Offloading Load Balancers onto SmartNICs

Tianyi Cui
University of Washington
cuity@cs.washington.edu

Wei Zhang
Microsoft
wei.zhang.gbs@gmail.com

Kaiyuan Zhang
University of Washington
kaiyuanz@cs.washington.edu

Arvind Krishnamurthy
University of Washington
arvind@cs.washington.edu

Abstract

Load balancers are pervasively used inside today's clouds to distribute network requests across data center servers at scale. While load balancers were initially built using dedicated and custom hardware, most cloud providers now use software-based load balancers. This allows the implementations to be more agile and also enables on-demand provisioning of load balancing workload on generic servers, but it comes with increased provisioning and operating costs.

We explore offloading load balancing onto programmable SmartNICs. To fully leverage the cost and energy efficiency of SmartNICs, our design proposes three key ideas. First, we argue that a full and complex TCP/IP stack is not required even for L7 load balancers and instead propose a design that uses a lightweight forwarding agent on the SmartNIC. Second, we develop connection management data structures that provide a high degree of concurrency with minimal synchronization when executed on multi-core SmartNICs. Finally, we describe how the load balancing logic could be accelerated using custom accelerators on SmartNICs. Our proof-of-concept implementations and preliminary results show that SmartNIC is a promising choice for navigating the underlying performance-cost tradeoffs.

CCS Concepts

• **Networks** → **Programmable networks**; **Transport protocols**.

ACM Reference Format:
Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. 2021. Offloading Load Balancers onto SmartNICs. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3476886.3477505

APSys '21, , Hong Kong, China

ACM ISBN 978-1-4503-8698-2...$15.00
https://doi.org/10.1145/3476886.3477505

1 Introduction

Load balancers are a fundamental building block for datacenters as they allow the service load to be balanced across a collection of application servers. Load balancers were initially built as specialized hardware appliances but are now typically deployed as software running on commodity servers or VMs. This deployment model provides a greater degree of customizability and adaptability than the older hardware-based designs, but it also can result in significant costs for cloud providers and application services given the purchase costs and the energy consumption of general-purpose servers.

Recently, data center operators are embracing the technological transition enabled by SmartNICs. These SmartNICs provide a cost-effective computing substrate for end-host network functionality, ranging from virtualization to security and storage. SmartNICs enclose cheap, energy-efficient (but relatively wimpy) multi-core processors that are aided by a rich array of accelerators. A SmartNIC's architecture not only places the SmartNIC's computing cores closer to the network but also enhances the computing capability with packet manipulation and cryptographic accelerators. Given the increasing relevance of this technological transition, we explore the question of how much we can accelerate the load-balancing network capability using SmartNICs.

There are a number of challenges that have to be addressed in offloading load-balancing functionality to SmartNICs. First, SmartNIC cores are wimpy, equipped with limited memory, and aren't suitable for running general-purpose computation. To the extent possible, we should use lightweight network stacks as opposed to generic, full-functionality stacks such as what is present inside OS kernels. Second, efficient multicore processing on the SmartNICs presumes lightweight synchronization for access to concurrent data structures, and this is particularly relevant as we slim down the network processing functionality. Third, effective use of accelerators both for packet transformations as well as network layer

operations, such as encryption/decryption, is necessary to enhance the computing capability of SmartNICs.

In this paper, we begin examining how to effectively accelerate different types of load balancers on different kinds of SmartNICs. We provide a preliminary design, called LB-NIC, that addresses the challenges raised above. We show that it is feasible to have a lightweight networking stack even for L7 load balancers, which are typically implemented using networking stacks that include generic TCP layer packet processing. We develop connection management data structures that are highly concurrent and minimize expensive mutual exclusion operations. We also discuss how abstract packet matching and packet manipulation accelerators can enhance the performance of load balancers.

