The Case for a Network Fast Path to the CPU

Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown
Stanford University

ABSTRACT

For the past two decades, the communication channel between the NIC and CPU has largely remained the same—issuing memory requests across a slow PCIe peripheral interconnect. Today, with application service times and network fabric delays measuring hundreds of nanoseconds, the NIC–CPU interface can account for most of the overhead when programming modern warehouse-scale computers.

In this paper, we tackle this issue head-on by proposing a design for a fast path between the NIC and CPU, called Lightning NIC (L-NIC), which deviates from the established norms of offloading computation onto the NIC (inflating latency), or using centralized dispatcher cores for packet scheduling (limiting throughput). L-NIC adds support for a fast path from the network to the core of the CPU by writing and reading packets directly to/from the CPU register file. This approach minimizes network IO latency, providing significant performance improvements over traditional NIC–CPU interfaces.

CCS CONCEPTS

• **Hardware → Networking hardware**;

KEYWORDS

Programmable NIC; PISA; and P4;

ACM Reference Format:
Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. 2019. The Case for a Network Fast Path to the CPU. In *Proceedings of Eighteenth ACM Workshop on Hot Topics in Networks (HotNets'19)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3365609.3365851

1 INTRODUCTION

This paper is about the design of network interfaces (NICs) for warehouse-scale computing. Traditionally considered the

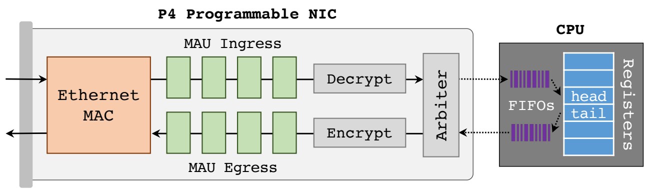

Figure 1: Simplified L-NIC architecture. Packets are transferred between a PISA NIC pipeline and CPU registers via a simple FIFO interface.

domain of computer architecture and operating systems, NIC design has become essential to making the network fast, low latency, and efficient for datacenter applications. So much so that—despite widely available commercial NICs—cloud computer companies are designing and deploying their own custom NICs in order to reduce latency, increase throughput, offload the networking stack, and connect efficiently to local accelerators, such as FPGAs and GPUs [1, 11, 14]. NIC design has become an essential part of cloud computing. While motivations vary, it is common to see designs of so-called "Smart NICs" with multiple CPU cores on the NIC to offload network operations. In some cases the NICs accelerate packet processing, perhaps in proprietary ways [22, 24]; in others, they offload work from the servers, allowing the liberated server cycles to be rented to paying customers. It begs the question as to why—when we want a packet to be processed as fast as possible—would we place a NIC-based CPU core in its way? CPUs are generally ill-suited to processing packets at speed because of their complex cache hierarchies, non-deterministic processing time, and low degree of parallelism (compared to an ASIC). A CPU on the NIC typically requires its own operating system, thread scheduler, and applications, leading to significant overhead. It likely has lower performance than the server CPU it is offloading. And while a CPU might help process packets at 10–25 Gb/s, it is a stretch to imagine a CPU processing packets for 100 Gb/s NICs and beyond; it is not a scalable solution. Our suspicion is that the added complexity is considered worthwhile in the short-term because there is uncertainty about the correct functions to place in the NIC. Hence, for now, NICs contain CPUs.

Our research is about high performance NIC designs further into the future. We ask: *What NIC architecture would minimize the latency of a remote procedure call (RPC)?* Our high-level approach is to co-design the CPU and NIC *together*, to bring message data as fast as possible into the heart of the server CPU and start executing code immediately (rather than bringing slower CPUs out onto the NIC). We call our architecture Lightning NIC, or L-NIC, and we argue that—compared to current PCIe interfaces—L-NIC reduces packet latency from the wire to the thread of execution in the server CPU by an order of magnitude (from 2 μs to less than 200 ns), while offloading transport, encryption, and thread scheduling to a P4-programmable NIC [8, 9].

