## TODO LIST

# The nanoPU: From Microservices to Nanoservices

Paper #52
12 Pages Body, 14 Pages Total


## ABSTRACT

Microservices have been shown to accelerate a broad class of applications by employing fine-grained parallelism using serverless tasks that run for just a few milliseconds. This paper takes the approach to the extreme: our goal is to enable a much finer-grained parallelism, with what we call *nanoservices*, built from *nanotasks* each running for less than 1 $\mu$s. Making this practical is primarily a networking problem. Our nanoservice Processing Unit (nanoPU), runs on a slightly modified CPU with a very low-latency network interface (NIC). The nanoPU is a domain-specific processor optimized to run sub-microsecond, cache-resident RPC nanotasks.

The nanoPU is a new networking-optimized compute platform for nanoservices. It contains a fast path from the network directly to the heart of a CPU core, minimizing communication latency, and it delegates thread scheduling and transport logic to the NIC, minimizing tail RPC-completion times and per-message overheads.

We designed 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. We also synthesized our design to run on an FPGA. The nanoPU reduces the average RPC request-response latency from the best reported (tens of microseconds) to below 1$\mu$s for nanoservice applications. We demonstrate NFV-like streaming applications running 3.5–4× faster and N-body simulations running four orders of magnitude faster than ChanGa [28] (an HPC simulation framework) on a modern system. The nanoPU additionally uses the NIC hardware to schedule high-priority threads so they start processing less than 100 ns after a message arrives, two orders of magnitude faster than the current state-of-the-art.

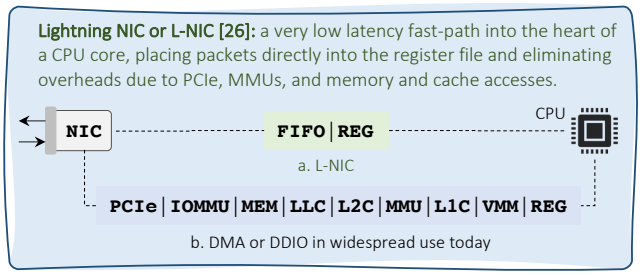


## KEYWORDS

Programmable NIC; PISA; and P4;


## 1 INTRODUCTION

Many distributed applications are routinely divided into fine-grained tasks that are executed simultaneously. The key is to harness as many cores as possible in order to minimize completion time. Recent advances in microservice and serverless computing (e.g., AWS Lambda [6], Azure Serverless Computing [8], Google Cloud Functions [22], and OpenFaaS [48]), have led to impressive speedup of parallel applications, including video encoding [19], video compression [4], and face recognition [13], by successfully employing thousands of CPU cores in parallel. Many applications could be accelerated even further if we break the application into tiny fine-grained fragments running on tens of thousands or even millions of cores. But, realistically, we quickly run into a minimum fragment size caused by the time taken to invoke a remote code fragment. Today, even with the best microservices and fastest RPCs, it takes milliseconds to execute code remotely. This limits the degree of parallelism in a cloud datacenter, making it impractical to harness millions of cores for just a short time to complete a large job quickly. So even though cloud data centers contain millions of CPU cores, any one application can only utilize a subset. We are interested in driving down the RPC time, to make it practical to harness many more CPUs simultaneously.




In the limit, if we segment an application into its smallest, indivisible software modules, *and if communication is instantaneous*, then an application could 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 [34, 51]. 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 $\mu$s 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 $\mu$s, then it makes sense for each object to run on its own core, and the simulation can complete one hundred times sooner.

Shahbaz: didn't get this point, a bit unclear

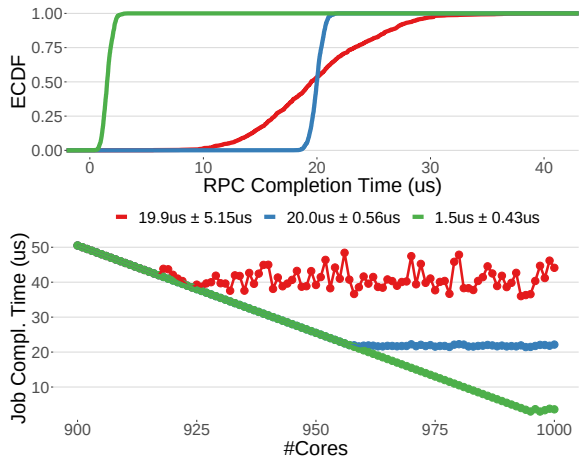



**Figure 1: Total job completion time of an example application with 1000 independent tasks (each with 500 $\mu$s processing time) for varying number of cores (i.e., one per RPC) under different RPC completion time distributions.**

More generally, consider a distributed application consisting of $N$ potentially parallelizable tasks, each of which can be executed locally (in $x$ $\mu$s) or on a remote core via RPC (in $r \cdot x$ $\mu$s, 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., $K = N - r$). Further increasing the degree of parallelism, $K$, 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 $(N-K) \cdot x = \max f_i$ 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 [15, 33, 50].

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 is designed for massive scale-out computing. Specifically, millions of nanoPUs could be deployed to run *nanoservices*: highly distributed, compute-intensive applications that process RPC requests in under 1 $\mu$s 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 [3, 24, 43] to reduce network congestion and minimize latency through switches, or thread scheduling to make sure an incoming RPC request starts executing promptly [33, 50]. 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 $\mu$s. This is mostly a networking problem, and 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 [26], 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 (not including the MAC).

(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 [24, 43].

(3) **Making it practical:** How can such an extreme approach to hardware and software be practically deployed, with minimal disruption? We have designed a hardware prototype, based on a RISC-V CPU [5], including 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, 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 [34]. 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 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 [5].
- 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 tail RPC-completion times by about 2–10× and 20×, respectively.

