The nanoPU: A Nanosecond Network Stack for Datacenters

Stephen Ibanez, Alex Mallery, Serhat Arslan, and Theo Jepsen, *Stanford University;* Muhammad Shahbaz, *Purdue University;* Changhoon Kim and Nick McKeown, *Stanford University*

https://www.usenix.org/conference/osdi21/presentation/ibanez

This paper is included in the Proceedings of the
15th USENIX Symposium on Operating Systems
Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

The nanoPU: A Nanosecond Network Stack for Datacenters

Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen,
Muhammad Shahbaz⋆, Changhoon Kim, and Nick McKeown
Stanford University ⋆*Purdue University*

Abstract

We present the nanoPU, a new NIC-CPU co-design to accelerate an increasingly pervasive class of datacenter applications: those that utilize many small Remote Procedure Calls (RPCs) with very short (μs-scale) processing times. The novel aspect of the nanoPU is the design of a *fast path* between the network and applications—bypassing the cache and memory hierarchy, and placing arriving messages directly into the CPU register file. This fast path contains programmable hardware support for low latency transport and congestion control as well as hardware support for efficient load balancing of RPCs to cores. A hardware-accelerated thread scheduler makes sub-nanosecond decisions, leading to high CPU utilization and low tail response time for RPCs.

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 RPC response time through the nanoPU is just 69ns, an order of magnitude quicker than the best-of-breed, low latency, commercial NICs. We demonstrate that the hardware thread scheduler is able to lower RPC tail response time by about 5× while enabling the system to sustain 20% higher load, relative to traditional thread scheduling techniques. We implement and evaluate a suite of applications, including MICA, Raft and Set Algebra for document retrieval; and we demonstrate that the nanoPU can be used as a high performance, programmable alternative for one-sided RDMA operations.

1 Introduction

Today, large online services are typically deployed as multiple tiers of software running in data centers. Tiers communicate with each other using Remote Procedure Calls (RPCs) of varying size and complexity [7, 28, 57]. Some RPCs call upon microservices lasting many milliseconds, while others call remote (serverless) functions, or retrieve a single piece of data and last only a few *micro*seconds. These are important workloads, and so it seems feasible that small messages with microsecond (and possibly nanosecond) service times will become more common in future data centers [7, 28]. For example, it is reported that a large fraction of messages communicated in Facebook data centers are for a single key-value memory reference [4, 7], and a growing number of papers describe fine-grained (typically cache-resident) computation based on very small RPCs [22, 23, 28, 57].

Three main metrics are useful when evaluating an RPC system's performance: (1) the *median response time* (i.e., time from when a client issues an RPC request until it receives a response) for applications invoking many sequential RPCs; (2) the *tail response time* (i.e., the longest or 99th %ile RPC response time) for applications with large fanouts (e.g., map-reduce jobs), because they must wait for all RPCs to complete before continuing [17]; and (3) the *communication overhead* (i.e., the communication-to-computation ratio). When communication overhead is high, it may not be worth farming out the request to a remote CPU at all [57]. We will sometimes need more specific metrics for portions of the processing pipeline, such as the *median wire-to-wire latency*, the time from when the first bit of an RPC request arrives at the server NIC until the last bit of the response departs.

Many authors have proposed exciting ways to accelerate RPCs by reducing the message processing overhead. These include specialized networking stacks, both in software (e.g., DPDK [18], ZygOS [51], Shinjuku [27], and Shenango [49]), and hardware (e.g., RSS [43], RDMA [9], Tonic [2], NeBuLa [57], and Optimus Prime [50]). Each proposal tackles one or more components of the RPC stack (i.e., network transport, congestion control, core selection, thread scheduling, and data marshalling). For example, DPDK removes the memory copying and network transport overhead of an OS and lets a developer handle them manually in user space. ZygOS implements a scheme to efficiently load balance messages across multiple cores. Shenango efficiently shares CPUs among services requiring RPC messages to be processed. eRPC [28] cleverly combines many software techniques to reduce median RPC response times by optimizing for the common case (i.e., small messages with short RPC handlers). These systems have successfully reduced the message-processing overhead from 100s of microseconds to 1–2 microseconds.

NeBuLa [57] is a radical hardware design that tries to further minimize response time by integrating the NIC with the CPU (bypassing PCIe), and dispatching RPC requests *directly into the L1 cache*. The approach effectively reduces the minimum wire-to-wire response time below 100ns.

Put another way, these results suggest that with the right hardware and software optimizations, it is practical and useful to remotely dispatch functions as small as a few microseconds. The goal of our work is to enable even smaller functions, with computation lasting less than 1μs, for which we need to minimize communication overhead. We call these very short RPCs *nanoRequests*.

The nanoPU, presented and evaluated here, is a combined NIC-CPU optimized to process nanoRequests very quickly. When designing nanoPU, we set out to answer two questions.

