Accelerator-Aware In-Network Load Balancing for Improved Application Performance

Hesam Tajbakhsh*§, Ricardo Parizotto†§, Miguel Neves*, Alberto Schaeffer-Filho†, Israat Haque*
*Dalhousie University, †UFRGS

***Abstract*—The end of Moore's law has sparked a surge on programmable accelerators (e.g., SmartNICs, TPUs) for continued scaling of application performance. However, despite the great success in offloading tasks from the CPU, we still lack proper mechanisms for balancing load among the multiple computing units present on current systems. On the one hand, traditional load balancers (either software or hardware-based) have no visibility of the different accelerators in a server and can only dispatch requests at a per-server granularity. On the other hand, emerging offloading engines can assign tasks at a finer-granularity (e.g., per-accelerator), but are hosted by the accelerator itself and thus waste precious resources for balancing load rather than processing it. This paper presents P4Mite, an accelerator-aware in-network load balancing system. P4Mite is based on two key insights: i) using programmable switches for load balancing traffic among different accelerators (and also the CPU) located in the same server; and ii) collecting statistics from each accelerator on demand for increased load visibility. We implement a P4Mite prototype on top of Intel Tofino and a Mellanox SmartNIC and evaluate it using real-world applications, including machine learning inference (VGG-16) and DNS. Our results show that P4Mite reduces flow latency by up to 50% and also makes the system handle 10-20% more load compared to standard server-level load balancing approaches. Moreover, it can process at least an order of magnitude more requests than a SmartNIC-based load balancer, with negligible latency and memory footprint.**

I. Introduction

The combination of recent slowdowns in CPU performance improvement, rapid increase in network speeds, and sharp growth in data volumes impose strain on system architects to cope with the ever-increasing application demands. As a viable alternative, many vendors are pushing the development of programmable accelerators to top up the server processing capacity. These accelerators include network (e.g., SmartNICs), storage (e.g., programmable SSDs), graphics (e.g., GPUs, visual compute accelerators), and miscellaneous (e.g., FPGAs, TPUs) devices, usually connected to the server through PCI express slots. Altogether, they have proven to be quite useful in offloading a variety of tasks from the CPU, freeing the latter to work on the most critical ones.

Current accelerators represent a significant amount of the total server processing capacity, e.g., our initial experiments showed that a single SoC-based SmartNIC could boost up to 20% of the original server performance. However, we still lack proper mechanisms for balancing load among multiple accelerators in a system. As a result, we see significant application performance degradation and resource under-utilization, especially at a larger scale with hundreds of servers in a pool. Traditional load balancers (e.g., Tiara [1], Cheetah [2], SilkRoad [3], Beamer [4], Maglev [5]) have no visibility of any accelerator in a server and can only dispatch requests at a per-server granularity. Therefore, these solutions may lead to severe imbalance and consequently latency inflation. On the other hand, emerging offloading engines ([6], [7]) can assign tasks at a finer granularity (i.e., per accelerator) but rely on either the CPU or the accelerator itself for balancing load and waste precious resources.

In this paper, we present P4Mite, an accelerator-aware in-network load balancer. P4Mite uses emerging hardware programmable switches, e.g., Intel Tofino [8], to balance connections among multiple CPUs and accelerators in a server pool and therefore explore their full capacity. To achieve its benefits, P4Mite needs to overcome a few challenges. First, the set of accelerators in a pool constitutes a heterogeneous environment and thus requires specific load balancing policies to avoid overwhelming the least capable resources. Second, balancing load at such a fine-granularity, i.e., per accelerator, puts an extra strain on the already constrained (e.g., in terms of memory) programmable switch ASICs. We overcome these two challenges by collecting load statistics from each accelerator on demand and combining a mix of data compression techniques for efficient policy representation at the switch. In particular, P4Mite relies heavily on hashing, bit mapping, and indirection to save switch memory space.

We implement a P4Mite prototype as a hybrid of P4 [9] and Python and evaluate it on a testbed containing an Intel Tofino switch, a Mellanox BlueField SmartNIC, and a commodity server. We evaluate the system performance and scalability using both microbenchmarks and real-world applications. Our results show that P4Mite reduces flow completion time by up to 50% and also makes the system handle 10-20% more load compared to standard server-level load balancing approaches. We also observed that our in-network solution outperforms load balancers running on SmartNICs by supporting at least 10x more requests. Finally, P4Mite incurs a small footprint on programmable switch ASICs, requiring less than 6% extra resources to support more than 50K concurrent connections.