But first, we must consider why latency matters so much, and why it needs to be reduced. In recent years there has been much focus (commercial and research) on reducing latency to remote storage in cloud data centers. Remote Directed Memory Access (RDMA), for example, allows one server to remotely read/write blocks of memory in another server's memory, with small requests completing in as low as 2–3 μs, by implementing everything in hardware. A customized NIC [5] accesses memory directly, without involving the CPU, freeing it for other work. While powerful for one-time read/write operations, RDMA is of less use to applications requiring multiple, dependent memory accesses. For example, a hash lookup in a key-value store requires an initial lookup in a hash table, followed by a second access to the value; these operations require two remote calls. Enterprise data-processing systems typically try to avoid multiple remote calls by requiring large data blocks be read sequentially (e.g., Spark [29], MapReduce [13]), making them only useful for large long-running batch jobs.

RAMCloud, on the other hand, was designed using low-latency RPCs to "achieve the lowest possible latency for small random accesses in large-scale applications" [27] and has reported access times in a large datacenter below 10 μs, claiming an improvement of 50–10,000 times over typical storage systems of the day. RAMCloud has larger per-transaction latency than RDMA, but implements a fast RPC mechanism involving waking up the remote CPU, executing a thread, and possibly multiple memory references before returning a response. The RAMCloud approach (low-latency RPC calls) fits well with modern microservices in which a frequently used service (e.g., a web server) is accessed by many cloud customers. It is particularly effective for services accessing data present in remote DRAM (rather than disk).

Future large datacenter applications will likely make heavy use of microservices. We aim to accelerate microservices significantly by allowing them to be built by stringing together hundreds or thousands of tiny, short-lived, low-latency serverless threads of execution (e.g., AWS Lambda [2], Google Cloud Functions [4], or Azure Functions [3]). If these threads operate on cache-resident data, we call them "nanoservices." These are small, lightweight, self-contained, serverless functions that are designed to start immediately (or close to) when a request arrives, then operate on that message from

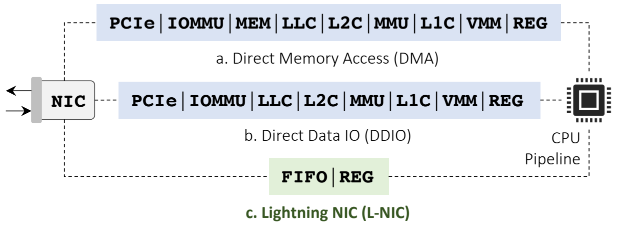
PCIe | IOMMU | MEM | LLC | L2C | MMU | L1C | VMM | REG
a. Direct Memory Access (DMA)
NIC
PCIe | IOMMU | LLC | L2C | MMU | L1C | VMM | REG
b. Direct Data IO (DDIO)
CPU Pipeline
FIFO | REG
c. Lightning NIC (L-NIC)

Figure 2: L-NIC places packet data directly into a CPU register, eliminating intermediate overheads such as interconnect delay (PCIe), address translations (IOMMU, MMU, and VMM), memory (MEM) and cache accesses (LLC, L2C, and L1C).

the CPU cache (the L1 cache on a modern CPU is about 200-times faster than accessing DRAM). Whereas RDMA uses hardware to avoid using the remote CPU, L-NIC is designed for applications requiring remote computation, and therefore require getting a message into the core of the CPU as soon as possible, bypassing the cache hierarchy, and starting the thread of execution. If the thread is small, and is responsible for a very small set of data, then it can run significantly faster if it is not held up (or scheduled out) waiting for a cache line to load from DRAM.