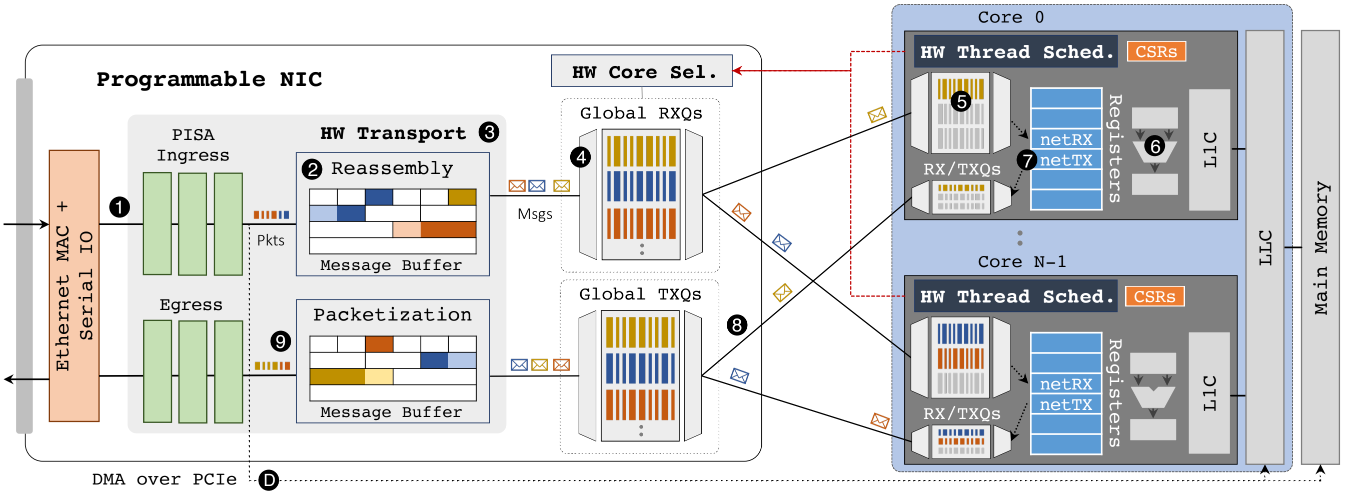

Figure 1: The nanoPU design. The NIC includes ingress and egress PISA pipelines as well as a hardware-terminated transport and a core selector with global RX queues; each CPU core is augmented with a hardware thread scheduler and local RX/TX queues connected directly to the register file.

The first is, **what is the absolute minimum communication overhead we can achieve for processing nanoRequests?** NanoRequests are simply very short-lived RPCs marked by the client and server NICs for special treatment. In nanoPU, nanoRequests follow a new low-overhead path through the NIC, bypassing the OS and the memory-cache hierarchy and arriving directly into running threads' registers. All message reassembly functions, transport and congestion control logic are moved to hardware, as are thread scheduling and core selection decisions. Incoming nanoRequests pass through only hardware before reaching application code. Our nanoPU prototype can deliver an arriving nanoRequest into a running application thread in less than 40ns (less than 15ns if we bypass the Ethernet MAC)—an order of magnitude faster than the fastest commercial NICs [20] and faster than the quickest reported research prototype [57]. For compatibility with existing applications, nanoPU allows all other network traffic (e.g., larger RPCs) to traverse a regular path through a DMA NIC, OS, and memory hierarchy.

Our second question is, **can we minimize tail response time by processing nanoRequests in a deterministic amount of time?** The answer is a qualified yes. Because nanoRequests are processed by a fixed-latency hardware pipeline, if a single-packet request arrives at a waiting core, its thread will always start processing the message in less than 40ns. On the other hand, if the core is busy, or another request is queued ahead, then processing can be delayed. In Section 2.2, we show how our novel hardware thread scheduler can bound the tail response time in this case too, under specific assumptions (e.g., that a nanoRequest can bound its CPU processing time, else its priority is downgraded). We believe nanoPU is the first system to bound the response time of short-lived requests.

In summary, the main contributions of the nanoPU are:

1. The nanoPU's median *wire-to-wire* response time for nanoRequests, from the wire through the header-processing pipeline, transport layer, core selection, and thread scheduling, plus a simple loopback application and back to the wire is just 69ns, an order of magnitude lower latency than the best commercial NICs [20]. Without the MAC and serial I/O, 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 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,[1] and runs on AWS F1 FPGAs using Firesim [31].
4. We evaluate a suite of applications including: the MICA key-value store [38], Raft consensus [47], set algebra and high dimensional search inspired from the μ-Suite benchmark [56].
5. We demonstrate that the nanoPU can be used to implement one-sided RDMA operations with lower latency and more flexibility than state-of-the-art commercial RDMA NICs.

The nanoPU ideas could be deployed in a variety of ways: by adding the low latency path to a conventional CPU, or by designing new RPC-optimized CPUs with only the low-latency path, or by adding the new path to embedded CPUs on smartNICs.

2 The nanoPU Design

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 1 depicts

[1] nanoPU Artifact: https://github.com/l-nic/chipyard/wiki

the key design components. The nanoPU has two independent network paths: (1) the traditional (unmodified) DMA path to/from the host's last-level [16] or L1 cache [57], 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 [27, 49, 51]; however, the OS delegates scheduling of nanoRequest threads to HTS.

To understand the flow of the nanoPU fast path, consider the numbered steps in Figure 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 the destination layer-4 port number in the transport header using a match-action table[2] to decide if the message should be delivered along the fast path. 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 2.3). When a message has 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 2.3). 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 2.2) to alert the core to run the message's thread and place the first word in the `netRX` register (Section 2.1). In ❻, the core processes the data and, if running a server application, will typically generate a response message for the client. The application transmits a message by issuing instructions that write one "word" at a time to the `netTX` register in ❼, where the word size is defined by the size of a CPU register, typically 64-bits (8B). These message words then flow into the global transmit queues in ❽. Messages are split into packets in ❾, before departing through the egress PISA pipeline.

Next, we detail the design of the main, novel 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.

2.1 Thread-Safe Register File Interface

Recent work [45] 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 CPU, to bring packets directly into the cache [12, 46, 57].

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 [45]). 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: A LLC miss takes about 50-100ns to resolve and creates extra traffic on the (shared) DRAM memory bus. DRAM access can be a bottleneck for a multicore CPU, and when congested, memory access times can increase by more than 200% [60]. Furthermore, contention for cache space and DRAM bandwidth is worse at network speeds above 100Gb/s [21].

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.

[2] It is the responsibility of the the host software to configure this table with entries for all nanoRequest processing applications.

2.1.1 How an application uses the interface

The J-Machine [13] first used the register file in 1989 for very low latency inter-core communication, followed by the Cray T3D [33]. 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 [14]. As we show below, our design solves this problem. We believe ours is the first design to add a 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.[3] In addition to the reserved GPRs, a small set of control & status registers (CSRs, described in Section 3.4) are used for the core and NIC hardware to coordinate with each other.