In summary, this paper makes the following contributions:

- we propose P4Mite, an accelerator-aware in-network load balancer that can use emerging programmable switches

§Hesam Tajbakhsh and Ricardo Parizotto contributed equally to this work as first authors.

ISBN 978-3-903176-48-5©2022 IFIP

to distribute connections among the multiple computing units (e.g., CPU, SmartNIC, FPGA) in a pool of servers.
- we prototype P4Mite on top of a hardware switch, a SoC-based SmartNIC, and a commodity server and make our code open source [10].
- we evaluate P4Mite using three real-world applications, including machine learning inference and a DNS server, and show its superiority compared to state-of-the-art load balancing designs.

II. Background and Motivation

This section summarizes server accelerators. We assess the performance of a SmartNIC accelerator compared to an x86 CPU and show how programmable switches can assist in dispatching tasks between them.

A. Server accelerators

Server accelerators are hardware devices connected to the CPU and that are capable of running (parts of) an application faster [11]. Over the last decades, they have evolved from simple co-processors specialized at a given functionality (e.g., floating point operations, encryption/decryption) to entire systems comprising programmable processors, onboard memory and networking capabilities. As a result, many modern cloud vendors leverage accelerators to provide services, such as Google's TPU-based AutoML [12] and Microsoft Azure's FPGA-based machine learning services [13]. Other examples of existing accelerators are SmartNICs [14], programmable SSDs [15], GPUs [16] and visual compute accelerators [17].

Server accelerators are typically attached to a PCIe slot and coordinate with each other and the CPU through the system bus. From a software perspective, many accelerators (e.g., Mellanox BlueField SmartNIC [14], Intel Visual Compute [17], Broadcom STT100 programmable SSD [15]) appear as an independent machine running their own operating system and having their own IP address. These accelerators can be directly reached through standard tools (e.g., TCP/UDP sockets) thanks to IP over PCIe tunneling [18]. Other accelerators (e.g., FPGAs) cannot run an operating system and rely on firmware to communicate using the TCP/IP stack [19]. Finally, a more general class of accelerators (e.g., GPUs, most SmartSSDs) typically do not implement a TCP/IP stack and rather adopt RDMA for inter-device communication [16]. In this case, a translator (e.g., [18]) can still be used to expose the network stack that is more convenient to the client application.

B. Roofline Benchmark

To understand to which extent we could benefit from existing accelerators to improve applications' performance and make a more informed load balancing design, we assess the performance of an example accelerator (i.e., a SoC-based SmartNIC [14]) compared to an x86 CPU in this section. We used the *roofline* tool [20] to characterize their performance. The tool runs a kernel driver inside the operating system of each processing unit, i.e., both SmartNIC and CPU, which measures the maximum throughput of the respective set of processing cores.

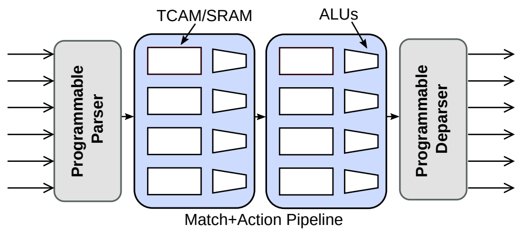


Fig. 1: PISA Architecture.

TABLE I: Roofline's Results.

Device	GFLOP/s
CPU (x86)	91.0
SmartNIC (arm)	17.6

Table I shows the results for the maximum number of floating point operations per second each processing unit can afford (in GFLOP/s). We can observe that even though the SmartNIC can handle around 5x less operations than the server CPU, it is able to provide up to 20% extra computing resources to the latter. Ultimately, that can significantly speed up the application request processing, specially in high load scenarios. Based on these observations, we argue that one can improve the performance of networked applications (e.g., machine learning inference, web serving) by carefully balancing their requests among the multiple accelerators in a server.

C. Programmable Switches

Switch programmability means that the switch functionality can be defined by the network owner using software artifacts. The switch functionality is often expressed using domain specific languages and then is tailored into a data plane model [9]. The resulting code is then compiled into a packet processing device that supports the data plane model. There are different programming models for the PDP, such as the data flow abstractions, and the protocol independent switch architecture (PISA) [21]. In this work, we will focus on the PISA architecture, which is shown in Figure 1. In the PISA programming model, packets go through a parser, which instantiates user-defined protocols. After the parser processes a packet, it follows a pipeline of control flows and *match+action* tables. Finally, packet headers are emitted by a deparser.