L-NIC is optimized for this type of computation. A high-level view of the L-NIC architecture is shown in Figure 1 (Section 3 goes into more detail). Arriving packets on the left are stream-processed in the NIC's P4-programmable *match-action* (MAU) pipeline, to check header values, translate virtual to physical network addresses, decrypt the payload, and implement a fast transport protocol (e.g., NDP [15] or Homa [23]). The interface from the NIC to the CPU has two key attributes: (1) It uses a fast, wide, synchronous FIFO interface directly into the register file; one register is dedicated to reading messages from the head of the FIFO, another for writing outgoing messages to the tail of the FIFO, (2) Message transfers are atomic; when a thread is scheduled to read a message from the FIFO, it keeps reading until the end of message before being eligible for swapping out. Similarly for sending messages. When messages arrive, the NIC identifies the correct thread (by indexing a local MAU table based on the packet header) and alerts the local thread scheduler running on the NIC. Next, the entire RPC message is sent over the FIFO interface, with the head appearing in the heart of the CPU, bypassing the entire memory hierarchy and avoiding any form of coherency protocols. Register accesses run at the speed of the CPU, and therefore a 128-bit wide register on a 3 GHz CPU corresponds to a data rate over 350 Gb/s. More importantly, the CPU can start executing on the message within 150 ns of it arriving at the NIC. Between two servers with L-NICs, with three 200 ns Ethernet switches in-between, an RPC could complete in less than 1 μs.

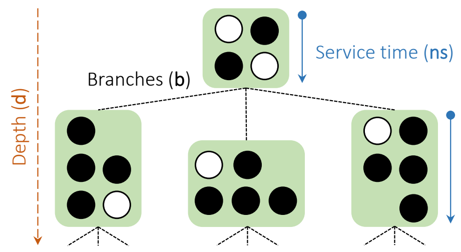

Figure 3: An example of a partial Othello game tree. Each node (a nanoservice) computes at most b moves and branches the new board states to b nodes. The game tree is searched d moves into the future.

Figure 2 shows why L-NIC (c, in green) has much lower latency than traditional DMA (a) or DDIO (b), which places data directly into the last-level cache. These alternatives are described in more detail in Section 5.

In the next section we motivate the need for L-NIC using example applications; specifically, a well-known board game that operates on cache-resident data. On a single-core machine, the application is limited by the CPU and cache-misses. If distributed to servers with L-NIC, we can accelerate processing and bring down the tail completion time by 30x. Furthermore, by using L-NIC we can achieve over 50% reduction in processing time relative to a distributed implementation that uses traditional network interfaces.

2 A NANOSERVICE APPLICATION

A nanoservice is a fast, cache-resident, and compute-intensive RPC with service times in nanoseconds. A group of such services—running in parallel across machines in a data center—forms a nanoservice application.

Let us consider an example of a simple state-space search application, we call *Othello-Player.* As the name suggests, the application plays the board-game Othello (*aka* Reversi [6]) by evaluating as many future moves as possible. Othello was invented in the 1800s and is played on a 8×8 grid board by two players, similar to chess or checkers. Players take turns to place a black or white disk onto an empty location on the board in an attempt to capture the opponent's discs, which then become their disks. The goal is to own more disks than the opponent by the end of the game.

Given an initial board state, Othello-Player must evaluate possible future moves (from an estimated total of 10^{28}) and decide which move to take next. A tree of such possible moves, starting from the initial state, is called an Othello game tree (Figure 3). Each node in the game tree, reads in a board state from the previous node and generates one or more new board states for the following nodes. The level at depth (d) of the Othello game tree represents all possible board states d moves into the future. We call the number of board states (or moves) originating from a node its branching factor, b—the average branching factor in Othello is 7, with a standard deviation of 3.

If we run Othello-Player on a single CPU core, not only will the serial computation take a long time but also, as the depth increases, the data will no longer fit in the CPU cache and require expensive main-memory accesses. This is a good example of a typical application to accelerate using nanoservices: (1) *Massively parallel.* Nodes in the same level in the Othello game tree can run independently in parallel. (2) *Cache resident.* The working set of each node (a few kilobytes) fits in the data caches, thus avoiding expensive accesses to DRAM. (3) *Short service times.* Each node can look one step ahead from the current board state very quickly (under a microsecond).

In our implementation, each node runs as a nanoservice that receives an initial board state, looks one move ahead, and sends new boards into the network, which then load balances these boards to nodes (or nanoservices) across many servers. The process will continue until the desired depth is reached, at which point the results propagate back up the tree to the root node.

