TODO LIST

Shahbaz: needs to be revised? 1

Shahbaz: add missing citations once the paper is ready. 1

Shahbaz: this is too much info at this point; also I'd recommend not empahsizing things that we haven't built. 4

Shahbaz: the NIC-Core interface seemed better, as it relates with figure and para headings? also the statements here are more verbose now. 4

Alex: The nanoPU layer we're calling "NIC-Core interface" might be easier to understand if we refer to it as something like the "NIC Message Reassembler and Packetizer" layer. Since right now we have to keep referring to "the interface between the NIC-Core interface and the CPU", which is a little hard to follow. I'm wondering if it might be easier to have a separate name for the hardware (i.e. the per-context FIFOs, reassembly, packetization, and scheduling) – then the NIC-core interface really is just the interface (i.e. the GPRs, CSRs and message formats. . . 4

Nick: I propose we axe this paragraph to save space and update the figure to remove the CDC. it is an implementation detail we don't need here 6

Nick: I will work on this part more in the morning. I think it can be shortened. If anyone wnats to take a shot, go for it.... 7

Steve: My main concern with using IceNIC as a baseline comparison is that it does not support TX/RX description rings, which means that the application must send packet descriptors one at a time to the NIC DMA engine for TX and the NIC DMA engine only fills up one packet buffer at a time for RX. I don't think this matters too much for latency of a single packet, but throughput will be affected. For example, the TX/RX throughput for 64B packets would be higher for IceNIC if it supported this feature. Additionally, the IceNIC applications might be written differently. For example, rather then using the same static buffer for every packet, they might use N packet buffers and process them in batches. So it's tough to say what the performance impact would be on the evaluated applications. 8

Steve: change IceNIC label to RISC-V w/ IceNIC and change L-NIC label to nanoPU in Figure 8 8

Steve: How does this compare to the latency of modern schedulers? 9

Steve: Do we want to mention isolates? 12

nanoPU: From Microservices to Nanoservices

Paper #52

13 Pages Body, 14 Pages Total

Shahbaz: needs to be re-vised?

ABSTRACT

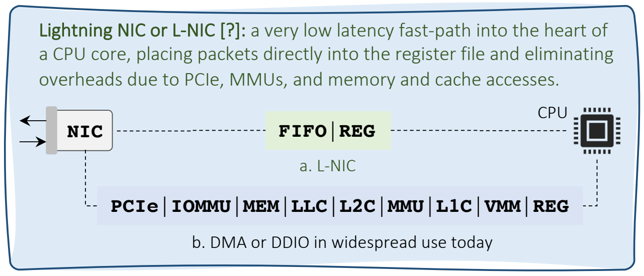
We present the nanoservice Processing Unit (nanoPU), a domain specific processor designed to accelerate compute-intensive, distributed applications. The nanoPU provides an order of magnitude lower average and standard deviation in RPC completion times over existing systems. We also introduce nanoservices, applications that are parallelizable into fine grained compute units with sub-microsecond response times and cache resident working sets. We believe that this class of applications will become increasingly popular, as many distributed applications have inherent parallelism that cannot be harnessed today because of large, unpredictable RPC completion times and substantial per-message overheads.

The nanoPU, as a new networking-optimized compute platform, helps to make nanoservices a reality. It contains a fast path from the network directly to CPU core, minimizing communication latency, and it delegates thread scheduling to the NIC, minimizing tail RPC completion times. The resulting architecture, which requires minimal changes to the CPU core, thus provides the necessary starting point for the development of nanoservice applications. We built a prototype nanoPU on top of an open source RISC-V CPU and evaluated its performance for a suite of real nanoservice applications using cycle-accurate hardware simulations.

1 INTRODUCTION

Cloud data centers allow thousands of client applications to share millions of CPU cores. Many of these client applications harness CPU cores using *coarse*-grained parallelism, such as MapReduce [21], Hadoop [6] and client-server web applications. Other, more specialized applications use *fine*-grained parallelism to employ hundreds of thousands of CPUs (e.g., search [12], HPC [54]). It is well-known to be difficult to write applications using fine-grained parallelism, and so the technique is generally reserved as a last resort, for when applications must respond quickly (e.g., search or ads), or must complete in a reasonable time (e.g., a large HPC physical simulation [2, 26, 31] or overnight ML training [1]). Recently, as applications have grown ever larger (e.g., mushrooming ML training data, big data analytics, and social media), the speed of individual cores has grown ever more slowly [28]. Increased parallelism seems inevitable.

Some recent products and open-source projects have used fine-grain modules, deployed as microservices (e.g., AWS Lambda [9], Azure Serverless Computing [8], Google Cloud


Lightning NIC or L-NIC [?]: a very low latency fast-path into the heart of a CPU core, placing packets directly into the register file and eliminating overheads due to PCIe, MMUs, and memory and cache accesses.
CPU
NIC
FIFO | REG
a. L-NIC
PCIe | IOMMU | MEM | LLC | L2C | MMU | L1C | VMM | REG
b. DMA or DDIO in widespread use today

Functions [25], and OpenFaaS [46]) to demonstrate impressive speedup of parallel applications, including video encoding [23], video compression [5], and face recognition [16], by successfully employing thousands of CPU cores in parallel. But, as we will see, this level of parallelism still yields far from the fastest possible execution time.

In the limit, if we segment an application into its smallest, indivisible software modules, *and if communication is instantaneous*, then an application can complete in the execution time of the longest sequence of serially-dependent modules. But, communication is not instantaneous—far from it. Many researchers have reported the (surprisingly) high communication latency between two cores, e.g., a request-response time of 0.7 ms [36, 49]. Intuitively, if it takes 0.7 ms to ask another CPU to execute a function, then it is only worth doing if we have more than 0.7 ms of work to do; otherwise we might as well execute the code locally.

For example, consider a physical simulation of one million objects, where each object needs to update its ten neighbors each time it moves. If each object executes 10 μs of code to calculate its move (about 40,000 instructions on a modern CPU), but takes 1 ms to tell its neighbors, then latency dominates the computation, and it makes sense for 10–100 objects to share one core. If instead, an RPC call could complete in less than 1 μs, then it makes sense for each object to run on its own core, and the simulation can complete one hundred times sooner.

Shahbaz: add missing citations once the paper is ready.

More generally, consider a distributed application consisting of N potentially parallelizable tasks, each of which can be executed locally (in x μs) or on a remote core via RPC (in $r \cdot x$ μs, where $r \geq 1$). If the degree of parallelism, K, is the number of issued RPCs (i.e., the number of remote cores), we minimize job-completion time when $(N - K) \cdot x = r \cdot x$ (i.e., $K = N - r$). Increasing the degree of parallelism, K, further yields no benefit; if we send more RPCs, the local core just sits idle waiting for the RPCs to complete. For example, if $r = 10$ (i.e., RPCs take 10 times as long as local execution) then we should send $N - 10$ RPCs while executing 10 tasks locally. If we reduce latency so that $r = 1$, the total job-completion time is reduced by 90%.

In practice, RPC completion times, f_i, are not fixed, but characterized by a probability distribution, $f_i \in \mathcal{F}$. The total job completion time is $(N - K)x = \max f_i$ where $1 \leq i \leq K$. Thus, if we want to improve performance, we must reduce the RPC tail latency $\max f_i$. This observation is not new and has been the motivation for much recent work [18, 35, 47].

The goal of our research is to design, build, and evaluate a domain-specific processor, which we call a *nanoPU*, optimized for minimizing the completion time of highly-parallel, compute-intensive jobs. The nanoPU (§3) is designed for massive scale-out computing. Specifically, millions of nanoPUs could be deployed to run what we call *nanoservices*: highly distributed, compute-intensive applications that process RPC requests in under 1 μs using fine-grained, cache-resident threads.