We acknowledge that our work is preliminary, and we raise more questions than what we can answer now, given that our work is in progress. Our hope is that our designs and the associated discussions would spur not only work on SmartNIC-based acceleration of networking components but also aid in studying what primitives should be supported by accelerators on SmartNICs. This is particularly relevant as many of the commercially available SmartNICs are still in flux concerning hardware support and are grappling with what should be supported in hardware and how to provide adequate access to some of their accelerators. Further, based on some preliminary data, we propose a flipped model wherein the SmartNIC cores serve as the primary compute elements, offloading packet rewrite operations to ASIC pipelines and expensive asymmetric cryptographic operations to the host cores. Our goal with this work is to spur the discussion on how to effectively use SmartNICs in the context of applications such as network load-balancing.

2 Background

2.1 Programmable Multi-core SmartNICs

We consider programmable SmartNICs equipped with multi-core processors. A typical SmartNIC is equipped with on-board memory, DMA engines, accelerators (e.g., engines for crypto, compression, and packet rewriting), in addition to the multi-core processor. Below, we discuss the two dominant categories, on-path and off-path SmartNICs [12].

On-Path SmartNICs are SmartNICs where the NIC processing cores are on the data path between the network port and the host processor (see Figure 1. Consequently, every packet received or transmitted by the host is also processed by the NIC cores. The performance of the NIC cores is critical to the throughput and latency characteristics of the NIC. To address this issue, these NICs typically augment the traditionally wimpy cores on the SmartNIC with specialized hardware support that enhances the packet processing capabilities of the core. For example, packet contents are prefetched and placed in a structure similar to the L1 cache, and there are hardware mechanisms for managing packet buffers. Further, the NIC cores can invoke specialized accelerators for tasks such as crypto and compression. Marvell LiquidIO [13] and Netronome NICs [14] are on-path SmartNICs.

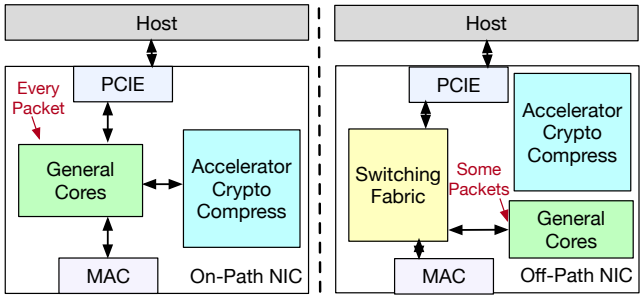


Figure 1: Common SmartNIC architectures

Off-Path SmartNICs are SmartNICs where the NIC's processing cores are off the data path connecting the host to the network. Instead, a NIC-level switching fabric (referred to as NIC-switch) provides connectivity between the network port, the host cores, and the NIC cores. The NIC-switch is a specialized hardware unit with match-action engines for selecting packet fields and rewriting them based on runtime-configurable rules. The NIC-switch rules can be used to route packets received from the network either to the host or the NIC, as well as rewrite them and immediately transmit them back into the network. Mellanox Bluefield [15] and Broadcom Stingray [1] are off-path SmartNICs.

While the on-path SmartNICs have only one type of computing (i.e., NIC cores), the off-path SmartNICs contain both general-purpose cores and packet match-action engines, and the packet processing logic could thus be split across them. However, in the case of on-path SmartNICs, the hardware constructs for packet buffers and packet prefetching enable more efficient communications and, therefore, greater efficiency for packet processing running on the NIC cores [12].

2.2 Load balancers

Load balancers operating at different network layers are widely deployed inside data centers to deliver traffic to services. They fall into two categories: layer 4 (L4) and layer 7 (L7).

2.2.1 L4 load balancer The primary function of an L4 load balancer is to map a virtual IP address (VIP) to a list of backend servers, with each server having its own dynamic IP address (DIP). As the L4 load balancers operate on the transport layer, the routing decision is solely based on the packet headers of the transport/IP layers (i.e., the 5-tuple of IP addresses and ports) without touching the payload.