***Ethics.*** This work does not raise any ethical issues.

## 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*, managed by a light-weight *nanokernel*.
**C2:** Each nanotask is able to process and respond to network messages in under 1 $\mu$s.
**C3:** The working set of each nanotask fits in the L1 cache.[1]

Figure 2 highlights the key differences between nanoservices and two other major application development frameworks: monoliths and microservices. We believe that many applications can benefit from the nanoservices framework, including applications such as physical simulations [23, 52], neural-network inference and training [1], graphics processing [21], and state-space search algorithms [60]. As we will show in Section 5, we can reduce the total application runtime by up to 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

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

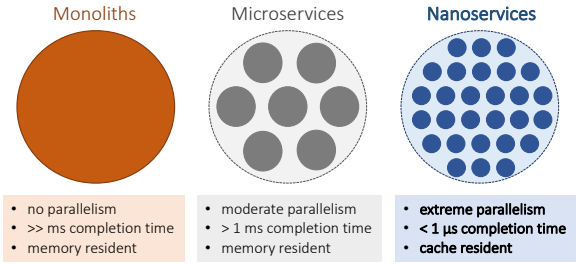



**Figure 2: A comparison of key characteristics of *nanoservices* vs. monoliths and microservices.**

are, ten and one hundred times slower than the L1 cache [17], respectively.

Building nanoservices requires more than simply dividing applications into nanotasks that access small amounts 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 [34, 51], 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—with low average and tail latency—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 [55]. The low-latency NIC is inspired by the recently proposed *Lightning NIC (L-NIC)* [26], 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 need, 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 [2, 10] and over a hundred 100 Gb/s

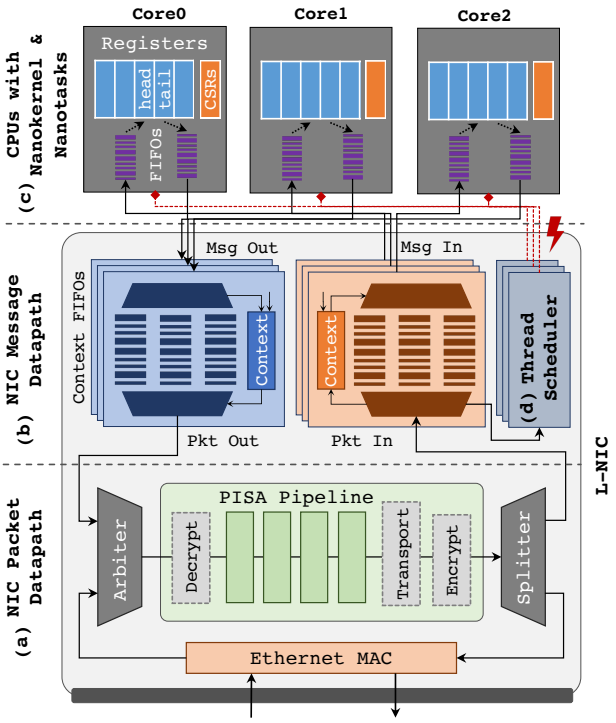



**Figure 3: The nanoPU architecture. The NIC includes an event-driven PISA pipeline, and provides an RPC message abstraction to a thread via dedicated RX/TX registers in the CPU. CPU cores run a nanokernel and user nanotasks.**

Ethernet interfaces;[2] in this example, the ratio would be five cores per NIC. Of course, many other ratios are possible and the ideal ratio depends on the application, technology, and economics.

The nanoPU is a domain-specific processor. With Moore's Law slowing down, new domain-specific processors are being widely used as accelerators for specific, high-volume workloads, such as graphics [47], machine learning [1] and networking [12]. 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. General-purpose CPUs put memory in the network and attach compute to memory, which makes perfect sense for memory-intensive applications. 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. The nanoPU is designed 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 Packet Datapath:** A programmable event-driven PISA "match action unit" (MAU) pipeline [25] on the NIC to process packet headers as they arrive and depart, terminate tunnels, encrypt/decrypt and compress/decompress data; and a low-latency transport protocol in hardware, such as Homa [43] or NDP [24].
- **NIC Message Datapath:** 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. This reduces latency by 1–2 orders of magnitude; we do not believe there is a lower latency path to a running thread.
- **Hardware Thread Scheduling:** In addition to moving network messages into the CPU quickly, the nanoPU must also make sure that the appropriate application thread is running on the core so that it can start processing messages promptly. For this, nanoPU includes a very low-latency, thread scheduler on the NIC and a light-weight nanokernel on the CPU.

These characteristics together enable the nanoPU to process RPCs quickly and predictably, making it ideal for building compute-intensive, distributed applications.

## 3.1 The NIC Packet Datapath

The NIC packet datapath (Figure 3a) processes packets as they enter and leave the network. The centerpiece of the NIC packet datapath is an event-driven PISA pipeline [25]. The original PISA architecture, proposed in the RMT chip [12] and later used by Tofino [58], 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 [25] enhances the basic model to support other event types, such as packet drops, timers and state-dependent events. Section 3.4 describes how an event-driven PISA pipeline can directly support transport protocols in hardware, offloading per-packet processing from the CPU.