Delimiting messages. Each message that is transmitted and received by an application begins with a fixed 8B "application header". On arriving messages, this header indicates the message length (as well as the source IP address and layer-4 port number), which allows software to identify the end of the message. Similarly, the application header on departing messages contains the message length (along with the destination IP address and layer-4 port number) so that the NIC can detect the end of the outgoing message. The programmable NIC pipeline replaces the application header with the appropriate Ethernet, IP, and transport headers on all transmitted packets.

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 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.[4] This turned out to be a key design choice, simplifying application development on the nanoPU; nanoRequest threads no longer need to read and write messages atomically.

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. 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. 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 1 lists applications that have been ported to efficiently use this new network interface and Section 4 further discusses applications on the nanoPU.

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 (100K rules)	0.90 / 2.20
Othello Player	Searches the Othello state space	0.90 / 1.70 [26]
One-sided RDMA	Performs one-sided RDMA operations in software	0.68 / N/A *

Table 1: Example applications that have been implemented on the nanoPU. These applications use small network messages, 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 wire-to-wire response time at low load. *Measured at client.

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 [52] or Thrift [59]. 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 [50]. In the nanoPU, the memory copy overhead involved in serialization and deserialization is little

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

[4] 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 (Section 5).

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.

2.2 Thread Scheduling in Hardware

Current best practice for low-latency applications is to either (1) pin threads to dedicated cores [18, 51], which is very inefficient when a thread is idle, or (2) devote one core to run a software thread scheduler for the other cores [27, 49].

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 [27, 49], 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-nanoseconds 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.

2.2.1 How the hardware thread scheduler works

Every core contains its own scheduler hardware. 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 lets the nanoPU hardware distinguish between threads and ensure 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 software 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 takes about 50ns to swap a previously idle thread onto the core, measured from the moment its first pending message arrives (Section 3.2).

If the thread to switch to 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 [27], but most of this time is spent making the scheduling decision [62]. 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: $t_p \leq t_n + kt_0 + (k-1)t_c$, where 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.

Writing well-behaved applications, which are able to process all messages within a short, bounded amount of time, is complicated by cache / TLB misses and CPU power management. Our approach so far has been to empirically verify that certain applications are well-behaved. However, we believe that there is substantial opportunity for future research to determine more systematic ways for developers to write well-behaved applications. One approach may be to propose modifications to the memory hierarchy in order to make access latency more predictable. Another approach may be to develop code verification tools to check whether threads meet execution time bounds. The eBPF [19] compiler, for example, is able to verify that a packet processing program will complete eventually; we believe a similar approach can be used to verify completion within a bounded amount of time.

2.3 The nanoPU NIC Pipeline

The NIC portion of the nanoPU fast path consists of two primary components: the programmable transport layer, and the core-selection algorithm. We describe each in turn.

Programmable transport layer. The nanoPU provides nanoRequest threads a *reliable one-way message* service. To be fast enough, the transport layer needs to be terminated in hardware in the NIC. For example, our prototype hardware NDP implementation (Section 3.3) 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 [2].

We only have space to give a high level overview of our programmable transport layer, leaving details to a follow-on paper. At the heart of our programmable transport layer is an event-driven, P4-programmable PISA pipeline [10, 25]. The pipeline can be programmed to do normal header processing, such as VXLAN, overlay tunnels, and telemetry data [35]. We enhance it for reliable message processing, including congestion control, and have programmed it to implement the NDP [24] and Homa [42] low-latency message protocols. Network operators can program custom message protocols tailored to their specific workloads.

Low-latency, message-oriented transport protocols are well-suited to hardware, compared to connection-oriented, reliable byte-stream protocols such as TCP. The NIC only needs to maintain a small amount of state for partially delivered messages. For example, our NDP implementation, beyond storing the actual message, keeps a per-message bitmap of received packets, and a few bytes for congestion control. This allows our design to be limited only by the number of outstanding messages, rather than the number of open connections, allowing large scale, highly-distributed applications across thousands of servers.

The transport layer (Figure 1) contains buffers to convert between the unreliable IP datagram domain and the reliable message domain. Outbound messages pass through a packetization buffer to split them into datagrams, which may need to be retransmitted out of order due to drops in the network. Inbound datagrams are placed into a reassembly buffer, re-ordering them as needed to prepare them for delivery to a CPU core.

Selecting a CPU core. If the NIC randomly sends messages to cores, some messages will inevitably sit in a queue waiting for a busy core while another core sits idle. Our NIC therefore implements a core-selection algorithm in hardware. Inspired by NeBuLa [57], our NIC load balances nanoRequest messages across cores using the Join-Bounded-Shortest-Queue or JBSQ(n) algorithm [36].

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 is impractical to implement efficiently at these speeds.

Our nanoPU prototype implements a JBSQ(2) load balancer in hardware *per application*. The NIC is connected to each core using dedicated wires, and 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 Section 5.2.3.

3 Our nanoPU Implementation

We designed a prototype quad-core nanoPU based on the open-source RISC-V Rocket core [54]. A block diagram of our prototype is shown in Figure 2.

Our prototype extends the open-source RISC-V Rocket-Chip SoC generator [3], adding 4,300 lines of Chisel [6] 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. Everything shown in Figure 2, except the MAC and Serial IO, is included in our prototype and is available as an open-source, reproducible artifact.[5] 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 Section 5, our prototype runs on AWS F1 FPGA instances, using the Firesim [31] framework. Our prototype adds about 15% more logic LUTs to an otherwise unmodified RISC-V Rocket core with a traditional DMA NIC.