In this work, we use programmable switches to build an accelerator-aware in-network load balancer. Load balancing between accelerators has a few characteristics that make them suitable for programmable switches. First, by running it on programmable switches, we decrease the amount of traffic sent to the device running the applications. Second, load balancing can be achieved with simple ALU operations, making it suitable for massive connection concurrency on the programmable data plane hardware. To show the potential of our idea, we consider SmartNICs as the accelerator to build a proof-of-concept. Still, our solution can be generalized to other accelerators, such as FPGAs and programmable SSDs.

III. P4Mite Design

This section presents P4Mite, a novel accelerator-aware load balancing system. P4Mite combines *inter-server* load balancing mechanisms (e.g., ECMP, connection ID hashing [5], power-of-*k*-choices [22]) with *intra-server* balancing at the accelerator level. More specifically, the latter includes dispatching connections to either the CPU or any server accelerators (e.g., a SmartNIC or GPU) based on estimations of their load. We can fully deploy our system on programmable switches, which enables it to support many connections with high-throughput and low latency. P4Mite also ensures per-connection consistency (PCC) by keeping track of existing connections on a highly optimized connection table.

A. Challenges

We address two major challenges in P4Mite:

Load balancing in a heterogeneous environment. Unlike different servers in a pool, which tend to be uniform [23], [24], accelerators inside the same server typically form a heterogeneous system (e.g. a GPU and a SmartNIC have substantially different architectures). Even two accelerators of the same kind, such as two SmartNICs, can present varying capabilities [25]. Unfortunately, policies that perform well in a homogeneous setting can suffer from unacceptably poor performance in heterogeneous ones. To address this issue, P4Mite carefully collects load statistics (e.g., CPU utilization, request processing latency) from each computing unit in a pool. That enables our system to make more informed decisions and balance traffic according to the actual resource spare capacity. In addition, it allows P4Mite to take micro-architectural differences into account and assign priorities to accelerators. For instance, it can prioritize assigning a new request to a slightly more loaded, but beefy, CPU rather than a spare but wimpy SmartNIC.

Processing a large number of concurrent flows. Modern programmable switches cannot support many concurrent flows due to their limited memory (around 50-100 MB SRAM) [3]. As a result, previous work has proposed alternatives such as hybrid load balancer architectures, i.e., hardware/software co-designs [1] and effective compression approaches, e.g., storing connection hashes rather than the actual 5-tuple or using indirect VIP-to-DIP mappings [3]. P4Mite requires an extra level of complexity on this matter as it needs to map connections to computing units rather than servers, i.e., a more fine-grained process. Even though P4Mite still relies on a connection table to keep track of a connection's state (and thus ensure PCC), we minimize its memory footprint with a combination of data compression techniques for efficient policy representation. More specifically, our system uses hashing and bitmaps to store connection and accelerator state, respectively, and breaks down the VIP-to-DIP mapping into a two-step process, i.e., mapping a VIP to a server code and then the latter into a DIP[1]. We assess the performance of our compression approach in Section V-E.

[1]This form of indirect mapping is complementary to the one in [3], which uses the same technique specifically to reduce the connection table size.

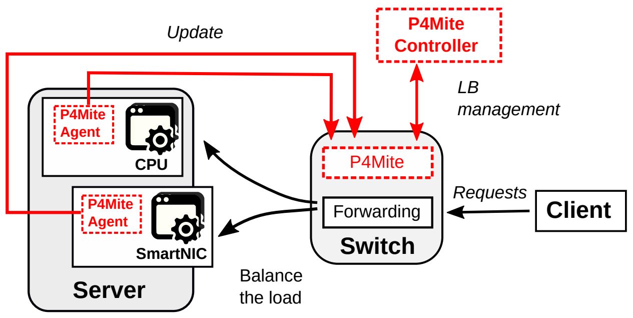


Fig. 2: P4Mite overview.

B. Overview

Figure 2 shows P4Mite's architecture overview. The system consists of a programmable switch, a controller, and server-based agents. P4Mite maintains the connection state at the programmable switch (*stateful* load balancing) and forwards requests entirely in the data plane. As a result, it can afford advanced load balancing policies with minimum impact on the flow performance. Running dedicated servers is a common practice on current data centers [26]. As such, we assume each processing unit, i.e., server CPU or accelerator, is dedicated to a single application (e.g, data analytics, VPN tunneling, or machine learning inference), even though the rack as a whole can run multiple distinct services.

