so many opportunities for resource contention and inefficient scheduling decisions. At the end of the day, the goal that an RPC stack needs to accomplish is dead simple: simply move data between the network and applications while ensuring that cores are used efficiently.

Ideally, the NIC should take care of translating between packets and messages as well as efficiently load balancing messages across cores. We want messages to be delivered directly to cores rather than via the memory hierarchy, and we want applications running on the cores to be multiplexed with such fine granularity that the performance of high-priority applications is essentially unaffected even when the core is shared with lower-priority applications. This is exactly what the nanoPU aims to enable for a class of performance-critical RPCs that we call *nanoRequests*. NanoRequests are small RPCs that are typically processed within 1μs. Hence, a network stack for nanoRequests must aim to minimize latency and message processing overheads.

The key characteristics of the nanoPU are as follows: first, it uses a novel, thread-safe CPU-Network interface via the CPU register file. This feature provides a direct path between the network and the applications running on the cores; it bypasses PCIe as well as the cache and memory hierarchy entirely, thus minimizing the latency of network operations. Additionally, by using a dedicated path for network IO, the approach eliminates contention on the PCIe bus, the memory bus, and in the caches, thus reducing tail response time. Furthermore, this new network interface is exposed to software using a simple API, which enables applications to issue and process RPCs with extremely low overheads.

Second, the nanoPU offloads the entire RPC stack to hardware, including: transport and congestion control, core selection, and thread scheduling. In addition to reducing communication overheads, this approach significantly reduces RPC tail response time and improves throughput by implementing key resource-scheduling decisions in hardware where they can operate much more efficiently than their corresponding software implementations.

Third, the nanoPU provides support for a unique thread-scheduling policy that bounds the interference caused by high-priority applications. This feature, combined with the deterministic latency of the hardware RPC processing pipeline, provides an opportunity for the nanoPU to bound the entire end-host RPC response time under certain conditions. We believe the nanoPU is the first system to offer such a service for performance-critical RPCs (i.e., nanoRequests).

For compatibility with existing applications, the nanoPU allows traditional network traffic, including RDMA and less performance-sensitive RPCs, to traverse a regular path through a DMA NIC, OS, and memory hierarchy.

The key contributions of this dissertation are as follows:

1. The nanoPU's median *wire-to-wire* response time for nanoRequests, from the wire through the header-processing pipeline, the transport layer, the core selection, and the thread scheduling, plus a simple no-op loopback application and back to the wire, is just 69ns, an order of magnitude lower latency than the best commercial NICs [27]. Without the MAC and serial I/0, loopback latency is only 17ns.

2. Our prototype's hardware thread scheduler continuously monitors processing status for nanoRequests and makes decisions in less than 1ns. The nanoPU sustains a 20% higher load than existing approaches while maintaining close to 1μs 99th %ile tail response times.

3. Our complete RISC-V-based prototype is available open-source and runs on AWS F1 FPGAs using Firesim [48].

4. We evaluate a suite of applications, including the MICA key-value store [60], Raft consensus [70], set algebra and high-dimensional search inspired by the μ-Suite benchmark [89].

1.5 Outline

The remainder of this thesis is organized as follows: **Chapter 2, Related Work,** describes alternative approaches pursued by other researchers to tackle some of the same problems that are addressed in this thesis— namely alternative ways to minimize RPC tail response time and communication overheads, as well as attempts to offload parts of the RPC stack to hardware. **Chapter 3, The nanoPU Architecture,** presents a detailed description of the nanoPU design, as well as a justification for each of the key design decisions. **Chapter 4, The nanoPU RISC-V Prototype,** explains how we have implemented each of the key aspects of the design by building upon an open source RISC-V core. In **Chapter 5, nanoPU Evaluations,** we evaluate the prototype using a combination of microbenchmarks and real applications. **Chapter 6, Event-Driven Packet Processing,** describes an additional contribution of this thesis: a new data plane programming model that enables more flexibility than what is possible using the modern data plane programming model, without sacrificing performance. We leverage this new model to develop the nanoPU's programmable transport module.

Chapter 2

Related Work

Given the increasing importance of RPCs in datacenters, it should come as no surprise that there has been a significant amount of research aimed at optimizing various aspects of the RPC stack. The following sections provide an overview of previous proposals and how they relate to the nanoPU.

2.1 Software Solutions

Kernel Bypass Networking. In the early days of the Internet, the ability of the Linux kernel network stack to operate reliably across a wide range of networks helped to enable the Internet to flourish. However, with the rise of data centers, performance requirements have become significantly more important. The Linux kernel network stack, which has not been optimized for low latency, adds about 10μs to the wire-to-wire response time in the best case, and usually much more. To provide some context, this is about 5-10$\times$ the latency through the network fabric of a modern datacenter, where each switch adds about 300ns of latency.

Researchers have realized that it is possible to achieve significantly lower response time by bypassing the Linux kernel and implementing the network stack in userspace [40, 11, 73, 45, 63, 79, 72]. For example, eRPC [45], which is a highly optimized RPC library, achieves a wire-to-wire response time of about 850ns (in the best case). However, despite its impressive best case performance, eRPC struggles with resource contention issues under high load, which leads to poor tail response time for applications. Dealing with resource contention issues requires efficient scheduling decisions, which is another area of related work.

RPC Scheduling on Cores. ZygOS [79], Shinjuku [44], and Shenango [72] are software systems that attempt to tackle the problem of efficiently scheduling RPCs across cores in order to drive down tail response time. These systems use techniques such as work-stealing [79] to approximate ideal single queue behavior, or they use a centralized dispatcher core [44, 72] to implement a sophisticated scheduling policy. However, as a result of the overheads associated with inter-core synchronization and software preemption, these systems are forced to make coarse scheduling decisions (at most every $5\mu s$) and are therefore ill-suited for nanoRequests, which require only $1\mu s$ of processing time. Furthermore, a centralized dispatcher core, which can only process up to about 10 million requests per second (Mrps), inevitably becomes a throughput bottleneck for the system.

In order to truly minimize tail response time we find that the best approach is to offload all RPC scheduling decisions to hardware, both the core selection decision as well as the thread scheduling decision. In hardware, these scheduling decisions can be made at line-rate (150Mrps at 100Gbps) within only a few nanoseconds. Thus the approach provides orders of magnitude lower latency and higher throughput than corresponding software systems.

Low Latency Transport Protocols. Another approach that researchers have pursued in an attempt to enable low latency for RPCs is to design new transport protocols. NDP [33] and Homa [66] are two promising examples of receiver-driven transport protocols that are well-suited for nanoRequests. By keeping queue occupancies low, bottleneck links fully utilized, and avoiding head-of-line blocking, these protocols are able to provide low latency and high throughput through the network. R2P2 [54] is a complimentary transport protocol whose goal is to efficiently load balance RPCs across servers by implementing the join-bounded-shortest-queue policy in the top-of-rack switch.

However, software implementations of transport protocols are fundamentally constrained by the efficiency of using general purpose cores for header processing, message reassembly/packetization, reliable delivery, and congestion control. One of the goals of this thesis is to demonstrate that it is possible to build a domain-specific hardware architecture that is able to perform these tasks at line rate with extremely low latency, and is also flexible enough to support multiple transport protocols. The nanoPU's NDP implementation is able to process packets at line rate while adding about 10ns to the wire-to-wire response time. This is about 1-2 orders of magnitude higher throughput and lower latency than an equivalent software implementation running on a single core.

2.2 Hardware Solutions

Recognizing the limitations of software-based approaches, many researchers have turned to new hardware designs in an effort to meet the strict performance requirements of datacenter RPCs.

Hardware Core Selection. The low throughput and high latency provided by software systems such as ZygOS [79] and Shinjuku [44] has led some researchers to offload core selection decisions to NIC hardware. RPCValet [19] and NeBuLa [90] are both examples of this approach. RPCvalet implements a single queue system, which in theory provides optimal performance, but it ran into memory bandwidth contention issues, which the authors later resolve in NeBuLa. NeBuLa and the nanoPU both use Join-Bounded-Shortest-Queue (JBSQ) [54] load balancing to steer requests to cores. However, in order to achieve low tail response time in a cloud environment where many applications share the limited number of CPU cores, steering requests to cores is only part of the solution. The nanoPU is the only system that efficiently implements hardware-accelerated thread scheduling in addition to core selection. Furthermore, in order to be able to load balance entire RPC messages (rather than packets) across cores using the NIC, the transport layer needs to be implemented in NIC hardware as well.

Hardware Transport Protocols. Aside from enabling the NIC to efficiently distribute RPC messages to cores, implementing transport logic in a fixed-latency hardware pipeline helps to reduce tail response time, improves throughput, and allows cores to focus on application processing. Furthermore, moving transport to the NIC reduces the latency of the congestion control loop, which helps to make more efficient use of the network resources.

We are not the first to suggest offloading the transport layer to the NIC. Modern NICs that support RDMA over Converged Ethernet (RoCE) already implement DCQN [100] in hardware. In the academic research community, Tonic [6] proposes a framework for implementing congestion control in hardware. The nanoPU's programmable transport layer draws upon ideas described by the Tonic authors, and goes further to build and evaluate an end-to-end system.

SmartNICs. In the commercial sector, many SmartNICs have embedded CPUs on them [8, 64, 67] and are being used to offload infrastructure software from the main server to CPUs on the NIC. However, adding embedded CPUs onto the critical path between the network and applications may actually increase RPC latency, unless they adopt nanoPU-like designs on the NIC.

RDMA. RDMA gives direct access to a remote server's memory, and many NICs now offer RDMA in hardware and can respond in a few microseconds. Several systems such as HERD [46], FaSST [47], and DrTM+R [16] exploit RDMA to build applications and services on top. The nanoPU compliments RDMA for nanoRequests that need low-latency access to remote CPUs rather than remote memory.

Integrated NICs. Recently, there have been a number of proposals for new network interface designs that are tightly integrated with memory, thus eliminating PCIe related overheads [2, 90, 56]. NetDIMM [2] integrates the NIC into the DRAM memory controller, and NeBuLa [90] goes further to dispatch RPCs all the way to the CPU's L1 cache. Dagger [56] proposes to use an FPGA-based NIC that interfaces with the host processor through a NUMA memory interconnect.

The nanoPU takes a different approach and integrates the NIC directly with the CPU core via the register file, rather than memory. Our approach requires slightly more involved changes to both applications and CPU cores, but provides about 2$\times$ lower average response time as well as an opportunity to reduce tail response time by orders of magnitude. By bypassing the memory hierarchy and delivering network messages directly to cores via the register file, the nanoPU reduces contention for cache space, memory bandwidth, and the TLB, all of which helps drive down application runtime variability. In addition, now that the core has a complete view of both network load and thread status, the opportunity arises to offload thread scheduling decisions to a hardware module on the core. At present, we do not know of a way to implement hardware-accelerated thread scheduling without using the register file network interface.

Register File Network Interface. The nanoPU's register file network interface is inspired by the J-Machine [20] from 1989, which used a similar approach to implement low-latency inter-core communication within a single machine. As will be explained further in Chapter 3, the design was eventually abandoned because of the difficulty of implementing thread-safety. Similar ideas have reappeared in several designs, including the RAW processor [97] and the SNAP processor for low-power sensor networks [50]. To our knowledge, the nanoPU is the first proposal to make the register file network interface thread-safe and thus available for use in datacenter servers.

2.3 Discussion

An RPC stack provides a variety of functions including transport processing, scheduling of RPCs across cores, thread scheduling, and a message interface for applications. An inefficient implementation of any one of these tasks can lead to high tail response time. Prior work has explored ways to optimize some of these tasks in isolation, but in order to truly minimize tail response time, the whole stack must be holistically optimized. The nanoPU is the first end-to-end system that brings together lessons learned from a wide variety of previous proposals along with our own novel ideas. The design described in this dissertation is the only way that we know how to build a network stack that can process minimum size messages at full line rate while providing sub-100ns wire-to-wire response times on average, and 1-2μs tail response times at high load.

Chapter 3

The nanoPU Architecture

3.1 Design Overview

The nanoPU is a new NIC-CPU co-design that adds a new fast path for nanoRequest messages requiring ultra-low and predictable network communication latency. Figure 3.1 depicts the key design components. The nanoPU has two independent network paths: (1) the traditional (unmodified) DMA path to/from the host's last-level [23] or L1 cache [90], and (2) an accelerated fast path for nanoRequests, directly into the CPU register file.

The traditional path can be any existing path through hardware and software; hence all network applications can run on the traditional path of the nanoPU unchanged, and perform at least as well as they do today. The fast path is a nanosecond-scale network stack optimized for nanoRequests. Applications should (ideally) be optimized to efficiently process nanoRequest messages directly out of the register file to fully harness the benefits of the fast path.

Each core has its own hardware thread scheduler (HTS), two small FIFO memories for network ingress and egress data, and two reserved general-purpose registers (GPRs): one as the tail of the egress FIFO for sending nanoRequest data, and the other as the head of the ingress FIFO for receiving. CPU cores are statically partitioned into two groups: those running normal applications and those running nanoRequest applications. Cores running regular applications use standard OS software thread scheduling mechanisms [72, 44, 79]; however, the OS delegates scheduling of the nanoRequest cores to the HTS.

To understand the flow of the nanoPU fast path, consider the numbered steps in Figure 3.1. In ❶, a packet arrives and enters the P4-programmable PISA pipeline. In addition to standard header processing (e.g. matching IP addresses, checking version and checksum, and removing tunnel encapsulations), the pipeline examines a header tag to decide if it is a nanoRequest. If so, it proceeds to ❷, else it follows the usual DMA processing path 🅓. In ❷, packets are reassembled into messages; a buffer is allocated for the entire message and packet data is (potentially) re-sequenced into the correct order. In ❸, the transport protocol ensures reliable message arrival; until all data has arrived, message data and signaling packets are exchanged with the peer depending on the protocol (e.g. NDP and Homa are both receiver driven using different grant mechanisms) (Section 3.4). One important consequence of implementing a message-oriented transport layer in hardware is that it enables the NIC to load balance entire messages across cores, rather than individual packets. Many applications are unable to process partial messages, hence, dispatching full messages to cores helps to reduce software overheads. When a message has fully arrived, in ❹ it is placed in a per-application receive queue where it waits to be assigned to a core by the core-selection logic (Section 3.5). When its turn comes, in ❺, the message is sent to the appropriate per-thread ingress FIFO on the assigned core, where it waits for HTS (Section 3.3) to alert the core to run the message's thread and place the first word in the `netRX` register (Section 3.2). In ❻, the core processes the data and, typically, generates a response message for the client. Words are written into the `netTX` register in ❼, then flow into the global transmit queues in ❽. Complete messages are sent to be split into packets in ❾, before departing through the egress PISA pipeline.

Next, we detail the design of the main components of the fast path: the thread-safe register file network interface, the hardware thread scheduler (HTS), and the programmable NIC pipeline, including transport and core selection.

3.2 Thread-Safe Register File Interface

Recent work [68] showed that PCIe latency contributes about 90% of the median wire-to-wire response time for small packets (800–900ns). Several authors have proposed integrating the NIC with the memory hierarchy in order to bring packets directly into the cache [69, 19, 90].

The nanoPU takes this one step further and connects the network fast path directly to the CPU core's register file. The high-level idea is to allow applications to send and receive network messages by writing/reading one word (8B) at a time to/from a pair of dedicated CPU registers.

There are several advantages to bringing packet data directly into the register file:

Message data bypasses the memory and cache hierarchy, minimizing the time from when a packet arrives on the wire until it is available for processing. In Section 5.2.1, we show that this reduces median wire-to-wire response time to 69ns, 50% lower than the state-of-the-art.

Reduces variability in processing time and therefore minimizes tail response time. For example, there is no variable waiting time to cross PCIe, no cache misses for message data (messages do not enter or leave through memory) and no IO-TLB misses (which lead to an expensive 300ns access to the page table [68]). And because nanoRequests are buffered in dedicated FIFOs, separate from the cache, nanoRequest data does not compete for cache space with other application data, further reducing cache misses for applications. Cache misses can be expensive: an LLC miss takes about 100ns to resolve and creates extra traffic on the (shared) DRAM memory bus. DRAM access can

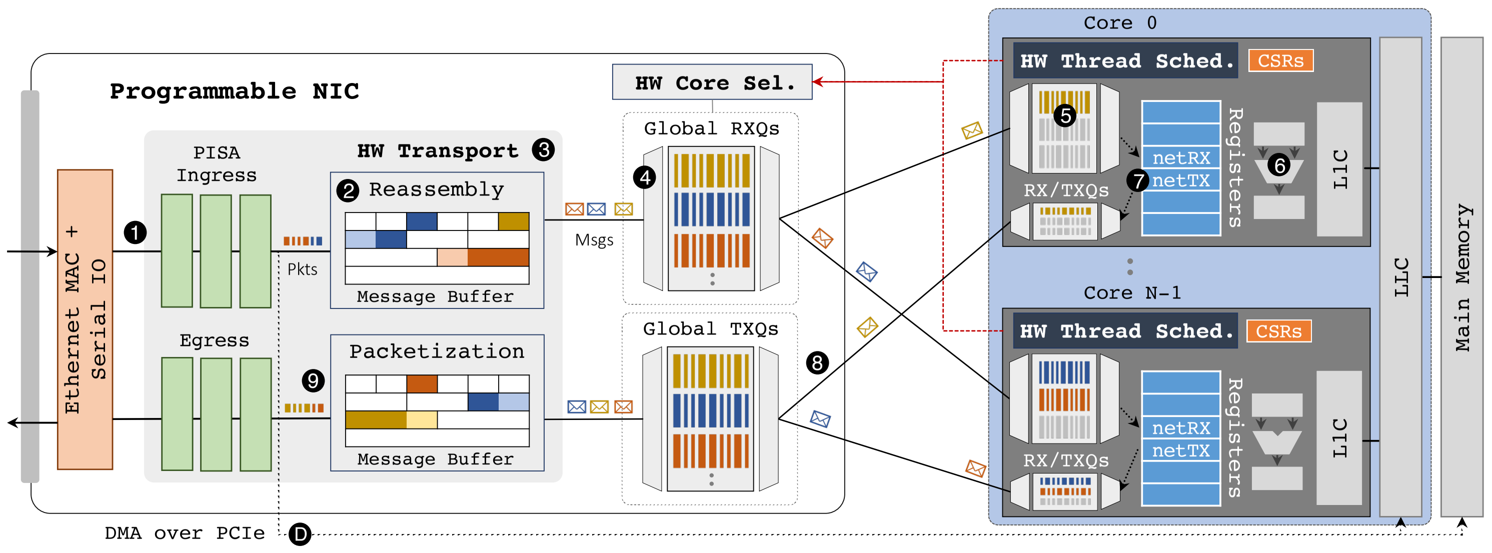

Figure 3.1: The nanoPU design. The NIC includes ingress and egress PISA pipelines as well as a hardware-terminated transport and a core selector with global RX queues; each CPU core is augmented with a hardware thread scheduler and local RX/TX queues connected directly to the register file.

be a bottleneck for a multicore CPU, and when congested, memory access times can increase by more than 200%. [94]. Furthermore, contention for cache space and DRAM bandwidth is worse at network speeds above 100Gb/s [28].