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

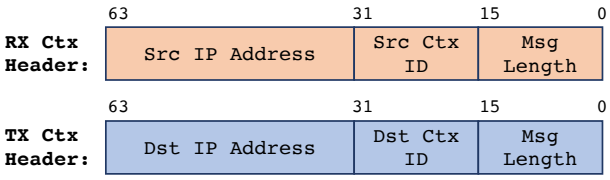



**Figure 4: NIC message (context) header formats.**

The PISA pipeline also performs protocol processing: depending on the loaded P4 program [11], it might add and remove Ethernet, IP, VXLAN, GRE, INT [36] and transport headers. We also program it to add or remove a small per-context header containing an IP address, a context ID, and the 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 [30], load-balancers [41], consensus protocols [29] and firewalls [54]). The L-NIC paper [26] describes how a PISA pipeline can accelerate nanoservices to search the Othello state-space.

## 3.2 The NIC Message Datapath

The nanoPU's NIC message datapath (Figure 3b) assembles arriving packets into a self-contained RPC message, then queues that message in a per-context FIFO until its destination thread is scheduled on a core by a nanokernel to be processed. The message then flows, one word at a time, into the core's register file. In the egress direction, messages sent by applications' threads are broken into packets before being sent to the NIC packet datapath.

We make two small modifications to the CPU core:

- Two general-purpose registers (GPRs) are now reserved for a special purpose: one is the `HEAD` of the network receive queue, 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 reserve registers via command-line options [49].
- New control status registers (CSRs) are added for out-of-band communication between the CPU and the NIC. These are used to configure the NIC with context IDs and to enable NIC-driven thread scheduling.

The NIC message datapath maintains transmit and receive FIFOs for each context (i.e., thread) that is currently pinned to each core. It ensures 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 the more traditional packet interface. This makes the nanoPU NIC hardware responsible for the transport layer, which includes breaking messages into packets, ensuring reliable delivery across the network, and performing message reassembly at the receiving end. To this end, messages between the NIC and core carry a small 8-byte header whose format is shown in Figure 4.

```
1  // Simple loopback & increment nanoservice
2  entry:
3    // Register context ID & priority with NIC
4    csrwi lcurcontext, 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
14   csrr  a5, lmsgsrdy
15   beqz  a5, idle
16
17 // Loopback and increment 16B message
18 loopback_plus1_16B:
19   mv t6, t5 // copy ctx hdr from head to tail
20   addi t6, t5, 1 // first data word + 1
21   addi t6, t5, 1 // second data word + 1
22   j wait_msg // wait for the next message
```

**Listing 1: Loopback with increment: a simple RISC-V assembly program for the nanoPU that waits for a 16 B message to arrive, increments each word of the message, and returns it to the sender.**

***A nanoPU program (loopback-increment).*** To illustrate how software on the nanoPU interacts with the NIC, Listing 1 shows a simple loopback-increment program in RISC-V assembly language. The program continuously reads 16-byte messages (two 8-byte integers) from the network, increments the integers, and sends the messages back to their sender. The program details are described below.

The `entry` procedure registers a context ID and its priority with the NIC by first writing a value to both the `lcurcontext` and `lcurpriority` CSRs, then subsequently writing the value 1 to the `lniccmd` CSR. The `lniccmd` CSR is a bit vector used by software to send commands to the NIC; in this case, it is used to tell the NIC to allocate an RX and TX queue for context ID = 0 with priority level = 0. The `lniccmd` CSR can also be used to remove context IDs or to update 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

[4]Registering a context ID with the NIC is roughly equivalent to opening a socket on a modern OS.

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 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 context header (the first word of every message) from the head register (`t5`) to the tail register (`t6`), shown on line 20 (Listing 1). 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 context 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.

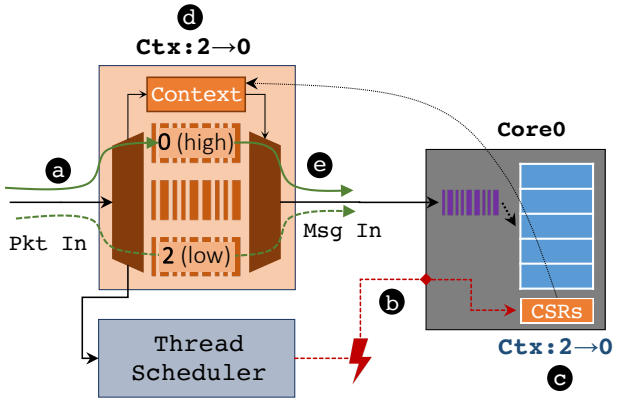



**Figure 5: The steps that occur when a low-priority thread running on a core is preempted by a higher-priority thread.**

## 3.3 NIC-Driven Thread Scheduling

In some specialized applications, thread scheduling might not be needed. Big HPC applications, for example, might pin every nanotask to a dedicated core for the duration of a run. But this would not work for a cloud provider who needs more economical sharing of nanoPUs by multiple nanoservice applications. Nanotask threads will therefore need to be switched in and out frequently, and context switches must be extremely fast. Otherwise, we will lose all the low latency benefits of nanoservices.

If nanoPU relied on software scheduling, it would be too slow. The fastest best-of-breed software schedulers use 5$\mu$s timer interrupts [33, 50] which are much too slow for our $< 1\mu$s nanotasks. And so, instead, the NIC hardware (Figure 3d) decides which thread to run next, and informs the nanoPU's minimal operating system—the nanokernel. 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 complete 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 step ⓐ, the arriving message is enqueued into the RX queue for the high-priority thread. In steps ⓑ and ⓒ, the thread scheduler tells the CPU that a high-priority message is waiting by updating the `ltargetcontext` CSR ($2 \rightarrow 0$) and firing an interrupt. The interrupt causes a trap into the nanokernel, which then reads the `ltargetcontext` CSR and performs a switch to the high-priority context in step ⓓ, updating the `lcurcontext` CSR. At this point, the high-priority context is running on the core and is able to process the message in its RX queue ⓔ. 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 then initiates a context switch back to the low-priority context because it still has messages to process.