If we are to minimize the average and tail latency of an RPC, we must minimize the latency at every step of the way. Starting from when a CPU thread issues an RPC, we must minimize latency through the network stack, through the cache, DRAM and DMA subsystem, through the NIC and onto the wire; across Ethernet links and through switches; and then from the destination NIC through the DMA, DRAM, and cache subsystem; through the network stack, and then wait for the RPC request's target thread to be scheduled for execution by the operating system and for the application to read and process the request. Most prior work has focused on one aspect of an RPC's latency–for example, low-latency transport layers [4, 27, 41] to reduce network congestion and minimize latency through switches, or thread scheduling to make sure an incoming RPC request starts executing promptly [35, 47]. Our goal is to reduce RPC response times by at least an order of magnitude from tens or hundreds of microseconds down to approximately 1 μs. This is mostly a networking problem, one that leads us to tackle three questions:

(1) **Minimizing *average* RPC completion time:** How can we minimize the time that a network message spends between the wire and an application thread? This requires minimizing the time through hardware and software. Our approach is to build upon the extremely low-latency *Lightning NIC* (L-NIC), described in [30], in which messages bypass the traditional networking stack, as well as the DMA, DRAM and cache hierarchy entirely, and are placed directly into the CPU register file. An arriving RPC can start execution in under 100 ns.
(2) **Minimizing *tail* RPC completion time:** How can we limit resource contention for the CPU, memory and network, and minimize the cost of context switching between threads? These are the primary causes of variance and hence tail RPC completion times. Our approach is to design a thread scheduler into the NIC hardware, as well as to enable the NIC to implement a very low latency hardware transport layer, such as Homa [41] or NDP [27].
(3) **Making it practical:** How can such an extreme approach to hardware and software be practically deployed, with minimal disruption? We have designed and implemented a hardware prototype, based on a RISC-V CPU [7], that includes a 100 Gb/s Ethernet L-NIC CPU interface and a hardware thread scheduler, and we report performance results for real nanoservice applications.

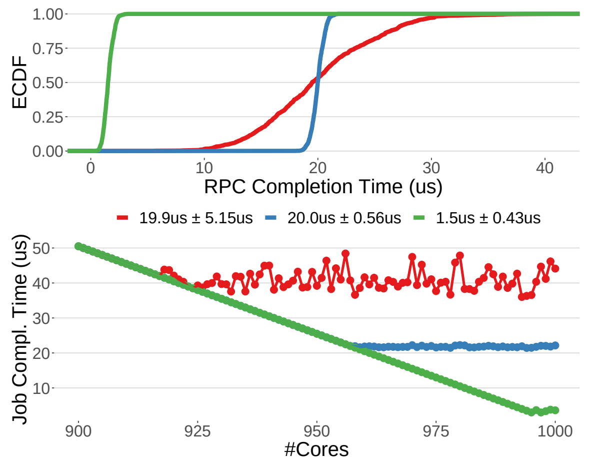


Figure 1: Total job completion time of an example application with 1000 independent tasks (each with 500 μs processing time) for varying number of cores (i.e. issued RPCs). These plots demonstrate the impact of the RPC completion time distribution on the total job completion time.

Figure 1, while based on synthetic numbers, illustrates what we aim to achieve. The first graph shows various distributions of RPC completion times. The red graph is intended to be representative of the current state-of-the-art RPC completion times [36]. The blue graph demonstrates the impact on total application runtime when the standard deviation of the RPC completion time distribution is reduced by an order of magnitude. If we can drive down the average *and* the standard deviation, represented by the green line, then the job-completion time falls linearly as we add cores, until eventually the per-RPC overhead limits parallelism.

In summary, we make the following contributions:

- A new compute framework, called nanoservices (§2), for highly parallelizable, compute-intensive distributed applications.
- The design and implementation of a nanoPU (§3) with a 100 Gb/s NIC datapath (§3.1), a message-based interface directly between the NIC and CPU register file (§3.2) and a NIC-driven thread scheduler (§3.3).
- An overview of the hardware mechanisms designed to support a low-latency transport protocol on the NIC (§3.4).
- An open-source nanoPU prototype (§4), based on a RISC-V Rocket Core [7].
- An evaluation of our nanoPU prototype against a traditional RISC-V CPU with a DMA-based NIC, for a suite of nanoservice applications (§5). Our evaluation demonstrates that nanoPU reduces both average and standard deviation in application processing times by about an order of magnitude.

2 NANOSERVICES

Nanoservices are a new framework for building compute-intensive, distributed applications with the following three characteristics:

C1: Each application is divisible into compute-intensive *nanotasks*.
C2: Each nanotask is able to process and respond to network messages in under 1 μs.
C3: The working set of each nanotask fits in the L1 cache.[1]

Figure 2 highlights the key differences between We believe that many applications can benefit from the nanoservices framework, including applications such as physical simulations [26, 50], neural-network inference & training [1], graphics processing [24], and state-space search algorithms [58]. As we will show in §5, total application runtime has the potential to be reduced by about three orders of magnitude if they are implemented using nanoservices.

The central idea is as follows: A nanotask must execute in the shortest, and most deterministic time possible. This implies that *everything* the nanotask needs must sit in registers and the L1 cache: Instructions in the L1 instruction cache, and data (in-progress messages, variables and program state) in registers or the L1 data cache. Cache misses are expensive, and the goal is to avoid them–L2 cache and external memory are, respectively, ten and one hundred times slower than the L1 cache [20].

Building nanoservices requires more than simply dividing applications into nanotasks that access small amounts

[1]L1 cache is typically 256 kB–1 MB at the time of writing, including the data and instruction cache.

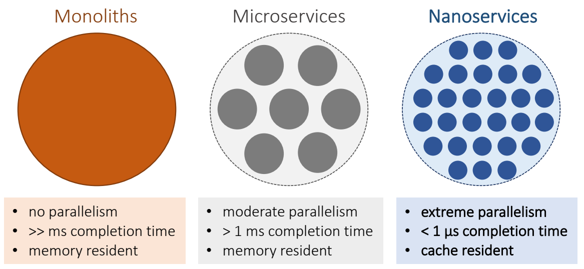


Figure 2: A comparison of key characteristics of *nanoservices* vs. monoliths and microservices.

of data. We also need to consider how to quickly and predictably transfer message data between the wire and an application thread, and how to efficiently schedule nanotasks on the available processor cores. It is not practical to run nanoservices on today's systems, which were designed to support applications that process RPCs in milliseconds, or at best tens of microseconds [36, 49], not hundreds of nanoseconds. Addressing these problems therefore requires hardware changes, which we address with our proposed design, the nanoPU, in the following section.

3 THE NANOPU

The nanoPU is a new domain-specific processor optimized for running nanotasks—quickly and predictably—for compute-intensive distributed applications based on nanoservices. A nanoPU consists of one or more CPU cores and one or more low-latency NICs. The CPU cores are slightly-modified cores; our design is based on the popular, open-source RISC-V ISA [52]. The low-latency NIC is inspired by the recently proposed *Lightning NIC (L-NIC)* [30], a novel approach that terminates the transport layer in hardware, then delivers message data right into the registers at the heart of the CPU core. The L-NIC approach minimizes latency (and unpredictability) by bypassing DMA, cache, and DRAM entirely. Our nanoPU design also adds a novel hardware thread scheduler, to minimize the time (and variability) from when a message arrives until the thread starts processing it.

Figure 3 shows a high-level block diagram of a nanoPU. This particular example shows one network interface shared by three cores, but in general the nanoPU is designed to work with any ratio of NICs to cores. Depending on the context, it may make sense to build nanoPUs with one core per NIC (e.g., for small embedded systems), all the way to hundreds of cores per NIC. For example, it would be practical today to design a chip with 500 cores [3, 13] and over a hundred 100 Gb/s Ethernet interfaces;[2] in this example, the ratio would be five cores per NIC. Of course, many other ratios are possible and

[2]Commercial switch chips exist with 128 × 100 Gb/s Ethernet MACs today.

the ideal ratio depends on the application, technology, and economics.