Less software overhead per message because software does not need to manage DMA buffers or perform memory-mapped IO (MMIO) handshakes with the NIC. In a conventional NIC, when an application sends a message, the OS first places the message into a DMA buffer and passes a message descriptor to the NIC. The NIC interrupts or otherwise notifies software when transmission completes, and software must step in again to reclaim the DMA buffer. The register file message interface has much lower overhead: when an application thread sends a message it simply writes the message directly into the `netTX` register, with no additional work. Section 5.2.1 shows how this leads to a much higher throughput interface.

3.2.1 How an application uses the interface

The J-Machine [20] first used the register file in 1989 for very low latency inter-core communication, followed by the Cray T3D [51]. The approach was abandoned because it proved difficult to protect messages from being read/written by other threads sharing the same core; both machines required atomic message reads and writes [21]. As we will see below, our design solves this problem. We believe ours is the first design to add the register file interface to a regular CPU for use in data centers.

The nanoPU reserves two general-purpose registers (GPRs) in the register file for network IO, which we call `netRX` and `netTX`. When an application issues an instruction that reads from `netRX`, it actually reads a message word from the head of the network receive queue. Similarly, when an application issues an instruction that writes to `netTX`, it actually writes a message word to the tail of the network transmit queue. The network receive and transmit queues are stored in small FIFO memories that are connected directly to the register file. These FIFO memories are about the same size as the L1 cache (i.e., 16–32KB).[1] In addition to the reserved GPRs, a small set of control & status registers (CSRs) are used for the core and NIC hardware to coordinate with each other. These CSRs are described in Section 4.5.

Delimiting messages. A short fixed header guides the NIC hardware modules; each arriving message starts with a header indicating the message length (as well as the source IP address and layer-4 port number), which allows applications to detect the end of message. Similarly, departing messages start with a header (with the length, destination IP address and layer-4 port number) so that the NIC knows when an outgoing message completes.

Inherent thread safety. We need to prevent an errant thread from reading or writing another thread's messages. The nanoPU prevents this using a novel hardware interlock. It maintains a

[1]We think of these FIFO memories as the equivalent of the L1 cache, but for network messages; both are built into the CPU pipeline and sit right next to the register file.

separate ingress and egress FIFO for each thread, and controls access to the FIFOs so that `netRX` and `netTX` are always mapped to the head and tail, respectively, of the FIFOs for the currently running thread only. Note our hardware design ensures this property even when a previous thread does not consume or finish writing a complete message.[2] This turned out to be a key design choice, simplifying application development on nanoPU; nanoRequest threads no longer need to read and write messages atomically.

Application software changes. The register file can be accessed in one CPU cycle, while the L1 cache typically takes three cycles. Therefore, an application thread will run faster if it can process data directly from the ingress FIFO by serially reading `netRX`. Ideally, the developer picks a message data structure with data arranged in the order it will be consumed—we did this for the message processing components of the applications evaluated in Section 5.3. But if an application needs to copy long messages entirely into memory so that it can randomly access each byte many times during processing, then the register file interface may not offer much advantage over the regular DMA path. Messages for these applications should probably not be tagged as nanoRequests. Our experience so far is that, with a little practice, it is practical to port latency-sensitive applications to efficiently use the nanoPU register file interface. Table 3.1 lists applications that have been ported to efficiently use this new network interface. We believe that if the nanoPU proves popular and useful, future advances in code generation, verification, and profiling techniques will help application designers more easily optimize message formats and application logic to most efficiently use the register file network interface. For example, a tool that analyzes the order in which an application accesses the the fields of a network message would be a helpful starting point.

A related issue is how, and at which stage of processing, to serialize/deserialize (also known as marshall/unmarshall) message data. In modern RPC applications this processing is typically implemented in libraries such as Protobuf [80] or Thrift [92]. Recent work pointed out that on conventional CPUs, where network data passes through the memory hierarchy, the serialize/deserialize logic is dominated by scatter/gather memory-copy operations and subword-level data transformation operations, suggesting a separate hardware accelerator might help [78].

In the nanoPU, the memory copy overhead involved in serialization and deserialization is little or none; only a few copies between registers and the L1 cache may be necessary when a working set is larger than the register file. The remaining subword data-transformation tasks can be done either in the applications (in software) or on the NIC (in hardware) using a PISA-like pipeline, but still operating at the message level. We currently take the former approach for the applications we evaluate in Section 5.3, but intend to explore the latter approach in future work.

[2]Our interlock logic would have been prohibitively expensive in the early days; but since 1989, Moore's Law lets us put four orders of magnitude more gates on a chip, making the logic quite manageable.

Application	**Description**	**Response Time p50 / p99 (μs)**
MICA	Implements a fast in-memory key-value store	0.40 / 0.50
Raft	Runs leader-based state machine replication	3.08 / 3.26
Chain Repl.	Runs a vertical Paxos consensus algorithm	1.10 / 1.40
Set Algebra	Processes data-mining and text-analytics workloads	0.60 / 1.50
HD Search	Analyzes and processes image, video, and speech data	0.80 / 1.20
N-Body Sim.	Computes gravitational force for simulated bodies	0.35 / N/A
INT Processing	Processes network telemetry data (e.g., path latency)	0.13 / N/A
Packet Classifier	Classifies packets for intrusion detection and access control	0.70 / 1.40 1K 0.90 / 2.20 100K
Othello Player	Searches the Othello state space	0.90 / 1.70 [39]

Table 3.1: Example applications that have been ported to and accelerated by the nanoPU. These applications use small RPCs, few memory references, and cache-resident function stack and variables (in the common case), and are designed to efficiently process messages out of the register file. Table indicates median and 99th %ile tail response time at low load (showing results with 1K & 100K rules for Packet Classifier).

3.3 Thread Scheduling in Hardware

Current best practice for low-latency applications is to either (1) pin threads to specific cores [79, 24], which is very inefficient when a thread is idle, or (2) devote one core to run a software thread scheduler for the other cores [44, 72].

The fastest software-based thread schedulers are not fast enough for nanoRequests. Software schedulers need to run periodically so as to avoid being overwhelmed by interrupts and associated overheads, which means deciding how frequently they should run. If it runs too often, resources are wasted; too infrequently and threads are unnecessarily delayed. The fastest state-of-the-art operating systems make periodic scheduling decisions every 5μs [44, 72], which is too coarse-grained for nanoRequests requiring only 1μs of computation.

We therefore moved the nanoRequest thread scheduler to hardware, which continuously monitors message processing status as well as the network receive queues and makes sub-nanosecond scheduling decisions. Our new hardware thread scheduler (HTS) is both faster and more efficient; a core never sits on an idle thread when another thread with a pending message could run.

3.3.1 How the hardware thread scheduler works

Every core contains its own hardware thread scheduler (HTS). When a new thread initializes, it must register itself with its core's HTS by binding to a layer-4 port number and selecting a strict priority level (0 is the highest). The layer-4 port number uniquely identifies each application, which may consist of multiple threads that are running on distinct cores. When threads on different cores bind to the same layer-4 port number the NIC will load balance messages across the cores, as explained in Section 3.5. Each core's HTS is responsible for scheduling all threads running on that core. It also ensures that `netRX` and `netTX` are always the head and tail of the FIFOs for the currently running thread.

HTS tracks the running thread's priority and its time spent on the CPU core. When a new message arrives, if its destination thread's priority is lower than or equal to the current thread, the new message is queued. If the incoming message is for a higher priority thread, the running thread is suspended and the destination thread is swapped onto the core. Whenever HTS determines that threads must be swapped, it (1) asserts a new, NIC-specific interrupt that traps into a small interrupt handler (only on the relevant core), and (2) tells the interrupt handler which thread to switch to by writing the target's layer-4 port number to a dedicated CSR. Our current HTS implementation is able to react to events such as the arrival of a high priority message or a thread becoming idle and fire an interrupt to initiate a context switch in a single cycle (0.3ns at 3.2GHz). It then takes about 50ns to swap a previously idle thread onto the core, measured from the moment the HTS fires the scheduling interrupt to when the target thread executes its first instruction (Section 4.2).

If the next thread to run belongs to a different process, the software interrupt handler must perform additional work: notably, it must change privilege modes and swap address spaces. A typical context switch in Linux takes about 1μs [44], but most of this time is spent making the scheduling decision [95]. Our HTS design makes this decision entirely in hardware and the software scheduler simply needs to read a CSR to determine which thread to swap to.

The scheduling policy. HTS implements a *bounded strict priority* scheduling policy to ensure that the highest priority thread with pending work is running on the core at all times. Threads are marked `active` or `idle`. A thread is marked `active` if it is eligible for scheduling, which means it has been registered (a port number and RX/TX FIFOs have been allocated) and a message is waiting in the thread's RX FIFO. The thread remains `active` until it explicitly indicates that it is `idle` and its RX FIFO is empty. HTS tries to ensure that the highest priority `active` thread is always running.

Bounded response time. HTS supports a unique feature to bound how long one high-priority application can hold up another. If a priority 0 thread takes longer than t_0 to process a message, the scheduler will immediately downgrade its priority from 0 to 1, allowing it to be preempted by a different priority 0 thread with pending messages. (By default, $t_0 = 1\mu$s.) We define a *well-behaved* application as one that processes all of its messages in less than t_0.

As a consequence, HTS guarantees an upper bound on the response time for well-behaved applications. If a core is configured to run at most k priority 0 application threads, each with at most one outstanding message at a time, then the total message processing time, t_p for well-behaved applications is bounded by the following equation:

$$t_p \leq t_n + kt_0 + (k-1)t_c \tag{3.1}$$

In this equation, t_n is the NIC latency and t_c is the context-switch latency. In practice, this means an application developer who writes a well-behaved application can have full confidence that no other applications will delay it beyond a predetermined bound. If application writers do not wish to use the time-bounded service, they may assign all their application threads priority 1.

We have written small well-behaved threads with bounded processing time by keeping threads small, using bounded loops and avoiding cache and TLB misses. But a more general approach may be worthwhile for broader adoption of the nanoPU. In a trusting environment, developers can empirically confirm whether their applications are well-behaved during the testing or early deployment phases. In a non-trusting environment, it may be possible to rely on code verification [26, 35] to check whether threads meet execution time bounds. The eBPF [26] compiler, for example, verifies code in a restricted environment; we believe similar ideas can be applied to nanoRequest threads.

3.4 Programmable Transport in Hardware

3.4.1 Background

There are two main reasons why transport protocols are implemented in software today: (1) there is no clear consensus on “the right” transport protocol to use for every workload in every network. Over the past decade, dozens of papers have been published at top-tier conferences that propose new or modified transport protocols. (2) In general, transport protocols utilize many complex features that do not lend themselves to an efficient hardware implementation. Thus, the flexibility provided by a general purpose processor provides a convenient environment to deploy transport logic.

However, this flexibility to support arbitrarily sophisticated processing comes at a cost. A single core is barely able to keep up with a 10Gbps network (when processing large packets), and at least 10 cores are required for a 100Gbps network [63]. As network speeds continue to increase, this problem is exacerbated. Dedicating this many cores for transport processing is expensive, especially for cloud providers who have a financial incentive to sell CPU cycles to paying customers. Additionally, using many cores leads to a load balancing problem that needs to be solved. Imperfect load balancing across the transport cores can lead to high latency variability. Furthermore, high inter-core synchronization costs lead to increased latency and lower throughput.

Processing packets in software incurs high overheads. In fact, processing small packets in software is almost as expensive as processing large packets. This is particularly problematic when processing datacenter RPCs where either the request or response (usually both) consists of a single small packet [66]. As a result of the high overheads, software solutions often process packets in batches in order to amortize the overheads across many packets and thus improve throughput. However, this technique has the undesirable effect of increasing latency variability. Clearly, when aiming to deploy high performance transport protocols, a general purpose CPU architecture is not the optimal solution.

On the other hand, a hardware solution is able to guarantee line rate packet processing with deterministic per-packet latency. For example, our prototype hardware NDP implementation (Chapter 4) runs in 7ns (fixed) per packet and at 200Gb/s for minimum size packets (64B). Such low latency means a tight congestion-control loop between end-points, and hence more efficient use of the network. Moreover, moving transport to hardware frees CPU cycles for application logic. However, the key challenge with this approach is ensuring sufficient flexibility in the hardware architecture in order to support the various requirements of real transport protocols and to enable innovation.

3.4.2 Our Approach

Rather than attempting to support every possible transport protocol in hardware, our approach is to build a programmable domain specific architecture that supports one important class of transport protocols. Inspired by recent proposals NDP [33] and Homa [66], the protocols we aim to support provide the abstraction of *reliable, one-way message delivery* to applications, rather than the more traditional reliable, long-lived, bi-directional, byte-stream.

Message-oriented transport protocols have several advantages over connection-oriented protocols. Connection-oriented transport protocols have no way to identify distinct messages within a connection and hence are unable to prioritize the delivery of certain messages over others. Message-oriented protocols, on the other hand, treat each message independently and thus are able to reduce the head-of-line blocking that occurs when short messages are queued behind long messages, reducing tail latency. Moreover, message-oriented protocols have the potential to significantly reduce state requirements at the transport layer, which facilitates an efficient hardware implementation and enables more scalable distributed systems. As one concrete point of comparison, a TCP socket structure in Linux contains about 2000 bytes of state whereas our hardware NDP implementation only needs to maintain about 20 bytes of state for each message, until it is successfully delivered to the destination. This means that in a 100Gbps network with a 2μs round trip time (RTT), one bandwidth-delay-product of single packet messages requires about 400 bytes of state for NDP, which is $5\times$ less than a single TCP flow. Furthermore, state requirements continue to decrease as the network RTT is reduced. As a consequence, optimizing network latency enables more scalable distributed systems because state requirements are often the limiting factor for scalability.

In general, transport protocols perform four main tasks: header processing, message reassembly/packetization, reliable delivery, and congestion control.

Header processing is required to parse and deparse packet headers such as VXLAN, overlay tunnels, telemetry data, and transport data. Modern multi-Tbps switches have already demonstrated the feasibility of implementing programmable header processing at line rate [93, 14]. We leverage these designs to provide this functionality in the nanoPU's transport architecture.

Message reassembly/packetization is required to reassemble incoming data packets, which might arrive out-of-order, into messages, as well as to split outgoing messages into packets, which might need to be retransmitted out-of-order due to drops in the network. This logic is consistent across transport protocols and thus does not necessarily need to be made programmable.

Reliable delivery of data packets over a lossy network is an important aspect of every transport protocol. Both endpoints of a message transfer must have a way to determine when the entire message has been successfully delivered to the destination. In addition, the only way to ensure efficient reliable delivery over a lossy network fabric is to provide support for a timeout mechanism. Timeouts are used to infer packet loss and trigger retransmissions, as well as to clean up transport state upon message delivery failure in extreme situations, such as a network partition or a crashed server.

Congestion control is responsible for deciding when to send individual packets into the network. The goal is to reduce queueing in the network, which then results in low latency. Congestion control algorithms use many different congestion signals to make packet scheduling decisions. Examples include: number of outstanding packets, number of active messages, active message state (e.g. message length or remaining bytes), round trip time (RTT) measurements, or explicit signals provided by the network such as queue occupancy and packet drop indicators. In order to compute and utilize these signals, algorithms maintain a wide range of state variables, and maintaining this state properly is a key challenge.

Handling of shared state is a key design decision that impacts both performance and efficiency. Transport protocols utilize a number of state variables that must be shared between independent event processing threads (i.e., state machines). For example, state to track the packets of each message that have been successfully delivered to the destination would be initialized when an application starts transmitting a message and updated as ACK packets arrive over the network. Since both of these events can occur at the same time, the state machines that process these events must share some state.

In order to provide guaranteed line rate throughput and fixed latency, we must avoid synchronization mechanisms that are commonly used in software implementations to serialize accesses to state variables. There are two reasons synchronization may be required. First, if there is insufficient memory bandwidth to the shared memory then the independent threads must coordinate to share this bandwidth. The hardware architecture should instead provide sufficient memory bandwidth so that distinct threads can process the state completely independently of one another. Additional memory bandwidth can be provided for state variables at the cost of additional hardware resources.

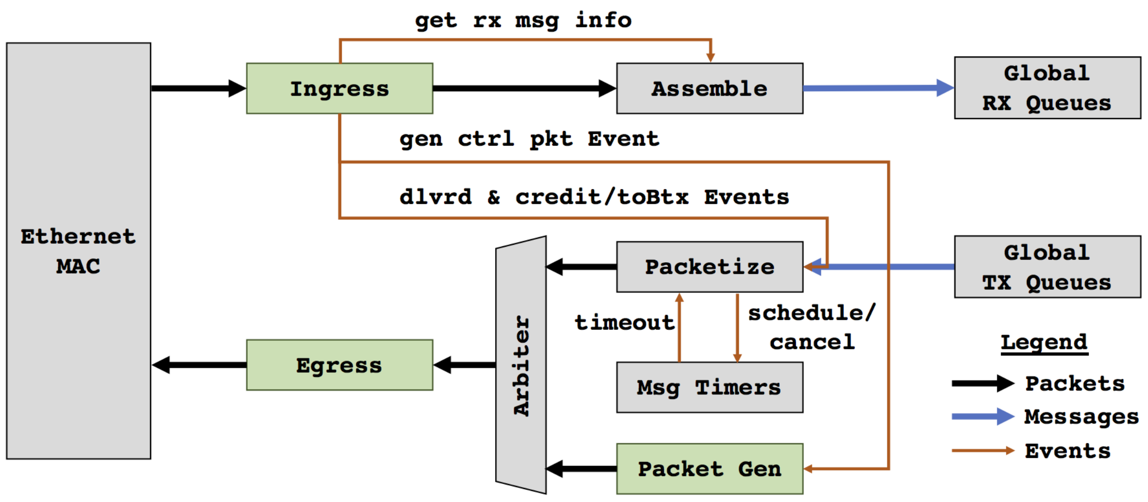

Figure 3.2: The nanoPU programmable transport architecture.

Thus, deciding the amount of memory bandwidth to use is an engineering trade off. In the nanoPU transport architecture, we aim to utilize only dual-ported state variables. We empirically find that this provides a sufficient amount of flexibility while keeping resource requirements low. Second, if the independent threads attempt to access the exact same state within the shared memory then the hardware must detect and resolve the collision by either aggregating or discarding the operations. For simple stateful operations such as increment or reset, it is easy for the hardware to resolve collisions. However, that may not be the case for all stateful operations.

It is important to consider how state update logic will be exposed to the data plane programmer. It would be naive to assume that we will be able to support arbitrary stateful operations. Prior work has already demonstrated that the target line rate constrains the expressiveness of stateful operations in hardware [87]. Instead, we take the approach proposed by Sivaraman et al. [87]. The nanoPU transport architecture supports a predefined set of stateful operations, called atoms, which are built into the design and guaranteed to operate at line rate. The developer's transport logic must then map onto the atoms provided by the architecture.