## 3.4 Terminating Transport in the NIC

If the nanoPU is to meet our aggressive latency target of $< 1\mu$s from when an RPC message arrives over 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, with support from the NIC's programmable pipeline. Note that transport protocols for low-latency applications (e.g., DCQCN for RDMA) are already fully implemented in hardware [27, 40] on state-of-the-art high-speed NICs available in today's market, vouching for the feasibility of a fully-offloaded transport protocol running in hardware.

We, however, don't mandate a specific transport protocol in this paper because we seem to be entering an era when different cloud providers prefer different transport protocols [38, 42, 61]. Instead we aim to provide some choices, 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 assume

that the transport layer provides a reliable message abstraction to each thread, as well as network congestion control. It therefore must handle retransmissions and decide when packets should be sent. To guide our design, we assume it must be possible for the NIC to be programmed to implement Homa [43] and NDP [24]. Between them, they mandate most of the building blocks we need, including reliable transmission, immediate startup rate, receiver driven scheduling, a message abstraction to the CPU, and data-trimming (in the switches).

We observe that realizing a transport protocol in hardware requires the following functions in the programmable pipeline of a NIC.

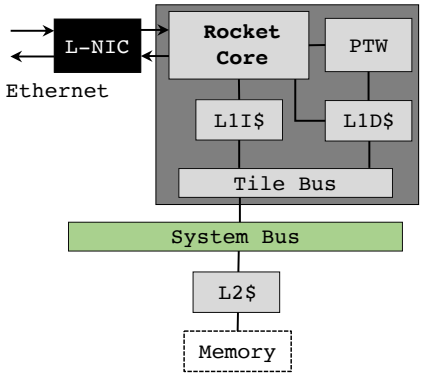
- Timers and timer-based event processing logic to realize various timer-based state transitions, such as retransmissions and timeouts.
- Pacers to rate-limit individual flows.
- State machines to maintain per-flow state, including current rate or congestion-window size, sequence and ack numbers, connection status, counters, etc.
- Packetization/retransmission buffer to break a message into packets and hold packets until they are acknowledged by a receiver.
- Reordering buffer to handle out-of-order packets.
- Packet generators to realize receiver-driven transport protocols that keep generating credit packets for senders.

The event-driven PISA pipeline [25] already provides most of the mechanisms we need for sophisticated stateful operations (e.g., state machines, timer events and packet generation), and can be extended to add support for message reassembly and retransmit buffers [53, 56]. Because nanoservice messages sizes will be very small, the amount of SRAM needed to realize these buffers will be sufficiently small for a cost- and power-efficient hardware implementation. With these mechanisms, we believe the nanoPU can run a complete transport stack on the NIC with very low latency. While we have a design for each of these building blocks, we will explain the details of those in a follow-up paper.

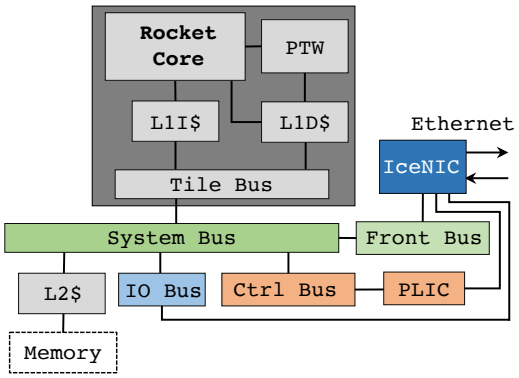
## 4 NANOPU PROTOTYPE

In order to evaluate the expected performance of a nanoPU, we built a prototype in Chisel [9] and C on top of the open source RISC-V Rocket core system [5] and evaluated performance using cycle-accurate hardware simulations [59].

Figure 6a 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:




**(a) The nanoPU Prototype**




**(b) Traditional RISC-V with IceNIC**

**Figure 6: Block diagram of (a) the nanoPU prototype and (b) a traditional RISC-V system with IceNIC.**

- The NIC packet datapath, without hardware transport protocol, encryption/decryption or MAC.
- The NIC message interface, including small changes to the core to add the fast path into the register file.
- The NIC hardware thread scheduler and nanokernel.

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 traditional DMA-based NIC called IceNIC [35]. Figure 6b depicts a 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 made one change to the original RISC-V IceNIC design to make up for the RISC-V ISA's omission of byte-order 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 change allows us to obtain a more fair performance comparison with our nanoPU prototype. 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.

# 5 EVALUATION

We evaluated our hardware design for a nanoPU running at 3 GHz, and compared its performance to an unmodified RISC-V core with a standard NIC (IceNIC), which we will call "Traditional."

## 5.1 Microbenchmarks

We wrote and ran microbenchmarks for three main aspects of our nanoPU design.

***Timing Analysis.*** We wanted to know if a nanoPU runs more slowly than the Traditional design. We synthesized both designs to run on a modern FPGA (Xilinx Ultrascale+). In both cases the critical timing path was the L2 cache, and hence our nanoPU prototype runs at the same speed as the unmodified RISC-V system.

***Basic Latency/Throughput Performance Analysis.*** Figure 7a compares the loopback latency, as a function of packet length, for both designs. For short packets, nanoPU latency is 4× shorter than for the Traditional design. This should come as no surprise: The nanoPU transmit path is only 11 clock cycles (< 4ns at 3GHz) for a single word message, measured from when an application writes the word into the register until it is placed on the network (not including the MAC processing). And the nanoPU receive RX path is only 6 cycles (not including the MAC processing). For longer packets the loopback for the two designs eventually converges as the latency becomes dominated by store-and-forward delays.