Programmable switch. The switch is the core component of P4Mite. It is responsible for load balancing connections at the transport layer (i.e., L4 load balancing). P4Mite is fully compatible with the TCP/IP stack and does not require any modification on applications' client or server sides[2]. We assume every accelerator has its own IP address, i.e., can be uniquely identified, and rely on PCIe switching [27], [28], [29] to deliver packets to the right accelerator once the packet reaches a server port. As modern accelerators can usually run their own network stack [30], sometimes even a whole operating system [14], [31], it makes sense assigning a separate network namespace (or IP address) for each of them.

Controller. The P4Mite controller is responsible for implementing the desired load balancing policy (e.g., prioritizing the CPU rather than the SmartNIC when both have spare resources) on the switch by installing the associated forwarding rules. It also updates the switch configuration whenever there is a change in the server pool such as the addition/removal of a server or accelerator. All connections are handled by the switch which avoids making the controller a bottleneck. Furthermore, the controller does not interfere with other protocols or network functions as it only manages its own state, including the load balancing structures from the data plane.

Server agents. P4Mite runs an agent on each processing unit (i.e., CPU or accelerator) in an application server. The agent monitors both system and service level metrics (e.g.,

[2]Some accelerators such as FPGAs may require re-implementing an x86 application (e.g., in VHDL) to be able to run it on their specific hardware. That is transparent to P4Mite though.

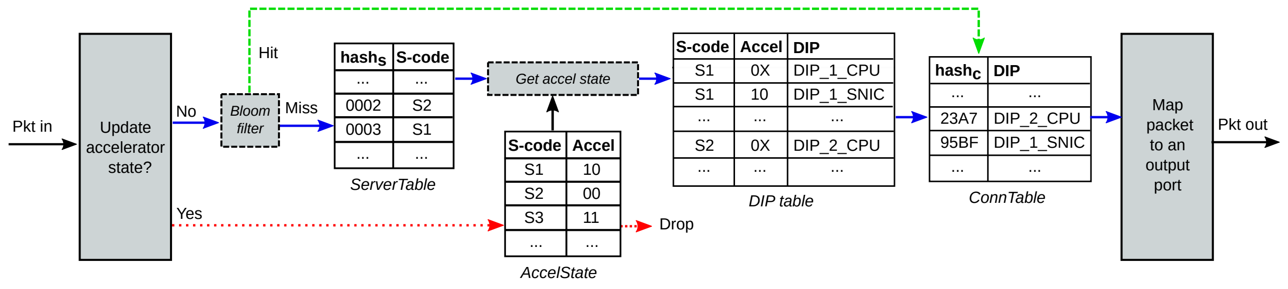


Fig. 3: P4Mite's data plane layout. Dotted red lines represent packets for updating an accelerator state. The Blue solid line represents the path for packets from connections. Green dashed lines represent packets from existing connections.

CPU utilization, request processing latency) and provides load status updates to the network load balancer. We reserved an L4 port to distinguish status update packets from ordinary data packets. Using a dedicated packet for status updates rather than piggybacking this information on data packets has two main advantages: i) servers can send replies back directly to clients instead of traversing the load balancer (a.k.a. direct source return [32]), which can be useful depending on where the load balancer is placed; and ii) agents can report more accurate information as the lag between the measurement time and its submission to the switch is minimized. To reduce the agent overhead, we also opted for it to report on a threshold-basis, meaning a status update packet will only be sent when the threshold (e.g., a maximum CPU utilization) is triggered. For simplicity we assume agents report binary states (e.g., busy or available) through the rest of this paper, although more fine-grain statuses (e.g., percentage of usage) can be considered as part of our future work.

C. Data Plane Design

Figure 3 shows the pipeline layout of P4Mite's switch data plane. Upon receiving a packet, the switch first checks whether it is a load update or a connection data packet by looking at its L4 ports. P4Mite performs a register update for the former (red dotted arrows). In this case, the register array (`AccelState`) key and values are, respectively, a unique identifier to a server (`S-code`) and a bitmap depicting the server accelerators' state, including the CPU. For example, the bitmap "10" represents a state in which the CPU is busy (and thus cannot serve new requests) but the accelerator, e.g., a SmartNIC, is available. The length of the bitmap is equal to the number of accelerators in a server plus one while the size of the register array, on the other hand, is proportional to the number of servers in the pool. Using bitmaps and identifiers (rather than actual IP addresses) helps us reduce memory consumption in the switch. Moreover, both the server code and status information are embedded in an update packet by a P4Mite agent.