3.4.3 nanoPU Programmable Transport Architecture

Figure 3.2 shows a block diagram of the nanoPU's programmable transport architecture. The design uses what we call an event-driven PISA programming model [37]. An event-driven PISA architecture is an extension of the traditional PISA architecture [14]; it exposes a more general programming model that allows data plane developers to express more sophisticated algorithms by utilizing a broader set of data plane events. A detailed description of event-driven PISA architectures will be provided in Chapter 6.

On the left, the Ethernet MAC is connected to the external Ethernet network. On the right, the global RX/TX queues send messages to / receive messages from the CPU cores, as shown in Figure 3.1. The elements in the architecture interact by passing packets, messages, as well as metadata (a.k.a. data plane events). The programmable elements of the architecture (shaded in green) include the ingress/egress pipelines, as well as the packet generation module. These elements are configured by the data plane developer to implement custom transport protocols. To explain the architecture details, we will walk through the processing on both the TX and RX paths.

On the TX path, message words transmitted by applications are loaded into the global TX queues, where they are buffered in per-application queues. Those messages are then passed to the packetization module as buffer space becomes available.

The packetization module is responsible for splitting application messages into data packets, as well as maintaining a few important state variables that are used for reliable delivery and congestion control: `delivered`, `toBtx`, and `credit`. The `delivered` state tracks the packets of each message that have been successfully delivered to the destination. The `toBtx` state tracks the packets of each message that still need to be transmitted (or retransmitted) to the receiver eventually, and the `credit` state tracks the packets that are currently eligible for transmission. Inspired by Tonic [6], these state variables are implemented as bitmaps to efficiently store one bit of state for each packet in each message. Upon receiving the first word of a message, the packetization module will allocate and initialize each of these state variables as well as trigger an event to initialize a timer for the message. After receiving either the full message or a maximum transmission unit (MTU) of data, a packet descriptor will be enqueued into an internal scheduling module which uses a configurable policy to prioritize data packets.

Upon receiving control packets from the peer, the programmable ingress pipeline triggers events containing metadata and instruction opcodes to update the `delivered`, `toBtx`, and `credit` state. When a message's credit increases or a packet retransmission is requested, the corresponding packet descriptors are scheduled for transmission within the packetization module. When a message timeout occurs, the packetization module will attempt to identify and schedule any packets that need to be retransmitted. Once all packets of a message have been successfully delivered to the peer, the corresponding message state in the packetization module is freed and the message timer is cancelled.

The arbiter schedules between data packets and control packets produced by the packetization and packet generator modules, respectively. In general, control packets are scheduled with higher priority in order to provide a low-latency feedback loop for the congestion control algorithm. The policy used to schedule the transmission of data packets can be protocol dependent and can affect the tail latency of messages through the network. For example, Homa [66] uses the shortest remaining processing time (SRPT) policy to schedule data packet transmissions. Ideally, this packet scheduling logic would be programmable. Recent work has proposed mechanisms for implementing programmable packet scheduling in hardware [88, 86, 1]. We believe it would be natural to incorporate these designs into the nanoPU architecture in the future.

The programmable egress pipeline consumes packet metadata and generates the appropriate Ethernet, IP, and transport headers for outgoing packets which are then serialized onto the wire by the Ethernet MAC.

On the RX path, deserialized network packets are first processed by the programmable ingress pipeline. This module parses packet header fields and drives the congestion control logic by triggering data plane events that are processed by other modules in the architecture.

For arriving data packets, the ingress pipeline will fire the `get rx msg info` event which is processed by the assembly module. If the assembly module determines that this is the first packet of a new message, it will attempt to allocate sufficient buffer space for the whole message. Upon success, it returns a unique message identifier, which can be used by the ingress pipeline to maintain state associated with the message. If the assembly module fails to allocate a buffer for the message, the packet is dropped.

The ingress pipeline can also be configured to trigger an event that causes the packet generation module to generate and transmit custom control packets. These control packets can be used to, for example, indicate successful delivery of a packet or elicit a retransmission from the peer.

The assembly module reassembles data packets, which might arrive out-of-order, into application messages. Once the final packet of a message is received, a message descriptor is scheduled for delivery to the global RX queues. The following section will describe how the nanoPU NIC assigns messages to cores after they arrive in the global RX queues.

3.5 Core Selection in Hardware

As mentioned in Section 3.3, applications can have multiple threads that are running on different cores and the NIC's goal is to balance incoming load across these cores. If the NIC randomly sends messages to cores then some messages will inevitably sit in a queue waiting for a busy core while another core sits idle.

Ideally, we would like to use a single queue for each application and allow cores to pull messages out of the appropriate queue when they need another one to process. However, it is impossible for the NIC to dispatch independent messages to different cores at the same time because the memory bandwidth that is available to read messages is inherently limited. For example, if the available memory bandwidth is 100Gb/s, then it would take 80ns to read a 1KB message. The NIC must dispatch messages to cores one at a time, and some cores will sit idle while it waits for the next message to arrive. In order to avoid this CPU inefficiency, it is usually better to build up a small queue of two or three, or more generally, n, messages at each core. That way, each core has something to work on while it waits for the NIC to dispatch the next message to the core. As n increases, the possibility of load imbalance also increases; this is why it is important to use small values of n.

Inspired by NeBuLa [90], the nanoPU NIC load balances nanoRequest messages across cores using the Join-Bounded-Shortest-Queue or JBSQ(n) algorithm [54]. JBSQ(n) approximates an ideal, work-conserving single queue policy using a combination of a single central queue, and short bounded queues at each core, with a maximum depth of n messages. The centralized queue replenishes the shortest per-core queues first. JBSQ(1) is equivalent to the theoretically ideal single-queue model, but, as explained above, is impractical to implement efficiently at these speeds.

Our nanoPU prototype implements a JBSQ(2) load balancer in hardware *per application*. The RX FIFOs on each core have space for at least two messages per thread running on the core. We chose JBSQ(2) based on the communication latency between the NIC and the cores as well as the available memory bandwidth for the centralized queues. We evaluate its performance in Chapter 5.

Chapter 4

The nanoPU RISC-V Prototype

We designed a prototype quad-core nanoPU based on the open-source RISC-V Rocket core [84]. A block diagram of our prototype is shown in Figure 4.1.

Our prototype extends the open-source RISC-V Rocket-Chip SoC generator [7], adding 4,300 lines of Chisel [9] to the code base. The Rocket core is a simple five-stage, in-order, single-issue processor. We use the default Rocket core configuration (16KB L1 instruction and data caches, a 512KB shared L2 cache, and 16GB of external DRAM memory) and simulate it running at 3.2GHz. Everything shown in Figure 4.1, except the MAC and Serial IO, is included in our prototype and is available open-source.[1] Our prototype does not include the traditional DMA path between the NIC and memory hierarchy. Instead, we focus our efforts on building the nanoPU fast path for nanoRequests.

To improve simulation speed, we do not run a full operating system on our prototype, but rather just enough to boot the system, initialize one or more threads on the cores, and perform context switches between threads when instructed to do so by the hardware thread scheduler (HTS). In total, this consists of about 1,200 lines of C code and RISC-V assembly instructions. All applications run as bare-metal applications linked with the C standard library.

The nanoPU design is intended to be fabricated as an ASIC, but we use an FPGA to build the initial prototype. As we will discuss further in Chapter 5, our prototype runs on AWS F1 FPGA instances, using the Firesim [48] framework. Our prototype adds about 15% more logic LUTs to an otherwise unmodified RISC-V Rocket core with a traditional DMA NIC.

4.1 RISC-V Register File Interface

The RISC-V Rocket core required surprisingly few changes to add the nanoPU register file network interface. The main change, naturally, involves the register file read-write logic. Each core has 32

[1]nanoPU source code: https://github.com/l-nic/chipyard/blob/lnic-dev/ARTIFACT.md

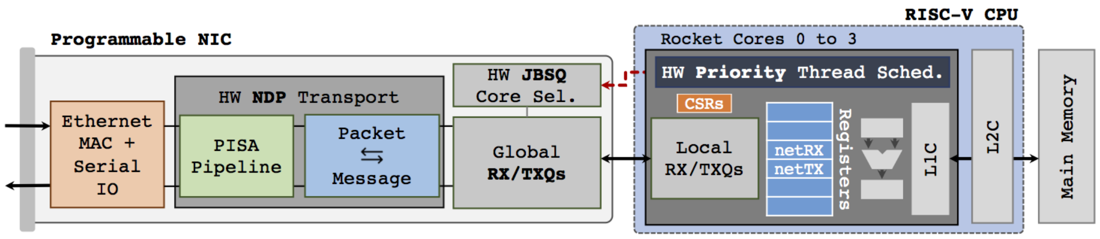

Figure 4.1: Block diagram of the nanoPU prototype.

GPRs, each 64-bits wide, and we reserve two for network communication (shared by all threads). Applications must be compiled to avoid using the reserved GPRs for temporary storage. Fortunately, `gcc` makes it easy to reserve registers via command-line options [71].

The core also required changes to the control logic that handles pipeline flushes. A pipeline flush can occur for a number of reasons (e.g. a branch misprediction). On a traditional five-stage RISC-V Rocket core, architectural state is not modified until an instruction reaches the write-back stage (Rocket Stage 5). However, with the addition of our network register file interface, reading **`netRX`** now causes a state modification (FIFO read) in the decode stage (Rocket Stage 2). The destructive read operation must be undone when there is a pipeline flush. The CPU pipeline depth is an upper bound on how many read operations need to be undone; in our case, at most two reads require undoing. It is straightforward to implement a FIFO queue supporting this operation.

4.2 Bounded Thread Scheduling in Hardware

The nanoPU core implements thread scheduling in hardware, as described in Section 3.3. Our prototype can run up to four threads on each core; each thread can be configured with a unique priority value. Priority 0 has a configurable maximum message processing time in order to implement the bounded priority thread scheduling policy. We added a new *thread-scheduling interrupt* to the RISC-V core, along with an accompanying control & status register (CSR) set by the hardware thread scheduling module to tell the interrupt's trap handler which thread it should run next. When processing nanoRequests, we disable all other interrupts to avoid unnecessary interrupt handling overheads.

Our prototype is able to react to events such as message arrivals, a thread becoming idle, or a thread hitting a timeout in a single cycle (0.3ns at 3.2GHz). This is possible because the scheduling logic is very simple; after using the event to update thread state, HTS identifies the highest priority active thread and checks if it is currently running on the thread. If it's not, HTS will fire a thread scheduling interrupt to initiate a context switch. We define the context-switch latency to be the time from when the scheduler fires the interrupt to when the first instruction of the target thread is executed. Our prototype has a measured context-switch latency of 160 cycles, or 50ns on a 3.2GHz

CPU. This is much faster than a typical Linux context switch, partly because the thread scheduling decision is offloaded to hardware, and partly because the core only runs bare-metal applications in the same address space with the highest privilege mode. Therefore, nanoPU hardware thread scheduling in a Linux environment would be less efficient than our bare-metal prototype.

4.3 Prototype Transport Architecture

We configured our programmable transport module to implement NDP [33] entirely in hardware. We chose NDP because it has promising low-latency performance, is well-suited to handle small RPC messages, and requires simple stateful primitives that are easy to implement in hardware. However, the nanoPU does not depend on NDP. As explained in Section 3.4, our NIC transport layer is programmable. We are in the process of porting additional transport protocols to run on the nanoPU (Section 7.1).

4.3.1 NDP Implementation

NDP [33] employs a few clever tricks to minimize end-to-end latency. First, packets are load-balanced over paths packet-by-packet (rather than flow-by-flow). This reduces congestion, but causes packet reordering. The assembly module described in Section 3.4 is responsible for resequencing received packet data into correctly ordered messages.

Second, NDP is receiver-driven; the receiving host decides when a sender can transmit packets, particularly during incast storms, by sending `PULL` packets to allow the sender to transmit new `DATA` packets or retransmit dropped packets. Each time a `DATA` packet arrives at the receiver, the ingress pipeline in the nanoPU transport architecture (Figure 3.2) triggers an event which causes the packet generation module to generate both an `ACK` and `PULL` packet for the corresponding message. The ingress pipeline maintains a small amount of state for each message to compute the offset that should be included in the generated `PULL` packet. The packet generator paces the outgoing rate of `PULL` packets to ensure that `DATA` packets arrive at the bottleneck link at line-rate, avoiding further congestion. When the `ACK` arrives back at the sender, the ingress pipeline updates the `delivered` state in the packetization module; the message state is freed after the entire message has been `ACK`ed. The `PULL` packets cause the ingress pipeline to update the `credit` state, which may result in additional `DATA` packets of the message being transmitted.

Third, if a packet encounters a full switch buffer, the header is trimmed and sent to the destination as a `TRIM` packet; the packet data is dropped. `TRIM` packets cause the receiver to generate both a `NACK` and `PULL` packet for the corresponding message. The sender processes `NACK` packets by updating the `toBtx` state in the packetization module to mark the corresponding packet as in need of retransmission. If a packet is dropped entirely, a message timeout event will trigger retransmission of `DATA` packets that are presumed lost.

```
typedef bit<16> l4port_t;

header ndp_t {
    bit<8>     flags; // DATA, ACK, NACK, PULL, TRIM
    l4port_t   src;
    l4port_t   dst;
    bit<16>    msg_len;
    bit<8>     pkt_offset;
    bit<16>    pull_offset;
    bit<16>    tx_msg_id;
    bit<16>    buf_ptr;
    bit<8>     buf_size_class;
    bit<120>   padding;
}
```

Listing 4.1: NDP header format used in the nanoPU prototype.

Fourth, network switches forward control packets (`PULL`, `TRIM`, `ACK`, `NACK`) with high priority over `DATA` packets in order to provide a low latency control loop between the network endpoints.

Listing 4.1 shows the format of the NDP header that we use in our prototype. The `flags` field indicates the type of packet (i.e. `DATA`, `ACK`, `NACK`, `PULL`, or `TRIM`'ed `DATA` pkt). Most of the fields in this header are equivalent to the ones described by the NDP authors [33]. However, we use a few additional fields to help simplify the implementation. The `tx_msg_id` field uniquely identifies the message amongst all others currently being transmitted at the sender. The Packetization module allocates the `tx_msg_id` from a free list at the same time that it allocates a buffer to store the message data; the `tx_msg_id` is freed once the message is fully `ACK`'ed. The sender includes this ID, along with a pointer to the message buffer (`buf_ptr`) and the size class from which it was allocated (`buf_size_class`) into all transmitted `DATA` packets; these fields will be explained further in Section 4.3.2. The receiver copies these fields into all control packets that it generates and transmits back to the sender. Including this information in the transport header helps to reduce state requirements at the sender because it avoids the need to maintain an additional table that maps message ID to buffer info. The NDP header also includes 15 bytes of padding to ensure that the Ethernet, IP, and NDP headers together are 64 bytes, which helps to simplify the packet parsing and deparsing logic. A highly optimized implementation should not include this additional padding in order to minimize header overhead.

We evaluate our hardware NDP implementation in Chapter 5.

4.3.2 Message Buffer Implementation

Here, we provide a brief overview of the buffer design used to perform message packetization and reassembly. Our message buffer is divided into buffers of several different fixed sizes, and a free list for each size class keeps track of which buffers are available. When a buffer is allocated, the smallest available buffer that is large enough to store the whole message is selected. For message reassembly, a buffer is allocated when the first packet of the message arrives from the network and is freed when the message is forwarded to the global RX queues.[2] For message packetization, a buffer is allocated when the application writes the first word of the message and is freed when the entire message has been successfully delivered to the receiver. The design uses a table indexed by message identifier to keep track of where each message is stored (the buffer pointer).

One of the benefits of using fixed size buffers to store messages is that it helps simplify out-of-order reassembly and retransmission: to find the position of a particular packet within the message, the hardware simply adds the appropriate offset to the message's buffer pointer. In addition, the logic that is required to manage memory buffers is very simple and can run at line rate. Buffer allocation simply requires one dequeue from a free list and buffer deallocation requires one enqueue into a free list; there is no need for complex partitioning and merging of variable size buffers.

The primary drawback of using fixed size buffers is that it can potentially lead to memory fragmentation and poor utilization of the buffer space. This is why it is important to properly configure these message buffer modules. Configuration involves selecting how to divide up the total buffer space into fixed size buffers. If the message size distribution is known at configuration time, then it is often possible to achieve very efficient buffer space utilization. For example, if a workload consists of 50% messages that are 100B and 50% messages that are 500B then the best option is to use two size classes, each with an equal number of buffers. For our evaluated workloads, we found it was possible to configure the buffers such that the design only drops packets when the total buffer space exceeds 96% utilization.

4.4 JBSQ Core Selection

As explained in Section 3.5, our NIC implements JBSQ(2) [54] to load balance messages across cores on a per-application basis. JBSQ(2) is implemented using two tables. The first maps the message's destination layer-4 port number to a per-core bitmap, indicating whether or not each core is running a thread bound to the port number. The second maps the layer-4 port number to a count of how many messages are outstanding at each core for the given port number. When a new message arrives, the algorithm checks if any of the cores that are running an application thread bound to the destination port are holding fewer than two of the application's messages. If so, it will immediately forward the message to the core with the smallest message count. If all target cores are holding two or more messages for this port number, the algorithm waits until one of the cores indicates that it has finished processing a message for the destination port. It then forwards the next message to that core. We evaluate our JBSQ implementation in Chapter 5.

[2]An arriving packet is dropped at the ingress of the reassembly module if it is unable to allocate a buffer for the message

CSR Name	Description
`lcurport`	The layer-4 port number of the current thread.
`lcurpriority`	The priority of the current thread.
`lniccmd`	A bitvector that can be used to bind or unbind a layer-4 port number from the a thread.
`lmsgsrdy`	A read-only register that produces the value 0 when the current thread's RX queue is empty and 1 otherwise.
`lidle`	Writing the value 1 to this register tells the hardware that the thread is idle and can be evicted if needed.
`lmsgdone`	Writing the value 1 to this register tells the hardware that the thread has finished processing the current message.
`ltargetcontext`	Populated by the hardware thread scheduler to indicate which thread to swap to.
`lmsgcycles`	The number of cycles a high priority (priority 0) thread is allowed to process a message before its priority is lowered to 1.

Table 4.1: Special control and status registers (CSRs) defined by the nanoPU architecture.

4.5 The nanoPU Hardware/Software Interface

Software running on the nanoPU interacts with the hardware by issuing instructions that read and write the `netRX` and `netTX` registers, as well as a special set of control and status registers (CSRs) that are described in Table 4.1. To illustrate how software on the nanoPU interacts with the hardware, we will describe two example applications. Section 4.5.1 will examine a minimal RISC-V assembly program in order to explain the basic mechanisms; and Section 4.5.2 will demonstrate how to write a simple application that computes the dot product of a vector from memory and a vector contained in a network message.