Table 1 compares the TX and RX throughput for 64B packets for both designs; the nanoPU runs at 6–8× higher throughput than the traditional design, mostly because of the huge reductions in per-packet overheads in the nanoPU. A nanoPU application does not need to take time to instruct a DMA engine to send/receive packets, it simply reads and writes the NIC queues directly. We could expect the IceNIC throughput to increase slightly if it supported descriptor rings, which means the application doesn't need to program the DMA for every packet. However, this would also increase latency, because of batching.

| | RX (Gb/s) | TX (Gb/s) |
|---|---|---|
| **The nanoPU** | 116 | 186 |
| **RISC-V w/ IceNIC** | 14 | 28 |

**Table 1: Throughput for 64B packets. The nanoPU prototype versus traditional RISC-V core with IceNIC.**

***Thread Scheduler Latency.*** We define the latency of the thread scheduler to be the time from when an interrupt fires to the time when the first instruction of the newly selected thread is executed. We measured it to be 60 ns (180 clock cycles) on the nanoPU with almost no variation. This is at least two orders of magnitude faster than the best-known software schedulers [50]; and better still, has almost no variance, reducing the tail latency.

## 5.2 Bare-Metal Application Benchmarks

We implemented and evaluated four nanoservice applications, and compared their performance on our nanoPU prototype with the Traditional design.

***NFV-like Streaming Application.*** This application emulates a very simple streaming NFV application. It treats an arriving message as a list of 8 byte unsigned integers, increments each integer, and then returns the resulting message back to the sender. 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. Figure 7b shows that the nanoPU has 3.5–4× lower latency for all packet lengths. Note that the nanoPU runs almost as fast as it did for the basic loopback, whereas the Traditional design runs much slower because the CPU is forced to touch every data byte in the packet. Every byte of the packet must be copied from memory into registers, then copied back to memory and then wait for the DMA engine to send it to the NIC. The nanoPU eliminates all this overhead.

***Neural Network Inference.*** The nanoPU core can minimize the latency required for inference by exploiting the maximum amount of model parallelism. In inference, if each node runs a small number of multiply-accumulate operations, it can be turned into a nanoservice, particularly if the weights and data use reduced precision integers, as is typical in today's inference engines [32].

We benchmarked inference for one node in a neural network by implementing a multiply-accumulate operation. 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

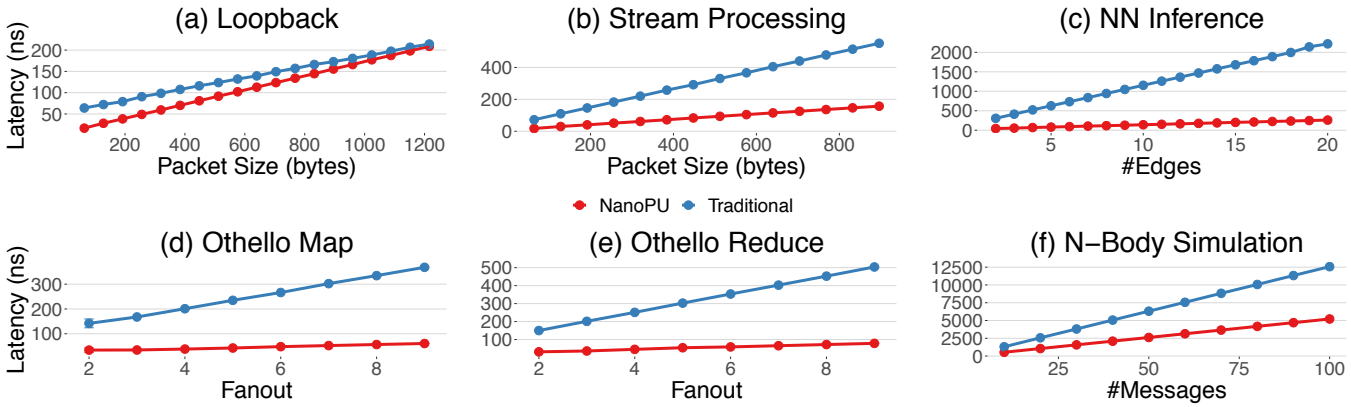



**Figure 7: Nanoservice application latency comparisons for nanoPU vs a traditional RISC-V core with IceNIC.**

of the first incoming message to the last byte of the response. Figure 7c shows that the nanoPU implementation has an order of magnitude lower latency than the Traditional design. This is because (1) packet data does not need to be copied from memory to registers before using the ALU, and (2) the nanoPU has a much higher RX throughput for small packets (as shown in Table 1).

***Othello Map/Reduce.*** We re-implemented the Othello application (used as a benchmark in [26]), comparing its performance on the nanoPU with a Traditional design. An Othello board can be represented by just two 64-bit unsigned integers, hence the working set fits in the L1 cache. The set of possible next moves can be calculated in less than one microsecond, as demonstrated in [26].

We measured the latency of the "map" and "reduce" operations 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 operations by a factor of 4–6× (Figure 7d,e).

***N-body Simulation.*** Nanoservices can efficiently implement and execute N-body simulations. This scientific computing application is typically run on HPC clusters. Astronomers use N-body simulations to model the gravitational interaction of celestial bodies. These simulations are computationally heavy, with most time spent computing the gravitational force each body exerts on every other body. A popular data structure for N-body simulations is a quad-tree, as described in the Barnes-Hut algorithm [23]. This algorithm reduces the computational complexity required for the gravity computation step from $O(N^2)$ to $O(N \log N)$.