We will show in Section 4 that distributed Othello-Player is limited by the communication delay. By reducing communication latency, the nanoservice and the application speedup dramatically.

Nanoservices in the wild. Nanoservices are not just limited to Othello-Player, but are applicable to a broader class of applications including state-space search [30], theorem proving [20], symbolic execution [10], physical simulations [16], and more. Moreover, emerging serverless computing platforms [2–4] may lead to widespread use of nanoservices, especially, for massively parallel applications that are small, compute intensive with strict tail-latency service level objective (SLOs) (e.g., Internet of Things (IoT), image and video processing, autonomous vehicle communication, and event streaming) [7].

3 THE L-NIC ARCHITECTURE

Figure 4 shows the L-NIC architecture. It consists of a P4 programmable NIC pipeline, which contains standard MAC/-PHY logic, an MAU-based PISA pipeline, and dedicated encryption/decryption logic. The NIC connects directly to the register file of each CPU core using a simple synchronous FIFO interface—this is the fast path for low-latency RPC messages. The NIC also supports a slower and more traditional (RDMA-like) path to transfer packets to/from the last-level cache (LLC).

The main features of the L-NIC design are as follows:

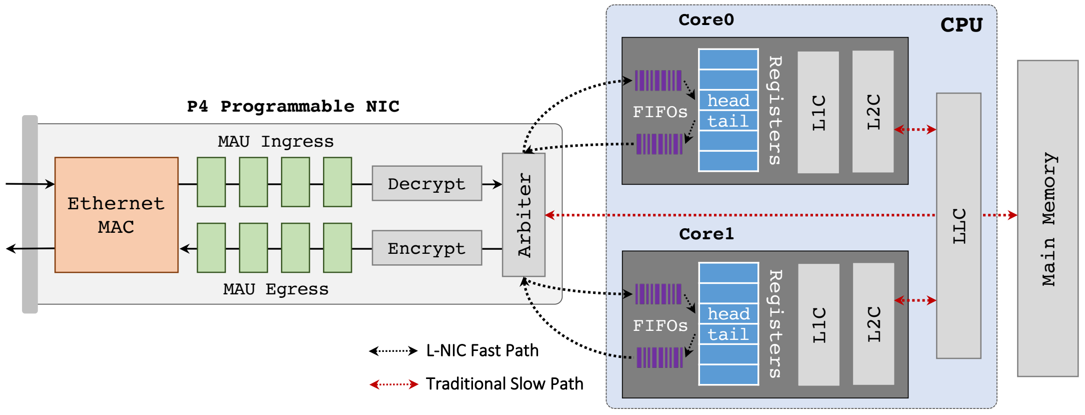

Figure 4: The proposed L-NIC architecture.

- *Fast path to the CPU (F1)*: A wide, fast synchronous FIFO interface between the NIC and the CPU register file.
- *Message dispatching (F2)*: Dispatches full RPC messages to the CPU, not individual packets. Decides if message should be sent over the fast path (to CPU register file) or slow path (through memory hierarchy).
- *Thread scheduling (F3)*: Ensures the target application thread is running, ready to read a message from the register file. Application threads perform atomic read-/writes to/from registers on message boundaries.
- *Network transport (F4)*: Implements transport protocol such as Homa [23] or NDP [15], but the NIC pipeline is programmable allowing other transport protocols to be implemented, instead.

Next, we explore each of these features in detail.

F1: Fast path to the CPU. In order to minimize the latency between the network and the CPU core, L-NIC supports a dedicated fast path to each processor's register file. This fast path is implemented using a simple synchronous FIFO interface running over a wide bus of fast serial I/Os. Today, a serial I/O line routinely runs at 50 Gb/s, and with eight lines a host can do 200 Gb/s I/O from NIC to CPU. The head of the receive FIFO is stored in a dedicated register in the processor's register file (Figure 4). Similarly, the tail of the send FIFO is stored in a separate dedicated register. By reading and writing the head and tail registers, an application can access the network with the minimum possible latency—there is no need for messages to traverse the memory hierarchy.