The nanoPU is a domain-specific processor. With the slowing of Moore's Law, new domain-specific processors are being widely used as accelerators for specific, high-volume workloads, such as graphics [45], machine-learning [1] and networking [15]. While the nanoPU is not nearly as radical a departure from a general-purpose CPU as, say, a GPU, TPU or programmable switch (after all, our design relies heavily on an existing core), the nanoPU shares the approach of tailoring the chip design for a specific class of applications.[3]

If nanoPU is so fast, why are not all CPUs designed this way? It is because general-purpose CPUs are optimized for general-purpose workloads, most of which are memory-intensive. They are "load-store" machines, with memory as a first-class citizen. Arriving and departing packet data must pass through the memory subsystem first, on its way in and out of the CPU. Our approach is, instead, to make network messages first-class citizens in their own right, independent of memory. Network messages arrive directly into the CPU, without needing to make their way through the memory hierarchy. General-purpose CPUs put memory in the network and attach compute to memory, which makes perfect sense for memory-intensive applications. Our approach in the nanoPU is to tightly couple compute directly to the network, and then attach memory on the side as needed.

The nanoPU has the following key characteristics that we visit, in turn, in the next few subsections:

- **NIC Datapath:** A programmable event-driven PISA "Match-Action Unit" (MAU) pipeline [29] on the NIC to process packet headers as they arrive and depart over the network, terminate tunnels, encrypt/decrypt and compress/decompress data; and a low-latency transport protocol in hardware, such as Homa [41] or NDP [27]. The NIC Datapath presents a packet interface to the NIC Message Reassembler and Packetizer.
- **NIC Message Reassembler and Packetizer:** Hardware FIFOS programmatically bound to each running nanotask CPU context that handle the reassembly of inbound packets into messages and the packetization of outbound messages into packets. This allows the NIC Message Reassembler and Packetizer to present a message interface to each nanotask executing on the CPU.
- **Hardware Thread Scheduling:** The NIC ensures that arriving message data reaches the core quickly. We also need to make sure the core switches to the message's nanotask recipient to process the message quickly; to support this requirement, the hardware includes a very low-latency thread scheduler [56].

[3]As an aside, it is only possible to consider prototyping a nanoPU in a university because of two recent trends: RISC-V provides a remarkably stable starting point for heavy-lifting, and the narrow performance gap between the leading edge process (currently 7 nm) and the closest already-paid-for process (16 nm) makes it economically feasible to build an interesting nanoPU.

Shahbaz: this is too much info at this point; also I'd recommend not emphasizing things that we haven't built.

Shahbaz: the NIC-Core interface seemed better, as it relates with figure and para headings? also the

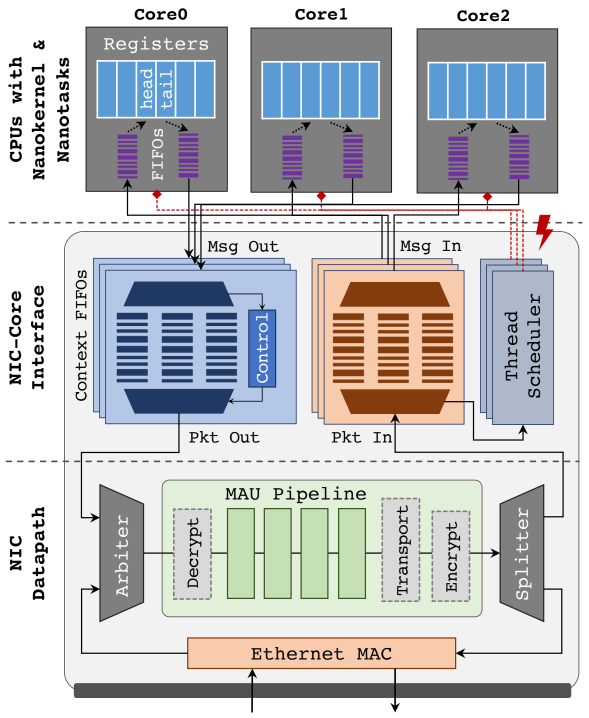


Figure 3: A high-level architecture of nanoPU having a NIC datapath with an event-driven MAU pipeline, a NIC-Core Interface with RX/TX context FIFOs per core and a thread scheduler, as well as CPU cores running the nanoKernel and nanotasks.

These characteristics together yield a very low latency path—just a few clock cycles—from the network right to the very heart of the CPU core—its register file, giving mean and tail latency reduction of one to two order of magnitude. We do not believe there is a lower latency path to a running thread.

3.1 The NIC Datapath

The NIC datapath processes packets as they enter and leave the network. Figure 3 shows a high-level block diagram of the NIC packet-processing datapath.

The centerpiece of the NIC datapath is an event-driven PISA pipeline [29]. The original PISA architecture, proposed in the RMT chip [15] and later used by Tofino [55], is designed for mostly-stateless match-action processing of packet headers; for example for lookups, encapsulation, tunnels and telemetry. Basic PISA only supports one type of event: the arrival of new packets. *Event-driven* PISA enhances the basic model to support other event types, such as packet drops,

Alex: The nanoPU layer we're calling "NIC-Core interface" might be easier to understand if we refer to it as something like the "NIC Message Reassembler and Packetizer" layer. Since

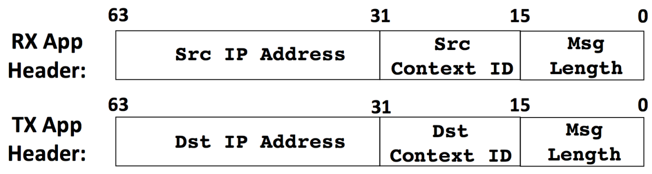


Figure 4: Application header formats used in the NIC-Core interface.

timers and state-dependent events. Section 3.4 describes how an event-driven PISA pipeline can directly support transport protocols in hardware, thus offloading per-packet processing from the CPU.

The PISA pipeline also performs protocol processing: depending on the loaded P4 [14] program, it might add and remove Ethernet, IP, VXLAN, GRE, INT [38] and transport headers. We also program it to add or remove a small per-context header containing the source IP address, a context ID, and message length Figure 4.

As others have noted, a P4-programmable PISA pipeline can also be used to accelerate some applications by offloading processing from the CPU (e.g., memory and disk caches [33], load-balancers [40], consensus protocols [32] and firewalls [51]). The L-NIC paper [30] describes how a PISA pipeline can accelerate nanoservices to search the Othello state-space.

3.2 The NIC-Core Interface

The nanoPU's NIC Message Reassembler and Packetizer, shown in Figure 3, is responsible for assembling arriving packets into a self-contained RPC message, and then queuing the message into its destination nanotask's FIFO, where the message will remain until that nanotask's thread is scheduled. The message then flows, one word at a time, into the core's head register as the nanotask application processes the words. In the egress direction, the Message Reassembler and Packetizer breaks messages into packets before being sending them to the NIC Datapath.

We make two small modifications to allow the CPU core to interface with this layer of the NIC:

- Two former general purpose registers (GPRs) are now reserved for a special purpose: one is the HEAD of the network receive queue, and the other is the TAIL of the network transmit queue. Applications must be compiled to avoid using reserved GPRs for temporary storage. Fortunately, gcc makes it easy to restrict register allocation via command line options.
- New control status registers (CSRs) are added for out-of-band communication between the CPU and the NIC. These are used to pass nanotask thread IDs and NIC-driven scheduling information between the NIC and the CPU.

The NIC Message Reassembler and Packetizer hardware maintains transmit and receive FIFOs for each nanotask context (i.e. thread) that is currently pinned to the core. The hardware makes sure that the currently running thread only sees (i.e. reads from and writes to) its own per-context FIFO via the HEAD and TAIL GPRs.

The NIC exposes a *message* interface to applications running on the core, instead of a more traditional packet interface. As a result, the nanoPU NIC hardware must be responsible for the termination of the transport layer, including breaking messages into packets, ensuring reliable delivery, and processing message reassembly.