A naïve nanoservices implementation of the Barnes-Hut algorithm would simply implement each node of the quad-tree as a separate nanotask and pass messages between nanotasks to compute gravitational forces, as shown in Figure 8. Leaf nodes represent bodies being simulated and internal nodes represent regions of space. To compute the force exerted on body F, node F will send a message intended to traverse the quadtree, starting at the root. If the center of mass of the receiving node is sufficiently far away, the current node will compute the force that it exerts on body F and return the result in a response message. Otherwise, the node will forward the request to all of its children, which recursively implement the same procedure.

In the Barnes-Hut algorithm, the state per node would be L1 cache resident, consisting primarily of the node's position and mass. A node computes the force it exerts on the body identified by the arriving message, using the 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 $< 1\mu$s. This is a radical departure from how the Barnes-Hut algorithm is implemented today in Changa [28], in which the quad-tree is stored as a large in-memory data structure distributed across many machines.

We implemented and measured a single node of an N-body physical simulation. 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 7f, the nanoPU reduces latency by 2.5× compared to a Traditional design. Section 5.4 describes a large-scale evaluation of the N-body physical simulation across multiple nodes.

## 5.3 Thread Scheduling Benchmarks

We compare two thread scheduling policies. The first is our NIC-driven hardware thread scheduling policy described in Section 3.3, in which the NIC decides which thread to run

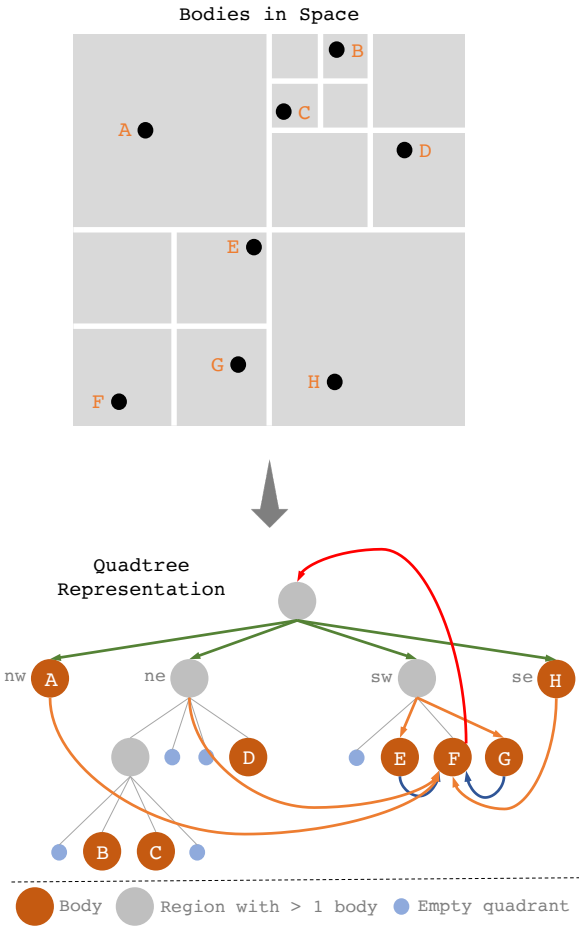



**Figure 8: An example message-passing pattern to compute the gravitational forces exerted on body F using a simple nanoservices implementation of the Barnes-Hut algorithm [23].**

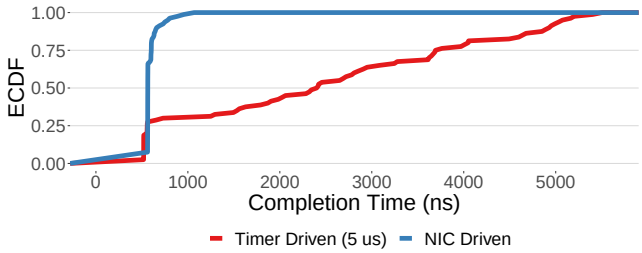



**Figure 9: Latency reduction as a result of NIC-driven thread scheduling relative to timer-driven thread scheduling.**

next and tells the CPU via the CSR registers. The second represents a more traditional Linux-style, timer-driven thread scheduler, in which timer interrupts are configured to fire periodically to decide which thread to run next and NIC hardware interrupts are disabled. In both cases, when an interrupt fires, the processor traps into the nanokernel, which then swaps to the highest priority context with messages to process. The key difference is that in the timer-driven approach, context switches only happen on periodic time boundaries, hence we can expect them to have longer latency (both average and tail).

To compare the two scheduling policies we pin one high-priority context and one low-priority context on the core, then send in a stream of randomly interleaved high- and low-priority requests (separated by an interpacket gap of 500 ns). Each application services the request for 500 ns (1500 instructions) before sending back a response.

Figure 9 compares the two scheduling policies by measuring the distribution of response times for high-priority messages. Our hardware NIC-driven scheduling policy reduces the tail response latency by a factor of 5.5× and the standard deviation of the response times by almost 20×. This result should not be surprising–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. The NIC-driven scheduling policy also accelerates the low-priority application because the timer-driven policy causes many traps into the nanokernel for no particular reasons, degrading its throughput. Our evaluation demonstrates the huge potential benefit of allowing the NIC hardware to drive the context-switch logic.

## 5.4 Large-Scale N-Body Simulation

We demonstrate the huge performance gains of a large-scale nanoservice deployment using N-body simulation and the naïve nanoservice implementation, described in §5.2. In this approach, the root node of the quad-tree becomes the bottleneck, because it must process one message from each body being simulated, and the root node dictates the total runtime of the gravity computation step. We compare the performance of our naïve nanoservice application with the performance obtained when using ChanGa [28], a popular N-body gravitational simulator.