There are several L4 load balancers designed and deployed by cloud providers (e.g., Ananta [17], Maglev [2], and Katran [4]). These load balancers differ in terms of the implementation strategy, e.g., user-level [2] or inside the kernel

using a driver [17] or eBPF code [4]. Most L4 load balancers, except Beamer [16], use a connection table data structure to provide per-connection consistency, i.e., route all packets within a flow to the same backend.

2.2.2 L7 load balancer L7 load balancers operate on the application layer of the OSI model, and application content (e.g., HTTP data) could be used for routing. Many services inside a data center, including microservices, may share a common application gateway implementing the L7 load balancing functionality. The L7 load balancer dispatches requests to the corresponding backend servers based on the service requested, e.g., different services are commonly differentiated by the URL [7]. The load balancer would then reassemble the stream, match the URL against various patterns to route the request to the corresponding services.

It is common for an L7 load balancer to modify the streamed application data, e.g., insert an `x-forwarded-for` header to inform the backend server of the real IP address of the client. The load balancer may also modify the reply from the server to inject a cookie into the response. As a result, for further requests from the same client, the load balancer can then route them to the same backend server based on the cookie. L7 load balancers also support transport layer security (i.e., TLS) to guarantee privacy and data integrity properties.

As the L7 load balancer works at the application layer of the networking stack, often parsing and modifying stream content and routing based on it, it is typically implemented on top of the TCP layer provided by the operating system. There are plenty of feature-rich L7 load balancers, including Nginx [3], Envoy [6], HAProxy, etc. However, given the overheads of the OS's networking stack, 50% to 90% of processing time is spent inside the kernel [8, 10].

3 SmartNIC-based Load Balancers

We consider offloading the load-balancing logic onto SmartNICs, given the cost and energy efficiency of the SmartNICs compared to host processors. Our work targets both L4 and L7 load balancers. We identify three key challenges to be addressed in order to make effective use of SmartNICs.

Lightweight networking stack: SmartNIC cores have limited processing power, e.g., the LiquidIO ARM cores have about 0.3x the processing power of host x86 cores (see Section 4). To cope with this, we provide a lightweight design that avoids the use of TCP layer processing even for L7 LBs.

Lightweight synchronization: As we streamline the processing logic, the synchronization costs for concurrent access to shared data structures would limit performance. We, therefore, design highly concurrent connection table management mechanisms for both L4 and L7 load balancers.

Effective use of accelerators: Our design considers the NIC-switch on off-path SmartNICs as a packet processing accelerator on which we can offload L4 and L7 load-balancing logic. Further, both on-path and off-path SmartNICs have accelerators for crypto that can be utilized for TLS processing.

3.1 Lightweight Networking Stack

As discussed above, L7 load balancers typically rely on the OS's kernel TCP stack to provide reliable and sequenced delivery channels that are responsive to congestion. As this approach results in significant overheads, especially if we were to deploy it on the SmartNIC cores, we propose an alternate design that utilizes a lightweight packet forwarding stack on the LB and relies on the end-hosts themselves to achieve the desired end-to-end properties. We focus on two desired functionalities of L7 LBs.

Routing based on the fields inside the HTTP header. The L7 LB can be configured to route an HTTP request based on the URL prefix. L7 load balancers buffer the entire HTTP header and then identify the backend using configured match rules.

Modify HTTP headers. L7 load balancers update HTTP header requests and responses by adding specific header values. For example, headers protect against XSS or CSRF attacks. Another common usage is to insert the client's IP address into the request header or a server identifier into a response cookie to enable a consistent level of service per client.

LB-NIC utilizes a simple forwarding agent whose operations are limited to constructing a limited portion of the stream, for routing, and buffering only the modified stream content. For reliable delivery, the forwarding agent falls back on end-host logic for reliable delivery of the rest of the stream and end-to-end congestion control.

3.1.1 State maintained We maintain a table entry for every connection that maintains a state machine for the connection. The connection state is one of the following.

