An Architecture for Nanoservices

Paper # XXX, XXX pages

ABSTRACT

Nanoservices are highly parallelizable, compute intensive applications with a cache resident working set and sub-microsecond response times. We believe that nanoservices will become popular in the future for two main reasons. One, Moore's Law is preventing increases in compute speed so we cannot rely on faster processors for performance gains. Two, many applications have inherent parallelizability that cannot be harnessed today because of large, unpredictable network delays and large overheads for small messages. In order to make nanoservices a reality, we need a new networking-optimized compute platform. Thus, in this paper we present a new domain specific architecture for compute intensive distributed applications: the Nanoservice Processing Unit (NanoPU). The NanoPU has the following characteristics: a fast path between the network and the core of the CPU to minimize average communication latency, NIC-driven thread scheduling along with dedicated memory & network resources per-core to minimize tail response times. The NanoPU provides an order of magnitude lower average and tail latency than existing systems for nanoservice applications.

ACM Reference Format:
Paper # XXX, XXX pages. 2020. An Architecture for Nanoservices. In *Proceedings of (Submitted for review to SIGCOMM)*. ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

- If we remove the constraint on the number of physical processors we have available to us, and if all processors could communicate with minimal, predictable latency, *how would we design distributed applications?*
- We believe that many applications have inherent parallelizability that cannot be harnessed today because of large, unpredictable network latencies provided by today's communication infrastructure.
- Describe simple example to give reader an intuitive sense of parallelism that cannot be harnessed today.
- If we eliminate these issues then we believe that we would re-implement distributed applications at a much finer granularity to harness as much parallelism as possible.
- The NanoPU is the first attempt to realize this vision.
  - Explicitly designed to scale out. In order to harness as much parallelism as possible we need to think beyond a single chip and build an architecture that is highly optimized for compute intensive networking applications.
  - Goal is to put as many cores into the network in the most cost effective way possible. We want to put *cores* into the network without letting memory get in the way. CK: I think we should justify the memory bypassing design.
  - Goal is to minimize *both* average and tail latency to provide both optimal and predictable performance.
- The nanoservices framework is a new way of building distributed applications that is very different from how distributed applications are designed today.
- We believe that many applications can be re-implemented using the nanoservices framework:
  - Physical simulations
  - Neural Network Inference and Training (or matrix operations more generally)
  - Graphics processing
  - State space search algorithms
- Today's best practices for designing distributed applications encourage developers to hide network latency as much as possible by overlapping communication with computation. Developers are forced to batch what would be many small messages into fewer large messages in order to amortize high communication overheads. This approach prevents modern implementations from harnessing all of the parallelism in distributed applications, thus sacrificing performance.
- In order to overcome this significant problem we propose a new domain specific processor that is designed for the domain of highly parallelizable, compute intensive, distributed applications.
- We argue that we should no longer rely on general purpose hardware for building distributed applications. Our reasons are simple, (1) general purpose hardware is not (and should not be) highly optimized for latency-critical, compute-intensive, network applications, and (2) distributed applications are highly sensitive to tail latency, which cannot be effectively controlled on a general purpose platform.
- Aspects of the NanoPU that allow it to eliminate tail latency:
  - NIC-driven strict priority thread scheduling ensures that the highest priority thread with work to do is always running on the core

  - Dedicated SRAM memory per core with no cache coherency or shared memory - eliminates contention in caches and memory bandwidth
  - Dedicated network interface per core eliminates NIC contention issues.
  - Programmable PISA pipeline enables us to implement transport protocols that make effective use of network bandwidth, such as Homa.
- By building a domain specific processor, we can be more power and area efficient and at the same time achieve much higher (and predictable) performance.