3.1 RISC-V Register File Network 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 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

[5] nanoPU Artifact: https://github.com/l-nic/chipyard/wiki

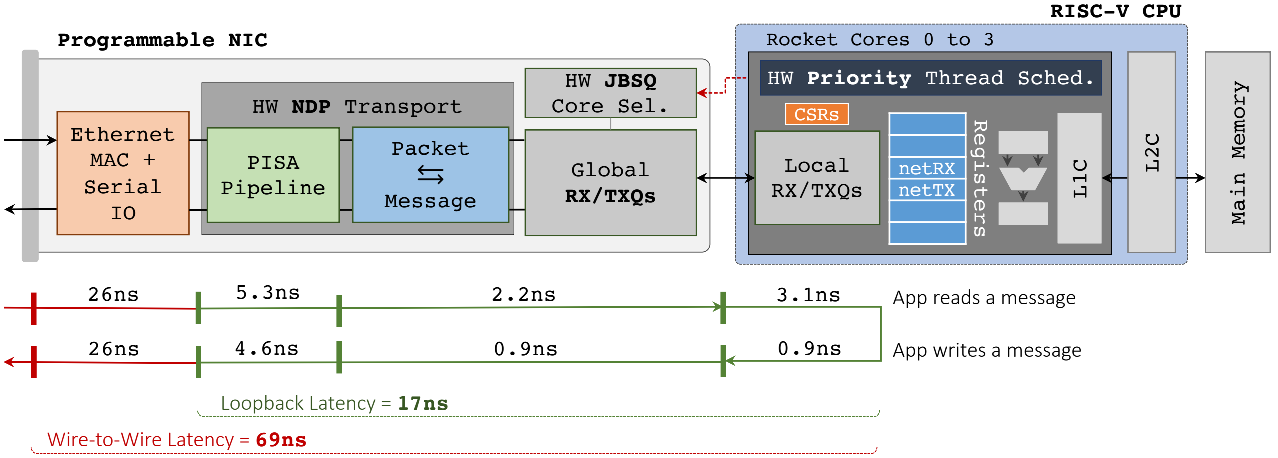

Figure 2: Our nanoPU prototype latency breakdown. Total wire-to-wire latency for an 8B message (72B packet) is 69ns.

temporary storage. Fortunately, `gcc` makes it easy to reserve registers via command-line options [48].

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.

3.2 Bounded Thread Scheduling in Hardware

The nanoPU core implements thread scheduling in hardware, as described in Section 2.2. The number of threads that can run on each core is primarily determined by the amount of buffering available for the local RX/TX queues. In order to implement the JBSQ(2) core selection policy, as described in Section 2.3, the local RX queue for each thread must be able to hold at least two maximum size messages. We use a maximum message size of 2KB (two packets)[6] and allocate 16KB of buffer for the local RX queues. Therefore, the prototype supports 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 HTS 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.

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.

3.3 Prototype NIC Pipeline

The NIC portion of the nanoPU fast path consists of the programmable transport module and the core selection module. Our prototype implements both.

Transport hardware. We configured our programmable transport module to implement NDP [24] entirely in hardware. We chose NDP because it has promising low-latency performance, and is well-suited to handle small RPC messages (the class of messages we are most interested in accelerating, i.e., nanoRequests). However, the nanoPU does not depend on NDP. As explained in Section 2.3, our NIC transport layer is programmable. It has already been shown to support several other protocols, including Homa [42]. We evaluate our hardware NDP implementation in Section 5.2.3.

JBSQ hardware. As explained in Section 2.3, our NIC implements JBSQ(2) [36] 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

[6] The maximum message size is a configurable parameter of the architecture and we have experimented with messages as long as 38 packets.

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 Section 5.2.3.

3.4 The nanoPU HW/SW Interface

To illustrate how software on the nanoPU core interacts with the hardware, Listing 1 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.[7] 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 2.1) from the `netRX` register to the `netTX` register, shown on line 19 (Listing 1), and thus the RX application header becomes the TX application header.[8] Upon writing the TX application header, the hardware ensures that there is sufficient buffer space for the entire message; otherwise, it generates an exception which should be handled by the application accordingly. The procedure then increments every integer in the received message and appends them to the message being transmitted. 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.[9] Finally, the procedure waits for the next message to arrive.

```
 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: 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 done
23    j wait_msg // wait for the next message
```

Listing 1: Loopback with increment. A nanoPU assembly program that waits for a 16B message, increments each word, and returns it to the sender.

3.5 How It All Fits Together

Next, we walk through a more representative nanoRequest processing application, written in C, to compute the dot product of a vector stored in memory and a vector contained in arriving RPC request messages. Listing 2 is the C code for the routine, based on a small library of C macros (`lnic_*`) we wrote to allow applications to interact with the nanoPU hardware (`netRX` and `netTX` GPRs, and the CSRs). The `lnic_wait()` macro corresponds to the `wait_msg` procedure on lines 9-15 in Listing 1. The `lnic_read()` and `lnic_write_*()` macros generate instructions that either read from or write to `netRX` or `netTX` using either registers, memory, or an immediate; and the `lnic_msg_done()` macro writes to the `lmsgdone` CSR, corresponding to line 22 of Listing 1. Our library also includes other macros as well such as `lnic_branch()` which branches control flow based on the value in `netRX`.

The dot product C application waits for a message to arrive then extracts the application header (the first word of every message), followed by the message type in the second word. It checks that it is a `DATA_TYPE` message, and reads the third word to know how many 8B words the vector contains. The vector identifies the in-memory weight to use for each word

[7] It is the responsibility of the application to ensure that it does not try to read `netRX` when the local RX queue is empty; doing so results in undefined behavior.

[8] Note that this instruction also sets the TX message length to be equal to the RX message length because the message length is included in the TX/RX application headers.