`FRONT_ESTABLISHED`: the client has created a TCP connection with the load balancer, but a backend has not determined; `SYN_SENT`: The backend has been identified, but the connection hasn't yet been fully set up; and `ESTABLISHED`: the connection to the backend has been set up.

LB-NIC also maintains a shallow buffer for each connection to buffer the packets received before the backend connection is established. This buffer is required to construct the HTTP headers so that the forwarding agent can use information such as the URL to route the request.

At its core, the forwarding agent splices two TCP connections and simply relays packets after rewriting the packet and ACK sequence numbers appropriately. The forwarding agent has to maintain the mapping between the sequence

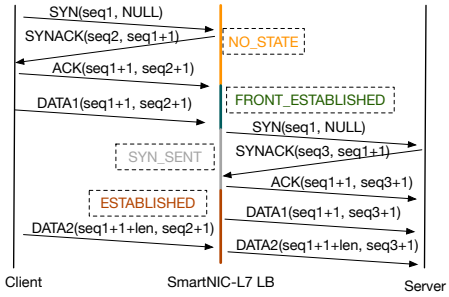


Figure 2: Connection setup flow chart

number spaces of the two connections. Since the load balancer can insert new header fields that will change the sequence number mappings, we maintain an array of "insertion points". Each insertion point records two pieces of information: the data offset where content is inserted and the size of the inserted data. For each packet, the load balancer performs a linear scan through the array, computes the total amount of inserted data before the packet, and uses this size value as an offset to adjust the sequence and ACK numbers.

3.1.2 Connection setup Figure 2 demonstrate the workflow of the entire connection establishment process.

Client's SYN received: The load balancer sends an SYNACK packet with a sequence number chosen according to the SYN cookie and the same TCP options as backend servers.

Client data received: LB-NIC buffers the packets received from the client till it can determine a backend. In particular, the load balancer will buffer client packets until it can reconstruct and parse the header fields, e.g., the hostname and the URL of the request. A connection table entry is created when the first client packet with payload is received.

Backend connection setup: After the load balancer has received sufficient client data, it can determine the backend. It then sends a SYN to the backend and completes the three-way handshake. The sequence and ACK numbers are recorded in the connection table. The buffered client packets are then forwarded to the backend server, possibly after implementing any desired header modifications. If the headers were modified, the forwarding agents holds on to the buffers until it receives the ACKs from the server; or else it releases them immediately. In the latter case, it will pass along duplicated ACKs to the client, which will then retransmit the data.

Relay established: From this point, both the connection to the client and the connection to the backend server are established and bridged. The subsequent packets will be forwarded directly without any buffering, as we discuss next.

3.1.3 Packet processing We discuss how the forwarding agent relays packets by appropriately modifying the sequence

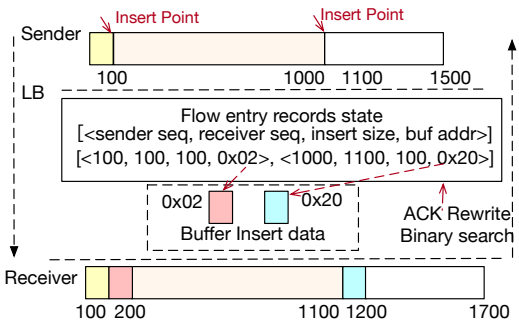


Figure 3: An example of content insertion

and ACK numbers. With HTTP 1.1 [5], a persistent connection can convey multiple HTTP requests over the same TCP connection. As a result, content insertion or modification at different locations of a TCP flow (i.e., sequence numbers) is required to support the modification of multiple requests.

Figure 3 shows an example of a TCP connection handled by the load balancer. The sender sends a flow of 1500B to the receiver, which contains two locations where we need to perform content insertions. For simplicity, we number the sequence number space starting from zero. The insertion locations are at 100B and 1000B when viewed from the sender side and at 100B and 1100B when viewed from the receiver.