Connection data packets can be either incoming or outgoing ones, which contain packets received from clients and request replies, respectively. Whenever handling an incoming packet, P4Mite uses a bloom filter to check whether it is a new connection or not. The bloom filter enables P4Mite to read and update a connection state without any involvement from the control plane. Even though false positives may happen (e.g., due to hash collisions [33]), these are negligible as long as the filter size is large enough to handle the number of connections. We leave incorporating more efficient data structures (e.g., Cuckoo filters [34]) into P4Mite's design as future work.

If a packet comes from a new connection (bloom filter `Miss`), P4Mite hashes the extracted header fields, i.e., 5-tuple, to determine a destination server (`ServerTable`). The lookup result is a server-level load balancing decision and any stateless load balancing policy (e.g., ECMP [35], WCMP [36]) can be deployed at this point. We opted for stateless load balancing policies at the server-level as those do not depend on the number of active connections and thus can save us a significant amount of memory. The downside of this approach is the non-negligible imbalance among servers (up to 30% in the worst case [5]) that can appear depending on the traffic distribution. The fact accelerators can also process requests in P4Mite's design (in addition to the CPU) can significantly amortize this imbalance though.

After a server is chosen, P4Mite retrieves its load state and uses the information, i.e., server and load, to decide the final packet destination (`DIPTable`). Each entry in the `DIPTable` can direct packets to a different accelerator and a network operator can use it to deploy various load balancing policies at the intra-server level. We explore the performance of different intra-server load balancing policies in Section V-C. P4Mite stores its load balancing decisions (i.e., each connection state) at a connection table (`ConnTable`). As a result, it ensures all packets from a connection are always delivered to the same DIP even if the pool of servers or the load balancing policy changes (a.k.a. per-connection consistency [3]). Subsequent packets from an existing connection can also be directly matched by the connection state table (bloom filter `Hit`), saving the network device a few cycles.

While the connection table is the most memory-expensive structure on P4Mite's design, its scalability is a well-known research problem and a number of techniques (e.g., hashing and indirect mapping [3]) are widely discussed in the literature to try to reduce its size. These techniques are orthogonal to P4Mite and can be used in a complementary way. Unlike incoming data packets, outgoing ones do not need to be balanced and can skip most of P4Mite's blocks (the only operation really needed is a backwards translation from a DIP to a VIP). We omit blocks associated with outgoing packets from Figure 3 for simplicity.

IV. Implementation

This section describes the P4Mite prototype implementation. Our code is publicly available at [10].

P4Mite controller and switch. We have developed P4Mite switch and its controller as a hybrid of P4-16 and Python, respectively. In total, both modules have approximately 350 lines of code. We implement the `ConnTable` as an array of 50K registers each containing a 16-bit index and use CRC16 to compute hashes. The `ServerTable` is also an array of registers with a size corresponding to the target number of servers. We retrieve server state (`AccelState`) as packet metadata and use it to match the `DIPTable`, which is a match-action table based on exact match. The switch uses standard IPv4 tables to do packet forwarding. Finally, we target our code to the Tofino Native Architecture (TNA) model.

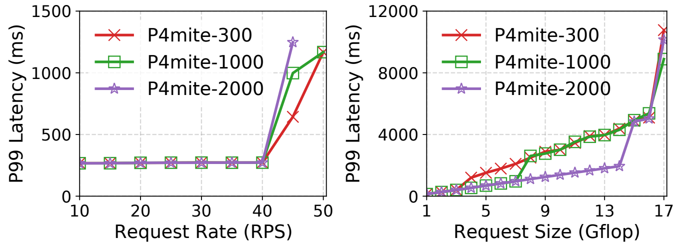
P4Mite Agents. We deployed the P4Mite agents in Python ($\sim$150 lines of code). The agents monitor the request processing latency using `tcpdump` and use it as metric to decide whether to report to the load balancer or not. We run tcpdump in "immediate-mode" so that agents can process mirrored packets on-the-fly. Even though we report results assuming the request processing latency as the main metric, our agents can be easily extended to also monitor additional metrics such as CPU utilization (e.g., using `top`) and/or network statistics (e.g., using `iftop`). We configured the P4Mite agents to encapsulate load reports and send them to the switch using UDP packets.

V. Evaluation

In this section, we use testbed experiments to evaluate P4Mite's prototype. The experiments focus on answering the following questions: (i) how P4Mite scales with an increasing load (i.e., number of requests and request size); (ii) how our system compares to traditional load balancing approaches (e.g., ECMP, WRR); and (iii) how P4Mite compares to other load balancer designs.