2 NANOSERVICES

- Nanoservices are a framework for building compute-intensive distributed applications.
- We believe that the performance of many compute-intensive distributed applications is severely limited by scalability of modern systems.
  - For instance, consider performing an N-body cosmological simulation of 1 million particles, which is considered to be a small benchmark in the field of astronomy [1]. This application is designed to simulate motion of the particles under the influence of gravity. Processing can often be parallelized for each particle in the simulation, for instance to compute the force that each particle exerts on every other particle. Today, these simulations are run the modern super computers using parallel programming frameworks such as CHARM++ [2]. However, these systems are only able to scale to O(10K) processors before performance starts to degrade [1]. Therefore, modern systems are unable to exploit all possible parallelism in this application, thus limiting performance.
  - Similarly, consider multiplying two 100x100 matrices, which is a small operation in today's world of machine learning and AI. This simple operation requires performing 1 million multiplications that could potentially be parallelized. However, even a modern TPU v3 chip, which is explicitly designed to do matrix multiplications, only has about 64K multiply accumulate processors, so cannot fully parallelize this operation. The nanoservice infrastructure would allow application developers to harness about as many cores as there are multiply accumulate processors in a large TPU pod - O(10M). And since NanoPU cores are more flexible than a TPU's multiply accumulate processors, the nanoservice infrastructure is more general purpose than a TPU pod and can be used to run machine learning workloads such as neural network inference and training.
  - We can make a similar argument for graphics processing on GPUs if needed.
- We believe that these scalability limitations exist for the following reasons:

(1) Large and unpredictable RPC completion times force application developers to prepare for the worst and provision resources accordingly. There is also the well known performance degradation caused by the straggler effect that plagues many modern distributed applications. Additionally, large overheads for small RPCs force application developers to batch what would be many small messages into fewer large messages, thus sacrificing opportunities for parallel computation.

(2) Modern network transport protocols are unable to handle massive degrees of incast (e.g. 10K-to-1) without ever overflowing or underflowing the receiver buffer.

- In this paper, we propose an architecture to deal with issue (1), while leaving issue (2) as an area for future research.
- Applications that are best suited for implementation using nanoservices are those that are highly parallelizable and compute intensive.
- Target applications should be divisible into compute intensive nanotasks that all service network messages in less than 1 us and have a small enough working set such that they fit in on-chip SRAM so that memory accesses are as fast as accessing the L1 cache on an general purpose CPU. CK: Working set must be contained within register files or L1 cache? I thought we were assuming the former. Also, if we tell the latter, some people may ask for the difference between ours and DDIO.
- Nanotasks should not perform DRAM memory accesses because a single DRAM memory access requires about 100 ns, which is already a large fraction of the maximum response time allowed by nanotasks.
- In order to make nanoservices work, we need new infrastructure. This new infrastructure must:
  - Minimize overheads for network messages in order to make it practical to send and process millions of requests per second.
  - Minimize average network latency to improve performance for the common case.
  - Minimize tail RPC completion time because end-to-end performance of distributed applications are often dominated by this metric. Minimizing tail RPC completion time will require infrastructure that is able to enforce strict performance isolation by eliminating resource contention.

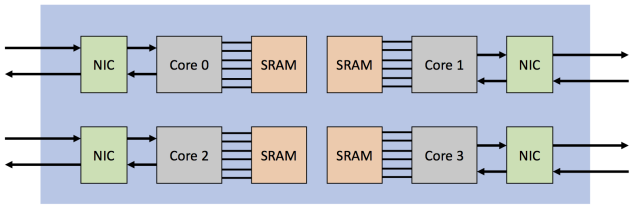

Figure 1: High-level NanoPU architecture with 4 cores.