When the load balancer performs an insert, it records the insertion point's sequence numbers, as viewed by the sender and the receiver. Further, it splits the original packet at the point of insertion and transmits the original packet fragments and the inserted content as separate packets.

We need to be careful about how the ACK numbers are rewritten by the forwarding agent in the presence of multiple insertions. There are several cases to consider. If the receiver's ACK is far beyond the inserted location (i.e., if the ACK number is higher than $1200+MTU$ in our example), we simply use the last offset stored in the connection table and rewrite the ACK using this offset. If the ACK is between two insertion points (i.e., it is in the range of $(201+MTU, 1100)$ in our example), we use the offset of the previous insertion point to calculate the ACK sent to the sender. If the ACK number is exactly before an insertion point and if multiple such duplicate ACKs have been received, then the load balancer retransmits the inserted content. The last case is that the ACK number is exactly after an insertion point (e.g., 201 or 1201 in our example), in which case we suppress the ACK as opposed to relaying it and triggering duplicate ACK processing on the sender. ACK packets are also used as the signal to garbage collect all the buffers buffered at the load balancer. The load balancer will check the ACKs against the stored buffers and release those that have been ACKed.

Note that the ACK number transformations ensure that the sender can use duplicate ACKs to detect lost packets and

retransmit them. The forwarding agent is responsible for reliable delivery of only the inserted content, with the sender being responsible for the reliability of all other content.

3.2 Lightweight synchronization for shared data

We now address the issue of providing lightweight and efficient synchronization for the load balancer's data structures, e.g., the connection table. Ideally, we would employ a scheme such as receive-side scaling (RSS), which would allow each NIC core having exclusive and lock-free access to its shard and avoid sharing of data structures across NIC cores. This is hard in the context of both L4 and L7 LBs as the connection table state would be accessed by traffic from both directions, and RSS would invariably map the forward and reverse directions to different cores. Thus, we need to develop lightweight synchronization techniques.

We focus first on the L4 load balancer. The primary operation of the L4 load balancer is to select a backend server, update the IP and port attributes in a packet header, and then send the packet to the chosen destination. The load balancer records the IP and port information of each flow in a connection table. A flow entry can be garbage-collected if the connection is idle beyond a configured TTL (time to live). Other than TTL, all other attributes in a flow entry are immutable during a connection's lifetime; the TTL is updated as we receive new packets in either direction for a flow.

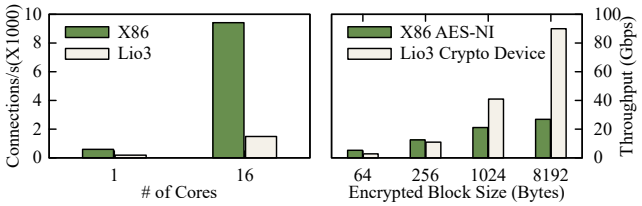
We present a design guided by the following principles. First, we avoid the use of locks for the common path. Once a backend is selected for a new flow, the majority of the packets in that flow only perform lookups, and we need to ensure that this is lock-free. Second, we reduce the scope of critical sections. Instead of locking multiple entries or even the entire table, LB-NIC utilizes fine-granularity locking and uses critical sections sparingly even during inserts. Third, we use domain-specific knowledge to pursue safe but approximate updates for connection table states such as TTL.

Concurrent connection table design. We develop a concurrent connection table design based on the cuckoo hash table [11]. A cuckoo hash table allows for hash conflicts and bounds the number of entries that have to be probed during lookup. In a cuckoo hash table, if a key exists, it can only be stored in one of two buckets, with one primary and the other secondary. If a key cannot be accommodated in its primary or secondary bucket, one of the current residents would have to be moved out. Each bucket has a configurable number of slots for collision resolution, with the number of slots providing a tradeoff between lookup cost and frequency of data movement in the presence of conflicts.