A. Experimental Setup

The experiments were conducted in a testbed with two hosts connected by a Wedge 100BF-32X 32-port programmable switch with a 3.2Tbps Tofino ASIC [8]. One host is used as the client, and the other acts as the server, which contains a SoC SmartNIC. The server is an Intel(R) Xeon(R) Silver 4210R CPU @ 2.4GHz, with 10 cores and 32GB memory. The SmartNIC, on its turn, is a dual-port SFP28, PCIe Gen3.0/4.0 x8, BlueField(R) G-Series, with 16 cores, 16GB on-board DDR4 RAM and crypto accelerators enabled. We prioritized the host CPU when balancing packets in our experiments



(a) Evaluating request rate (b) Evaluating request size

Fig. 4: Microbenchmarking results.

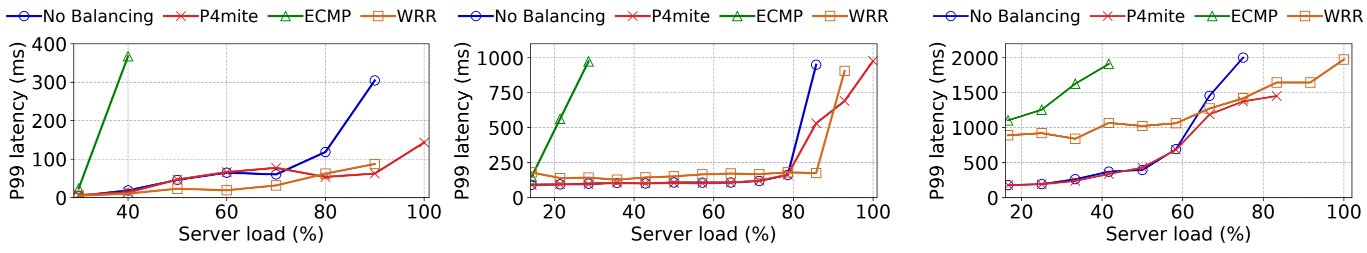
and configured its agent to report based on a combination of thresholds and time intervals for highly computation-intensive applications. That was necessary to compensate the increased performance gap between the CPU and the SmartNIC in these scenarios, when a single request may be enough to overwhelm the SmartNIC. In this case, we configured the CPU agent to send a second load status report immediately (i.e., within 5 miliseconds) after a threshold is triggered to avoid late arrivals from SmartNIC reports. Note to mention, the agent adds approximately 5% load on the server's CPU.

B. Microbenchmarks

We use synthetic workloads to investigate the effects of the request rate and request size on the performance of P4Mite. We implemented both an application capable of performing different amounts of floating-point operations for the workloads and a client application capable of sending different numbers of requests per second (RPS). Each request triggers a specific amount of computation in the server, which aims to keep the server busy and overload its CPU usage. Figure 4 presents the results of using the synthetic workloads. We performed two types of experiments using the micro-benchmarks: firstly, we evaluate how our solution behaves by varying the request rate; secondly, we assess how our solution behaves by varying the request sizes. In both experiments, we tested three different thresholds for the agents to send a load report. Next, we discuss these experiments in detail.

Request Rate. Figure 4a presents results showing how P4Mite behaves under different request rates. We configured the microbenchmark to run workloads of 2 Gflop of computation per request, varying the request rate. On the flip side, we should set a threshold for the agent running on the server. The agent sends reports once the computation time in the server hits the threshold. To rephrase it, we assume computation time in the server as the decision metric. We run the same workload using different thresholds for P4Mite (300ms, 1000ms, and 2000ms), which are referenced in the figure as P4Mite-300, P4Mite-1000, and P4Mite-2000, respectively. We stop an experiment run if we observe that packets are being dropped[3], which means the server can not handle the number of requests.

[3]In our experiments, the client assumes that a packet has been lost if either the latency of a request is 10x higher than when the system is not overloaded, or if the packet is dropped by the server.



(a) DNS (b) VGG16 (c) KNN

Fig. 5: The 99th percentile latency for specific applications.

We observe that all scenarios operate similarly up to the rate of 40 RPS, considering the latency is smaller than the smallest threshold. However, for 45 RPS, P4Mite-300 has the best performance since the agent is triggered earlier compared to the others, and the switch starts sending the requests to the SmartNIC. For the same reason, P4Mite-1000 has lower latency than P4Mite-2000. In fact, for P4Mite-2000, the agent is not triggered at all, as the latency is approximately 1200ms. The maximum increase in latency occurs for P4Mite-2000 because the server becomes overloaded and, consequently, multiple packets need to wait in the queue to be processed. As we can see in the figure, choosing a suitable threshold for the agent is essential to get a more satisfactory outcome. Moreover, this gives us evidence that while setting thresholds, we must consider thresholds that are lower than the latency of the server when it is overloaded.