[9] A future implementation may also want to use this signal to flush any unread bytes of the current message from the local 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.

```
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 2: Example nanoPU application that computes the dot product between a vector in a network message and in-memory weights.

when computing the dot product. Note that the application processes message data directly out of the register file and message data never needs to be copied into memory, allowing it to run faster than on a traditional system. Finally, the application sends a response message back to the sender containing the dot product.

4 The nanoPU Applications

Applications that will benefit most from using the nanoPU fast path exhibit one or both of the following characteristics: (i) strict tail response time requirements for network messages; or (ii) short (μs-scale) on-core service times. It should come as no surprise that applications with strict tail response time requirements will benefit from using the nanoPU fast path. Enabling low tail response time was one of our primary goals that guided many of the design decisions described in Section 2. For the latter, when an application's on-core service time is short, any CPU cycles spent sending or receiving network messages become comparatively more expensive. The nanoPU's extremely low per-message overheads help to ensure that these applications are able to dedicate close to 100% of CPU cycles to performing useful processing and thus achieve their maximum possible message processing throughput. Furthermore, the nanoPU can also help to reduce on-core service times by reducing pressure on the cache-hierarchy and allowing message data to be processed directly out of the register file. Another consequence of having short on-core service times is that the end-to-end completion time of each RPC becomes dominated by communication latency. By moving the entire network stack into hardware and by using the register file interface, the nanoPU fast path efficiently reduces communication latency and, hence, the RPC completion time. Therefore, the relative benefit provided by the nanoPU will increase as on-core service time decreases. An application's on-core service time does not necessarily need to be sub-1μs in order to benefit from using the nanoPU. The following section describes a few specific classes of applications that we believe are well-suited for the nanoPU.

4.1 Example Application Classes

μs-scale (or ns-scale) Services. An increasing number of datacenter applications are implemented as a collection of independent software modules called microservices. It is common for a single user request to invoke microservices across thousands of servers. At such large scale, the *tail* RPC response time dominates the end-to-end performance of these applications [17]. Furthermore, many microservices exhibit very short on-core service times; a key-value store is one such example that has sub-1μs service time. Therefore, these applications exhibit both of the characteristics described in the previous section and are ideal candidates to accelerate with the nanoPU.

Programmable One-sided RDMA. Modern NICs support RDMA for quick read and write access to remote memory. Some NICs support further "one-sided" operations in hardware: a single RDMA request leads to very low latency *compare-and-swap*, or *fetch-and-add*. It is natural to consider extending the set of one-sided operations to further accelerate remote memory operations [40, 55], for example *indirect read* (dereferencing a memory pointer in one round-trip time, rather than two), *scan and read* (scan a small memory region to match an argument and fetch data from a pointer associated with the match), *return max*, and so on. Changing fixed-function NIC hardware requires a new hardware design and fork-lift upgrade, and so, instead, Google Snap [40] implements a suite of custom one-sided operations in software in the kernel. This idea would run much faster on the nanoPU, for example as an embedded core on a NIC, and could implement arbitrary one-sided RDMA operations in software (Section 5.3).

High Performance Computing (HPC) and Flash Bursts. HPC workloads (e.g., N-body simulations [34]) as well as flash bursts [37], a new class of data center applications that utilize hundreds or thousands of machines for a short amount of time (e.g., one millisecond), are both examples of highly parallelizable application classes that are partitioned into fine-grained tasks distributed across many machines. These applications tend to be very communication intensive and spend a significant amount of time sending and receiving small messages [37]. We believe that the nanoPU's extremely low per-message overheads and low communication latency can help to accelerate these applications.

Network Function Virtualization (NFV). NFV is a well-known class of applications with μs-scale on-core service times [60, 66]. The nanoPU's low per-message overhead, register file interface, and programmable PISA pipelines allow it to excel at stream processing network data and thus is an excellent platform for deploying NFV applications.

5 Evaluation

Our evaluations address the following four 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 latency under high load and bounded tail latency for well-behaved applications (Section 5.2.2)?
3. How does our prototype NIC pipeline (i.e., transport and core selection) perform under high incast and service-time variance (Section 5.2.3)?
4. 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 [31]), 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.

Benchmark tools. We use two different cycle-accurate simulation tools to perform our evaluations: (1) the Verilator [63] software simulator, and (2) the Firesim [31] FPGA-accelerated simulator. Firesim enables us to run large-scale, cycle-accurate simulations with hundreds of nanoPU cores using FPGAs in AWS F1 [1]. 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 2 shows a breakdown of the latency through each component 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).[10] 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 3 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 the nanoPU. 12ns (of the 51ns) are spent performing `memcpy`'s to swap the Ethernet source and destination addresses, something that is unnecessary for the nanoPU, because it is handled by the NIC hardware. The speedup of the nanoPU fast path decreases as the message size increases because the response time becomes 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 [57], the loopback response time would be faster, but the nanoPU fast path would still have 50% lower response time for small nanoRequests.

Loopback throughput. Figure 4 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. 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 the nanoPU than on the traditional NIC. For small 8B messages (72B packets), the nanoPU loopback application achieves 68Gb/s, or 118Mrps – 7× higher than the traditional system. For 1KB messages, the nanoPU achieves a 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.