4.5.1 Loopback with Increment Example

Listing 4.2 shows a simple bare-metal loopback-with-increment program in RISC-V assembly. The program continuously reads 16B messages (two 8B integers) from the network, increments the integers, and sends the messages back to their sender. The program details are described below.

The `entry` procedure binds the thread to a layer-4 port number at the given priority level by first writing a value to both the `lcurport` and `lcurpriority` CSRs, then writing the value 1 to the `lniccmd` CSR. The `lniccmd` CSR is a bit-vector used by software to send commands to the networking hardware; in this case, it is used to tell the hardware to allocate RX/TX queues both in the core and the NIC for port 0 with priority 0. The `lniccmd` CSR can also be used to unbind a port or to update the priority level.

The `wait_msg` procedure waits for a message to arrive in the core's local RX queue by polling the `lmsgsrdy` CSR until it is set by the hardware. While it is waiting, the application tells HTS that it is idle by writing to the `lidle` CSR during the polling loop. The scheduler uses the idle signal to evict idle threads in order to schedule a new thread that has messages waiting to be processed.

The `loopback_plus1_16B` procedure simply swaps the source and destination addresses by moving the RX application header (the first word of every received message, see Section 3.2) from the

```
 1 // Simple loopback & increment application
 2 entry:
 3   // Register port number & priority with NIC
 4   csrwi lcurport, 0
 5   csrwi lcurpriority, 0
 6   csrwi lniccmd, 1
 7
 8 // Wait for a message to arrive
 9 wait_msg:
10   csrr  a5, lmsgsrdy
11   bnez  a5, loopback_plus1_16B
12 idle:
13   csrwi lidle, 1 // app is idle
14   csrr  a5, lmsgsrdy
15   beqz  a5, idle
16
17 // Loopback and increment 16B message
18 loopback_plus1_16B:
19   mv netTX, netRX // copy app hdr from rx to tx
20   addi netTX, netRX, 1 // send word one + 1
21   addi netTX, netRX, 1 // send word two + 1
22   csrwi lmsgdone, 1 // msg processing complete
23   j wait_msg // wait for the next message
```

Listing 4.2: Loopback with increment. A nanoPU assembly program that waits for a 16B message, increments each word, and returns it to the sender.

`netRX` register to the `netTX` register, shown on line 19 (Listing 4.2). It then increments every integer in the received message and appends them to the message being transmitted. An instruction that attempts to read an empty RX queue triggers an exception that can be handled by the application. After the procedure has finished processing the message, it tells HTS it is done by writing to the `lmsgdone` CSR. The scheduler uses this write signal to: (1) reset the message processing timer for the thread, and (2) tell the NIC to dispatch the next message for this application to the core. A future implementation may also want to use this signal to flush any unread bytes of the current message from the RX queue. Doing so would guarantee that the next read to `netRX` would yield the application header of the subsequent message and help prevent application logic from becoming desynchronized with message boundaries. Finally, the procedure waits for the next message to arrive.

4.5.2 Dot Product Example

The previous section described a minimal program written in RISC-V assembly in order to explain the key concepts of the nanoPU HW/SW interface. However, applications are actually written in a higher level language like C. Listing 4.3 describes a set of convenient C macros that allow applications to interact with the nanoPU hardware; namely, the `netRX` and `netTX` GPRs as well as the CSRs. These C macros are simple wrappers around in-line assembly instructions that access the appropriate hardware registers.

```c
#define NET_RX "x30"
#define NET_TX "x31"

// Wait for a msg to arrive
#define lnic_wait() while (read_csr(lmsgsrdy) == 0) { write_csr(lidle, 1); }
// Read a msg word --- move from netRX to another register
#define lnic_read() __extension__({ uint64_t __tmp; \
  asm volatile ("mv %0, " NET_RX  : "=r"(__tmp)); \
  __tmp; })
// Write a msg word --- move from register to netTX
#define lnic_write_r(val) asm volatile ("mv " NET_TX ", %0" : : "r"(val))
// Write a msg word --- from an immediate
#define lnic_write_i(val) asm volatile ("li " NET_TX ", %0" : : "i"(val))
// Write a msg word --- from memory
#define lnic_write_m(val) asm volatile ("ld " NET_TX ", %0" : : "m"(val))
// Branch based on the current msg word
#define lnic_branch(inst, val, target) asm goto (inst" %0, " NET_RX ", %1\n\
    t" : : "r"(val) : : target)
// Indicate processing completion of the current msg
#define lnic_msg_done() write_csr(lmsgdone, 1)
```

Listing 4.3: A few helper macros that applications can use to interact with the nanoPU hardware. Namely, the `netRX` and `netTX` GPRs, as well as the CSRs.

Listing 4.4 shows the main processing loop of a simple C application that computes the dot product of a vector stored in memory and a vector contained within network messages. The application first waits for a message to arrive then extracts the application header (the first word of every message). The second word of the message indicates the message type. This application is only designed to process `DATA_TYPE` messages and hence it checks this field to verify the message type. The third word indicates the number of 8B words in the vector contained within the message. For each word of the vector, the message also indicates which in-memory weight to use when computing the dot product. Note that the application processes message data directly out of the register file and hence message data is never copied into memory. This feature allows this application to run faster on the nanoPU than a traditional system. Finally, the application sends a response message back to the sender which contains the resulting dot product.

4.5.3 OS Integration Considerations

This section describes a number of additional features to consider when deploying the nanoPU as an actual system rather than a research prototype. Our current nanoPU prototype runs only bare metal applications without an operating system. We believe this is sufficient to evaluate the key benefits provided by the nanoPU architecture. However, a real deployment would likely require support for an operating system that implements features such as virtual memory, privilege modes, process isolation, and exception/interrupt handling.

The simple example described in Section 4.5.1 conflates application processing with OS logic. In particular, the instructions to bind the thread to a layer-4 port number at a given priority level

```
while (1) {
  // Wait for a msg to arrive
  lnic_wait();

  // Extract application header from RX msg and check msg type
  app_hdr = lnic_read();
  if (lnic_read() != DATA_TYPE) {
    printf("Expected Data msg.\n");
    return -1;
  }

  // Compute the dot product of the msg vector with in-memory data
  uint64_t num_words = lnic_read();
  uint64_t result = 0;
  for (i = 0; i < num_words; i++) {
    uint64_t idx = lnic_read();
    uint64_t word = lnic_read();
    result += word * weights[idx];
  }

  // Send response message
  lnic_write_r((app_hdr & (IP_MASK | PORT_MASK)) | RESP_MSG_LEN);
  lnic_write_i(RESP_TYPE);
  lnic_write_r(result);
  lnic_msg_done();
}
```

Listing 4.4: Dot product example.

(lines 4-6 in Listing 4.2) should instead be performed by a system call that first verifies whether or not the thread is allowed to bind to the requested port number.

Additionally, operating systems may require the ability to migrate a thread between cores. This is tricky because it would require the OS to move any message data out of the thread's current RX queue an into the RX queue on the destination core. This means that the local RX queue must provide a means for privileged software to inject data into a local RX queue.

Furthermore, the nanoPU's hardware thread scheduler (HTS) module should be carefully integrated with the operating system's thread scheduling logic. In the current prototype, the HTS indicates which thread to swap to by providing the layer-4 port number of the target thread; it is not aware of any "traditional" threads that are not using the register file network interface. The interface between the OS and the HTS would need to be modified if nanoRequest processing threads are to share cores with traditional threads.

Chapter 5

nanoPU Evaluations

Our evaluations address the following five questions:

1. How does the performance of the nanoPU register file interface compare to a traditional DMA-based network interface (Section 5.2.1)?
2. Is the hardware thread scheduler (HTS) able to provide low tail response time under high load and bounded tail response time for well-behaved applications (Section 5.2.2)?
3. How does our hardware NDP implementation perform under a high incast workload?
4. Is the nanoPU's JBSQ core selector able to efficiently load balance messages across cores?
5. How do real applications perform using the nanoRequest fast path (Section 5.3)?

5.1 Methodology

We compare our nanoPU prototype against an unmodified RISC-V Rocket core with a standard NIC (IceNIC [48]), which we call a *traditional* NIC. The traditional NIC is implemented in the same simulation environment as our nanoPU prototype and performs DMA operations directly with the last-level (L2) cache. The traditional NIC does not support hardware-terminated transport or multi-core network applications, however, an ideal traditional NIC would support both of these. Therefore, for our evaluations, we do not implement transport in software for the traditional NIC baseline; we omit the overhead that would be introduced by this logic.

Our evaluations ignore the overheads of translating addresses because we run bare-metal applications using physical addresses. When using virtual memory, the traditional design would perform worse than reported here, because the message buffer descriptors would need to be translated resulting in additional latency, and more TLB misses. There is no need to translate addresses when processing nanoRequests from the register file.

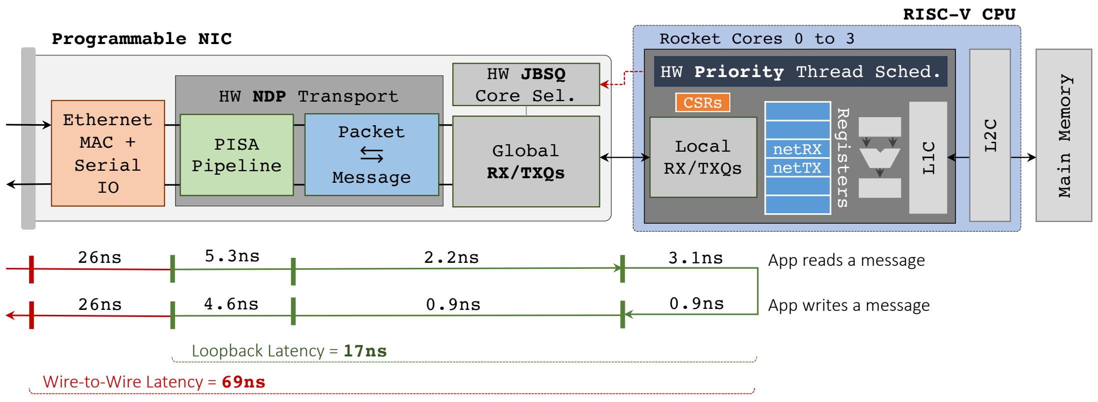

Figure 5.1: nanoPU prototype latency breakdown. Total wire-to-wire latency for an 8B message (72B packet) is **69ns**.

Benchmark tools: We use two different cycle-accurate simulation tools to perform our evaluations: (1) the Verilator [96] software simulator, and (2) the Firesim [48] FPGA-accelerated simulator. Firesim enables us to run large-scale, cycle-accurate simulations with hundreds of nanoPU cores using FPGAs in AWS F1 [4]. The FPGAs run at 90MHz, and we simulate a target clock rate of 3.2GHz—all reported results are in terms of this target clock rate. The simulated servers are connected by C++ switch models running on the AWS x86 host CPUs.

5.2 Microbenchmarks

5.2.1 Register file interface

Loopback response time: Figure 5.1 shows a breakdown of the latency through each component of the nanoPU fast path for a single 8B nanoRequest message (in a 72B packet) measured from the Ethernet wire through a simple loopback application in the core, then back to the wire (first bit in to last bit out).[1] As shown, the loopback response time through the nanoPU fast path is only 17ns, but in practice we also need an Ethernet MAC and serial I/O, leading to a wire-to-wire response time of 69ns.

For comparison, Figure 5.2 shows the median loopback response time for both the nanoPU fast path and the traditional design for different messages sizes. For an 8B nanoRequest, the traditional design has a 51ns loopback response time, or about 3× higher than nanoPU. 12ns (of the 51ns) are spent performing `memcpy`'s to swap the Ethernet source and destination addresses, something that is unnecessary for nanoPU, because it is handled by the NIC hardware. The relative speedup of the nanoPU fast path decreases as the message size increases because the response time becomes

[1]Our prototype does not include MAC & Serial IO, so we add real values measured on a 100GE switch (with Forward Error Correction disabled).

dominated by store-and-forward delays and message-serialization time.

If instead the traditional NIC placed arriving messages directly in the L1 cache, as NeBuLa proposes [90], the loopback response time would be faster, but we estimate that the nanoPU fast path would still have roughly 50% lower response time for small nanoRequests.

Loopback throughput: Figure 5.3 shows the throughput of the simple loopback application running on a single core for both the nanoPU fast path and the traditional NIC. The traditional NIC processes batches of 30 packets, which fit comfortably in the LLC. Batching allows the traditional NIC to overlap computation (e.g. Ethernet address swapping) with NIC DMA send/receive operations.

Throughput is dominated by the software overhead to process each message because that is the only component of the loopback latency that is not pipelined. For the register file interface, the software overhead is: read the `lmsgsrdy` CSR to check if a message is available for processing, read the message length from the application header, and write to the `lmsgdone` CSR after forwarding the message. For the traditional design, the software overhead is: perform MMIO operations to pass RX/TX descriptors to the NIC and to check for RX/TX DMA completions, and `memcpy`'s to swap the Ethernet source and destination addresses.

Because of lower overheads, the application has 2–7× higher throughput on nanoPU than on the traditional NIC. For 1KB messages, the nanoPU application has a loopback throughput of 166Gb/s (83% of the line-rate). When we add the per-packet NDP control packets sent/received by the NIC, the 200Gb/s link is completely saturated.

Stateless nanoRequest jobs: The nanoPU is well-suited for compute-intensive applications that transform the data carried by self-contained nanoRequests. We use a very simple benchmark application that increments each word of the message by one and forwards the message back into the network; this is similar to the program described in Section 4.5.

Figure 5.4 shows that the nanoPU accelerates the throughput of this application by up to 10×. NanoRequest data is read from the register file and passed directly through the ALU; no memory operations are required at all. On the other hand, when using the traditional NIC, each word of the message must be read from the last-level cache (LLC), passed through the ALU, and the final result is written back to memory. If instead the traditional NIC loaded words into the L1 cache, as in [90], we estimate a throughput about 1.3× faster than via the LLC. This would still be 7.5× slower than the nanoPU fast path. In Section 5.3, we will compare benchmarks for real applications.

Stateful nanoRequest jobs: These are applications that process both message data and local memory data. Our simple microbenchmark computes the dot-product of two vectors of 64-bit integers, one from the arriving message and a *weight vector* in local memory. The weight vector is randomly chosen from enough vectors to fill the L1 cache (16kB).

There are two ways to implement the application on the nanoPU. The *optimal* method is to process each message word directly from the register file, multiplying and accumulating each word

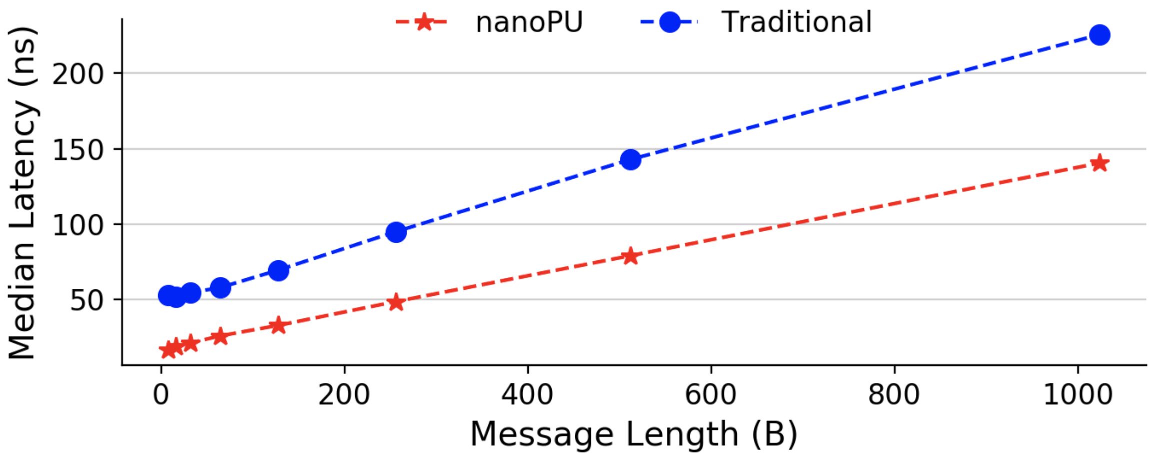

Figure 5.2: Loopback median response time across various message lengths for nanoPU fast path vs traditional NIC.

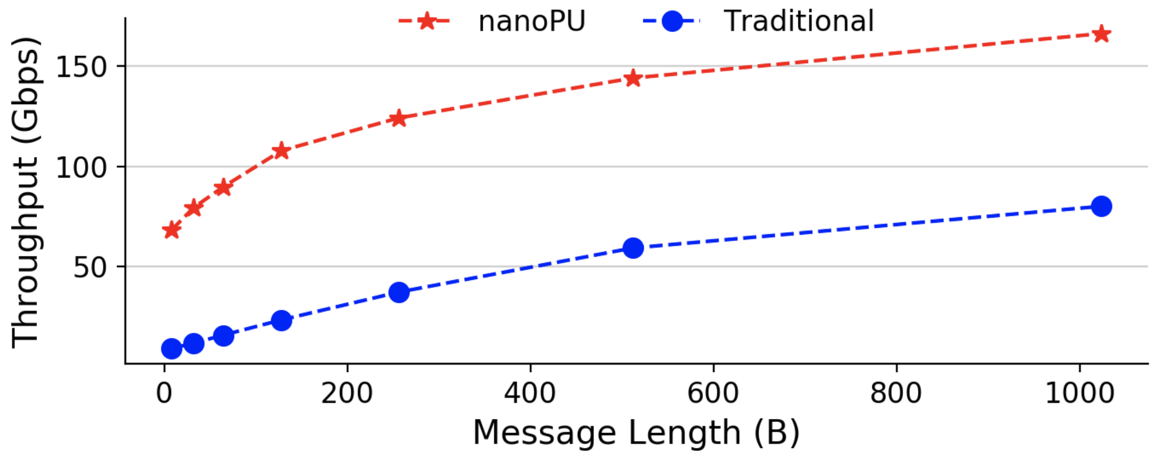

Figure 5.3: Loopback throughput across various message lengths for nanoPU fast path vs traditional NIC.

with the corresponding weight value from memory. The *naive* method copies the entire message from `netRX` into memory before computing the dot product with the weight vector. The *traditional* design processes messages in batches of 30 to overlap dot-product computation with DMA operations. Figure 5.5 shows the throughput speedup of the *optimal* and *naive* methods relative to the traditional application, for different vector sizes.

- *Small messages:* For small vectors, nanoPU is 4–5× faster because of fewer per-message software overheads.