Lookup: We do not obtain a lock during the lookup operation, which means that there is a possibility that the underlying record might be concurrently moved. LB-NIC performs the following sequence of operations to avoid reading inconsistent data. It first locates a key, reads the value associated with the record (e.g., the destination IP/port), and then checks the record's key again to confirm that a concurrent move operation hasn't occurred. If the key has changed in the meantime, LB-NIC repeats the entire process of locating the record and reading its value. Upon a successful lookup, the TTL is updated to a value Δ seconds into the future using a blind write, followed by a further check that a concurrent move hasn't occurred. LB-NIC could spuriously update the TTL of a different record, thereby delaying its garbage collection, but it ensures that a received packet would always bump up the corresponding flow's TTL.

Insert: To insert a flow entry, we need a free slot in the primary or secondary buckets. If all of the slots are occupied in these two buckets, one of the existing records must be moved to its primary/secondary bucket, possibly triggering additional movements. We use a breadth-first search to identify a sequence of record moves that can help accommodate the new record. The search phase is performed without acquiring any locks on the examined records. Once a sequence of moves has been determined, LB-NIC transitions to the move phase, which will sequentially move the elements in reverse order to eventually generate an empty slot for the record that is to be inserted. During each move in this sequence, LB-NIC locks the destination slots, checks that the destination slot is still empty, verifies the source unchanged during the move and then commits. If the destination slot isn't empty, LB-NIC reenters the search phase to identify a new plan. Further, while performing a move, the record fields are written in a specific order: the record's key field is written and flushed to the memory system (using a memory fence operation) before the rest of the record data is written. This allows a concurrent lookup performed without a lock to infer that an unmodified key field also means that the record values correspond to the desired key.

L7 extensions: The L7 load balancer poses several additional challenges due to its complex logic. First, the data in a connection table entry contains a state machine, and the state transitions require synchronization. We observe that some state transitions are performed before the backend connection is fully established, and we execute them without needing locks. Second, since there are more forms of concurrent state access, a connection table entry now maintains pointers to the flow data structures instead of inlining them like in the L4 load balancer. Finally, the insertion point list has a producer-consumer relationship between the sender and the receiver, so we use lock-free mechanisms for appending to and reading from this list.



(a) Asymmetric encryption **(b) Symmetric encryption**

Figure 4: Encryption performance on x86 host and smartNIC

3.3 Hardware acceleration

Packet rewrite engines: On off-path SmartNICs, the NIC-switch or the flow engine can play a dominant role in packet processing. Abstractly, it works similar to a P4 capable switch, though there are differences between hardware vendors. Multiple match-action tables can exist in the hardware. Match and rewrite rules for various packet fields of a packet can be inserted dynamically.

For L4 LBs, our design only transfers the initial packets to a general core for figuring out the destination. Once the backend is determined, LB-NIC inserts flow engine rules to match the flow's 5-tuple and rewrites the destination IP/port. Future packets can be directly processed by the fast-path flow engine and transmitted back into the network. To evict stale flow entries, we have a background thread that will periodically determine the timed-out flow entries and then remove the corresponding rules. For L7 LBs, a flow engine with the ability to do simple arithmetic calculations to offset the sequence and ACK numbers is required. Some of the SmartNICs do have this functionality, but the software support to expose this capability is currently unavailable.

Crypto accelerators: SmartNICs provide crypto accelerators that can be used to optimize TLS. Unlike prior work that offloads just the TLS handshake onto SmartNICs [9], we take a different approach that advocates a "flipped model"; asymmetric crypto for the handshake is onloaded onto the host, and symmetric crypto is accelerated using SmartNIC's crypto engines. This is motivated by the performance of asymmetric and symmetric operations on the x86 host and the SmartNIC. Since x86 has beefy powerful cores, it can handle a higher number of TLS handshakes than lio3 cores (almost 6.3x as shown in Figure 4a). On the other hand, although x86 supports efficient AES-NI sets for speeding up symmetric crypto, using the SmartNIC's accelerators for symmetric operations can still achieve significant benefits, especially for large packets (see Figure 4b). The lio3 crypto device is about 2x faster than x86 AES-NI for 1024B packets and about 3.3x faster for 8192B jumbo packets. We therefore advocate utilizing SmartNIC's accelerators for symmetric crypto and onloading asymmetric operations onto the host.