F2: Message dispatching. A low-latency NIC must decide how to load balance messages across CPU cores. If load is imbalanced, some cores will be overloaded while others sit idle, resulting in long queueing delays and poor tail latency.

Rather than sending individual packets to CPU cores, L-NIC dispatches entire messages. This is because applications deal with messages not packets; if an application only receives one packet of a multi-packet message, it may need to sit idle waiting for the rest of the message to arrive. The P4 pipeline on L-NIC is ideally suited for performing efficient message dispatching. The P4 pipeline keeps track of packets within a message as well as the current state of the CPU cores, making its dispatching decisions based on which cores are idle.

The L-NIC also decides which path to send the message along: the fast path or the slow path, based on the type of application and its sensitivity to latency. The decision is made by indexing into a MAU table based on the packet header. Nanoservice messages take the fast path into the register file, while RDMA and data-intensive apps take the slower path into LLC or main memory.

F3: Thread scheduling. L-NIC must also participate in the operating system's thread scheduling logic. L-NIC ensures the target thread is running on the selected CPU core so that it can immediately start reading the message from the register file. The thread reads/writes atomically to/from the head/tail registers on message boundaries. Latency-sensitive application threads can poll or be woken; they can decide. We expect both models to be used, depending on application and CPU utilization requirements. In an extreme case, if a message takes a long time to process, L-NIC may interrupt a thread to prevent it from hogging the core [18]; but this is not the common case.

F4: Network transport. L-NIC implements the transport logic in the P4 pipeline. We draw inspiration from the Homa [23] transport protocol, which achieves tail latencies within 3–4x of the minimum possible latency on an unloaded network. We believe that, for nanoservices, the transport logic would be too slow if run in software. As network speeds increase, CPU cores will quickly become the bottleneck. Instead, the transport logic runs in a P4-programmable NIC with an MAU

Metric	Parameter: Values
Latency	Net to Fabric, Mem, LLC: 1 μs Net to Reg: 200 ns, 400 ns, 600 ns
Access time	Memory: 100 ns, LLC: 10 ns, Reg: 1 ns
Service time	Map: $X \sim MapDist$, Reduce: 500 ns
Others	#Hosts: 2500, #Branches: $Y \sim BranchDist$

Table 1: Parameter values used in simulations.

pipeline. Such a pipeline is well-suited to perform per-packet processing, and we believe that the P4 language is expressive enough to implement the required transport functions.

Design observations. Traditional RPCs that traverse the cache and memory hierarchy are often bottlenecked by expensive address translations between device address space and physical address space (IOMMU), between virtual address space and physical address space (MMU), or between guest address space and host address space (VMM). By placing data directly into the register file, it is address-less—the application can decide whether or not to store it in memory, and if so the address translation is immediate. Furthermore, in a virtualized environment an RPC to a VM will have the same low latency as a bare metal server, because the message does not need to traverse a complex network stack in the host OS or the guest's VM.

In some cases, the P4 pipeline might help accelerate lightweight streaming applications even further. Many recent papers demonstrate applications which can be greatly accelerated this way [12, 17, 21]. In our Othello-Player state-space search example, moves are evaluated using a *minimax* search [19], which can be partially offloaded to a P4 pipeline, as explained in Section 4.

4 PRELIMINARY EVALUATION

We evaluate the potential benefits of our proposed L-NIC design using the Othello-Player application described in Section 2.

4.1 Experiment Setup

Single-core Othello testbed. We gather baseline measurements by implementing and evaluating an optimized Othello-Player[1] application running on a single Intel Xeon CPU running at 2.40 GHz. This implementation can search a single move ahead in 890 ns with 99-th percentile of 1.68 μs.