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

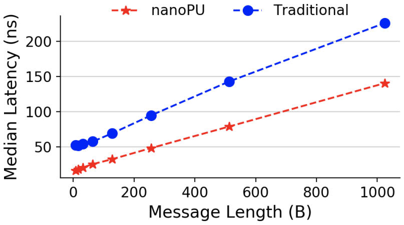

Figure 3: Loopback median response time vs. message length; nanoPU fast path and traditional.

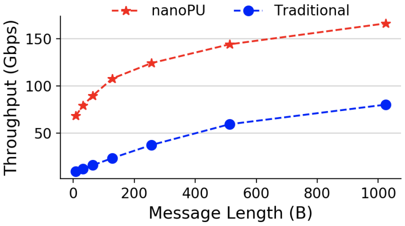

Figure 4: Loopback throughput vs. message length; nanoPU fast path and traditional.

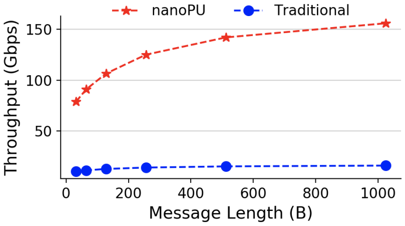

Figure 5: Loopback-with-increment throughput vs. message length; nanoPU fast path and traditional.

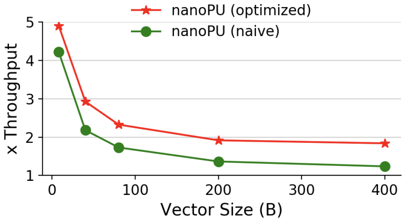

Figure 6: Dot-product throughput speedup for various vector sizes; nanoPU fast path (naive & optimal) relative to traditional NIC.

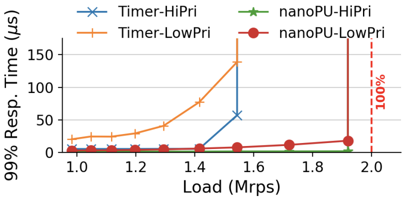

Figure 7: 99th %ile response time vs load; hardware thread scheduler (HTS) vs. traditional timer-interrupt driven scheduler (TIS).

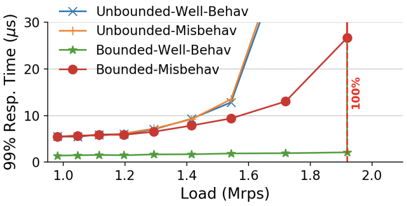

Figure 8: 99th %ile response time vs load for well-behaved and misbehaved threads, with and without bounded message processing time.

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; similar to the program described in Section 3.4.

Figure 5 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 [57], 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 more realistic benchmarks for real applications.

Stateful nanoRequest jobs. These are applications that process both message data and local memory data. Similar to the example described in Section 3.5, 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 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 6 shows the throughput speedup of the *optimal* and *naive* methods relative to the traditional application, for different message lengths.

- *Small messages:* For small messages below 100bytes, the 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 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 [57], we estimate the traditional design would run 1.5× faster for large messages. *Optimal* would still be 30% faster for large messages.

The benefits are clear when an application processes message data directly from the `netRX` register. While this may seem like a big constraint, we have found that it is generally feasible and 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 in Firesim [31]. It generates nanoRequests with Poisson inter-arrival times, and measures the end-to-end response time.

Priority thread scheduling. We compare our hardware thread scheduler (HTS) against a more traditional timer-interrupt driven scheduler (TIS) interrupted by the kernel every 5μs to swap in the highest-priority active thread. We run both schedulers in hardware on our prototype.[11] TIS uses a 5μs timer interrupt to match the granularity of state-of-the-art low-latency operating systems [27, 49].

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 7 shows the 99th %ile tail response time vs load for both thread scheduling policies, with a high and low priority thread. HTS reduces tail response time by 4× and 6.5× at high and low load, respectively; and can sustain 96% load.[12]

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. 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 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.

We expect an application with at most one message at a time in the RX queue, to have a tail response time bounded by $2 \cdot 43ns + 17ns + 2 \cdot 1000ns + 50ns = 2.15\mu s$. 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).[13] 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.

5.2.3 Prototype NIC pipeline

Hardware NDP transport. We verify our hardware NDP implementation by running a large 80-to-1 incast experiment on Firesim, with 324 cores simulated on 81 AWS F1 FPGAs. All hosts are connected to one simulated switch; 80 clients send a single packet message to the same server at the same time. The switch has insufficient buffer capacity to store all 80 messages and hence some are dropped. When NDP is disabled, dropped packets are detected by the sender using a timeout and therefore the maximum latency through the network is dictated by the timeout interval. When NDP is enabled, the dropped messages are quickly retransmitted by NDP's packet trimming and NACKing mechanisms, lowering maximum network latency by a factor of three.

Hardware JBSQ core selection. We evaluate our JBSQ implementation using a bimodal service-time distribution: 99.5% of nanoRequests have a service time of 500ns and 0.5% have a service time of 5μs. When using a random core assignment technique, like receive side scaling (RSS), to balance requests across four cores, short requests occasionally get queued behind long requests, resulting in high tail response time. With JBSQ enabled, tail response time is reduced 5× at low load, and can sustain 15% higher load than RSS.

5.3 Application Benchmarks

As shown in Table 1, we implemented and evaluated many applications on our nanoPU prototype. Below, we present the evaluation results for a few of these applications.

MICA. We ported the MICA key-value store [38] and compared it running on the nanoPU and traditional NIC designs. MICA is implemented as a library with an API that allows applications to GET and SET key-value pairs. Traditionally, this API uses in-memory buffers to pass key-value pairs between the MICA library and application code. The *naive* way to port MICA to the nanoPU is to copy key-value pairs in network messages between the register file and in-memory buffers, using the MICA library without modification. However, we find it more efficient to modify the MICA library to read and write the register file directly when performing GET and SET operations. This avoids unnecessary `memcpy`s in the MICA library. Optimizing the MICA library to use the register file only required changes to 36 lines of code.

Our evaluation stores 10k key-value pairs (16B keys and 512B values). The load generator sends a 50/50 mix of read-/write nanoRequest queries with keys picked uniformly. Figure 9 compares the 99th %ile tail response time vs load for the traditional, nanoPU naive, and nanoPU optimized versions of this application. The naive nanoPU implementation outperforms the traditional implementation, likely because it is able to use an L1-cache resident in-memory buffer rather than an LLC-resident DMA buffer. The optimized nanoPU imple-

[11] TIS would run in software in practice, likely on a separate core, and would therefore be slower than in hardware.

[12] Our prototype does not currently allocate NIC buffer space per-application, causing high-priority requests to be dropped when the low-priority queue is fill. This will be fixed in the next version.

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

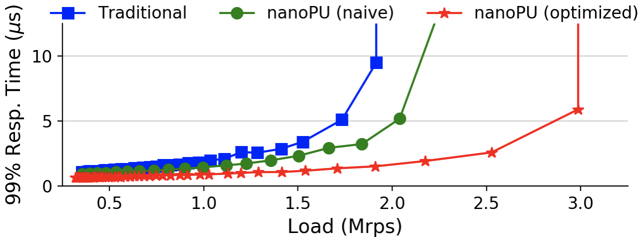

Figure 9: MICA KV store: 99th %ile tail response time for READ and WRITE requests.

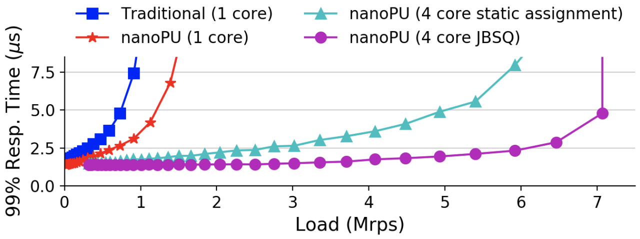

Figure 10: Set intersection: 99th %ile tail response time.

mentation is able to achieve about 30% higher throughput and lower response times by efficiently using the register file interface when processing network messages.