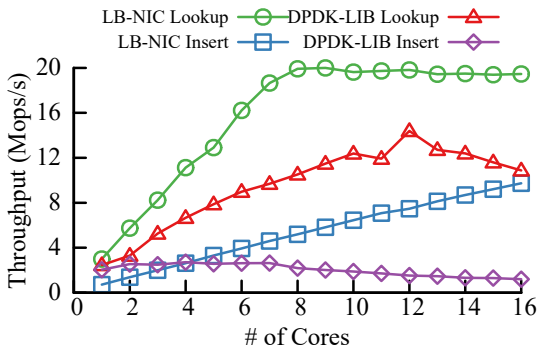


Figure 5: Hash table performance with varying cores

4 Evaluation

Our implementations are in progress, and we report preliminary results on only some aspects of our system. In our experimental setup, we use two servers. One server contains a 2x50GbE Marvell LiquidIO3 with 24 2.2GHz ARM cores and 16GB DDR4 DRAM. Both servers have two Intel Xeon CPUs (16 cores, 2.3GHz) and 96GB DDR4 DRAM, with a 100GbE Mellanox CX5 NIC. The other server acts as the client for evaluation. Both servers connect to the same 100G switch.

Hash table performance with a microbenchmark: We first evaluate the LB-NIC hash table performance by comparing it against the DPDK hash table implementation. To eliminate the performance impact from other factors such as NIC bandwidth and overhead, we pre-generate the access requests inside the test server and then show how fast the hash table performance can be. In Figure 5, LB-NIC can linearly increase the performance until the maximum throughput of 20 million packets per second with just 8 cores for the lookup operation. However, with the same number of 8 cores, the DPDK library only reaches half the performance. For inserts with 50% capacity occupation, LB-NIC can linearly increase up to 10 million packets per second, significantly better than the DPDK library, thanks to fine-grained locks and minimal critical sessions.

Hash table performance with real traffic: We next measure the performance of the L4 load balancer with real traffic. As Figure 6 shows, LB-NIC can reach the maximum NIC bandwidth of 50Gbits/s for 1024B packets with a single core. Even 64B packets can quickly scale up to the maximum speed with just 7 cores. However, the implementation based on the DPDK hash table scales badly. For small packet sizes, adding more cores sometimes even degrades performance. Even for 1024B packets, it takes 3 cores to reach the link bandwidth.

Performance gap with lio3 and x86 architectures: Finally, we compare the LB-NIC performance on two different architectures - Marvell's lio3 and x86. Figure 7 shows the throughput ratio of lio3 vs. x86, with the x86 performance as