- *Large messages:* For large vectors, throughput is limited by the longer dot-product computation time. The *optimal* application consistently doubles throughput by keeping message data out of the L1 cache and reducing cache misses. The *naive* application is slowed by the extra

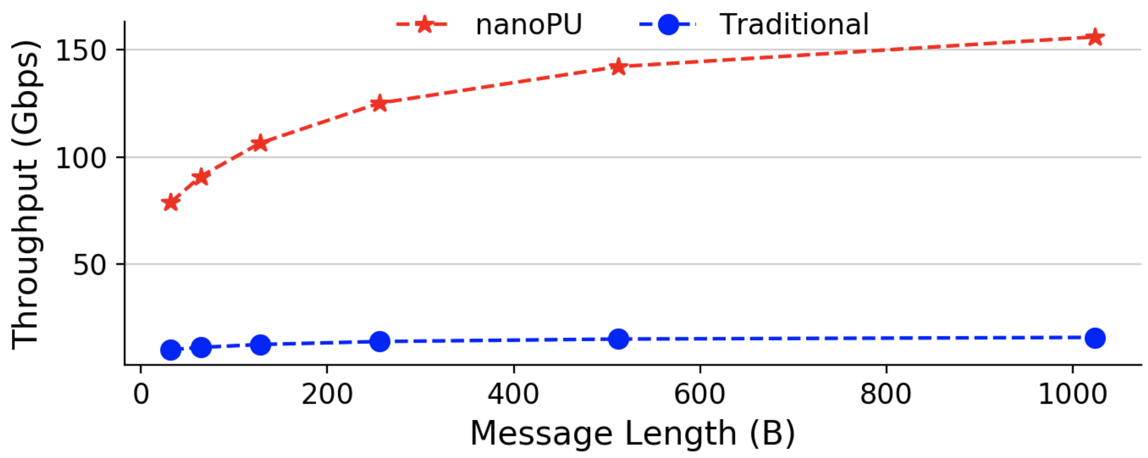

Figure 5.4: Loopback-with-increment throughput across various message lengths for nanoPU fast path vs traditional NIC.

copy, and about twice as many L1 data cache misses. The *traditional* application has 10× as many L1 data cache misses as *optimal* because message data must be fetched from the LLC, which pollutes the L1 cache, evicting weight data. If we speed up the traditional NIC by placing message data directly in the L1 cache, as NeBuLa proposes [90], we estimate the traditional design would run 1.5× faster for large vectors, however, *optimal* would still be about 30% faster.

The benefits are clear when an application processes message data directly from the `netRX` register. While this may initially seem like a big constraint, we have found that it is generally quite feasible, and even natural to design applications this way. We demonstrate example applications in Section 5.3.

5.2.2 Hardware thread scheduling

Next, we evaluate how much the hardware thread scheduler (HTS) can reduce tail response time under high load.

Methodology: We evaluate tail response time under load by connecting a custom (C++) load generator to our nanoPU prototype via a simulated Ethernet network in Firesim [48]. The load generator produces nanoRequests with Poisson inter-arrival times, and measures the end-to-end response time. We plot the 99th %ile tail response time versus load. The system becomes saturated when requests arrive faster than they are processed, which causes queue build up and requests to be dropped. Dropped requests are treated as having infinite response time.

Priority thread scheduling: We compare our hardware thread scheduler (HTS) against a timer-interrupt driven thread scheduler (TIS) that is intended to be representative of state-of-the-art

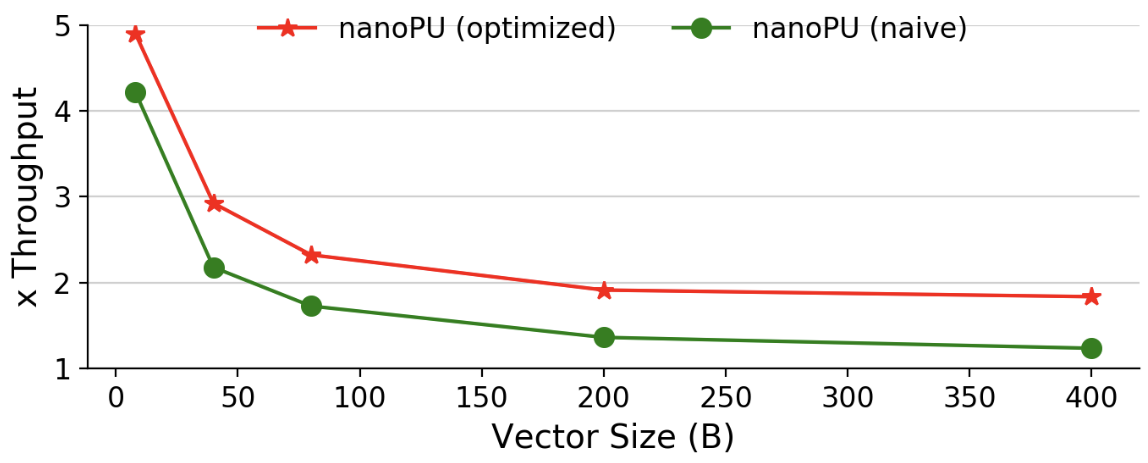

Figure 5.5: Dot-product microbenchmark throughput speedup for various vector sizes; nanoPU fast path (naive & optimal) relative to traditional NIC.

low-latency operating systems such as Shinjuku [44] and Shenango [72]. The timer-interrupt driven scheduler checks to see if a thread swap is required whenever a timer interrupt fires (every 5μs). The hardware is constantly monitoring the RX queues and thread status to identify the highest priority active thread. The timer interrupt handler reads a CSR that is populated by the hardware to determine if a context switch is required.

For both TIS and the nanoPU 's HTS, the decision as to which thread to run next is made by the hardware; the main difference between the two is that with TIS, context switches can only happen at 5μs intervals; whereas with HTS, a context switch is initiated the instant the current thread is no longer the highest priority active thread. The Shinjuku [44] and Shenango [72] systems indicate that software overheads required for inter core synchronization limit scheduling decisions to once every 5μs. If we were to design a software thread scheduler running on a dedicated core that is performing the same tasks as the nanoPU HTS, we would expect roughly the same limitation to arise. That is why we are using 5μs intervals for our TIS baseline. We evaluate both schedulers when they are scheduling two threads: one with priority 0 (high) and one with priority 1 (low). The load generator issues 10K requests for each thread, randomly interleaved, each with an on-core service time of 500ns (i.e. an ideal system will process 2Mrps).

Figure 5.6 shows the 99th %ile tail response time vs load for both thread scheduling policies, with a high and low priority thread. Figure 5.7 shows the same data using a smaller range on the y-axis in order to clearly see the data. For timer based approach, if the high priority thread has any work to do when a timer interrupt fires (every 5μs) then the high priority thread will remain on the core. The entire time the high priority thread is on the core, messages can arrive for the low priority thread (orange line) causing queueing delay, which increases the tail response time. The curves saturate once the low priority RX queue fills up and messages are dropped. The nanoPU

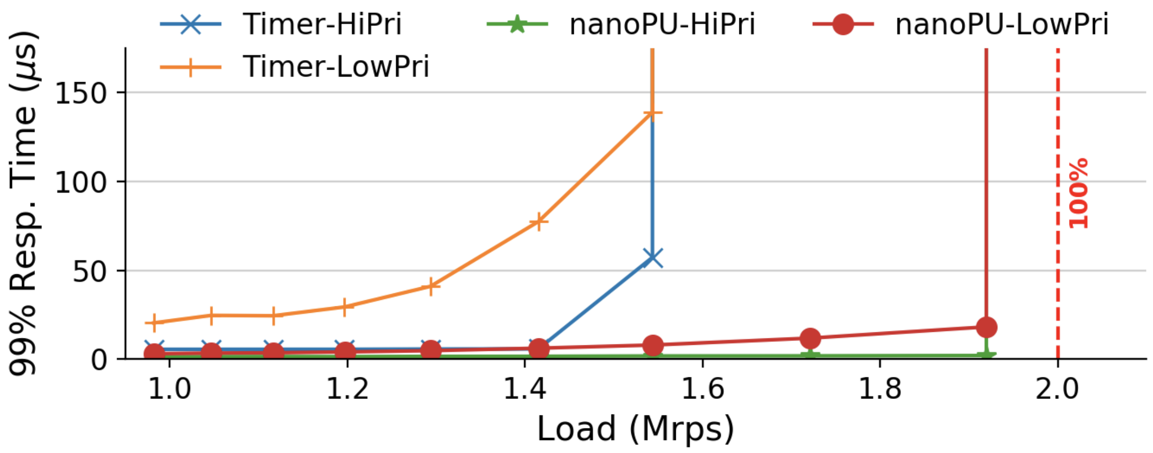

Figure 5.6: Comparing hardware thread scheduler (HTS) performance against a traditional timer-interrupt driven scheduler (TIS); 99th %tile tail response time vs load for both a high-priority and low-priority thread for each scheduler.

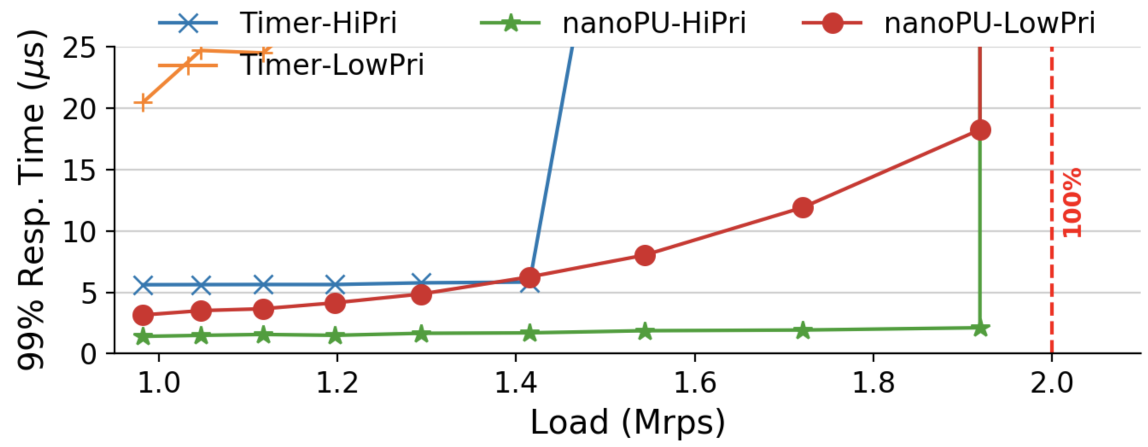

Figure 5.7: The same data as Figure 5.6 but using a zoomed in y-axis.

design does not currently partition buffer space on a per-application basis, which means that when the low priority RX queue fills up, high priority messages can be dropped. This is why the high and low priority curves saturate at the same load. The curves for the timer based approach (orange and blue) saturate before the nanoPU curves because the nanoPU HTS is able to keep the low priority RX queue smaller by being essentially work conserving. With HTS, the core is always processing a message if there is one to work on, which is not the case for the timer based scheduler - the high priority thread can be sitting on the core idle while the low priority thread has work to do. HTS reduces the tail response time of the high and low priority thread by 4× and 6.5× at low load, respectively; and can sustain at least 96% load.

Bounded message processing time: HTS is designed to bound the tail response time of well-behaved applications, even when they are sharing a core with misbehaving applications. To test this, we configure a core to run a well-behaved thread and a misbehaving thread, both configured to run at priority 0. As a reminder, a well-behaved thread is one that process all of its messages within a bounded amount of time. All requests have an on-core service time of 500ns, except when a thread misbehaves (once every 100 requests), in which case the request processing time increases to 5μs.

Figure 5.8 shows the 99th %ile tail response time vs load for both threads with, and without, the bounded message processing time feature enabled. When enabled, if a priority 0 thread takes longer than 1μs to process a request, HTS lowers its priority to 1. When disabled, all requests are processed by the core in FIFO order.

Using Equation (3.1), we expect an application with at most one message at a time in the RX queue, to have a tail response time bounded by the link latency (43ns), the NIC latency (17ns), the maximum message processing time (1000ns), and the context switch latency: $2 \cdot 43ns + 17ns + 2 \cdot 1000ns + 50ns = 2.15\mu s$. This matches our experiments: with the feature enabled, the tail response time of the well-behaved thread never exceeds 2.1μs, until the offered load on the system exceeds 100% (1.9 Mrps).[2] HTS lowers the priority of the misbehaving application the first time it takes longer than 1μs to process a request. Hence, the well-behaved thread quickly becomes strictly higher priority and its 500ns requests are never trapped behind a long 5μs one. Note also that by bounding message processing times, shorter requests are processed first, queues are smaller and the system can sustain higher load.

At the beginning of this dissertation, we asked the question: *is it possible to bound RPC response time?* The conventional wisdom tells us that this is not possible. But as we have demonstrated here, this is not as crazy as it seems. For *high priority*, *well-behaved* applications under *controlled load*, it is indeed possible to bound the *server's* RPC response time. There is still work to be done in refining and eliminating the caveats required to answer this question, but we think of this more as the first word rather than last word. Questions that still need to be answered include: how do we enable developers to write well-behaved applications? How do we ensure controlled load within applications in order to bound queueing delay at the server? How can we bound latency through the network in order to bound *end-to-end* RPC response time? We hope our results will encourage researchers to pursue these important questions further.

5.2.3 Hardware NDP transport

If the nanoPU and extremely fine-grained computing become prevalent, then applications will be distributed across many more servers than they are today and we can expect significant amounts of incast. We therefore evaluate our NDP implementation by running an 80-to-1 incast experiment.

[2]This is despite our Poisson arrival process occasionally allowing more than one message in the RX queue.

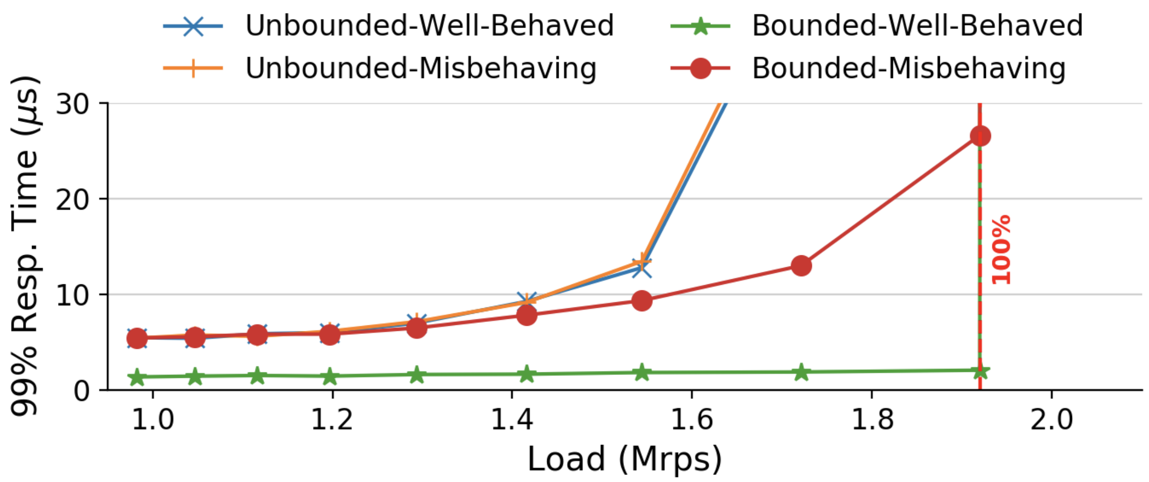

Figure 5.8: 99th %ile response time vs load for well-behaved and misbehaved threads, with and without bounded message processing time.

The experiment runs on 81 AWS FPGAs simulating 81 nanoPUs with a total of 324 cores; the experiment is coordinated by Firesim. The 81 nanoPUs connect to a single switch via 200 Gb/s links; the RTT of the network is 3μs. All 80 clients send a single 1024B message (in a 1088B packet) to the server at the same time. The bottleneck queue size is 81KB, and is therefore only large enough to hold 74 of the 80 packets; therefore, most of the packets will be queued, while others will be trimmed (when we enable NDP) or dropped (otherwise). We run two experiments, one with NDP congestion control enabled and one with it disabled (by disabling packet trimming in the switch).

Figure 5.9 shows a time series of the occupancy of the bottleneck queue at the switch, with and without NDP enabled. At the beginning, we see all 80 packets arrive at the same time and filling up the switch queue. Without NDP (blue line), six packets are silently dropped at the onset of the incast. The senders must infer that their packets were dropped using a timeout. All of the retransmitted packets arrive at the same time, causing a smaller secondary incast. After 13μs the final byte of the final packet arrives.

On the other hand, with NDP enabled, six packets are trimmed and their headers are placed into the control queue and forwarded with high priority. For each `TRIM` packet received, the server generates a `NACK` packet and a paced `PULL` packet to tell the client to retransmit the data that was lost. `PULL` packets are scheduled so that the retransmitted packets arrive at the bottleneck link at line-rate. In total, it takes 4.2μs for the final byte of the final packet to be serialized onto the bottleneck link, which is about three times quicker than without NDP enabled.

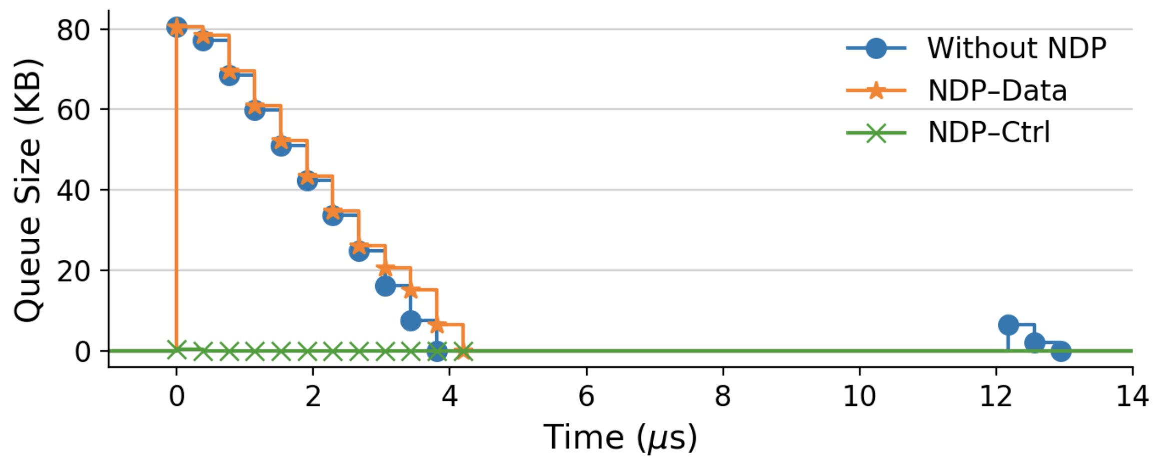

Figure 5.9: Occupancy of the bottleneck queue in the switch for 80-to-1 incast experiment, with and without NDP enabled.

5.2.4 Hardware JBSQ core selection

The hardware core selection algorithm steers incoming messages to nanoPU cores for processing. We evaluate and compare three different algorithms using a workload representative of an application like Redis [83]. We assume that 99.5% of messages are simple get/put requests (modeled by a nanoPU service time of 500ns) and 0.5% of messages are complex range queries (with a 5μs service time). We compare three core selection techniques:

- **RSS (Receive Side Scaling):** This is a simple load-balancing algorithm commonly used by modern NICs. One thread runs on each core and is fed by a separate global RX queue (one per-thread, which is also one per-core). Each thread is assigned a unique port number, and the load generator selects a port number uniformly at random.