3 NANOPU DESIGN OVERVIEW

- Domain specific architectures are becoming more and more popular today for good reason.
- The slowing of Moore's Law is preventing general purpose compute from getting any faster, so we must turn to domain specific hardware to achieve performance and efficiency gains.
- We believe that the domain of compute-intensive distributed applications is in dire need of domain specific hardware.
  - General purpose hardware is not optimized to put compute in the network. Instead, it puts memory in the network and attaches compute to memory. CK: Perhaps we should tell why general-purpose CPUs evolved that way. Is it because the initial designs for CPUs were done when standalone computing (as opposed to distributed computing) was predominant? Or was it because in the early days of CPUs networks were just way too slow (relatively to the speed of the CPUs of those early days), and hence it just didn't make sense to put compute to the network directly?
  - General purpose hardware has far too many sources of resource contention which inflate tail latency and kill performance of distributed applications: cache contention, memory bandwidth, CPU cores, and NIC queues. CK: Isn't dedication or hard partitioning kind of orthogonal to our point? What if someone introduces hard-partitioned cache, memory, memory bandwidth, NIC, etc. for general-purpose CPUs? Sure, it's gonna be expensive, but it may be able to bound the tail latency very well, right? What's wrong with such an architecture for nanoservices?
- We present the NanoPU, a domain specific processor for compute-intensive distributed apps.
- The NanoPU has the following characteristics that make it well suited for the aforementioned domain:
  - A fast path from the network to the heart of the CPU core, the register file. This minimizes average communication latency for network communications, drastically improving the common case. This architecture is explicitly designed to scale out beyond a single chip. One of the main goals of the NanoPU is to put as many cores in the network in the most cost effective way possible. The NanoPU will allow developers to harness more cores than would ever be possible or practical to fit on a single chip (e.g. millions of cores).
  - An event-driven PISA pipeline on the NIC which allows it to efficiently terminate a low latency transport protocol like Homa in hardware. This minimizes the overhead required to send messages because software no longer needs to deal with maintaining timers, sending retransmissions, or reassembling messages because these things can be implemented very efficiently in hardware. We will describe the NanoPU transport protocol and its implementation in a future paper due to lack of sufficient space here.
  - The NanoPU has the following traits that allow it to minimize tail latency:
    * NIC-driven thread scheduling, which enforces strict priority scheduling of nanoserver threads on the core and does not rely on slow software based scheduling mechanisms.
    * Each core has dedicated SRAM that is not shared with anyone else and is the only memory available to applications (i.e. there is no off chip DRAM). This eliminates both cache contention and memory bandwidth contention issues, which contribute to tail latency. It also has a number of additional benefits:
      · A two-way set associative cache uses 2.5 times as much power as an equivalent software controlled scratchpad memory. So since the SRAM no longer needs to function as a cache, the NanoPU can be more power efficient.
      · All communication between cores is done via explicit message passing over the external network, which means the NanoPU does not need an on-chip network and thus the additional silicon area can be used to fit more SRAM onto the chip.
    * Each core has its own dedicated NIC interface, which eliminates NIC contention issues.
- The following sections describe the NIC data-path, the NIC-Core interface, and the NanoPU's minimal operating system (the NanoKernel) in more detail.

3.1 NIC Datapath

- The NIC Datapath is responsible for processing packets received from both the network and the attached CPU core and forwarding packets either to the network or to the CPU core.

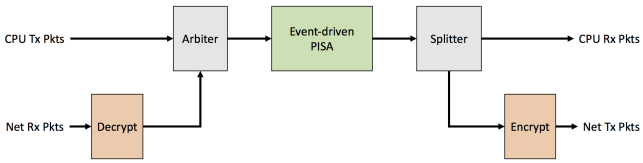

Figure 2: High-level NanoPU NIC datapath architecture.

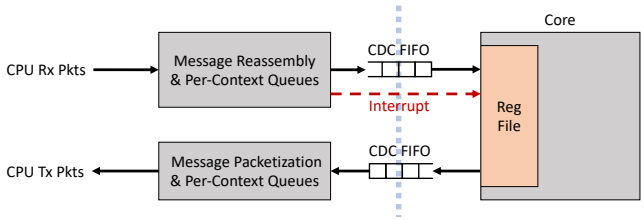

Figure 3: Block diagram of the NanoPU's NIC-Core interface.

- The centerpiece of the NIC datapath is an event-driven PISA pipeline.
- Event-driven PISA is a recently proposed programmable packet processing architecture, which supports more expressive stateful data-plane programming than its predecessor: the baseline PISA architecture. We strongly believe that an event-driven PISA architecture is flexible enough to terminate a low latency transport protocol like Homa. Thus minimizing the overheads required for sending network messages.
- The PISA pipeline is mainly responsible for processing packet headers. It removes Ethernet, IP, and L-NIC transport headers, and adds a small application header for packets that arrive from the network and are going to the CPU. It performs the opposite tasks for packets sent from the CPU and are going out the network.
- The Event-driven PISA pipeline can also be used to offload some application specific logic in order to improve performance. CK: what do we mean by this? It sounds vague.
- The data-path also contains a block to decrypt packets as they arrive from the network and a block to encryt packets just before they are sent over the network. This avoids the software overhead that would be incurred if these tasks needed to be performed in the application or the operating system.