Distributed Othello simulator. We built a Python-based discrete event simulator[2] to evaluate performance of a distributed Othello-Player implementation. Table 1 shows the

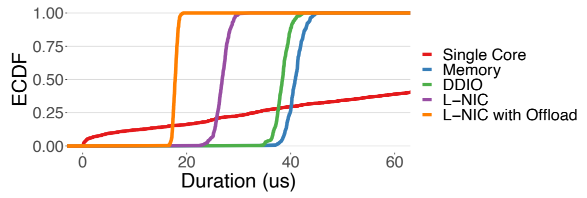

Figure 5: Completion time CDFs for searching 5 levels deep in the Othello game tree for a single-core implementation and distributed Othello implementation using traditional DMA to main memory, DMA to LLC (like DDIO), and L-NIC.

main simulation parameters. Note that the *map* message service time and the branching factor are both random variables drawn from distributions computed using the single core Othello-Player implementation.

The simulator models a topology of hosts connected to a network. Each host maintains a single queue to store requests that arrive from the network. Our simulations assume that the network has infinite capacity with no queueing in the switches.

The simulation proceeds in two phases, a *map* phase followed by a *reduce* phase. Upon initialization, one host generates a random Othello board, evaluates all possible boards one move into the future, then sends the new boards into the network, load-balancing requests across the hosts to evaluate the next move. Our simulated network load-balances map requests by hashing the message ID and mapping the result to a host. This process continues until the desired depth has been reached, at which point the map phase of the simulation is complete and the reduce phase begins. During the reduce phase, the hosts at the leaves send their results back up the Othello game tree; results are captured in reduce messages. All non-leaf hosts must wait for all the corresponding reduce messages to arrive before forwarding its own results up the tree.

Offloading Reduce processing. Upon receiving a reduce message, the host must lookup the state associated with the message to keep track of two pieces of state: (1) the number of responses that have arrived so far, and (2) the running maximum (or minimum) of the responses. These two pieces of state allow the host to decide which value to send up the tree. It turns out that this processing logic maps very efficiently onto L-NIC's P4 pipeline and thus can be offloaded. In fact, the reduce processing in a map-reduce implementation of the minimax search algorithm can always be offloaded to L-NIC's P4 pipeline.

[1]Othello-Player GitHub: https://github.com/l-nic/othello
[2]Othello-Simulator GitHub: https://github.com/l-nic/othello-sim

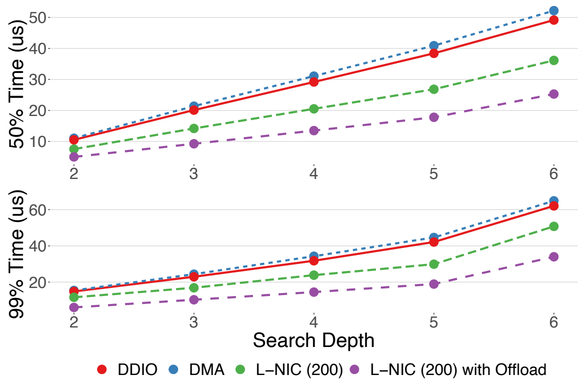

Figure 6: The 50th- and 99th-percentile completion time against varying search depths for the Othello game with different NIC implementations.

4.2 Comparing NIC–CPU Interfaces

The simulator is designed to model three types of network interfaces: DMA, DDIO, and L-NIC. DMA models the traditional network interface with a DMA engine to move packets between the network and the CPU through the host's main memory. DDIO is intended to model Intel DDIO with a DMA engine to move packets between the network and CPU through LLC. Finally, L-NIC directly moves packets between the network and CPU's register file without going through the memory hierarchy. The simulator also models the time it takes the CPU to fetch each message from the appropriate memory location (main memory, LLC, or register).

Search time evaluation. We ran the simulator to evaluate how long it takes to search *five* moves ahead for each type of NIC–CPU interface. We also evaluate performance improvements by offloading the reduce message processing to L-NIC's P4 pipeline.