- **JBSQ:** This is the algorithm described in Section 3.5. We run one thread per core, allocate one global RX queue for all threads (i.e., all threads share the same port number). The JBSQ algorithm load balances requests to cores.

- **JBSQ-PRE:** In this prioritized version, the short requests are assigned priority 0 (high), and long requests run at priority 1 (low). Each type of request has its own port number. We run two threads on each core (one per-priority) and run JBSQ with strict priority thread scheduling at each core as new messages arrive (described in Section 3.3).

Figure 5.10 shows the 99% tail response time vs load for the three techniques described above. The tail response time of JBSQ is less than RSS because short requests do not get stuck behind long requests, unless all cores are busy processing long requests. In that case, JBSQ-PRE is even better, because the nanoPU thread scheduler will strictly prioritize processing short requests over

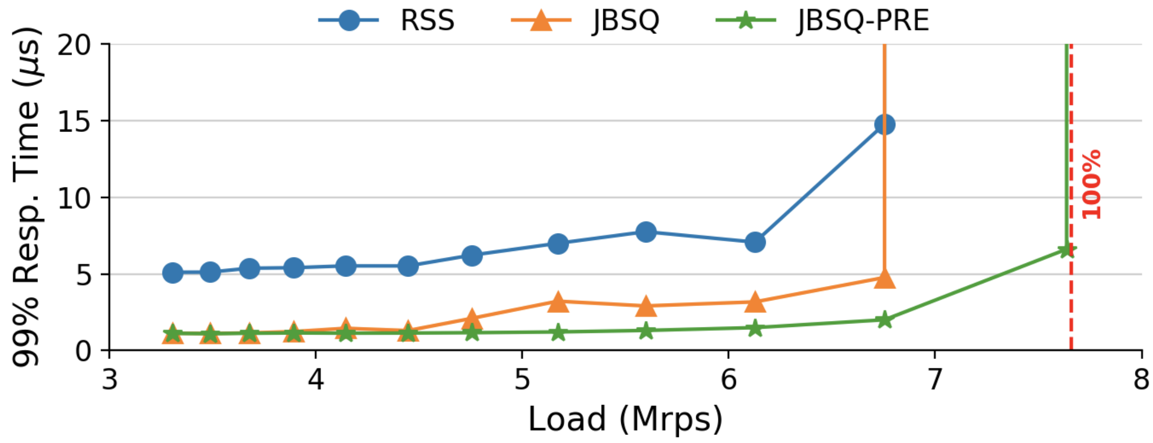

Figure 5.10: 99% tail response time vs load for three core-selection algorithms: RSS, JBSQ and JSBQ-PRE (with two priorities). Bimodal message service times: 99.5% - 500ns, 0.5% - 5μs.

long requests, preemptively if necessary. JBSQ-PRE sustains higher overall load (almost 100%) because it keeps the queues smaller by processing short requests first.

Our evaluation shows that with the combination of an efficient core selection algorithm and a fast per-message, preemptive, prioritized thread scheduling algorithm, we can sustain very high load and low response time from the nanoPU cores.

5.3 Application Benchmarks

As shown in Table 3.1, we have implemented and evaluated many applications on our nanoPU prototype. Below, we present the evaluation results for a few of these applications in detail.

5.3.1 MICA

MICA [60] is a high performance, key-value store application. We ran MICA on both the nanoPU and the traditional NIC designs. Porting to the nanoPU required modifying only 36 lines of functional code.

In our evaluation, we configure MICA to maintain a database of 10K key-value pairs (16B keys and 512B values) using a single core. The load generator sends a 50/50 mix of read/write nanoRequest queries with keys picked uniformly at random from the set. Figure 5.11 compares the 99th %ile tail response time vs load for both the traditional and nanoPU versions of this application.

Response time is shorter on nanoPU because we can process requests directly from `netRX` into, and out of, the MICA table, eliminating the need for `memcpy`'s to/from DMA buffers. Additionally, the nanoPU minimizes cache misses by avoiding the need to pollute the cache hierarchy with message

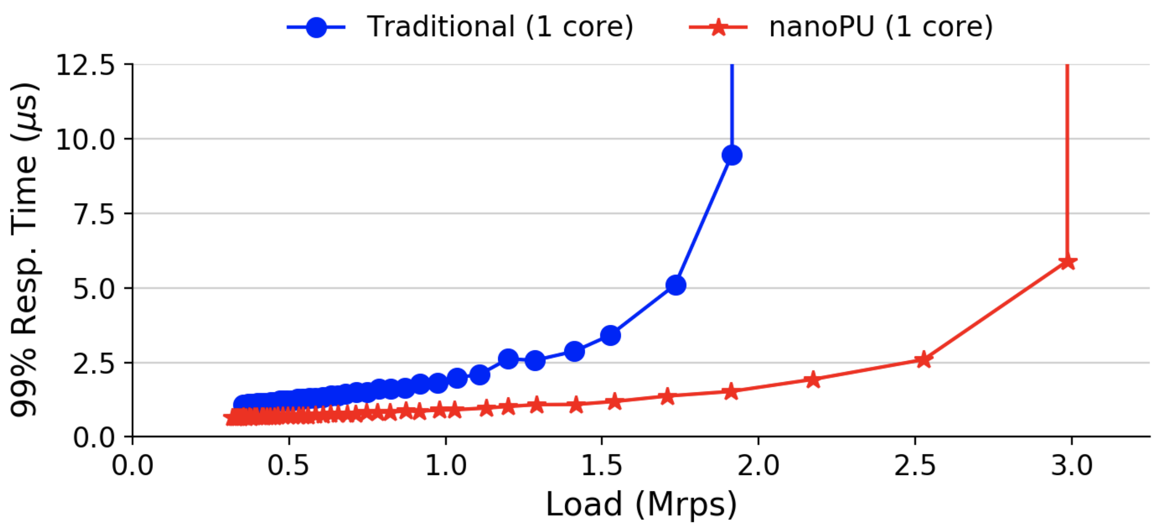

Figure 5.11: MICA KV store: 99th %ile wire-to-wire tail response time vs load for READ and WRITE requests.

data. As a result of both of these factors, the on-core processing time of each message is lower and the nanoPU is able to achieve about 50% higher load.

5.3.2 Raft

Raft [70] is a widely used consensus protocol for implementing fault-tolerant, state machine replication. We ported a production grade Raft implementation [82] to the nanoPU and used it to build a three-way replicated key-value store with MICA [60] (16B keys, 64B values). The Raft cluster correctly implements leader election, can tolerate server failure, and our client can automatically identify a new Raft leader. We evaluate the response time of the Raft cluster under steady-state, failure-free conditions.

Figure 5.12 depicts the topology and communication pattern used in our evaluation. The client and Raft servers are all connected to a single switch. The switch has a forwarding latency of 300ns (typical of modern cut-through commercial switch ASICs [91]) and all links have a latency of 43ns. The response time of the cluster is measured at the client; it includes the time it takes the leader to replicate the requested write operation across both followers. In 10K trials, the median, zero-load, response time was 3.08μs, with a 3.26μs 99th %ile tail response time. eRPC [45], a high performance, highly-optimized RPC library reports a 5.5μs median and 6.3μs 99th %ile tail response time for the same operation — about a factor of two slower. Given the different environments in which the nanoPU and eRPC implementations are running, it is difficult to compare the two directly. Hence, this comparison should be taken with a grain of salt. However, it goes to show that the nanoPU is able to achieve state-of-the-art performance for real applications.

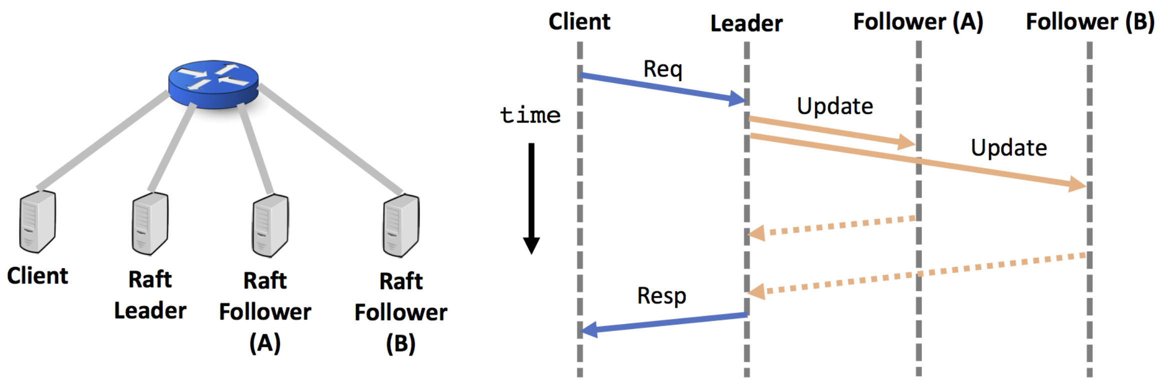

Figure 5.12: The topology (left) and communication pattern (right) used to evaluate Raft.

5.3.3 Set Algebra

Information retrieval systems such as Lucene [12] use set algebra for data mining, text analytics, and search. For example, in order to find all documents related to a particular search query, a common technique is to start with a reverse index that maps words to a set of related document IDs, then compute the intersection of all sets corresponding to the words in the query.

We implemented this set intersection application on both the nanoPU and the traditional design. We created a reverse index of 100 Wikipedia [98] articles with 200 common English words. This is significantly smaller than what would be used in a modern production application. However, we believe that if we had a system like the nanoPU which offered ultra low and predictable RPC response time then we would deploy production applications differently than we do today. In particular, we would likely distribute the application across many more servers to increase parallelism and reduce runtime.

Our load generator sends search queries with 1–4 words chosen from a Zipf distribution based on word frequency. Figure 5.13 shows the tail response time for both systems. By efficiently processing messages directly out of the register file with very minimal per-message overhead for network IO, the nanoPU is able to achieve about 40% higher load than the traditional design. Furthermore, JBSQ load balancing enables the application running on nanoPU to achieve linear scalability with the number of cores, while ensuring low 99th %ile tail response time. This can be seen from the fact that a single nanoPU core achieves a max load of about 1.75Mrps while 4 cores is able to achieve about 7Mrps.

5.4 Reproducibility

This section provides a brief overview of the artifact that we have put together to make it easy for others to reproduce the experiments described in this chapter. The artifact documentation can be

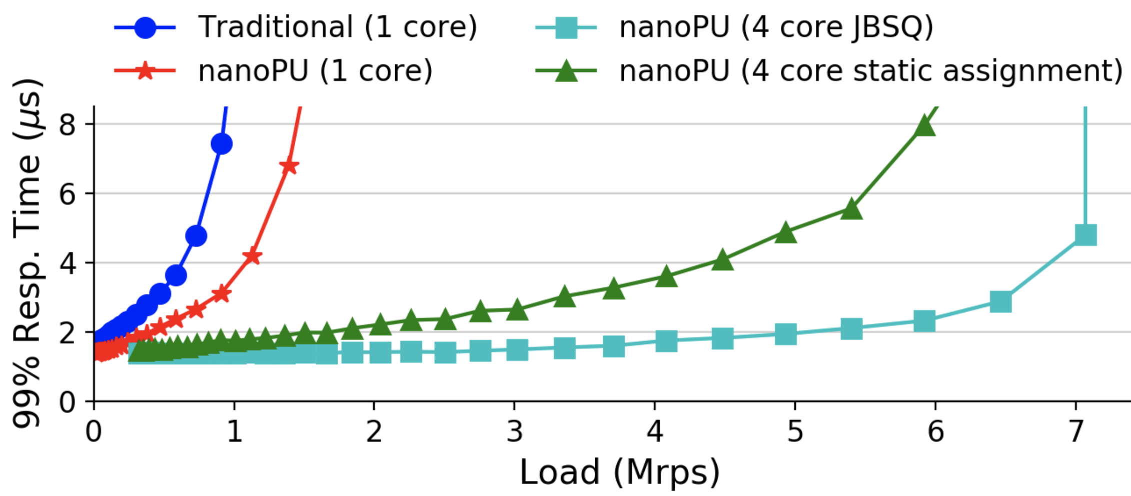

Figure 5.13: Set intersection: 99th %ile wire-to-wire tail response time vs load.

found in our public GitHub repository.[3] The documentation provides an overview of the repository structure as well as instructions to set up and run the nanoPU simulations.

Some of the simulations are conducted using the Verilator [96] cycle-accurate software simulator and others use the Firesim [48] AWS FPGA-accelerated simulation platform. We have developed a custom AWS EC2 image and made it publicly available on the community marketplace. This image has all of the necessary tools and repositories pre-installed to make reproducibility as easy as possible. The artifact documentation includes detailed instructions to: (1) configure an AWS account for Firesim experiments, (2) launch our custom image on EC2, and (3) reproduce all of the nanoPU experiments.

We hope this artifact will provide a useful starting point for other researchers to build upon the nanoPU.

[3]https://github.com/l-nic/chipyard/blob/lnic-dev/ARTIFACT.md

Chapter 6

Event-Driven Packet Processing

Chapter 3 describes the nanoPU design, which includes programmable support for transport protocols in hardware. The transport architecture that we use belongs to a new class of programmable architecture that we call event-driven PISA. While previous chapters have focused on the end host network stack, this chapter will provide a detailed discussion of event-driven PISA architectures within the context of network switch design.

Programmable network devices have been gaining significant traction within the networking community as a result of their unique ability to deploy custom algorithms that operate at line rate. There have already been many interesting applications that take advantage of this new found ability to program the data plane [22, 42, 41, 36, 65]. P4 [13] has emerged as the *de facto* language for programming the data plane. P4 programs are designed to be compiled onto a class of data-plane architectures called Protocol Independent Switch Architecture (PISA) [14]. PISA architectures are composed of programmable parsers, match-action pipelines, and deparsers and are designed to process packets at line rate. Each instance of a PISA architecture exposes a certain data-plane programming model to the P4 programmer who then works within the confines of the provided programming model to implement their custom processing logic. Every data-plane programming model is driven by a set of data-plane events, where a data-plane event is an architectural state change that triggers processing in the programming model.

The simple PISA architecture introduced in [14] consists of a single programmable parser, match-action pipeline, and deparser connected in series. The P4 language consortium recently defined a different PISA architecture called the Portable Switch Architecture (PSA), which is depicted in Figure 6.1. The PSA consists of two P4 programmable pipelines, one to process packets on ingress and one to process packets on egress as they leave the device. Both of these architectures are what we call *baseline PISA* architectures. A baseline PISA architecture supports a programming model that exposes synchronous packet-by-packet processing to the P4 programmer. That is, the programming model only allows developers to define how to handle a small set of packet-related events, usually

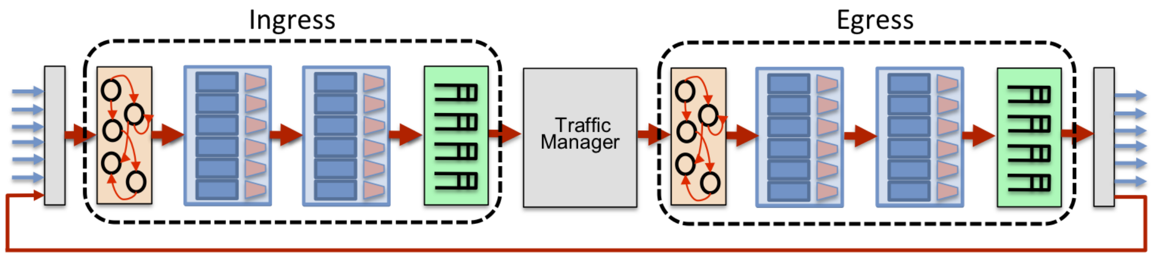

Figure 6.1: Simplified diagram of the portable switch architecture (PSA), which consists of separate ingress and egress pipelines to handle packet arrival and departure events, respectively.

ingress and egress packet events.

We observe that many data-plane algorithms do not naturally fit into this synchronous packet-by-packet programming model. Some applications need to execute logic independently of packet arrivals and departures. For example, HULA [49] is a load balancing application that must periodically generate probe packets to measure link utilization. When deployed on a baseline PISA architecture, these HULA probe packets must be generated by either the control plane or end hosts because the programming model provides no means to perform periodic tasks or generate packets. Similarly, Count-Min Sketch (CMS) [18] is a commonly used data-plane primitive that must be periodically reset. When a CMS is used in a baseline PISA architecture, the control plane must be responsible for performing the reset operation. This can lead to significant overhead for the control plane, especially if the data structure must be frequently reset. Other data-plane operations, such as measuring flow rates or computing average queue occupancies, must compute functions of a signal over a moving window of time. While this type of operation is sometimes possible to implement using only packet events, it is often cumbersome and challenging to do so. Furthermore, many data-plane applications can benefit from the ability to update algorithmic state multiple times while processing a packet. For instance, computing congestion signals such the number of buffered flows inherently requires state updates both as packets are enqueued and dequeued from the buffer.

Event-driven PISA architectures provide a programming model that explicitly exposes a rich set of data-plane events to the P4 programmer. As packets traverse the architecture, they generate events such as buffer enqueue, dequeue, or overflow events, which are subsequently handled by dedicated processing threads that share state with the packet processing threads. Events may also be generated independently of any packet processing logic, such as based on a timer configuration, a link status change, or a control-plane command. An event-driven programming model allows a P4 programmer to express how each of the individual data-plane events are handled. Event-driven PISA architectures alleviate many of the key limitations of baseline PISA architectures. In particular, they enable more interesting stateful packet processing applications as they allow data-plane programs to spawn threads that perform background maintenance of algorithmic state, as well as perform periodic tasks such as generating packets.

The main contributions presented in this chapter are:

- We propose a common, general way to express line rate data-plane *event processing* beyond just packet arrival and departure events.
- We identify a set of useful data-plane events that can be used to implement a wide range of data-plane algorithms.
- We identify classes of applications that will benefit from the proposed programming model.
- We demonstrate feasibility of the approach at line rate by architecting an event-driven architecture on the NetFPGA SUME platform.

6.1 Event-Driven Programming

P4 programs are often compiled to run on a PISA pipeline comprising multiple *match+action* stages. The baseline PISA architecture only supports events that are triggered by packet arrivals and departures. In this chapter we will explore how to enhance the baseline programming model to support a richer set of data-plane events.

We start by examining a broad set of data-plane applications in order to identify a set of useful data-plane events. Our list is shown in Table 6.1. The first three are packet events (ingress, egress, and recirculated) and are commonly supported in the baseline programming model. The remaining events, such as when a packet is enqueued or dequeued, a buffer overflows or underflows, a timer expires, or a link status changes, are sometimes available from the hardware, but are not exposed by the programming model.