Table 2 compares the average gravity computation time for 80K bodies using both the ChanGa simulator (with recommended parameter settings) and the naïve nanoservices implementation. We obtain the expected nanoservice performance by extrapolating the performance of the nanoPU bare-metal N-body node, evaluated in §5.2, to 80K messages, mimicking the processing required by the root node of the quad-tree. Even our simple, naïve nanoservice implementation will reduce the total gravity computation time by four

| | Avg. Gravity Computation Time |
|---|---|
| **Nanoservices** | 4 ms |
| **ChanGa** | 30,000 ms |

**Table 2: Comparing the average gravity computation time in an N-body simulation of 80K bodies using a theoretical nanoservices deployment and a real implementation using ChanGa [28].**

orders of magnitude. We can drive it down further by replicating certain nodes of the quad-tree. Although ChanGa is widely used, we can significantly outperform it because, unlike nanoservices, it uses very coarse-grained parallelism. Note that our evaluation assumes the transport protocol is able to handle an 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 or adopting transport protocols to deal with this degree of incast may be challenging and is a subject of future research.

## 6 REPRODUCIBILITY

Our nanoPU prototype and programs are public domain and open-source. We will make them publicly available with the published paper. We will provide a virtual machine with all tools and data pre-installed to make it easy for others to reproduce or extend our results.

## 7 DESIGN CONSIDERATIONS

Our nanoPU is deliberately made simple, in part because of the constraints of an academic research project. The designer of a complete nanoPU will need to consider many additional aspects of the design. We address some of the key tradeoffs.

***GPRs vs CSRs.*** Our nanoPU prototype on the RISC-V Rocket core repurposes two of its 32 general-purpose registers (GPRs) for the head and tail of the network queues. In a CPU with 32 registers, we can likely afford to lose two; our benchmark nanoservice applications did not suffer from the loss of two GPRs. However, if they had suffered, the cost would have been high as, variables would have spilled over into memory, increasing latency. This would be exacerbated in embedded applications for a CPU with fewer GPRs. We therefore experimented with a design where the head and tail registers are implemented using control status registers (CSRs) instead, which might need to be considered in settings with limited GPRs.

***Floating point.*** Our nanoPU design minimizes the latency from the network into the *integer* register file. Many scientific computing applications, such as the N-body simulation, rely on *floating point* arithmetic, with their own set of floating point GPRs. The default RISC-V ISA does not include instructions to copy an integer register to a floating point register; the value must first be stored into memory and then loaded into the floating point register file, increasing latency. If necessary, a designer should add instructions to the RISC-V ISA to copy between the two types of GPR.

***In-order Execution.*** Our nanoPU prototype is built from a simple 5-stage, in-order RISC-V Rocket core. While our prototype required very minor modifications to the CPU pipeline, more invasive changes would be required for an out-of-order processor in order to ensure that words are read from the RX queue in FIFO order. This is even evident on our simple 5-stage core: if the network RX queue is read as a result of a branch mis-prediction, this state change must be undone when the pipeline is flushed. We modified the RX queue to "unread" the last two words that were removed. A more complete design would be needed for a CPU with out-of-order, speculative execution.

***Domain-specific versus general-purpose.*** For clarity, we describe a nanoPU here that exclusively runs nanoservices. This might be ideal, particularly if many applications can exploit its low-latency design. We can imagine a single chip with hundreds or thousands of nanoPU cores, with little or no DRAM. But in other settings, for example in a warehouse computer, it might be necessary (or economically prudent) to support general purpose workloads too. As-is, nanoPU does not support *memory-intensive* distributed applications, such as a Key-Value store [45] (unless the database can be sharded so small that it fits in the L1 cache). In this case, the nanoPU optimizations might be added to an otherwise general-purpose CPU. The NIC, for example, could steer only nanotask messages to the GPRs, and steer memory intensive traffic (e.g. RDMA or regular TCP) to the traditional DMA, memory, and cache path. This would require a more sophisticated thread scheduler for very low latency.

***Domain specific languages.*** We assume that the NIC pipeline is programmed using the P4 language, or similar, for fast header processing. It is worth considering whether C is the correct language for nanotasks: Perhaps a set of libraries, restricted C, or a special language could help programmers craft very fast nanotasks using explicit message processing.

***Protection and security.*** The J-machine [16] first used GPRs for low-latency communication, but the approach was abandoned because it was felt that isolating multiple contexts running on the same core required too much hardware support. With many generations of improvement in process technology (from $10^6$ transistors per chip in 1989 to over

$10^{10}$ today), the per-context queues in the nanoPU solve this problem in a manner not feasible at the time of the J-machine.

***Kernel.*** Our prototype replaces Linux with our own minimal operating system, the nanokernel, designed to minimize average and tail latency for context switches. Our nanokernel currently lacks virtual memory and different privilege modes; applications and the nanokernel currently execute in the same address space and at the same privilege level. Our next generation nanokernel will address these issues, and we do not anticipate much increase in latency. A real system might also need a runtime resource scheduler to ensure high utilization of nanoPU cores and NICs by dispatching nanoservice apps to cores, monitoring the utilization of the cores and NICs, the status of contexts, adjusting their priorities, and rebalancing over/under-utilized nanoPUs on a longer timescale.

## 8 RELATED WORK

***Building Distributed Apps.*** Microservices and serverless compute is growing more in popularity for finer-grained computing than in the past [6, 8, 22], and has been used to implement fine grained video encoding [19], compilers [18], and scientific computing applications [31]. Our nanoservices framework would drive this approach to the extreme. Our nanoPU is designed to make it possible.