3.2 NIC-Core Interface

- The key component of the NanoPU's NIC-Core interface is its fast path directly into the CPU's register file. This is implemented using a simple FIFO interface between the NIC and the register file.
- We reserve two general purpose registers (GPRs), one for the head of the RX queue and one for the tail of the TX queue of the current context (i.e. application thread).
- The underlying hardware supports per-context TX and RX queues and only exposes the queues of the currently running context via the head/tail GPRs.
- As a result of the fact that the CPU pipeline and the NIC may run at different clock frequencies, we use simple clock-domain crossing FIFO queues between the per-context queues and the CPU pipeline.
- The NIC exposes a message interface to applications running on the core. This means that the hardware is responsible for performing message packetization and reassembly. Both of these are tasks that we will describe in more detail in our subsequent paper, which will include a description of the NanoPU transport protocol. In this paper, we will assume that all application messages fit in a single Ethernet packet. We expect that this will be sufficient to implement the vast majority of nanoservice applications.
- The NIC-Core interface also includes a few additional control status registers (CSRs) and memory-mapped registers, which are used to perform out-of-band configuration of the NIC, such as registering/deregistering context IDs with the NIC, supporting NIC-driven thread scheduling, and adding/removing PISA table entries.
- The first 8 bytes of every message sent and received by the application consists of a small application header.
- The application header indicates message length as well as the source IP address and context ID for received messages, or destination IP address and context ID for transmitted messages. This header allows the application to know when it has reached the end of a message. It also allows the NIC to know when the last byte of a message has been written to the TX queue by the application.

3.3 Transport Termination in the NIC

- This section describes the hardware mechanisms in the NanoPU architecture that enable the NIC to terminate a low latency transport protocol such as Homa.
- We describe the hardware mechanisms, but the actual implementation and evaluation of a specific transport protocol is outside the scope of this paper and will be presented in a future paper.
- By terminating the transport protocol in hardware, the overhead required for applications to send/receive small messages is significantly reduced. It is very inefficient for software to deal with per-message timers,

packet retransmissions, message reassembly and packetization, while hardware can implement this functionality much more efficiently.

- The key hardware components that enable the NIC to terminate a low latency transport protocol are as follows:
  - An Event-driven PISA pipeline in the NIC data path. This programmable data-plane architecture enables us to 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.
  - A timer event generation module 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 (i.e. add a new time), reschedule (i.e. restart an existing timer), and cancel (i.e. remove the state associated with an existing timer). 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 has expired. 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.
  - In addition to data packet processing and timer event processing, the Event-driven PISA pipeline also supports background state clean up event processing, which is triggered either when: (1) message transmission / reception is complete, or (2) message transmission / reception has expired (i.e. timed out).
  - A programmable packet generation module that can be programmed to generate acknowledgement and/or message completion packets.
  - A packetization buffer that 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.
  - A message reassembly buffer that 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.
  - The transport header has the following fields:
    * Source / destination context ID
    * Message ID
    * Message length
    * Packet offset within the message
    * Reserved bits for layering RPC request/response support on top of a one-way reliable message protocol

3.4 The NanoKernel

- One of the main goals of the NanoPU is to put as many cores in the network in the most cost effective way possible. Ideally, nanoservice application developers will be able to utilize enough cores to exploit all possible parallelism in their application, which could reach beyond millions of cores. Application parallelism could exceed the number of physical cores that are available. CK: What do we mean by this last sentence? What's application parallelism? The total demands of applications for nanoservices?
- Thus, NanoPU cores must be multiplexed very efficiently. We cannot rely on slow software-based scheduling mechanisms to schedule nanoservice threads. Instead, the NIC drives thread scheduling. The NIC keeps track of the highest priority context with messages to process and interrupts the CPU pipeline 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 sees that there is another context with messages to process.
- When the NIC interrupts the CPU pipeline it tells the NanoKernel scheduler which context to switch to. The NanoKernel thread scheduler simply performs a context switch to the context indicated by the NIC.
- When an application wants to perform network IO it issues a system call providing its desired context ID and priority. The privileged system call code will then ensure that the desired context ID is available and will register the context ID with the NIC along with the indicated priority level by performing writes to the appropriate CSRs. This operation is roughly equivalent to opening a socket on a modern operating system.
- Maybe also explain how the current implementation of the NanoKernel is different from a traditional operating system: applications are compiled into a single binary, shared address space, single privilege mode, etc.
- We require a minimal operating system to ensure that all code (and data) fits in on-chip SRAM.