Our event-driven PISA programming model explicitly exposes data-plane events to the P4 programmer by allowing them to define custom event handling logic. A particular target device exposes the precise set of events that it supports via the P4 architecture description file. This generalization from packet events to data-plane events gives data-plane programs much more flexibility, and if designed appropriately, still allows packets to be processed at line rate. We next describe how these events are exposed to data-plane developers within our proposed event-driven programming model.

Example Logical Architecture Model. We consider a simple event-driven architecture that only supports ingress packet events, enqueue events, and dequeue events. Figure 6.2 depicts a block diagram of this logical architecture model. Ingress packet events trigger processing in the ingress PISA pipeline. Every time a packet is enqueued in the switch buffer, the traffic manager extracts some metadata from the packet and uses it to fire an enqueue event which then triggers the logical enqueue pipeline. A similar procedure occurs for dequeue events. Each of these pipelines have some notion of local state as well as global shared state.

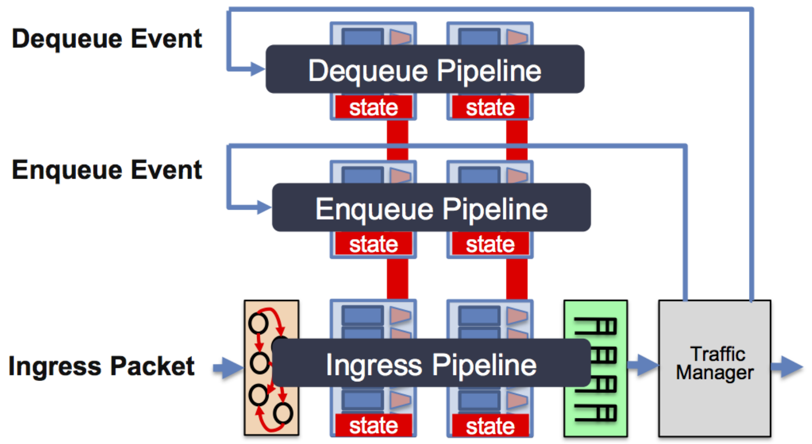

Figure 6.2: A diagram of a logical event-driven data-plane architecture. Each event triggers processing in a separate logical pipeline.

Writing Event-Driven Programs. Let us see how we can write an event driven P4 program for our example architecture model. Our P4 program will monitor the buffer occupancy of each active flow. It will use this information to identify microburst culprits: flows that contribute to a sudden, significant increase in buffer usage. As noted in [15], this is very challenging to do on baseline PISA architectures because the programming model does not allow state to be updated both when packets are enqueued and dequeued from the buffer. The authors needed to maintain multiple, complex, stateful data structures to keep track of the (approximate) queue occupancy in the *egress* pipeline. If instead we use our event-driven programming model, we can reduce the stateful requirements at least four-fold and can perform the detection in the ingress pipeline *before* packets are enqueued in the switch buffer.

To do this, our P4 target architecture will need to support events as well as a new type of extern. An extern is an element whose functionality is not described in P4, and provides an interface for P4 programs to interact with it. This is the mechanism through which an architecture can expose stateful operations to P4 programmers. Our target event-driven architecture will support a new

Table 6.1: Set of useful data-plane events to support in an event-driven packet processing architecture.

Data-Plane Events	
Ingress Packet	Buffer Overflow
Egress Packet	Buffer Underflow
Recirculated Packet	Timer Expiration
Generated Packet	Control-Plane Triggered
Packet Transmitted	Link Status Change
Buffer Enqueue	User Event
Buffer Dequeue	

type of extern called `shared_register` to allow event processing threads to share state.

The user's program `microburst.p4` (shown below), instantiates one of these new extern objects to track the buffer occupancy on a per-flow basis (`bufSize_reg`). The `bufSize_reg` should be allocated with enough entries to track state for every flow that has at least one packet in the buffer.[1]

When a packet arrives, the ingress packet event processing thread computes the packet's flow ID by hashing the IP source and destination addresses, and initializes the packet's metadata so that it can carry the enqueue and dequeue events through the pipeline. Next, the ingress logic reads the flow's buffer occupancy and checks if it exceeds a pre-configured threshold to determine if the flow is a microburst culprit. Upon successful detection of a microburst culprit, the program may then decide to take corrective action such as dropping the packet, lowering its scheduling priority, or notifying a controller.

```
// microburst.p4
shared_register<bit<32>>(NUM_REGS) bufSize_reg;

// Ingress Packet Event Logic
control Ingress(/* hdrs and metadata */) {
  bit<32> bufSize;
  bit<32> flowID;
  apply {
    // compute flowID
    hash(hdr.ip.src ++ hdr.ip.dst, flowID);
    // initialize enq & deq metadata for this pkt
    enq_meta.flowID = flowID;
    enq_meta.pkt_len = meta.pkt_len;
    deq_meta.flowID = flowID;
    deq_meta.pkt_len = meta.pkt_len;
    // read buffer occupancy of this flow
    bufSize_reg.read(flowID, bufSize);
    // detect microburst
    if (bufSize > FLOW_THRESH) { /* microburst culprit! */ }
  }
}
```

The user also needs to implement event handling logic for enqueue and dequeue events to update the per-flow buffer occupancy state. The enqueue logic increments the appropriate entry in the `bufSize_reg` by the length of the enqueued packet, while the dequeue logic decrements this state by the length of the packet that was just removed from the buffer. Here we show how the enqueue event handling logic can be implemented, the dequeue event handling logic is very similar.

```
// Enqueue Event Logic
control Enqueue(inout enq_data_t meta) {
  bit<32> bufSize;
  apply {
```

[1]If needed, a count-min-sketch data structure can be used to reduce state requirements even further.

```
    // increment buffer occupancy of this flow
    bufSize_reg.read(meta.flowID, bufSize);
    bufSize = bufSize + meta.pkt_len;
    bufSize_reg.write(meta.flowID, bufSize);
  }
}
```

Perhaps the biggest benefit to the P4 programmer is that event handling logic can now be expressed in separate threads of execution with shared state. The issues surrounding such state are considered in Section 6.3.

6.2 Event-Driven Applications

Table 6.2 summarizes five classes of applications that we believe will greatly benefit from event-driven programming:

Congestion Aware Forwarding applications base their forwarding decisions on recent congestion signals. We can derive congestion signals, such as (per-active-flow) queue occupancy, link utilization, and packet loss from the enqueue, dequeue, and buffer overflow events. This allows for variants of ECN marking, with packets carrying multiple bits rather than just one, to communicate queue occupancy along the path, or just the maximum queue occupancy at the bottleneck. If the programmer uses timer events as well, congestion signals can be periodically transmitted along various paths in the network, as is the case in HULA [49] or can be used in the ingress pipeline to make priority forwarding decisions, as in NDP [33].

Network Management encompasses a broad range of tasks typically handled by the network control plane. For example, re-routing traffic when links fail usually requires the control plane to detect the failure, re-route the affected flows, and potentially migrate data-plane state from a flow's old path to its new one. By introducing link status change events, the data plane can immediately respond to link failures, autonomously re-route affected flows and migrate data-plane state. This

Table 6.2: Various application classes that can benefit from event-driven programming.

Application Classes	Examples	Events Used
Congestion Aware Forwarding	Load Balancing [49, 3], Congestion Control [33]	Enqueue, Dequeue, Buffer Overflow, Timer
Network Management	Neighbor/Link/Tunnel Failure Detection, Data-plane State Migration [62], Fast Re-Route [85]	Timer, Link Status
Network Monitoring	Sketches [18, 58], Time Window Functions, Microburst Detection [15], INT [52]	Timer, Enqueue, Dequeue, Buffer Overflow
Traffic Management	AQM [61, 75], Policing, Packet Scheduling [88]	Enqueue, Dequeue, Buffer Overflow/Underflow, Timer
In-Network Computing	Coordination [41], Caching [42]	Timer, Link Status

makes it much easier to implement Fast Re-Route (FRR) [85] and swing-state [62]. Furthermore, timer events allow data-planes to reliably and quickly probe and detect failed neighbors and tunnels.

Network Monitoring with extremely fine-grain measurements made possible by In-band Network Telemetry (INT) [52] is becoming increasingly popular. One challenge with INT is the potentially huge volume of measurement data, which might overwhelm a software-based logging and analysis system. But if we can expose event-driven programming to the programmer, data-plane applications can analyze, pre-process and reduce the amount of data reports, using filters and watchlists. For example, data planes can use timer events to aggregate congestion information (e.g. queue size, packet loss, or active flow count) and only report anomalous events to the monitoring system periodically. Furthermore, given it is now easy to write programs using enqueue and dequeue events, applications such as microburst detection are now much simpler to write than before [15].

Traffic Management functions such as active queue management (AQM), policing, and packet scheduling are challenging to implement in P4 today, but can be enabled by an event-driven programming model. AQM algorithms, such as RED [29], AFD [75], FRED [61], and PIE [76], need to monitor and manage the packet queues, and need access to several congestion signals in the ingress pipeline. These include: current queue occupancy, queue service rate, queueing delay, packet loss volume, rate of change of the queue size, per-active-flow queue occupancy, and number of active flows. Event-driven programming gives the user access to all of these congestion signals (and more). Thus, AQM is a natural use case of this approach, and was one of the motivating applications for our work. Similarly, policing often requires a leaky token bucket meter [34]. While baseline PISA architectures might expose fixed-function meters to P4 programmers as primitive elements [74], if we use timer events, token bucket meters can be constructed from simple registers. This approach allows data-plane developers to build and customize their own policing algorithms. Taking this one step further, we can construct a complete, programmable packet scheduler using our event-driven model in combination with the recently proposed Push-In-First-Out (PIFO) queue [88].

In-Network Computing became a hot topic once researchers realized that programmable data planes can be used to accelerate some end-host applications. For example, NetCache [42] demonstrated improvements in throughput and tail-latency of key-value storage systems by caching hot items within a P4-based data-plane. Timer events allows the programmer to write more sophisticated cache replacement policies, such as approximate least-recently-used (LRU), entirely in the data-plane. Timer events can also be used to quickly clear all NetCache statistics, which, as the authors point out, would allow the cache to more rapidly react to workload changes. Link status change events enable coordination services, such as NetChain [41], to quickly react to network failures.

Overall, we have found that P4 programs for a wide range of applications can be simplified using the event-driven model. We conclude (perhaps unsurprisingly with hindsight) that *network algorithms* *are inherently event-driven.*

6.3 Global vs Distributed State

One of the most important design decisions, when building an event-driven data plane, is how state is shared (or not) among different processing elements. If state is private and local to a pipeline stage, we need a way to share state between stages, potentially maintaining multiple copies. Things get more complicated when a device has multiple independent pipelines (e.g. Tofino has four independent pipelines). Deciding how state is shared turns out to be a key design decision.

The answer depends on the line rate. Lower line rate devices (e.g. a WiFi AP or an end host NIC) can use multi-ported memory to directly implement the logical event processing pipelines described in Section 6.1 as separate physical pipelines, each with a dedicated read/write port to global shared state. The memory would need as many ports as the number of event processing threads that access the state.

For high line rate devices, where multi-ported memory is impractical to implement due to the high silicon area cost, we require a different approach. For these devices, we can merge the logically separate event processing pipelines into a single physical pipeline so that state is local to a single physical pipeline stage as in the baseline PISA model. Metadata, created by enqueue and dequeue events, propagates through the pipeline (on its own, or alongside arriving packets), allowing processing to proceed at line rate.

While this model is conceptually simple, we need to make sure the pipeline is wide enough to carry all the events, and at the same time, be able to handle all the required stateful operations. For example, suppose we write a P4 program to compute queue sizes. On a single clock cycle, an enqueue event wants to increment the size of queue 0, a dequeue event wants to decrement the size of queue 1, and an ingress packet event wants to read the size of queue 2 in order to make a forwarding decision. Is it possible to support all of these memory operations simultaneously without resorting to multi-ported memory?

Rather than use multi-ported memory we can instead use multiple single-ported register arrays that are suitably coordinated. Packet event read-modify-write operations always operate on the main register that maintains the algorithmic state, the queue size in our example. The read-modify-write operations for enqueue and dequeue events are aggregated in separate register arrays, in the same or potentially a different pipeline stage. During idle clock cycles when there is spare memory bandwidth available, the aggregated operations are applied to the main register that maintains the algorithmic state. Idle clock cycles occur when the workload contains larger than minimum size packets or when the PISA pipeline is configured to run faster than line rate, which is typical in modern switch chips [93]. Figure 6.3 depicts how this mechanism can be used to process enqueue, dequeue, and packet events to maintain queue sizes.

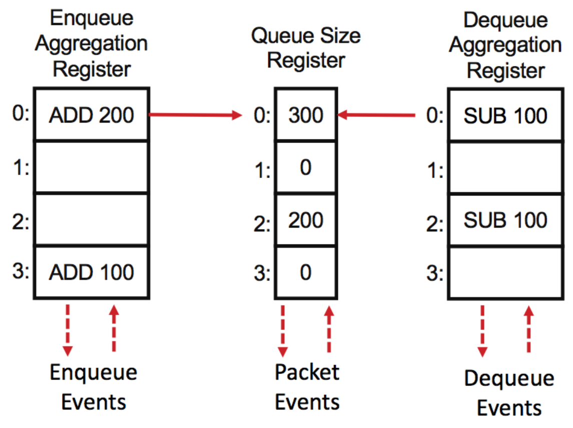

Figure 6.3: Updating algorithmic state across multiple pipeline stages. The goal: keep the algorithmic state (queue size) up to date. Low-priority enqueue and dequeue events are aggregated in separate register arrays, then applied to the main register when memory bandwidth is available.

Stale state. It is important to note that whenever state is distributed across pipeline stages, the algorithmic state will sometimes be stale because of the time it takes state to propagate through a pipeline, or from one pipeline to another. One redeeming feature is that staleness is bounded if the pipeline runs slightly faster than the line rate (as is typical). So, while the state may be temporarily imprecise, the resulting algorithm has well-defined behavior. For example, an application detecting heavy hitters might detect a flow a few nanoseconds late, which is unlikely to matter. On the other hand, some applications require more care (e.g. for consensus algorithms) and the programmer needs to be aware of the staleness. If needed, staleness can be reduced by freeing up processing capacity in the pipeline, for example by not using some of the external ports. This means there is more capacity available to carry metadata from one stage to another, to update algorithmic state. It also opens up another design trade-off: packet processing bandwidth versus accuracy of the data-plane algorithm. This trade-off closely resembles the one provided by sketch algorithms: switch memory versus accuracy of the data-plane algorithm.

When distributing state between stages, we also need to consider how memory accesses are scheduled, depending on which events are the most important and urgent, and whether priorities are assigned by the programmer, the compiler, or the hardware. We plan to address these questions in future work.

6.4 Hardware Feasibility

SUME Event Switch Architecture. To demonstrate that our event-driven architecture is feasible to implement in hardware while processing packets at full line rate, we developed a prototype on the

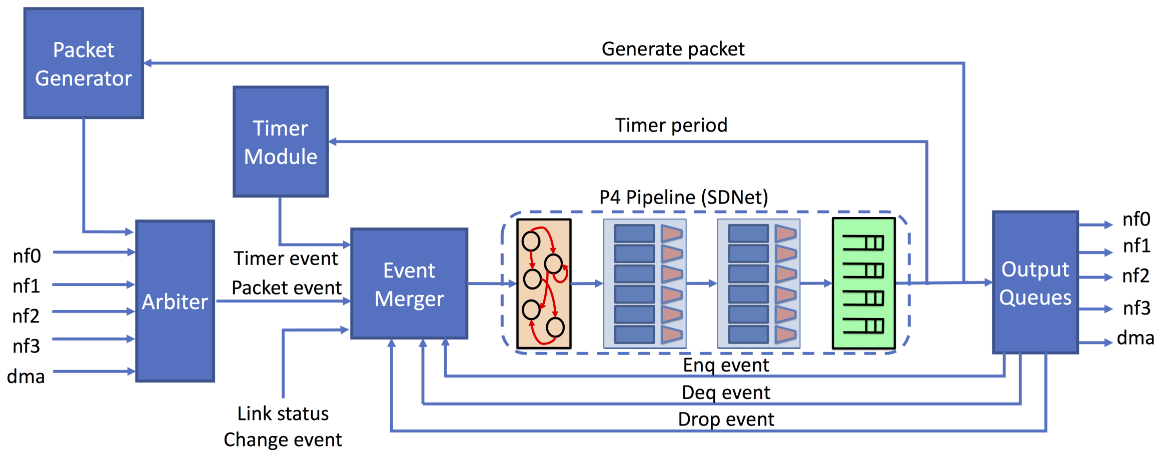

Figure 6.4: The SUME Event Switch architecture implemented on the NetFPGA SUME platform.

NetFPGA SUME platform using the P4→NetFPGA tools [38]. Our prototype, which we call the *SUME Event Switch* supports regular P4 packet events, plus enqueue, dequeue, and drop events, timer events, link status change events, and a configurable packet generator. Figure 6.4 shows a block diagram. The Event Merger is responsible for gathering all new events and placing them into metadata that flows through the pipeline. If there are no ingress packets for the metadata to piggyback onto, the Event Merger generates an empty packet, attaches the event metadata and injects it into the P4 pipeline. The pipeline functions themselves are described in P4, and compiled using Xilinx SDNet [99] to run on the FPGA.

The event handling is very efficient, requiring relatively few FPGA resources. Table 6.3 shows that on a Xilinx Virtex-7 FPGA, the event logic consumes at most 2% additional resources.

In practice. We teach a graduate networking class at Stanford in which students build the hardware and software components of an Internet router; then extend it to add new features of their own choosing. The class uses P4 to define the forwarding behavior. In 2019, the students built their projects on the SUME Event Switch and implemented several different data-plane applications. We highlight a few here.

Liveness Monitoring in the Data Plane. The event-driven programming model was used to

Table 6.3: The cost of adding support for events in the SUME Event Switch architecture. The increase in resources are shown as a percentage of the total resources available in a Xilinx Virtex-7 FPGA.

FPGA Resource	**% Increase**
Lookup Tables	0.5
Flip Flops	0.4
Block RAM	2.0

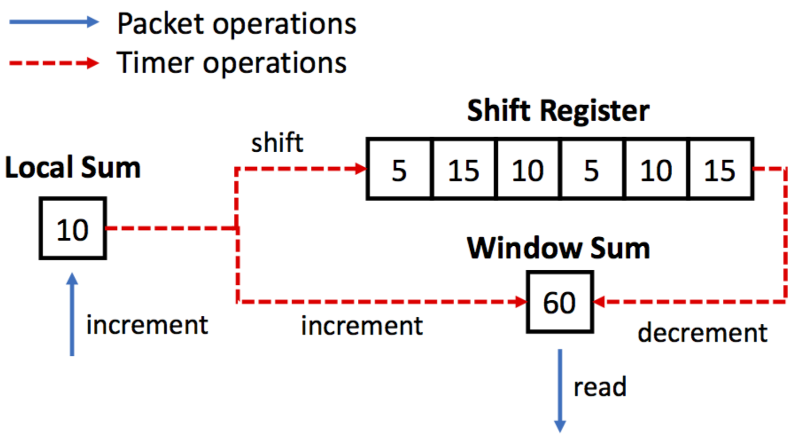

Figure 6.5: A diagram of the time window data-plane primitive being used to compute a sum over a sliding window of time.

implement a protocol in the data plane that periodically checks the liveness of neighboring network devices by transmitting echo request packets and waiting for replies. Upon detecting failure of a neighbor, the data plane transmits notifications to a central monitor, with no intervention by the control plane.