***Minimizing RPC Tail Latency.*** Several authors have recently described novel approaches to minimize tail RPC completion time (Shinjuku [33], Shenango [50], and RPC-Valet [15]) by efficiently load balancing incoming requests across cores. While demonstrably good for *micro*services, software schedulers are too slow for nanoservices. We believe load balancing messages to nanotasks is best done by the sender or by the network, not the receiving host where it is already too late, especially for the massive degrees of incast that we expect nanoservices to exhibit. Fine-grained load-balancing across nanoPUs, plus receiver-driven scheduling in the transport layer (a la Homa [43] and NDP [24]) should balance load better than a software scheduler could at the receiver.

***Global Clock Synchronization.*** If nanoPUs have tightly synchronized clocks (within say 100ns), like SIMON [20], PTP [57], or DTP [37], the sender can make informed load balancing decisions, or delay sending an RPC to avoid queues and drops. This is now much easier to deploy with the advent of per-packet INT [36] with nanosecond resolution to eliminate noisy synchronization signals caused by queueing delay.

***NIC Hardware.*** Smart NICs (NICs with CPUs on them) are becoming more popular, particularly for shared cloud data centers [7, 39, 44]. They aim to offload infrastructure software from the revenue-generating host CPUs onto smaller, lower power CPUs on the NIC. These designs are likely to increase latency for nanoservices, unless they contain a dedicated nanoPU-like fastpath for RPC messages on the NIC's data path. Alternatively, the CPUs on the NIC could include nanoPUs, to provide low latency RPCs without bothering the main host CPU.

***Integrated NIC.*** We are not the first to propose an integrated network interface. Scale-out NUMA [46] accelerates RDMA-style applications by integrating the NIC into the machine's local cache-coherence hierarchy. The recent Compute Express Link (CXL) [14] standard is similar; it maintains coherence between the CPU memory and the memory on PCIe-attached devices. These approaches, while good for throughput, are not optimal for nanoservices, which require minimal latency into the CPU, not memory.

## 9 CONCLUSION

There are several conclusions to draw from this extreme approach to distributed computing. On one hand, the nanoPU suggests a fairly simple, low-cost, non-invasive way to modify a CPU+NIC to accelerate distributed applications that can exploit very fine-grained nanotasks. We could expect dramatic speedups for this class of applications. On the other hand, one might dismiss nanoservices as too fine-grained for the remaining broad class of applications that require accesses to large pools of memory; these are unlikely to benefit from nanoservices because they cannot be cache resident.

We envisage three deployment scenarios: First, racks of nanoPUs in an otherwise unchanged datacenter, to which nanoservice jobs are directed. A single chip might contain over a thousand nanoPU cores sharing over a hundred on-chip low-latency L-NIC interfaces. This would be a formidable platform for running nanoservice applications. Further, one can imagine extending stream processing and load-balancing into the NIC pipeline and network switches, offloading the CPU cores from processing tasks better suited to in-network computation. The second scenario is where an existing CPU has the nanoPU features added to it, in addition to its normal cache, DRAM, DMA and PCIe hardware. In this case, the NIC would steer nanoservice RPC messages directly to the registers, while steering legacy network traffic via DMA into memory. A third scenario is for embedded applications, for example the CPUs on a modern smart NIC. These could be designed as a nanoPU either to accelerate control of the smart NIC, or to host nanoservices to offload the end-host server. All three scenarios are technically possible.

Our nanoPU prototype design indicates that in all three scenarios, nanoservice applications would run several times faster, with more predictable completion times.

# 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] 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).

[3] 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).

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

[5] 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).

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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).

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

[15] Daglis, A., Sutherland, M., and Falsafi, B. Rpcvalet: Ni-driven tail-aware balancing of $\mu$s-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 35–48.

[16] 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.

[17] 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.

[18] 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.

[19] 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).

[20] Geng, Y., Liu, S., Yin, Z., Naik, A., Prabhakar, B., Rosenblum, M., and Vahdat, A. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *USENIX NSDI* (2019).

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Intel Ethernet 800 Series NIC. https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet.html. Accessed on 2020-01-29.

[28] 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.

[29] 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.

[30] 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.

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

[32] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), pp. 1–12.

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

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

[35] 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.

[36] 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).

[37] Lee, K. S., Wang, H., Shrivastav, V., and Weatherspoon, H. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 454–467.

[38] Li, Y., Miao, R., Liu, H. H., Zhuang, Y., Feng, F., Tang, L., Cao, Z., Zhang, M., Kelly, F., Alizadeh, M., et al. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. ACM New York, NY, USA, 2019, pp. 44–58.

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

[40] Mellanox ConnectX-6 200GbE Ethernet Adapter IC. https://www.mellanox.com/sites/default/files/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf. Accessed on 2020-01-29.

[41] 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.

[42] Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review 45*, 4 (2015), 537–550.

[43] 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.

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

[45] 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.

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

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

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

[49] Options for Code Generation Conventions. https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html#Code-Gen-Options. Accessed on 02/04/2020.

[50] 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.

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

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

[53] RapidIO Intel FPGA IP User Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_rapidio.pdf. Accessed on 02/04/2020.

[54] 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.

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

[56] Schuehler, D. V., and Lockwood, J. W. A Modular System for FPGA-based TCP Flow Processing in High-Speed networks. In *FPL* (2004), Springer.

[57] Shankarkumar, V., Montini, L., Frost, T., and Dowd, G. Precision Time Protocol Version 2 (PTPv2) Management Information Base. RFC 8173, June 2017.

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

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

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

[61] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review 45*, 4 (2015), 523–536.