Figure 5 is a CDF of the search completion times for 300 moves for each NIC–CPU interface, as well as the single-core implementation. We see that DDIO provides a very small improvement over the traditional DMA approach. This is a result of the reduction in the time to fetch each message from memory, 10 ns rather than 100 ns. By using the L-NIC interface, we are able to reduce the average completion time by 34% and 30% over the DMA implementation and DDIO implementation, respectively. The performance is improved because of the reduction in network communication latency, which in turn increases CPU utilization allowing the search to complete more quickly. If we offload the *reduce* processing onto L-NIC's P4 pipeline, it runs 56% and 54% faster than DMA and DDIO, respectively. For comparison, the 99% search time for the L-NIC implementation with reduce offload enabled is 30x lower than the 99% search time of the single-core implementation.

	DMA	DDIO	L-NIC (600 ns)	L-NIC (400 ns)	L-NIC (200 ns)
CPU (%)	70%	77%	80%	82%	84%

Table 2: The average CPU utilization across all hosts when searching 8 levels deep using 1000 host machines, a map message service time of 1 us, and a branching factor of 5.

Figure 6 shows how the median and 99% completion times change with the depth of the search. As long as there is a sufficient number of hosts to handle all of the map requests in parallel at each level of the search tree, then host queues will never build up during the map phase, and the completion time is a linear function of the depth that is searched into the tree. We see that this linear dependence is true until we reach a search depth of 6 levels. At 6 levels deep, the number of hosts required to process all the map messages in parallel exceeds the number of hosts in the topology (2500) and, thus, the host queues build up causing the completion time to be a function of both search depth and queueing delay.

CPU utilization. By reducing the communication latency between hosts, messages spend less time in the network and more time being processed by CPUs, which means that CPU utilization is increased. Table 2 shows average CPU utilization for different NIC interfaces in an experiment that simulates searching 8 levels deep on 1000 hosts. We see that, in this situation, if L-NIC is able to move packets between the network and the register file in 200 ns, then it can increase CPU utilization by 14% over the traditional DMA interface.

Typically, to increase CPU utilization we need to multiplex many application threads on the same CPU core. The context switch overheads associated with this approach lead to disastrous consequences for tail latency sensitive applications. Fortunately, L-NIC provides an alternative solution by simply decreasing network communication latency.

5 RELATED WORK

In hardware, the NIC–CPU interface has seen little attention for over two decades. Modern computer systems still use traditional (remote) direct memory access (DMA) over a slow PCIe bus. The authors of [25] demonstrated that PCIe is responsible for 80–90% of the latency between the network and CPU, which can easily exceed 1 μs. In 2012, Intel introduced Direct Data IO (DDIO) to move packets directly into LLC, reducing cache misses and memory bus contention. However, DDIO can only use 10% of the LLC, and if an application reads network data too slowly it is evicted into the main memory, eliminating the benefits of DDIO. Today, unlike L-NIC, the network industry is largely focused on offloading computation south of the PCIe onto CPUs running on

Smart NICs. But putting more CPUs in the way of packets will increase end-to-end communication latency. Furthermore, new serialization and encoding schemes are pushing down network switch latencies. Infiniband switches, for example, have latencies below 100 ns [5], and it is reasonable to assume Ethernet will follow suit.

In software, recent proposals tend to reduce latency using lightweight mechanisms (e.g., Shenango [26], Arachne [28], and Shinjuku [18]) that can benefit I/O operations taking tens of microseconds (e.g., adding an overhead of 0.55% for flash storage with service times in 10 μs). Moreover, new transport protocols (e.g., Homa [23]) demonstrate that in-network priority queues can bring down 99th-percentile latencies within 2x of the ideal. In L-NIC, we plan to leverage Homa.

6 CONCLUSION

In the past, accelerating applications meant faster CPUs and clever cache mechanisms. Today, with almost all computation done at warehouse-scale, network latency is becoming the bottleneck to distributed computing. It no longer makes sense to think of the network as a peripheral hanging off a PCIe bus; packets need to be treated as first-class citizens alongside memory transactions, and L-NIC is the first NIC design to attempt to do so. Rather than throwing more CPUs in the way, L-NIC tries to judiciously process messages in a programmable pipeline, then starts executing on them as soon as possible. We believe this is how future NICs will be architected.