Raft. Raft is a widely-used consensus algorithm for distributed applications [47]. We evaluate a production grade version of Raft [53] using a 16B-key, 64B-value MICA key-value store state machine, with three servers and one client connected to a single switch. The switch has a forwarding latency of 300ns (typical of modern cut-through commercial switch ASICs [58]) and all links have a latency of 43ns. Although our Raft cluster correctly implements leader election, can tolerate server failure, and our client can automatically identify a new Raft leader, we evaluate our Raft cluster in the steady-state, failure-free case, with a single leader and three fully-functioning replicas.

We define the response time to be from when the client issues a three-way replicated write request to the Raft cluster, until the client hears back from the cluster leader that the request has been fully replicated and committed across all three Raft servers. In 10K trials, the median response time was 3.08μs, with a 3.26μs 99th %ile tail response time. eRPC [28], a high performance, highly-optimized RPC library reports a 5.5μs median and 6.3μs 99th %ile tail response time — about a factor of two slower.

Set algebra. In information retrieval systems, set intersections are commonly performed for data mining, text analytics, and search. For example, Lucene [8] uses a reverse index that maps each word to a set of documents that contain the word. Searches yield a document set for each search word, then compute the intersection of these sets.

We created a reverse index of 100 Wikipedia [65] articles with 200 common English words. Our load generator sends search requests with 1-4 words chosen from a Zipf distribution based on word frequency. Porting the set intersection application to the nanoPU was straight forward. The only difference between the nanoPU and traditional versions of the applications is the logic to send and receive network messages (~50 LOC). We did not need to make any modifications to the application logic that computes the intersection between sets of document IDs.

One-sided RDMA	Latency (ns)	
	Median	90th %ile
Read	678	680
Write	679	686
Compare-and-Swap	687	690
Fetch-and-Add	688	692
Indirect Read	691	715

Table 2: Median and 90th %ile latency of one-sided RDMA operations implemented on the nanoPU. Measurements are made at the client, and the one-way latency through the switch and links is 300ns.

Figure 10 shows the tail response time for searches. The traditional design has a low-load tail response time of 1.7μs, compared to 1.4μs on a single nanoPU core. JBSQ helps to ensure that long running requests do not get stuck behind short ones. With JBSQ enabled for four cores, the 99th %ile tail response time remains low until 7Mrps.

One-sided RDMA operations. As described in Section 4.1, the nanoPU can implement flexible, low latency one-sided RDMA operations. As a baseline, the median end-to-end latency of one-sided operations between two hosts using state-of-the-art RDMA NICs, connected by a single switch with a port-to-port latency of 300ns is about 2μs [28]. [14] Table 2 shows the median and 90% tail latency of several one-sided RDMA operations implemented on the nanoPU, using the same topology as the baseline. The median latency, measured by the nanoPU client, is 680-690ns with a 90% tail latency of approximately 700ns, 65% lower latency than state-of-the-art RDMA NICs. Most of the latency reduction is from eliminating the traversal of PCIe on the client and server.

In addition to the standard one-sided RDMA operations (read, write, compare-and-swap, fetch-and-add) we also implement *indirect read*, in which the server simply dereferences a pointer to determine the actual memory address to read. This operation would require two network round trips on a standard RDMA NIC; on the nanoPU, it takes only a few nanoseconds longer than a standard read.

6 Discussion

nanoPU deployment possibilities. We believe there are a number of ways to deploy nanoPU ideas, in addition to a modified regular CPU. For example, the nanoPU fast path

[14] Note that when using an ARM-based smartNIC, such as the Mellanox BlueField [41], the time to traverse the embedded cores will increase this end-to-end latency by at least a factor of two [39, 61].

could be added to embedded CPUs on smartNICs for the data center [5, 41, 44]. This could be a less invasive way to introduce nanoPU ideas without needing to modify server CPUs. A more extreme approach would be to build a nanoPU domain-specific architecture explicitly for nanoRequests. For example, it would be practical today to build a single chip 512-core nanoPU, similar to Celerity [15], with one hundred 100GE interfaces, capable of servicing RPCs at up to 10Tb/s.

In-order execution. Our prototype is based on a simple 5-stage, in-order RISC-V Rocket core and required only minor modifications to the CPU pipeline. An out-of-order processor would require bigger changes to ensure that words read from `netRX` are delivered to the application in FIFO order.

7 Related Work

Low-latency RPCs (software). Recent work focuses on algorithms to choose a core by approximating a single-queue system using work-stealing (like ZygOS [51]) or preempting requests at microsecond timescales (Shinjuku [27]). However, the overheads associated with inter-core synchronization and software preemption make these approaches too slow and coarse-grained for nanoRequests.

eRPC [28] takes the other extreme to the nanoPU and runs everything in software, and through clever optimizations, achieves impressively low latency on a commodity server for the common case. eRPC has good median response times, but its common-case optimizations sacrifice tail response times, which often dictate application performance. The nanoPU's hardware pipeline makes median and tail RPC response times almost identical.

Low-latency RPCs (hardware). We are not the first to implement core-selection algorithms in hardware. RPCvalet [12] and NeBuLa [57] are both built on the Scale-out NUMA architecture [46]. RPCvalet implements a single queue system, which in theory provides optimal performance. However, it ran into memory bandwidth contention issues, which they later resolve in NeBuLa. Both NeBuLa and R2P2 [36] implement the JBSQ load balancing policy; NeBuLa runs JBSQ on the server whereas R2P2 runs JBSQ in a programmable switch. Like NeBuLa, the nanoPU also implements JBSQ to steer requests to cores.

Many NICs support RDMA in hardware. Several systems (HERD [29], FaSST [30], and DrTM+R [11]) exploit RDMA to build applications on top. As described in Sections 4.1 and 5.3, the nanoPU can be used to implement programmable one-sided RDMA operations while providing lower latency than state-of-the-art commercial NICs.

SmartNICs (NICs with CPUs on them) [5, 41, 44] are being deployed to offload infrastructure software from the main server to CPUs on the NIC. However, these may actually increase the RPC latency, unless they adopt nanoPU-like designs on the NIC.

Transport protocols in hardware. We are not the first to implement the transport layer and congestion control in hardware. Modern NICs that support RDMA over Converged Ethernet (RoCE) implement DCQCN [67] in hardware. In the academic research community, Tonic [2] proposes a framework for implementing congestion control in hardware. The nanoPU's programmable transport layer (and NDP implementation) draws upon ideas in Tonic.