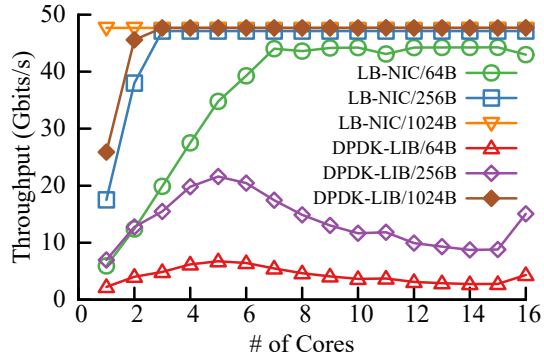


Figure 6: L4 with different number of cores

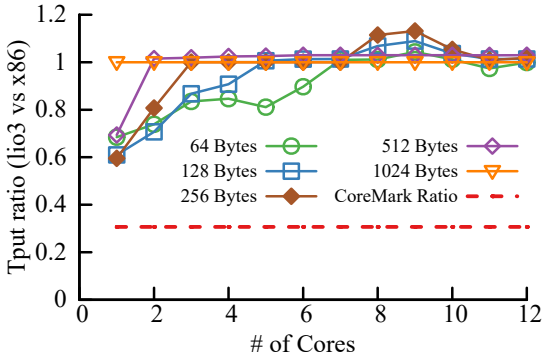


Figure 7: Performance ratio for LB-NIC: lio3 versus x86

a baseline. Note that the lio3 cores are wimpy. We characterize using the CoreMark benchmark score, which shows that the performance of lio3 cores is about 0.3x of the x86 cores (red dashed line). For the load balancing work, LB-NIC on lio3 gets much higher performance than this generic performance ratio. In fact, lio3 sometimes outperforms x86, which is significant since lio3 is cheaper and more energy-efficient than a blade server.

5 Conclusion

We examine the possibility of offloading load balancers onto SmartNICs. To leverage SmartNICs effectively, we propose a design that includes a light-weight networking stack, fast concurrent data structures, productive use of packet rewrite engines on some SmartNICs, and a flipped model for crypto traffic hardware acceleration. Our preliminary evaluations show that SmartNIC-based offload of load balancers is a promising avenue to explore.

Acknowledgments

This work is supported by NSF grant CNS-2028771. We would like to thank the anonymous reviewers for their comments and feedback.

References

[1] Broadcom. [n.d.]. Stingray SmartNIC Adapters and IC. [EB/OL]. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/smartnic Accessed Oct 25, 2020.

[2] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A fast and reliable software network load balancer. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 523–535.

[3] Inc F5. [n.d.]. NGINX high performace load balancer. [EB/OL]. https://www.nginx.com/ Accessed Oct 25, 2020.

[4] Facebook. [n.d.]. Katran. [EB/OL]. https://github.com/facebookincubator/katran Accessed Oct 25, 2020.

[5] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. 1999. RFC2616: Hypertext Transfer Protocol–HTTP/1.1.

[6] Cloud Native Computing Foundation. [n.d.]. Envoy Proxy-Home. [EB/OL]. https://www.envoyproxy.io/ Accessed Oct 25, 2020.

[7] Google. [n.d.]. Google Cloud Endpoints now generally available: a fast, scalable API gateway. [EB/OL]. https://cloud.google.com/blog/products/gcp/google-cloud-endpoints-now-ga-a-fast-scalable-api-gateway Accessed Oct 25, 2020.

[8] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 489–502.

[9] Duckwoo Kim, SeungEon Lee, and KyoungSoo Park. 2020. A Case for SmartNIC-Accelerated Private Communication. In *4th Asia-Pacific Workshop on Networking* (Seoul, Republic of Korea) *(APNet '20)*. Association for Computing Machinery, New York, NY, USA, 30–35.

[10] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. SocksDirect: Datacenter Sockets can be Fast and Compatible. In *ACM SIGCOMM Conference (SIGCOMM)*. https://www.microsoft.com/en-us/research/publication/socksdirect-datacenter-sockets-can-be-fast-and-compatible/

[11] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[12] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 318–333.

[13] Marvel. [n.d.]. OCTEON TX2 LiquidIO III SmartNIC. [EB/OL]. https://www.marvell.com/products/infrastructure-processors/multi-core-processors/liquidio-smart-nics.html Accessed Oct 25, 2020.

[14] Netronome. [n.d.]. Agilio CX SmartNICs. [EB/OL]. https://www.netronome.com/products/agilio-cx/ Accessed Oct 25, 2020.

[15] Nvidia. [n.d.]. BlueField SmartNIC Ethernet. [EB/OL]. https://www.mellanox.com/products/BlueField-SmartNIC-Ethernet Accessed Oct 25, 2020.

[16] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless datacenter load-balancing with beamer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 125–139.

[17] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. 2013. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 207–218.