ACKNOWLEDGMENTS

We thank members of the Stanford Platform Lab, Amin Vahdat, Jennifer Rexford, Mina Tahmasbi Arashloo and the anonymous HotNets reviewers for their valuable feedback that helped improve the quality of this paper. This research is supported in part by Stanford Platform Lab and Xilinx, Inc. This material is also based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7865. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and DARPA or the U.S. Government.

REFERENCES

[1] Amazon: Annapurna Labs. http://www.annapurnalabs.com/. Accessed on 06/28/2019.
[2] AWS Lambda. https://aws.amazon.com/lambda/. Accessed on 06/28/2019.
[3] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed on 06/28/2019.
[4] Google Cloud Functions. https://cloud.google.com/functions/. Accessed on 06/28/2019.
[5] Mellanox Technologies: Introducing 200G HDR InfiniBand Solutions. https://www.mellanox.com/related-docs/whitepapers/WP_Introducing_200G_HDR_InfiniBand_Solutions.pdf. Accessed on 06/28/2019.
[6] Reversi (Wikipedia). https://en.wikipedia.org/wiki/Reversi. Accessed on 06/28/2019.
[7] Serverless Use Cases. https://serverless.com/learn/use-cases/. Accessed on 06/28/2019.
[8] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM CCR 44*, 3 (July 2014), 87–95.
[9] 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. In *ACM SIGCOMM* (2013).
[10] Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C. S., Sen, K., Tillmann, N., and Visser, W. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *IEEE ICSE* (2011).
[11] Caulfield, A. M., Chung, E. S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., and Burger, D. A Cloud-scale Acceleration Architecture. In *IEEE/ACM MICRO* (2016).
[12] Dang, H. T., Sciascia, D., Canini, M., Pedone, F., and Soulé, R. NetPaxos: Consensus at Network Speed. In *ACM SOSR* (2015).
[13] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51*, 1 (Jan. 2008), 107–113.
[14] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., Chandrappa, H. K., Chaturmohta, S., Humphrey, M., Lavier, J., Lam, N., Liu, F., Ovtcharov, K., Padhye, J., Popuri, G., Raindel, S., Sapre, T., Shaw, M., Silva, G., Sivakumar, M., Srivastava, N., Verma, A., Zuhair, Q., Bansal, D., Burger, D., Vaid, K., Maltz, D. A., and Greenberg, A. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI* (2018).
[15] 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 *ACM SIGCOMM* (2017).
[16] 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.
[17] 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 *SOSP* (2017).
[18] Kaffes, K., Chong, T., Humphries, J. T., Belay, A., Mazières, D., and Kozyrakis, C. Shinjuku: Preemptive Scheduling for Microsecond-scale Tail Latency. In *USENIX NSDI* (2019).
[19] Kiefer, J. Sequential Minimax Search for a Maximum. *AMS 4*, 3 (1953), 502–506.
[20] Loveland, D. W. *Automated Theorem Proving: A Logical Basis.* Elsevier, 2016.
[21] 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 *ACM SIGCOMM* (2017).

[22] 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. In *ACM SIGCOMM* (2015).

[23] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In *ACM SIGCOMM* (2018).

[24] N. Dukkipatti, e. a. PicNIC: Predictable Virtualized NIC. In *ACM SIGCOMM* (2019).

[25] Neugebauer, R., Antichi, G., Zazo, J. F., Audzevich, Y., López-Buedo, S., and Moore, A. W. Understanding PCIe Performance for End Host Networking. In *ACM SIGCOMM* (2018).

[26] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI* (2019).

[27] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The RAMCloud Storage System. *ACM TOCS 33*, 3 (Aug. 2015), 7:1–7:55.

[28] Qin, H., Li, Q., Speiser, J., Kraft, P., and Ousterhout, J. Arachne: Core-aware Thread Management. In *USENIX OSDI* (2018).

[29] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM 59*, 11 (2016), 56–65.

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