To convey relevant transport information, all messages sent or received by an application carry an 8-byte application header whose format is shown in Figure 4. Applications must write the message's destination IP address, destination nanotask context ID and message length at the start of all outgoing messages. (Including the message length allows the NIC to determine when the application has finished writing a complete message.) Similarly, all incoming messages read by any application will begin with the message's source IP address, source nanotask context ID, and message length.

```
// Simple loopback nanoservice
entry:
  // First register context ID with NIC
  csrwi lcurcontext, 0
  csrwi lcurpriority, 0
  csrwi lniccmd, 1

// Wait for a message to arrive
wait_msg:
  csrr  a5, lmsgsrdy
  bnez  a5, loopback_plus1_16B
idle:
  csrwi lidle, 1
  csrr  a5, lmsgsrdy
  beqz  a5, idle

// Loopback and increment 16B message
loopback_plus1_16B:
  // Copy application header from head to tail
  mv t6, t5
  // Increment first data word
  addi t6, t5, 1
  // Increment second data word
  addi t6, t5, 1
  // Wait for the next message
  j wait_msg
```

Listing 1: Loopback with increment: A simple RISC-V nanoPU assembly program to register a context & priority with the NIC, wait for a 16B message to arrive then increment values during loopback from HEAD to TAIL FIFOs..

To illustrate how a nanoPU interacts with its NIC, Listing 1 shows a simple loopback+increment program in RISC-V assembly language. The program continuously reads 16 byte messages (consisting of two 8 byte integers) from the network, increments the integers, and sends the messages back to their sender. The program is described in more detail below.

The **entry** procedure registers a nanotask context ID with the NIC using the `lcurcontext` CSR (in the example, it sets the context ID to be 0). The procedure also sets the priority of the context (again to 0, the most urgent) using the `lcurpriority` CSR. It then executes the command by setting the `lniccmd` CSR to value 1. `lniccmd` is a bit vector used by supervisor mode software to send commands to the NIC–in this case, it is used to tell the NIC to allocate an RX and a TX queue for context ID= 0 with priority level= 0. The `lniccmd` CSR can also be used to remove context IDs or to update the priority level.[4]

The **wait_msg** procedure waits for a message to arrive in the RX queue by polling the `lmsgsrdy` CSR until it is set by the NIC, indicating that the context has messages to process. While it is waiting, the application tells the NIC that it is idle by setting the `lidle` CSR during the polling loop. The NIC thread scheduler uses the idle signal to evict waiting threads and, instead, schedule a new thread with a new message waiting to be processed.

The **loopback_plus1_16B** procedure simply swaps the source and destination addresses by moving the application header (the first word of every message) from the head register (`t5`) to the tail register (`t6`), shown on line 20. It then increments every integer in the received message, appends them to the message being transmitted, and waits for the next message to arrive.

Applications that use variable length messages can use the message length (in the application header) to read the correct number of words from the network RX queue. If an application reads an empty RX queue, the resulting behavior is undefined - similar to reading an uninitialized variable.

3.3 NIC-Driven Thread Scheduling

In some specialized applications, thread scheduling might not be needed: Big HPC applications, for example, might pin a nanoservice to its own core for the duration of a run. But this won't work for a cloud provider who needs more economic sharing of nanoPUs by multiple nanoservices. Nanotask threads will therefore be switched in and out frequently, and context switches must be extremely fast, else we will lose all the low latency benefits of nanoservices.

[4]Although the `lcurcontext`, `lcurpriority` and `lniccmd` CSRs convey hardware commands, they serve a similar purpose to opening or closing a socket in a traditional operating system.

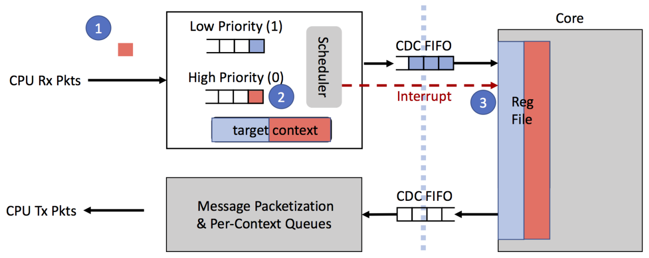


Figure 5: The steps that occur when a low priority thread is running on the core and then a message arrives for a higher priority context.

If nanoPU relied on software scheduling it would be too slow. The fastest best-of-breed software schedulers use 5μs timer interrupts [35, 47] which are much too slow for our $< 1\mu$s nanotasks.

And so, instead, nanoPU decides which thread to run next in hardware on the NIC, then tells the CPU core. The NIC keeps track of the highest priority nanotask with messages to process and interrupts the CPU to initiate a context switch under two conditions:

(1) A message arrives for a higher priority context than the one currently running on the core.
(2) The current context tells the NIC it is idle, and the NIC has messages for another context to process. This condition prevents an idle high priority context from starving a low priority context with messages waiting to be processed.

Figure 5 shows what happens when a message arrives for a higher priority context than the one currently running on the core. In steps 1 and 2, the arriving message is placed in the RX queue for the high priority context. The NIC scheduler tells the CPU that a high priority message is waiting by placing the context ID in the `ltargetcontext` CSR. In step 3, the NIC scheduler interrupts the CPU, causing a trap into the nanoKernel, which reads the `ltargetcontext` CSR and performs a switch to the high priority context. At this point, the high priority context is running on the core and is able to process the message in its RX queue.

Note that there may be words in the clock-domain crossing (CDC) FIFO for the low priority context when the NIC initiates a context switch. Since these words have not yet been read by the low priority context, they must be pushed back into the low priority RX queue. Fortunately, it is easy to implement a FIFO that is able to “unread” the last few words that were read out.

Nick: I propose we axe this paragraph to save space and update the figure to remove the CDC. it is an implementation detail we don't need here

When the high priority context finishes processing the message, it tells the NIC that it is now idle by writing to a dedicated CSR called `lidle`. The NIC initiates a context switch back to the low priority context because it still has messages to process.

3.4 Transport Termination in the NIC

If nanoPU is to meet our extremely aggressive latency target of $< 1\mu s$ from when an RPC message arrives from Ethernet until a response is sent back out, there is clearly no time to run a traditional transport networking stack in software.

Therefore, nanoPU runs a low latency transport protocol in hardware in the NIC. But we don't mandate a specific transport protocol; we seem to be entering an era when different cloud providers prefer different transport protocols [? ? ?] and so we aim to provide some choice, within our tight latency budget.

The NIC therefore places minimal constraints on the transport protocol, while providing programmable hardware blocks to allow some choice by the network owner. We do assume that the transport layer provides a reliable message abstraction to each thread. It therefore must handle retransmissions and decide when packets should be sent.

Nick: I will work on this part more in the morning. I think it can be shortened. If anyone wnats to take a shot, go for it....

This section describes the hardware mechanisms in the nanoPU architecture that enable the NIC to terminate a low latency transport protocol such as Homa [41] or NDP [27]. We describe the hardware mechanisms, however the implementation and evaluation of a specific transport protocol is outside the scope of this paper and is a topic for future research.

By terminating the transport protocol in hardware, the overhead required for applications to send and receive messages (especially small ones) is significantly reduced. While it is expensive for software to handle transport tasks such as per-message (or per-RPC) timers, packet retransmissions, message reassembly and packetization; hardware can implement these tasks very efficiently.

The key components that enable the NIC to terminate a transport protocol are described below.

Event-driven PISA Pipeline [29]. This programmable data-plane architecture enables us to write programs that process data-plane events in the background of data packet processing, that is, without affecting the rate at which data packets are processed. It does this by scheduling and aggregating memory accesses. This mechanism helps to enable transport logic processing which must perform more sophisticated stateful operations than basic packet forwarding.

Timer Event Generation Module. This is a hardware mechanism within the aforementioned event-driven PISA pipeline that is able to maintain N timers (e.g. one per active message or one per active RPC). The timer module supports the following three operations per-timer:

- Schedule - add a new timer
- Reschedule - restart an existing timer
- Cancel - remove the state associated with an existing timer

All timers must be constrained to have the same period in order to make the hardware design scalable. We do not anticipate this to be a major limitation. These timer events are processed in the background of data packet processing and are used to determine when a data packet retransmission must be sent or when a message (or RPC) has expired and its state must be cleaned up.

Programmable Packet Generation Module. This module is used to generate acknowledgement and/or message completion packets.

Packetization Buffer. This module buffers messages sent from the CPU and generates packets that are subsequently processed by the event-driven PISA pipeline before being transmitted. It also supports retransmitting data packets within a message when a packet drop is detected.

Message Reassembly Buffer. This module is able to reassemble potentially multi-packet messages with duplicate packets into a single message that is then delivered to the appropriate RX queue for the destination context.