Request Size. In Figure 4b we evaluate how P4Mite behaves according to different request sizes. We configured the microbenchmark to run workloads with a fixed request rate of 5 RPS, varying the request size. Similarly to what we did in the previous experiments, we stopped increasing the request size when the client started to see packet losses.

We observed that latency increases for all cases as we increase the amount of computation. For small request sizes, the difference between using various thresholds for P4Mite is negligible. For example, for 2 Gflop of computation per request, the latency is approximately 200ms in all cases. Going further, with 4 Gflop requests, the server's latency becomes higher than 300ms, triggering P4Mite-300's agent. Likewise, P4Mite-1000 is triggered at the size of 8 Gflop per request. Once P4Mite-1000's agent starts sending reports to the switch, it works similarly to P4Mite-300. Moreover, because the server is not overloaded up to this size, these two scenarios have poor performance since the switch forwards some requests to the SmartNIC, although the server could still process the requests faster.

Assuming that the server capacity is roughly 90 Gflops, and considering a request rate of 5 RPS and a request size of 15 Gflop, the server's usage reaches 75 Gflops. Further, operating system tasks and connection management take the remaining server capacity. At this point, the system starts to become overloaded, and the latency changes to 1800ms. Due to these circumstances, P4Mite-2000 performs significantly better for request sizes higher than 15 Gflop, as it utilizes the SmartNIC when the server is overloaded.

Finally, we observe that with heavier requests, P4Mite also starts to drop requests. This occurs because both the server and the SmartNIC become overloaded. However, this is acceptable since it takes 18 GFlop for P4Mite to start dropping packets while the server-only solution drops packets at 16 Gflop. We also clarify that we used a fixed threshold for different workload sizes to understand scalability aspects, but we can optimize different thresholds for each specific request size. Exploring ways for dynamically adapting the thresholds is a topic for future work.

C. Application Performance

We have implemented three client-server applications (two machine learning - VGG16 and KNN - and one network application - DNS) to compare P4Mite's performance with different load balancing strategies. For the sake of simplicity, all applications are based on UDP requests though P4Mite can also work seamlessly with TCP ones. We use DNSlib to implement both a DNS server and client. For (deep) neural networks, we adopted TensorFlow and TensorFlow Lite to implement VGG16 on the CPU and the SmartNIC, respectively[4]. Finally, for K-nearest neighbors, we used the *scikit-learn* framework.

We have deployed each application independently using P4Mite. We also deployed each application using Weighted Round Robin (WRR) and Equal Cost Multi-Path (ECMP) to compare our approach to existing ones. We observed in the Roofline results that the server CPU has $6x$ more capacity than the SmartNIC, so we configured WRR to send $1/6$ requests to the SmartNIC and $5/6$ to the server. Due to the computing demand of each application, we varied the rate to hit the maximum capacity of the system. The maximum rates the host and SmartNIC handle jointly are 2750, 70, and 24 RPS for DNS, VGG16, and KNN, respectively. Besides, for DNS servers, which are not CPU-intensive, we use only one core of each resource, while for the other applications, all cores are involved. Finally, we consider 100ms, 150ms, and 300ms as the threshold for DNS, VGG16, and KNN, separately. These are 20-30% more than usual delays for the applications.

We ran each test for 30 seconds and measured request loss and latency for each request. Figure 5 presents the 99th percentile latency for each application. Like what we did for the microbenchmarks, we stopped the execution as soon as we observed a packet loss.

[4]We were not able to run the TensorFlow framework on the SmartNIC due to architectural limitations from the latter.

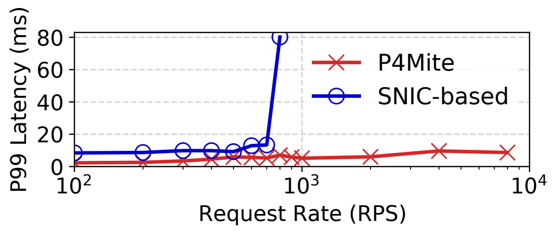


Fig. 6: P4Mite vs. SmartNIC-based Load balancer.

TABLE II: Amount of resources used by P4Mite.