Time-Windowed Network Measurement. A common data-plane task is to compute a function of a signal, such as a moving average, over a sliding window of time. This sort of operation is very natural to implement using timer events. One student group demonstrated how to use timer events in conjunction with a simple shift register to accurately measure flow rates in the data plane.

Computing Congestion Signals. In this project, the students implemented a simple AQM policy to enforce flow-level fairness, similar to FRED [61]. Enqueue and dequeue events were used to compute congestion signals (total buffer occupancy, per-active-flow buffer occupancy, and active flow count). Timer events periodically sample the buffer occupancy and send a report to a monitor which maintains a time series of the buffer occupancy.

Fast Re-Route. Link status change events make it easy to implement fast re-route policies in the data-plane. When a link failure is detected, the prototype updates its forwarding decisions immediately to send packets along a backup route.

Community. We have contributed the SUME Event Switch architecture to the P4→NetFPGA project[2] so that the community can experiment with their own event-driven programs on real hardware.

[2]GitHub Wiki: https://github.com/NetFPGA/P4-NetFPGA-public/wiki

6.5 Relation to Modern PISA Devices

Today's P4 programmable devices expose a programming model that resembles the one provided by baseline PISA architectures. That is, the only events that are explicitly exposed to the programmer are packet events. Some P4 targets can indirectly support other events as well. For example, Tofino [93] contains a configurable packet generator which the control-plane can configure to generate periodic packets and hence emulate timer events. Tofino also supports packet recirculation, which can emulate dequeue events that trigger the ingress pipeline. However, supporting all of the events listed in Table 6.1 requires changes to existing hardware.

6.6 Related Work

Our proposed Event-Driven PISA architecture builds directly upon the baseline PISA architecture described in [14]. The authors of dRMT [17] propose to modify the baseline PISA architecture by disaggregating table memory and compute resources. They demonstrate that this approach leads to higher resource utilization as well as more flexibility when applying match-action tables. However, the programming model that is used to configure their dRMT architecture is identical to the one provided by the baseline PISA architecture. Therefore, our event-driven PISA architecture is able to support the same programs as dRMT.

There has been a number of recent efforts to build new abstractions for programming the network data-plane. Domino [87] introduced the notion of packet transactions which are stateful read-modify-write operations that are performed atomically per packet. This per-packet atomic constraint enables Domino to provide consistency guarantees to data-plane programs. However, it also significantly limits the complexity of the read-modify-write operations. As a result, the authors of FlowBlaze [77] propose a new abstraction which distinguishes between global state and flow state. Operations on flow state can be more complex because they only need to complete atomically between packets of the same flow, rather than on a per-packet basis. Both of these proposals only consider single threaded data-plane programs. In an event-driven programming model there can be many event processing threads that share the same state. Defining a consistency model for multi-threaded data-plane programs remains an area of future work.

6.7 Discussion

All network algorithms are event-driven. As we have shown, P4 is actually a domain specific language suitable for expressing line rate event processing, not just packet processing. The set of data-plane algorithms that can be expressed in today's data-plane programming model is a strict subset of what can be expressed using our more general event-driven programming model. Events give data-plane programmers much more flexibility, enabling them to implement algorithms that derive and use congestion signals, update state multiple times and independently of packet arrivals and departures, and even compute functions over windows of time much more naturally. However, switch data-plane algorithms are not the only network algorithms that are event-driven. In particular, protocols running at end hosts are also event-driven. For example, the state machine for a simple reliable delivery protocol is driven by packet arrivals, packet departures, and timeout events. Chapter 3 describes how we can build an event-driven PISA architecture that enables line rate programmable transport logic in NIC hardware. Since most network algorithms are event-driven, we believe that data-plane architectures should be as well. This approach has the potential to offload much more functionality to high-speed data-plane hardware.

Chapter 7

Conclusions

To conclude this dissertation we outline areas for future work and wrap up with some final thoughts.

7.1 Future Work

There are a number of areas of future work, described below.

Beyond the Rocket core. Our prototype is based on the simple 5-stage, in-order, RISC-V Rocket core. As described in Chapter 4, we only needed to make very minor modifications to the core in order to support the nanoPU's register file network interface and hardware thread scheduler. The processors used in modern data centers are significantly more sophisticated than the Rocket core; they are typically much deeper, out-of-order, superscalar processors. Porting the nanoPU to such processors would require careful consideration to ensure that message words are delivered from `netRX` to applications in the correct order.

Improved transport support. The nanoPU's programmable transport architecture currently supports NDP [33] and we are in the process of adding support for additional protocols as well, such as Homa [66]. Both NDP and Homa are designed for fully provisioned networks and thus assume that congestion only occurs at last hop (i.e., the receiver downlink) and not within the network core. As such, both congestion control algorithms use a fixed window size of one bandwidth delay product.[1] We plan to explore how to enable dynamic window size adjustments by incorporating ideas from HPCC [59] and Swift [55].

As described in Chapter 3, stateful operations, called atoms, are built into the design as primitives. Transport logic must then be mapped onto these atoms. As we add support for additional transport protocols, we will also attempt to identify the ideal set of atoms to include in the design.

[1]Strictly speaking, Homa includes an optimization to handle self-inflicted incasts, which allows a sender to reduce its window size.

Additional programmability. Beyond the transport logic, it is also interesting to consider how to make other aspects of the nanoPU programmable.

There are a number of places in the architecture where scheduling decisions are made, such as when packets are selected to be transmitted onto the network link. The PIFO [88] is a promising abstraction for programmable packet scheduling and it would be natural to incorporate into the nanoPU.

The nanoPU's hardware thread scheduler, which currently implements the bounded strict priority policy (Section 3.3), is another prime opportunity to include programmable logic. For instance, some system administrators may be inclined to use a hierarchical scheduling policy in which threads that are assigned the same priority are given a fair share of CPU time. Or perhaps thread scheduling decisions should also take into consideration other external factors, such as disk interrupts.

Our choice to use the JBSQ core selection policy in the nanoPU is based upon the assumption that any core can process any message with equal cost. While this may be true for nanoRequests, for memory intensive applications, it may be important to take into consideration cache affinity when assigning messages to cores. Doing so would require more sophisticated core selection logic.

We have come across a number of applications that would benefit from being able to customize how packets are reassembled into messages before being delivered to the core. Aggregating many small messages into a few large ones enables applications to reduce message processing overheads and thus improve throughput. Perhaps we can gain inspiration from the DPDK generic receive offload library [25] and allow data plane developers to write custom message reassembly functions.

Does the P4 language provide the right level of abstraction to enable programmability of these diverse tasks? We believe that each of these are interesting areas for future research.

RPC serialization & deserialization. NanoRequests are wire-format messages and thus there is no need to perform application-level serialization or deserialization (ser/des). If we are to utilize the nanoPU fast path for traditional RPCs then the NIC would require additional support for message ser/des. Optimus Prime [78] has already demonstrated that it is feasible to implement the necessary data transformation operations in line rate hardware. It would be interesting to explore how to incorporate these ideas into the nanoPU architecture.

Facilitating application development is one of the key challenges that will need to be overcome in order to enable widespread adoption of the nanoPU.

Host software needs to cope with the fact that the nanoPU's transport layer provides the abstraction of reliable, one-way message delivery, rather than the more traditional abstraction of reliable, bi-directional, byte streams. Fortunately, these changes can be hidden from applications by porting popular RPC libraries, such as gRPC [32], to use the new transport API.

However, applications will need to be modified in order to most efficiently utilize the nanoPU's register file network interface. It is unlikely that these optimizations can be hidden within an RPC library because it requires applications to carefully control the order in which they process the bytes of each message. However, it may be possible to develop tools that allow developers to analyze their applications and determine optimal compute patterns and message formats.

Additionally, in order to provide bounded RPC response times we also need to enable applications to process messages within a short, bounded amount of time. Processing time variations caused by cache and TLB misses make this a challenging task; although, perhaps we will be able to leverage techniques used in real-time embedded systems [57].

Unified fast and traditional paths. In our description of the nanoPU, the fast path for nanoRequests and the traditional path are independent of each other. It is interesting to consider how best to synthesize the two paths, to get the best of both. For example, an application might benefit from part of a request passing through the register file, with the remainder going through the DMA path; or by initiating low latency RDMA operations through the register file. Also, the hardware thread scheduler (HTS) might be extended to schedule all message processing threads, even those using the traditional path.

nanoPU as a DSA. Another approach is to take the nanoPU's fast path design to the extreme and build a domain-specific architecture explicitly for nanoRequests. A cluster of hundreds of thousands of devices might be harnessed to accelerate finer-grain distributed applications than is possible today.

The prospect of nanoservice applications. The nanoPU is designed to enable applications that are highly parallelizable into extremely fine-grained tasks that have very small, cache-resident working sets and exchange tons of small RPC messages. We call this class of applications *nanoservices*. We believe that some modern applications can be refactored as nanoservices, and in doing so, achieve significant performance improvements by harnessing orders of magnitude more cores in parallel than is practical on modern systems. As a starting point, we are looking at applications from the HPC community (e.g. N-body simulations) as well as from the computer graphics community (e.g. massively distributed ray tracing). We hope that the availability of the nanoPU will encourage researchers to rethink the way modern distributed applications are designed.

Silicon implementation. The nanoPU is designed to run as an ASIC, however our current prototype runs on an FPGA. Our FPGA prototype has allowed us to verify the correctness of our RTL as well as simulate the full design orders of magnitude faster than is possible with cycle-accurate software simulators. The next step is to demonstrate the nanoPU using an ASIC prototype. Building an ASIC from scratch can take a few years and requires a large team of dedicated engineers. Fortunately, to build a nanoPU ASIC, we do not need to start from scratch. The Rocket core has been taped out over a dozen times and the RISC-V community provides a number of open source tools to help facilitate the ASIC development process. Therefore, we have a very solid starting point.

7.2 Final Thoughts

Today's CPUs are optimized for load-store operations to and from memory. Memory data is treated as a first-class citizen. But modern workloads frequently process huge numbers of small RPCs. Rather than burden RPC messages with traversing a hierarchy optimized for data sitting in memory, we propose providing them with a new optimized fast path, inserting them directly into the heart of the CPU, bypassing the unnecessary complications of caches, PCIe and address translation. Hence, we aim to elevate network data to the same importance as memory data.

As datacenter applications continue to scale out, with one request fanning out to generate many more, we must find ways to minimize not only the communication overhead, but also the *tail* response time. Long tail response times are inherently caused by resource contention (e.g., shared CPU cores, cache space, and memory and network bandwidths). By moving key scheduling decisions into hardware (i.e., congestion control, core selection, and thread scheduling), these resources can be scheduled extremely efficiently and predictably, leading to lower tail response times.

If future cloud providers can provide bounded, end-to-end RPC response times for very small nanoRequests, on shared servers also carrying regular workloads, we will likely see much bigger distributed applications based on finer grain parallelism. Our work helps to address part of the problem: bounding the RPC response time once the request arrives at the NIC. If coupled with efforts to bound network latency, it might complete the end-to-end story. We hope our results will encourage other researchers to push these ideas further.

Bibliography

[1] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. Sp-pifo: approximating push-in first-out behaviors using strict-priority queues. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 59–76, 2020.

[2] Mohammad Alian and Nam Sung Kim. Netdimm: Low-latency near-memory network interface architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 699–711, 2019.

[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM CCR*, volume 44, pages 503–514. ACM, 2014.

[4] Amazon ec2 f1 instances. https://aws.amazon.com/ec2/instance-types/f1/. Accessed on 2020-08-10.

[5] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *ACM SoCC*, 2018.

[6] 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, 2020.

[7] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[8] Aws nitro system. https://aws.amazon.com/ec2/nitro/. Accessed on 2020-12-10.

[9] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala

embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[10] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, 2018.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[12] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*, page 17, 2012.

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

[14] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[15] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. Catching the microburst culprits with snappy. In *Proc. of the Workshop on Self-Driving Networks*, pages 22–28. ACM, 2018.

[16] 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*, pages 1–17, 2016.

[17] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. drmt: Disaggregated programmable switching. In *Proc. of ACM Sigcomm*, pages 1–14. ACM, 2017.

[18] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[19] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[20] William J Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, and John Keen. The j-machine: A fine grain concurrent computer. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE MICROSYSTEMS RESEARCH CENTER, 1989.

[21] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[22] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM CCR*, 46(2):18–24, 2016.

[23] Intel corporation. intel data direct i/o technology (intel ddio): A primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf. Accessed on 2020-08-17.

[24] DPDK: Data Plane Development Kit. https://www.dpdk.org/. Accessed on 2020-12-04.

[25] DPDK Generic Receive Offload Library. https://doc.dpdk.org/guides/prog_guide/generic_receive_offload_lib.html. Accessed on 2020-12-10.

[26] eBPF – extended Berkeley Packet Filter. https://prototype-kernel.readthedocs.io/en/latest/bpf/. Accessed on 2020-12-08.

[27] Cisco Nexus X100 SmartNIC K3P-Q Data Sheet. https://www.cisco.com/c/en/us/products/collateral/interfaces-modules/nexus-smartnic/datasheet-c78-743828.html. Accessed on 2020-12-01.

[28] 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)*, pages 673–689, 2020.

[29] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking*, (4):397–413, 1993.

[30] Sadjad Fouladi, Dan Iter, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. A thunk to remember: make-j1000 (and other jobs) on functions-as-a-service infrastructure, 2017.

[31] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-latency Video Processing Using Thousands of Tiny Threads. In *USENIX NSDI*, 2017.

[32] gRPC. https://grpc.io/. Accessed on 2020-12-10.

[33] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[34] Juha Heinanen and Roch Guérin. A single rate three color marker. Technical report, 1999.

[35] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 54–66, 2018.

[36] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX NSDI Symposium*, pages 161–176, 2019.

[37] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 133–140, 2019.

[38] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4-netfpga workflow for line-rate packet processing. In *Proc. of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–9. ACM, 2019.

[39] 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*, pages 52–59, 2019.

[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, 2014.

[41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[42] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2017.

[43] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *ACM SoCC*, 2017.

[44] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[45] 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, 2019.

[46] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 295–306, 2014.

[47] 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, 2016.

[48] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2018.

[49] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. of the Symposium on SDN Research*, page 10. ACM, 2016.

[50] Clinton Kelly, Virantha Ekanayake, and Rajit Manohar. Snap: A sensor-network asynchronous processor. In *Ninth International Symposium on Asynchronous Circuits and Systems, 2003. Proceedings.*, pages 24–33. IEEE, 2003.

[51] Richard E Kessler and James L Schwarzmeier. CRAY T3D: A New Dimension for Cray Research. In *Digest of Papers. COMPCON Spring*, pages 176–182. IEEE, 1993.

[52] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[53] Youngbin Kim and Jimmy Lin. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455. IEEE, 2018.

[54] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 863–880, 2019.

[55] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. 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*, pages 514–528, 2020.

[56] Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou. Dagger: Towards efficient rpcs in cloud microservices with near-memory reconfigurable nics. *IEEE Computer Architecture Letters*, 19(2):134–138, 2020.

[57] Qing Li and Caroline Yao. *Real-time concepts for embedded systems.* CRC press, 2003.

[58] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. In *13th USENIX NSDI Symposium*, pages 311–324, 2016.

[59] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. 2019.

[60] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[61] Dong Lin and Robert Morris. Dynamics of random early detection. In *ACM SIGCOMM CCR*, volume 27, pages 127–137. ACM, 1997.

[62] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. Swing state: Consistent updates for stateful and programmable data planes. In *Proc. of the Symposium on SDN Research*, pages 115–121. ACM, 2017.

[63] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[64] Mellanox bluefield-2. https://www.mellanox.com/products/bluefield2-overview. Accessed on 2020-12-10.

[65] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. of the ACM Sigcomm Conference*, pages 15–28. ACM, 2017.

[66] 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)*, pages 221–235, 2018.

[67] Naples dsc-100 distributed services card. https://www.pensando.io/assets/documents/Naples_100_ProductBrief-10-2019.pdf. Accessed on 2020-12-10.

[68] 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*, pages 327–341, 2018.

[69] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. *ACM SIGPLAN Notices*, 49(4):3–18, 2014.

[70] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[71] Options for Code Generation Conventions. https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html#Code-Gen-Options. Accessed on 2020-11-11.

[72] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[73] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), August 2015.

[74] P4.org. P4 portable switch architecture (psa), 2018.

[75] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communication Review*, 33(2):23–39, 2003.

[76] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. Pie: A lightweight control scheme to address the

bufferbloat problem. In *IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, pages 148–155. IEEE, 2013.

[77] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. Flowblaze: Stateful packet processing in hardware. In *16th USENIX NSDI Symposium*, 2019.

[78] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*, 2020.

[79] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 17)*, pages 325–341, 2017.

[80] Google protocol buffers. `https://developers.google.com/protocol-buffers`. Accessed on 2020-12-08.

[81] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 193–206, 2019.

[82] raft GitHub. `https://github.com/willemt/raft`. Accessed on 2020-08-17.

[83] Redis. `https://redis.io/`. Accessed on 2020-08-12.

[84] Rocket-chip github. `https://github.com/chipsalliance/rocket-chip`. Accessed on 2020-08-17.

[85] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. Supporting emerging applications with low-latency failover in p4. 2018.

[86] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 367–379. 2019.

[87] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[88] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown.

Programmable packet scheduling at line rate. In *Proc. of ACM SIGCOMM Conference*, pages 44–57. ACM, 2016.

[89] Akshitha Sriraman and Thomas F Wenisch. μ suite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[90] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The NEBULA rpc-optimized architecture. Technical report, EcoCloud EPFL, 2020.

[91] Sx1036 product brief. https://www.mellanox.com/related-docs/prod_eth_switches/PB_SX1036.pdf. Accessed on 2020-09-12.

[92] Apache thrift. https://thrift.apache.org/. Accessed on 2020-12-08.

[93] Tofino - world's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/. Accessed on 2020-08-17.

[94] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 283–297, 2018.

[95] User level threads. http://pdxplumbers.osuosl.org/2013/ocw//system/presentations/1653/original/LPC%20-%20User%20Threading.pdf. Accessed on 2020-12-08.

[96] Verilator. https://www.veripool.org/wiki/verilator. Accessed on 2020-01-29.

[97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[98] Wikipedia:database download. https://en.wikipedia.org/wiki/Wikipedia:Database_download. Accessed on 2020-12-08.

[99] Xilinx. Sdnet, 2018.

[100] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review (CCR)*, 45(4):523–536, 2015.