4 NANOPU PROTOTYPE

In order to evaluate the expected performance of a nanoPU, we built a prototype in Chisel [11] and C on top of the open source RISC-V Rocket Core system [7] and evaluated performance using cycle-accurate hardware simulations [57].

Figure 6 shows a high-level block diagram of the prototype's most relevant components. The prototype implements the main components of the NIC described in §3 and connects it to a slightly modified RISC-V Rocket Core. The prototype implements the following aspects of the nanoPU architecture:

- The NIC Datapath, without hardware support for a transport protocol, encryption/decryption or MAC processing.
- The NIC-Core Interface, including the small changes to the core required to add the fast path into the register file and the NIC Message Reassembler and Packetizer's per-context FIFOs.
- The NIC hardware thread scheduler and the nanoKernel.

Since the NIC and core run at the same clock rate in our prototype, we have no need for clock domain crossing (CDC) FIFOs in the NIC-Core Interface. Furthermore, since the prototype does not implement a transport protocol, we assume that all messages sent and received by the applications consist of a single Ethernet packet, which we believe will be true for the vast majority of nanoservice applications.

To provide a useful reference, we compare the performance of our nanoPU prototype to a RISC-V Rocket Core with a DMA-based NIC called IceNIC [37]. Figure 7 depicts a

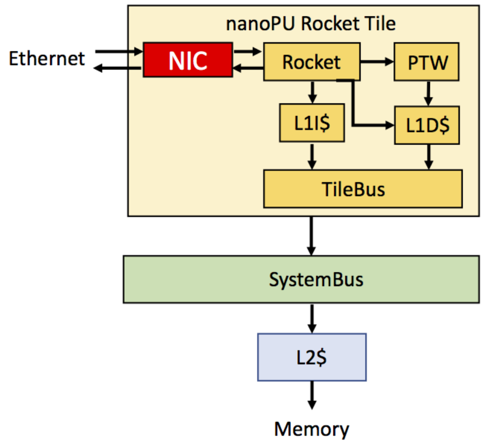


Figure 6: Block diagram of nanoPU RISC-V prototype.

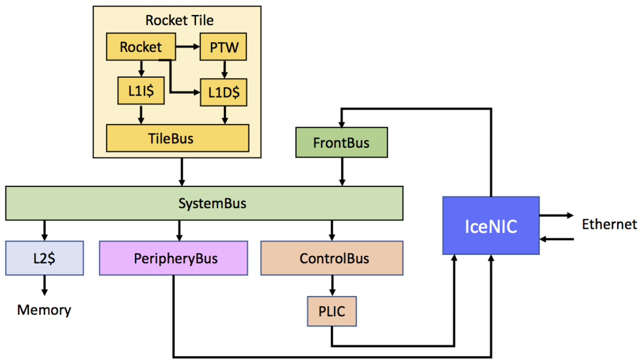


Figure 7: Block diagram of a traditional RISC-V system with a DMA-based NIC called IceNIC.

block diagram of the IceNIC RISC-V system. IceNIC behaves in a similar fashion to Intel DDIO based NICs. That is, it uses DMA to move network packets directly between the NIC and the CPU's last level cache. Since the IceNIC RISC-V design is an integrated solution, it does not include a PCIe bus–thus we expect it to exhibit lower latency than a typical modern NIC.

We still made one change to the original RISC-V IceNIC design to make up for the standard RISC-V ISA's omission of byte reversal instructions. Since this operation is extremely common in network applications, and because Intel processors have hardware support for this type of operation, we added these instructions to our RISC-V processor in order to accelerate IceNIC applications. This gives a more fair comparison with nanoPU. These instructions are not needed for nanoPU applications because the NIC swaps the byte order of message data before making it available to the application.

Steve: My main concern with using IceNIC as a baseline comparison is

	RX (Gbps)	TX (Gbps)
nanoPU	116	186
RISC-V w/ IceNIC	14	28

Table 1: TX and RX throughput achievable by both our nanoPU prototype and a traditional RISC-V core with IceNIC for 64B packets.

5 EVALUATION

This section describes our evaluation methodology and results. These evaluations assume a 3 GHz clock, which is a typical CPU clock speed.

Steve: change IceNIC label to RISC-V w/ IceNIC and change L-NIC label to nanoPU in Figure 8

5.1 Microbenchmarks

This section describes a set of microbenchmarks that characterize various aspects of the nanoPU architecture.

Timing Analysis. We synthesized both our nanoPU RISC-V prototype as well as a standard RISC-V core with a traditional NIC (IceNIC) to a modern FPGA (Xilinx Ultrascale+) in order to analyze the critical paths in each design. We found that the critical path in both designs is in the L2 cache, and hence our nanoPU prototype is able to achieve the same clock rate as a traditional RISC-V system.

Basic Latency/Throughput Performance Analysis. The latency of the nanoPU's TX path is 11 cycles for a single word message, which is measured from the cycle when an application writes the word to when the word is placed on the network (not including the MAC processing). The corresponding latency of the RX path is 6 cycles. These latency measurements do not include MAC processing. Figure 8 shows the loopback latency for various packet lengths for both our nanoPU prototype and the traditional DMA-based IceNIC. The nanoPU provides about a 4× latency reduction for small packets and a much smaller speedup for large packets due to the store-and-foreword nature of the current nanoPU prototype's TX path. In the case of IceNIC, the CPU is mostly idle during these simple loopback experiments because it only needs to instruct the NIC's DMA engine to send/receive packets and to swap source an destination header fields. However, in the case of the nanoPU, the CPU is fully utilized because it touches every byte of every packet.

Table1 shows the TX and RX throughput for 64B packets for both our nanoPU prototype and RISC-V with IceNIC. The nanoPU is able to achieve 6-8× higher throughput than the traditional network interface. This performance gain is mainly a result of the significantly reduced per-packet overheads in the nanoPU. A nanoPU application does not need to instruct a DMA engine to send/receive packets, it simply reads and writes the NIC queues directly.

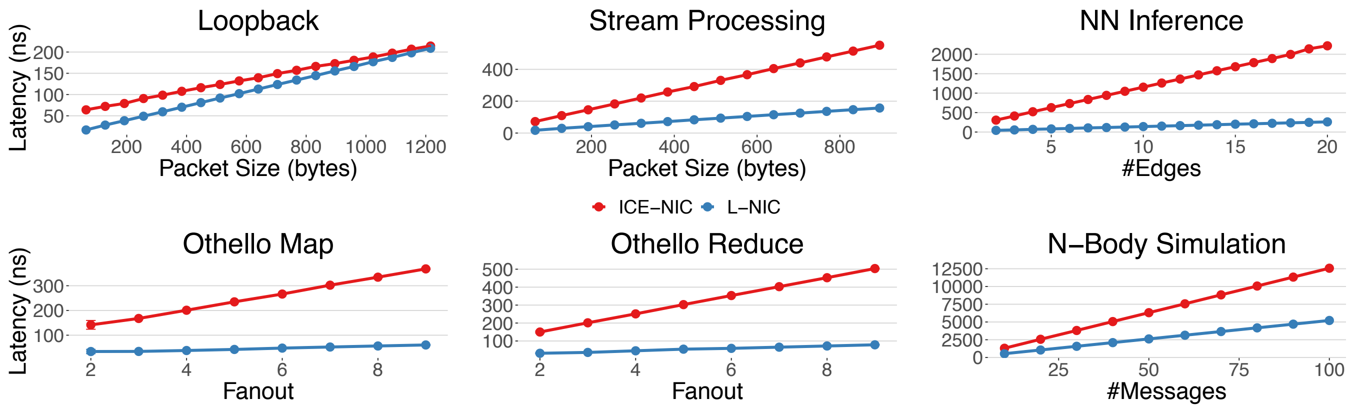


Figure 8: Nanoservice application latency comparisons for nanoPU vs a traditional RISC-V core with IceNIC.

nanoKernel Scheduler Latency. We measure the latency of the nanoKernel scheduler as the time from when an interrupt fires to the time when the first instruction of the newly selected thread is executed. We find that the nanoKernel scheduler latency is 60 ns (180 clock cycles).