Register file interface. GPRs were first used by the J-machine [13] for low-latency inter-core communication on the same machine, but were abandoned because of the difficulty implementing thread-safety. The idea has reappeared in several designs, including the RAW processor [64], and the SNAP processor for low-power sensor networks [32].

8 Conclusion

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.

Acknowledgements

We would like to thank our shepherd, Yiying Zhang, Amin Vahdat, John Ousterhout, and Kunle Olukotun for their invaluable suggestions throughout the duration of this project. This work was supported by Xilinx, Google, Stanford Platform Lab, and DARPA Contract Numbers HR0011-20-C-0107 and FA8650-18-2-7865.

A Artifact Appendix

Abstract

This artifact contains the Chisel source code of our nanoPU prototype as well as the application code and simulation infrastructure that is required to reproduce the key results presented in this paper. Our prototype is evaluated using both Verilator for cycle-accurate simulations in software, and Firesim for cycle-accurate simulations on FPGAs in AWS. The artifact is packaged as an AWS EC2 image with all of the dependencies pre-installed to make it easy for others to use and build upon our work.

Scope

The artifact can be used to reproduce the key results presented in the following figures:

- **Figure 3** – Loopback latency; nanoPU vs. traditional.
- **Figure 4** – Loopback throughput; nanoPU vs. traditional.
- **Figure 5** – Loopback-with-inc. throughput; nanoPU vs. traditional.
- **Figure 6** – Dot-product throughput speedup.
- **Figure 7** – Tail response time using priority thread scheduling.
- **Figure 8** – Tail response time using bounded message processing time.
- **Figure 9** – MICA tail response time.
- **Figure 10** – Set intersection tail response time.

Additionally, there are a number of ways to use the artifact to build upon our work. For example, you can write new applications for the nanoPU and evaluate them at close to real-time using a custom topology with Firesim. Alternatively, you can modify the nanoPU architecture and use the provided simulation infrastructure to easily test your changes.

Contents

The documentation for the nanoPU artifact can be found at: https://github.com/l-nic/chipyard/wiki. The primary repositories are briefly described below:

- Chipyard – The main top-level repository which contains the others listed below as git submodules. Contains application code as well as Verilator simulation infrastructure.
- Rocket Chip – Contains the chisel source code for our modified RISC-V Rocket core (as well as all of the other components that are needed to create a full SoC).
- L-NIC – Contains the chisel source code for the nanoPU NIC.
- Firesim – Provides all of the infrastructure that is required to run FPGA-accelerated, cycle-accurate simulations on AWS.

Hosting

The nanoPU artifact is hosted on GitHub:
https://github.com/l-nic/chipyard/tree/nanoPU-artifact-v1.0

The development branch is called `lnic-dev` and, at the time of this writing, the latest release is tagged `nanoPU-artifact-v1.0`. In order to make it easy for others to reuse and build upon our work, we have developed a custom Amazon Machine Image (AMI) with the artifact and all required dependencies pre-installed. See the documentation for detailed instructions regarding how to access and use this AMI.

Requirements

In order to use the nanoPU artifact, you will need access to an AWS account and you will need to subscribe to the AWS FPGA developer AMI. Additionally, you will need permission from AWS to launch F1 instances. These requirements are explained in greater detail in the online documentation.

References

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

[2] 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.

[3] 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.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

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

[6] 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.

[7] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

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

[9] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[10] 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.

[11] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–17, 2016.

[12] 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.

[13] 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.

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

[15] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, et al. The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 38(2):30–41, 2018.

[16] 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.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Zahra Khatami, Hartmut Kaiser, Patricia Grubel, Adrian Serio, and J Ramanujam. A massively parallel distributed n-body application implemented with hpx. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 57–64. IEEE, 2016.

[35] 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.

[36] 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.

[37] Yilong Li, Seo Jin Park, and John Ousterhout. Millisort and milliquery: Large-scale data-intensive computing in milliseconds. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, 2021.

[38] 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.

[39] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. Association for Computing Machinery, 2019.

[40] 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.

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

[42] 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.

[43] Microsoft: Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling. Accessed on 2020-12-07.

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

[45] 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.

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

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

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

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

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

[55] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[56] 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.

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

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

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

[60] 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.

[61] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 {USENIX} Annual Technical Conference ({USENIX} {ATC} 20)*, pages 33–48, 2020.

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

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

[64] 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.

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

[66] Yifan Yuan, Yipeng Wang, Ren Wang, and Jian Huang. Halo: accelerating flow classification for scalable packet processing in nfv. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 601–614. IEEE, 2019.

[67] 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.