CK: In addition to the NanoKernel and priority-based scheduling, it seems we also need a runtime system that ensures the high utilization of NPUs and L-NICs. It'll be responsible for dispatching nanoservice apps to NanoPU cores, monitoring the utiliztion of the cores and NICs, along with the status of contexts. If some cores or NICs are over/under-utilized for a longer period of time, it'll need to rebalance the system. If some contexts are starved for a long period of time, again it may needto address it. We don't need to design and

build such a runtime for this paper, but we may at least have to identify the needs for such one, while leaving the actual design to the next paper.

4 PROGRAMMING THE NANOPU

- Applications must be written such that they process network messages in FIFO order.
- Applications should have minimal memory requirements, such that the working set fits in on-chip SRAM. In fact, the code and data of all nanoservice threads pinned to the core must fit in on-chip SRAM.
- Nanoservice applications are composed of many single threaded nanoservers that are pinned to a specific NanoPU core. These single threaded applications exchange messages directly with one another. Load balancing decisions are performed either at the application level or in the network switches.
- The NanoPU transport protocol supports reliable one way message delivery. It also supports mechanisms to support RPC style request/response protocols layered on top of the transport.

5 EVALUATIONS

- We built a NanoPU prototype on top of the open source RISC-V Rocket Core. Cite chipyard XXX.
- For the prototype we focused on building the NanoPU NIC datapath, NIC-Core interface, and a minimal operating system (the NanoKernel).
- Our prototype does not implement a full NanoPU:
  - It does not make any modifications to the memory hierarchy.
  - It does not implement any encryption or decryption.
  - The NIC and the core run at the same clock rate, hence the CDC FIFOs are not included.
  - The prototype assumes all messages consist of a single packet. It does not implement a reliable transport protocol.
- Probably need to include a description of how well we believe IceNIC represents a modern state-of-the-art NIC.

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.

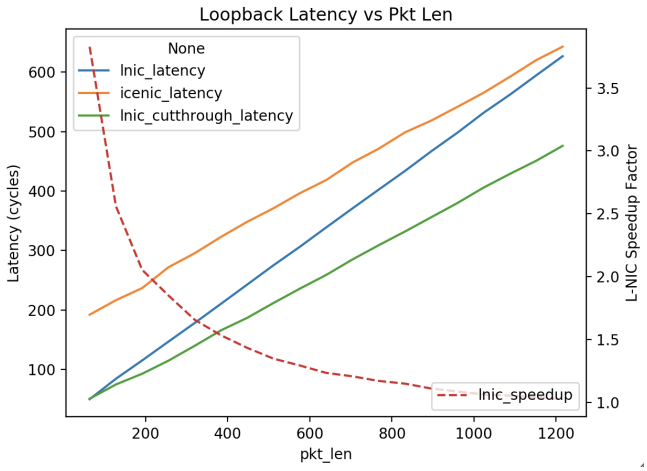

Figure 4: NanoPU vs Traditional net-core-net loopback latency for various packet sizes.

Basic Latency/Throughput Performance Analysis.

- Latency of the TX path from CPU to network: 11 cycles
- Latency of the RX path from network to CPU: 6 cycles
- Loopback latency for various packet sizes
- TX/RX throughput for 64B packets

Basic NanoKernel Performance Analaysis.

- NanoKernel context switch latency

5.2 Bare Metal Application Evaluations

- This section shows the reduction in average latency as a result of the hardware fast path to the core of the CPU.
- Streaming application (NFV style)
- Neural Network inference node
- Othello
- N-body simulation

5.3 Thread Scheduling Evaluations

- This section shows the reduction in tail latency as a result of NIC-driven thread scheduling for nanoservice applications.
- Compare Linux-style timer interrupt driven scheduling to NIC driven thread scheduling on the NanoPU.
- Explain the experiment set up.
- Show that NIC driven scheduling has much lower tail latency.

5.4 Large Scale Nanoservice Evaluation

- Show evaluations of large scale (1000's of NanoPU cores) nanoservice deployment being used to implement an N-body simulation.
- Maybe compare to ChanGa.