Steve: How does this compare to the latency of modern schedulers?

5.2 Bare Metal Application Evaluations

This section describes the reduction in average latency as a result of the nanoPU's hardware fast path to the core of the CPU. For our evaluations we consider 4 example nanoservice applications, each of which are implemented as bare metal apps on top of both our nanoPU prototype and a traditional IceNIC-based processor.

NFV Style Streaming Application. This simple application treats the arriving message as a list of 8 byte unsigned integers and increments each integer then returns the resulting message back to the sender. The latency is measured from the time when the first byte of the request enters the NIC to when the last byte of the response leaves the NIC. The nanoPU is able to achieve a 3.5-4× speedup over the IceNIC system across all packet lengths, as shown in Figure 8.

The most significant difference between this application and the loopback application is that the CPU is forced to touch every byte of every packet. This means that for the IceNIC implementation, every byte of the packet must be copied from memory into registers, sent through the ALU, and then copied back into memory before the DMA engine can send out the final packet. These operations lead to a lot of overhead and extra latency for IceNIC, none of which exist for the nanoPU because it never needs to touch memory.

Neural Network Inference. This application can be implemented using nanoservice where each node of the neural network is running on a separate nanoPU core. This exploits the maximum amount of model parallelism, minimizing the latency required to perform an inference task. Since the computation required by each node is essentially a running multiply-accumulate operation, it is reasonable to expect that the working set and message processing time will easily satisfy the constraints required to be considered a nanoservice. Especially if the weights and data use reduced precision integers, as is typical in today's inference tasks [48].

We demonstrate this application by implementing the multiply-accumulate operation for a single node in a neural network. The node receives N weight messages and N data messages, one for each incoming edge, then multiplies the corresponding data and weights and accumulates the result. When all messages have been processed it sends a response message with the final result. The latency is measured from the first byte of the first incoming message to the last byte of the response. Figure 8 shows that the nanoPU implementation is able to achieve about an order of magnitude latency reduction over the traditional IceNIC hardware. The main reasons for the nanoPU's performance improvement are twofold: (1) Packet data does not need to be copied into registers before using the ALU, and (2) nanoPU has a much higher RX throughput for small packets (as shown in Table 1).

Othello Map/Reduce. We re-implemented the Othello application (as described in [30]), comparing its performance on the nanoPU with a traditional RISC-V core and IceNIC. An Othello board can be represented using BitBoards [53] with just two 64-bit unsigned integers and hence the working set of the nanotasks are all L1 cache resident. The set of possible next moves can be calculated in less than one microsecond, as demonstrated in [30].

We measured the latency of the "map" and "reduce" functions as a function of the *fanout*, which is the number of possible next moves given an initial board state. Because of the very short execution latency and the nanoPU's superior

TX/RX throughput, the nanoPU reduces the time for both functions by a factor of 4 – 6×, as shown in Figure 8.

N-body Simulation. This large class of applications is extremely well-suited to implementation using nanoservices. An N-body simulation is a type of physical simulation commonly used in scientific computing and is typically run on HPC clusters. Astronomers uses N-body simulations to model the gravitational interaction of celestial bodies. These simulations are very computationally heavy, especially the steps that compute the gravitational force each body exerts on every other body. A commonly used data structure for implementing an N-body simulation is a quad-tree, as described in the Barnes-Hut algorithm [26]. This algorithm reduces the computational complexity required for the gravity computation step from $O(N^2)$ to $O(\log N)$. A naive nanoservices implementation of the Barnes Hut algorithm would simply implement each node of the quad tree as a separate nanotask. In this case, the state required by each node is very minimal as it mainly consists of the node's position and mass, which is easily cache resident. The computation required by each node consists of computing the force that the node exerts on the body indicated in the arriving message, which requires an evaluation of the following equation:

$$F_G = G\frac{m_1 \cdot m_2}{r^2}$$

Where, G is the gravitational constant, m_1 and m_2 are the masses of the two bodies, and r is the distance between them. Thus, the computation only requires a few floating point operations and the response time for each message is easily sub-microsecond.

We implemented a single node of an N-body physical simulation to evaluate the performance gain provided by the nanoPU. The node receives N update messages which indicate the position of each body, and then evaluates the equation shown above to compute the gravitational force. The result is included in a response message that is sent back into the network. Latency is measured from when the first byte of the first request message enters the NIC to when the last byte of the last response message leaves the NIC. As shown in Figure 8, the nanoPU provides a 2.5× latency reduction over the traditional IceNIC system.

5.3 Thread Scheduling Evaluations

This section demonstrates the reduction in tail latency as a result of the NIC-driven thread scheduling policy used in our nanoPU prototype. We compare our NIC-driven thread scheduling policy, which was described in §3.3, to a more traditional Linux-style, timer driven thread scheduling approach. In the timer-driven thread scheduling approach, timer interrupts are configured to fire periodically and NIC interrupts are disabled. When an interrupt fires, the processor traps into the nanoKernel which then swaps to the context indicated by the NIC (i.e. the highest priority context with messages to process). The key difference between the NIC-driven thread scheduling and the timer-driven thread scheduling approach is that in the later, context switches can only happen on periodic time boundaries.

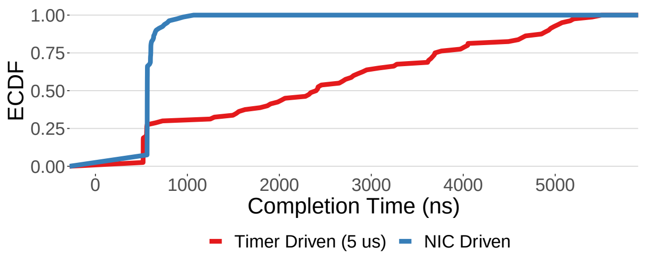


Figure 9: Latency reduction as a result of NIC-driven thread scheduling relative to timer-driven thread scheduling.

We evaluate these scheduling policies by pinning one high priority context and one low priority context on the core, then sending in a stream of randomly interleaved high and low priority requests. Each application will service the request for 500 ns then send back a response. The inter packet gap of arriving messages is also configured to be about 500 ns. Figure 9 shows the distribution of response times for the high priority messages for both scheduling policies. We see that NIC-driven scheduling reduces the tail response latency by a factor of about 5.5× and the standard deviation of the response times is reduced by almost 20×. This result should not be surprising because the NIC-driven scheduling policy is able to immediately evict the low priority thread once a high priority message arrives, whereas the timer-driven policy must wait for a timer interrupt. Furthermore, we also measure that the NIC-driven scheduling policy is able to achieve slightly higher throughput for the low priority application. This is because the timer-driven policy causes many traps into the nanoKernel for no reason, degrading its throughput.

We expect that both of these scheduler implementations are substantial improvements over what is commonly used in today's systems because they both use scheduling decisions made by the NIC hardware, whereas scheduling decisions in modern systems are made by slow software. The purpose of this evaluation is simply to demonstrate the benefit of allowing the NIC to drive the context switch logic.

5.4 Large Scale Nanoservice Evaluation

In order to demonstrate the expected performance gains of a large scale nanoservice deployment we will use the N-body simulation application and the naive nanoservice implementation described in §5.2. Using this approach, the root node

	Avg Gravity Computation Time
Nanoservices	4 ms
ChanGa	30 sec

Table 2: Average gravity computation time in an N-body simulation of 80K bodies. Compares a theoretical nanoservices deployment to a real implementation using ChanGa [31].

of the quad tree becomes the bottleneck, because it must process one message from each body being simulated. In this case, the root node dictates the total runtime of the gravity computation. We compare the theoretical performance of this naive nanoservice application with the performance obtained by using a commonly used N-body gravitational simulator, ChanGa [31]. Table 2 shows the average gravity computation time for 80K bodies using both the ChanGa simulator (with recommended parameter settings) and the naive nanoservices implementation. We obtain the expected nanoservice performance by extrapolating the performance of the bare metal application N-body node, evaluated in §5.2, to 80K messages. This mimics the processing required by the root node of the quad tree. We expect that even this naive nanoservice implementation would achieve about 3 orders of magnitude reduction in total gravity computation time. Further performance gains can be obtained by replicating certain nodes of the quad tree. The main reason for this massive performance gain is because modern N-body simulators, like ChanGa, are not designed to perform extremely fine grained parallel processing in the way that nanoservices are. Note that this preliminary evaluation assumes that the nanoPU's transport protocol is able to handle a 80K-to-1 incast without ever overflowing or underflowing the receiver buffer. This type of massive incast is not special to this one application, but we expect that this will be common across many nanoservice applications. Designing transport protocols to deal with this degree of incast may be challenging and is a subject of current research.

6 DISCUSSION

This section discusses a few additional details about the nanoPU's NIC-Core interface, as well as some thoughts regarding the nanoPU as a domain specific processor and using it to run nanoservice applications.

6.1 nanoPU NIC-Core Interface Design Considerations

As mentioned in §4, the changes to the RISC-V Rocket core that were required to support our fast path to the register file were very minimal. The most significant change resulted from the fact that reading the network RX queue in the CPU pipeline's decode stage causes a state change which must be undone when there is a pipeline flush resulting from a branch misprediction. We handle this situation by simply modifying the RX queue to be able to effectively "unread" the last two words that were removed.

Our nanoPU design reserves two general purpose registers for the head and tail of the network RX and TX queues respectively. We also explored an alternative design where the head and tail registers are instead implemented using control status registers (CSRs). This design decision of whether to use GPRs or CSRs has a number of trade offs. Using GPRs often results in lower latency for applications because GPR values feed directly into the CPU's ALU and thus avoids the need to copy every word of every message from the CSR to a GPR before using the ALU. However, if the number of GPRs is very limited, for example in a light weight embedded system, then sacrificing two GPRs may result in an excessive amount of register spill into memory, thus inflating latency. Our experiments suggest that reducing the number of GPRs in the RISC-V Rocket core from 32 to 30, does not have a significant impact on the amount of register spill for our benchmark suite of nanoservice applications.

Our nanoPU prototype is built on top of the RISC-V Rocket core, which is a simple 5-stage, in-order RISC processor. While our prototype required very minor modifications to the CPU pipeline, the changes may need to be more invasive on an out-of-order RISC processor in order to ensure that words are read from the RX queue in the correct order.

Our nanoPU design is optimized to minimize the latency from the network into the *integer* register file. Many scientific computing applications, such as the N-body simulation described in §5.2, require use of messages with *floating point* numbers. Since the RISC-V ISA does not define instructions to directly copy bytes from an integer register to a floating point register without converting the number format, the number must first be stored into memory and then loaded into the floating point register file, resulting in additional latency. We advocate for a minor change to the RISC-V ISA in order to support instructions that directly copy bytes between the integer and floating point register files.

6.2 nanoPU as a Domain Specific Processor

We consider our nanoPU prototype to be a first step towards building a domain specific processor for compute-intensive distributed applications. In this paper, we focus on what we believe to be the most important components of the architecture: the network interface and thread scheduler. That being said, we believe that future research on this topic should consider tailoring other aspects of the architecture including the memory hierarchy, on-chip network, CPU pipeline, and ISA for compute-intensive distributed applications.

Furthermore, it is not clear that writing C code is the best way to program the nanoPU. Network applications written for the nanoPU must process message data in FIFO order, have minimal memory requirements, and all communication is done via explicit message passing. Perhaps there is a higher-level domain specific language that developers can use to facilitate writing stateful message processing applications. This could be an area of interesting research.

As a result of being a domain specific processor, there are classes of applications that we believe to be ill-suited for the nanoPU, such as *memory-intensive* distributed applications. For example, a Key-Value store application [43] may or may not be well suited for the nanoPU, depending on the scale and latency requirements. If the size of the database is small enough to fit in the L1 caches of a reasonable amount of nanoPU cores then it would likely be good fit and would provide substantially lower latency than a DRAM based Key-Value store. However, if the size of the database is very large, then the nanoPU's fast path will probably not provide much benefit, although the nanoPU's NIC-driven scheduling policy may still be beneficial.

6.3 Running Nanoservice Applications

Rather than using a standard Linux distribution to test our scheduling policies we decided to use a minimal operating system of our own design - the nanoKernel. Our main reason for doing this was to eliminate all of the general purpose logic in Linux in an effort to measure the minimum possible scheduling latency. That being said, our current nanoKernel implementation is lacking a few Linux features that we believe would be beneficial for running nanoservice applications. This includes features such as virtual memory and multiple privilege modes. That is, all applications in the nanoKernel currently execute in the same address space and at the same privilege level. We expect that this will change in future iterations of the nanoKernel.

In addition to the nanoKernel and hardware thread scheduler, we suspect that we may also need a runtime system that ensures the high utilization of nanoPU cores and NICs. This runtime system would be responsible for dispatching nanoservice apps to nanoPU cores, monitoring the utilization of the cores and NICs, along with the status of contexts. If some cores or NICs are over/under-utilized for a long period of time, it will need to rebalance the system. If some contexts are starved for a long period of time, again it may need to address this.

7 RELATED WORK

Smart NICs (i.e. NICs with CPUs on them) have become a popular trend recently [10, 39, 42]. They are hell bent on offloading processing logic in order to free up the host CPUs so that those cycles can be sold to paying customers. Smart NICs are terrible for latency sensitive applications, in fact, they increase latency. In order to make nanoservices practical, we believe that the NIC should be a domain specific architecture designed to minimize latency to the core, perform thread scheduling, and implement a transport protocol. Throwing a CPU on a NIC is not what we would consider to be a domain specific architecture.

Recent trends have emerged which suggest that it is a good idea to break up large monolithic applications into smaller compute units. The concept of microservices emerged about a decade ago with the goal of creating more maintainable infrastructure, not necessarily to improve performance. As more and more applications were developed using the microservices framework, it became apparent that we need to find a way to minimize the latency overhead of RPCs and make their performance predictable.

Serverless compute [8, 9, 25] is another recent trend that enables developers to implement applications in a more fine grained manner than has every been feasible in the past. Serverless compute infrastructure has been used to implement fine grained video encoding [23], compilers [22], and scientific computing applications [34]. The nanoservices framework takes this approach to the absolute extreme and we believe that the nanoPU will help make this practical.

Steve: Do we want to mention isolates?

There are are number of recent works that attempt to minimize tail RPC completion time: Shinjuku [35], Shenango [47], RPCValet [18]. These works assume that incoming requests must be load balanced across cores. We believe that this assumption will generally be false for the vast majority of nanoservice applications. For the most part, nanoserver threads will need to directly send messages between one another. That is, each message will indicate its destination thread ID and hence the core ID. This is because in most cases, only one thread will have the state needed to processes a certain message. That being said, there will be some nanoservice apps that will want to send messages which can be processed by any one of N replicas. In this situation, we believe that the load balancing decision should be made at the machine which is sending the request or within the network, and not on machine(s) where the replicas reside. One reason for this belief because the replicas many not even reside on the same machine, much less attached to the same NIC. Another reason for this belief is that load balancing decisions made at the destination machine are already too late, especially with the massive degrees of incast that we expect nanoservices to exhibit. Ideally, the nanoserver thread that is about to send the message will be aware of who is about to send a message to each replica so that it can make the most informed load balancing decision or perhaps decide to delay sending the message in the first place in order to avoid drops within the network or the destination NIC, wasting network bandwidth. This would require an immense amount of coordination, which is perhaps achievable with tightly synchronized clocks across all machines. This is our current area of active research.

We are not the first to propose an integrated network interface. Scale-Out NUMA [44] integrates the network interface into the machine's local cache coherence hierarchy in order to accelerate RDMA style applications. This approach is similar in spirit to the recent Compute Express Link (CXL) [17] standard, which is designed to maintain memory coherence between the CPU memory and the memory on PCIe attached devices. Both of these approaches are sub-optimal for nanoservice applications, which require minimal latency between the network and the CPU pipeline, not memory. Thus, the nanoPU integrates the network interface and the CPU pipeline.

The design of the nanoPU's network fast path into the CPU register file is inspired by the J-machine [19] from 1989. This machine proposed to communicate between processors on the same chip via the register file in order to minimize inter-processor communication latency. The approach was eventually abandoned because it was believed to be challenging to maintain isolation between contexts running on the same core. We believe that our proposal to use per-context queues in NIC hardware overcomes this challenge.