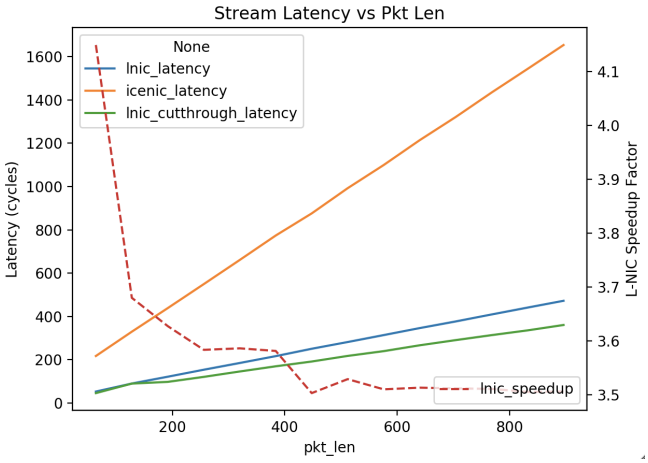

Figure 5: NanoPU vs Traditional NFV-style streaming application latency for various packet sizes.

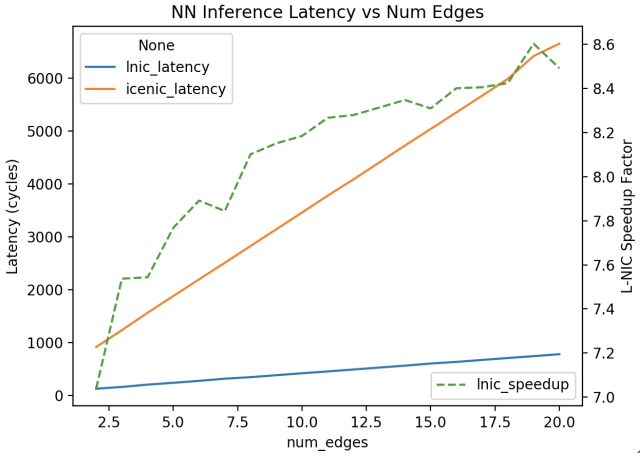

Figure 6: NanoPU vs Traditional neural network node latency for various number of incoming edges.

- Show that the nanoservices implementation can perform the simulation much faster.

6 RELATED WORK

7 CONCLUSION

REFERENCES

[1] 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.

[2] Kale, L. V., and Krishnan, S. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications* (1993), pp. 91–108.

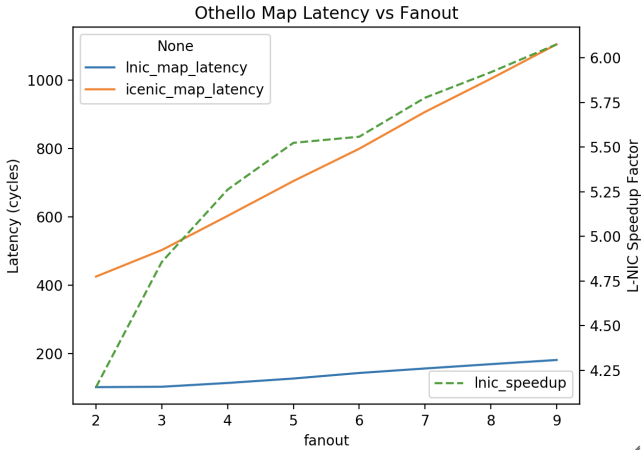

Figure 7: NanoPU vs Traditional Othello map latency for various degrees of fanout.

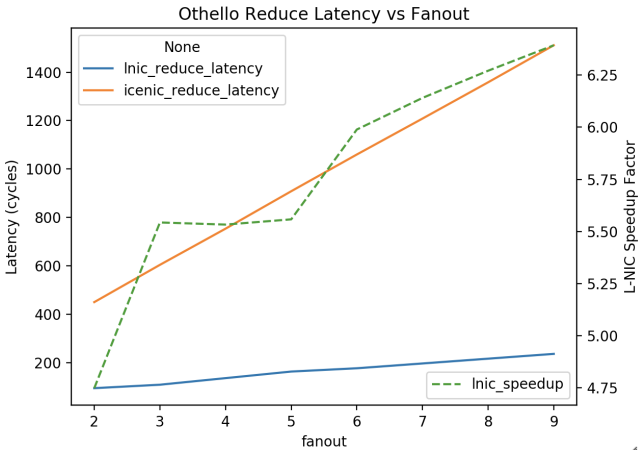

Figure 8: NanoPU vs Traditional Othello reduce latency for various degrees of fanout.