8 CONCLUSION

We believe that compute-intensive distributed applications would be designed very differently than they are today if utilizing remote computation is predictably only slightly more expensive than local computation. We believe that these applications would be implemented in an extremely fine grained manner - using the nanoservices framework. The NanoPU described in this paper is the first step towards making nanoservices a reality. There is still much work to be done. For example, designing an optimal transport protocol for nanoservices, rethinking the NanoPU's memory architecture, as well as potentially its instruction set architecture (ISA).

REFERENCES

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, 2015.

[2] Adiga, N. R., Almási, G., Almasi, G. S., Aridor, Y., Barik, R., Beece, D., Bellofatto, R., Bhanot, G., Bickford, R., Blumrich, M., et al. An overview of the BlueGene/L supercomputer. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (2002), IEEE, pp. 60–60.

[3] Ajayi, T., Al-Hawaj, K., Amarnath, A., Dai, S., Davidson, S., Gao, P., Liu, G., Lotfi, A., Puscar, J., Rao, A., et al. Celerity: An open source risc-v tiered accelerator fabric. In *Symp. on High Performance Chips (Hot Chips)* (2017).

[4] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal near-Optimal Datacenter Transport. In *ACM SIGCOMM* (2013).

[5] Ao, L., Izhikevich, L., Voelker, G. M., and Porter, G. Sprocket: A Serverless Video Processing Framework. In *ACM SoCC* (2018).

[6] Apache Hadoop. https://hadoop.apache.org/. Accessed on 02/03/2020.

[7] Asanovic, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[8] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed on 06/28/2019.

[9] AWS Lambda. https://aws.amazon.com/lambda/. Accessed on 06/28/2019.

[10] Aws nitro system. https://aws.amazon.com/ec2/nitro/. Accessed on 02/04/2020.

[11] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012* (2012), IEEE, pp. 1212–1221.

[12] Barroso, L. A., Dean, J., and Holzle, U. Web Search For a Planet: The Google Cluster Architecture. *IEEE Micro 23*, 2 (2003), 22–28.

[13] Bohnenstiehl, B., Stillmaker, A., Pimentel, J. J., Andreas, T., Liu, B., Tran, A. T., Adeagbo, E., and Baas, B. M. Kilocore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits 52*, 4 (2017), 891–902.

[14] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (2014), 87–95.

[15] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 99–110.

[16] Carreira, J., Fonseca, P., Tumanov, A., Zhang, A., and Katz, R. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *ACM SoCC* (2019).

[17] Compute express link: Breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/. Accessed on 2020-01-29.

[18] Daglis, A., Sutherland, M., and Falsafi, B. 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* (2019), pp. 35–48.

[19] Dally, W. J., Chien, A., Fiske, S., Horwat, W., and Keen, J. The j-machine: A fine grain concurrent computer. Tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE MICROSYSTEMS RESEARCH CENTER, 1989.

[20] Dean, J. Software engineering advice from building large-scale distributed systems. https://static.googleusercontent.com/media/research.google.com/en/us/people/jeff/stanford-295-talk.pdf. Accessed on 02/04/2020.

[21] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI* (2004).

[22] Fouladi, S., Iter, D., Chatterjee, S., Kozyrakis, C., Zaharia, M., and Winstein, K. A thunk to remember: make-j1000 (and other jobs) on functions-as-a-service infrastructure, 2017.

[23] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. Encoding, Fast and Slow: Low-latency Video Processing Using Thousands of Tiny Threads. In *USENIX NSDI* (2017).

[24] Glassner, A. S. *An introduction to ray tracing.* Elsevier, 1989.

[25] Google Cloud Functions. https://cloud.google.com/functions/. Accessed on 06/28/2019.

[26] Grama, A. Y., Kumar, V., and Sameh, A. Scalable parallel formulations of the barnes-hut method for n-body simulations. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing* (1994), IEEE, pp. 439–448.

[27] Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., and Wójcik, M. 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* (2017), pp. 29–42.

[28] Hennessy, J. L., and Patterson, D. A. *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2017.

[29] Ibanez, S., Antichi, G., Brebner, G., and McKeown, N. Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (2019), pp. 133–140.

[30] Ibanez, S., Shahbaz, M., and McKeown, N. The case for a network fast path to the cpu. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (2019), pp. 52–59.

[31] Jetley, P., Gioachin, F., Mendes, C., Kale, L. V., and Quinn, T. Massively parallel cosmological simulations with ChaNGa. In *2008 IEEE International Symposium on Parallel and Distributed Processing* (2008), IEEE, pp. 1–12.

[32] Jin, X., Li, X., Zhang, H., Foster, N., Lee, J., Soulé, R., Kim, C., and Stoica, I. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), pp. 35–49.

[33] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 121–136.

[34] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. Occupy the cloud: Distributed computing for the 99%. In *ACM SoCC* (2017).

[35] Kaffes, K., Chong, T., Humphries, J. T., Belay, A., Mazières, D., and Kozyrakis, C. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 345–360.

[36] Kalia, A., Kaminsky, M., and Andersen, D. G. Datacenter RPCs Can Be General and Fast. In *USENIX NSDI* (2019).

[37] Karandikar, S., Mao, H., Kim, D., Biancolin, D., Amid, A., Lee, D., Pemberton, N., Amaro, E., Schmidt, C., Chopra, A., 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)* (2018), IEEE, pp. 29–42.

[38] Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., and Wobker, L. J. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM* (2015).

[39] Mellanox bluefield-2. https://www.mellanox.com/products/bluefield2-overview. Accessed on 02/04/2020.

[40] Miao, R., Zeng, H., Kim, C., Lee, J., and Yu, M. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 15–28.

[41] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. 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* (2018), pp. 221–235.

[42] Naples dsc-100 distributed services card. https://www.pensando.io/assets/documents/Naples_100_ProductBrief-10-2019.pdf. Accessed on 02/04/2020.

[43] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. Scaling memcache at facebook. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)* (2013), pp. 385–398.

[44] Novakovic, S., Daglis, A., Bugnion, E., Falsafi, B., and Grot, B. Scale-out numa. *ACM SIGPLAN Notices 49*, 4 (2014), 3–18.

[45] Nvidia: GeForce Now. https://www.nvidia.com/en-us/geforce-now/. Accessed on 02/04/2020.

[46] OpenFaaS: Serverless Functions, Made Simple. https://www.openfaas.com/. Accessed on 02/03/2020.

[47] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 361–378.

[48] Park, E., Yoo, S., and Vajda, P. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 580–595.

[49] PerfKit: gRPC Performance. https://performance-dot-grpc-testing.appspot.com/explore?dashboard=5652536396611584&widget=1747498211&container=897341821. Accessed on 02/03/2020.

[50] Rapaport, D. C., and Rapaport, D. C. R. *The art of molecular dynamics simulation.* Cambridge university press, 2004.

[51] Ricart-Sanchez, R., Malagon, P., Alcaraz-Calero, J. M., and Wang, Q. Hardware-accelerated firewall for 5g mobile networks. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (2018), IEEE, pp. 446–447.

[52] Risc-v. https://riscv.org/. Accessed on 2020-01-29.

[53] San Segundo, P., Galan, R., Matia, F., Rodriguez-Losada, D., and Jimenez, A. Efficient search using bitboard models. In *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)* (2006), IEEE, pp. 132–138.

[54] Serverless computing in financial services: High performance computing—with more speed, accuracy and flexibility. https://www.ibm.com/downloads/cas/Q93RMQMN. Accessed on 02/03/2020.

[55] Tofino. https://www.barefootnetworks.com/products/brief-tofino/. Accessed on 02/04/2020.

[56] Tootoonchian, A., Panda, A., Lan, C., Walls, M., Argyraki, K., Ratnasamy, S., and Shenker, S. Resq: Enabling slos in network function virtualization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), pp. 283–297.

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

[58] Zhang, W. *State-space Search: Algorithms, Complexity, Extensions, and Applications.* Springer Science & Business Media, 1999.