Resource	Usage %
SRAM	5.1%
TCAM	0.0%
VLIW Instructions	2.6%
Hash Bit	4.0%
Stats ALU	2.1%
Map RAM	5.6%
Exact Match Input Xbar	2.9%

DNS. Figure 5a presents the results for the DNS application. Given that DNS is not CPU intensive, both server and SmartNIC have the same delay (around 3ms) when their CPU is not under stress. The blue curve (no balancing) shows that the server can handle requests up to 80% load. At higher loads, the server delay increases drastically. Using P4Mite, the agent sends reports to the switch when the server starts to get overloaded. Consequently, a portion of the load is forwarded to the SmartNIC, increasing the maximum rate by approximately 20%. P4Mite not only avoids overloading the server but also the offload of requests to the SmartNIC allows more injunctions to be processed. As such, one core of the SmartNIC's wimpy processor can handle 20% more DNS queries. Regarding ECMP, since it tries to split the requests between the server and SmartNIC evenly, and considering that the SmartNIC's computation power is one-sixth of the server, the SmartNIC becomes quickly overloaded. Conversely, in the weighted round-robin, the switch dispatched 5/6 of the requests to the server and the remaining to the SmartNIC, enabling it to run faster than both the baseline and P4Mite for lower rates. However, as we reach 90%, WRR drops packets. P4Mite beats WRR because it wisely balances the load, while WRR proactively distributes the load. Nevertheless, WRR has better performance for less load because P4Mite needs to wait for the server to start to get overloaded before it is triggered.

VGG16. Figure 5b presents the results for VGG16. Without any stress, the latencies of the server and the SmartNIC are approximately 80ms and 120ms for VGG16, respectively. In the no balancing scenario, the server can process until 80% load, but the latency increases sharply after that. In P4Mite, for loads higher than 80%, the latency grows and the agent sends reports to the switch, causing a portion of the requests to be sent to the SmartNIC. P4Mite not only improves the latency by 50%, but it also can handle 16% more requests. Figure 5b also indicates that if we use ECMP, the system can handle only 30% load. We investigated these results and noticed that the SmartNIC becomes fully utilized, causing packets to be dropped. Finally, considering WRR as the load balance approach, we observe that it increases the latency compared to the no balancing approach when the load is less than 80%. This occurs because WRR always sends 1/6 of traffic to the SmartNIC, which performs slower than the server. However, WRR enables the application to go up to 90% load because the requests sent to the SmartNIC alleviate the server.

KNN. Figure 5c presents the results for KNN, the most CPU-intensive application in our evaluation. In this scenario, the baseline solution can handle 75% load until packets start to be dropped. P4Mite improves the latency and maximum rate by 25% and 11%, respectively, compared to the no balancing solution. We could not get better results for KNN because our SmartNIC is not powerful enough to process heavier requests within the time constraints. We observe that ECMP handles at most 40% load, where 20% load is forwarded to the SmartNIC. Also, the latency for ECMP is higher than other approaches because the SmartNIC executes KNN requests far slower than the server. Finally, while WRR works poorly for loads below 70%, it can handle more load than P4Mite. Our investigation shows that for a load of 90% and higher, requests are lost for P4Mite because of request time expiration in the server, not because of packet drop.

D. *P4Mite vs. SmartNIC-based load balancer*

One of the advantages of P4Mite is its ability to perform load balancing operations at line-rate. To better assess our solution, we compare it with a SmartNIC-based load balancer that runs on a single core[5] of the Mellanox Bluefield. This alternative implementation is an L4, software-based load balancer, distributing the load among the host and the SmartNIC, and its load-balancing logic is similar to P4Mite's. Figure 6 compares P4Mite and the SmartNIC load balancer for an increasing number of requests per second. For less than 700 RPS, the SmartNIC-based load balancer is slower than P4Mite. This happens because all requests should go through the SmartNIC first, although most of them will be sent to the host anyway. Further, the SmartNIC-based load balancer starts dropping packets after 700 RPS. This happened because the networking resource was fully utilized and could not handle more connections. Conversely, P4Mite can handle up to 8K RPS without dropping packets.

E. *Resource Consumption*

Finally, we measured the resource overheads of P4Mite when deployed on our Tofino switch. We configured the switch to support up to 256 servers with two accelerators. Table II presents the resource usage of P4Mite assuming 50k concurrent connections. This is close to the maximum number of connections we can support with a single pipeline stage. Unrolling our mechanism across different pipeline stages can enable more concurrent connections [3].

[5]Although this SmartNIC-based load balancer could work on multiple cores, fewer resources would be available for processing requests.

We observe that for every switch resource, P4Mite's overhead is always less than 6%. P4Mite uses 5.1% of SRAM and 5.6% of Map RAM mainly due to the implementation of the bloom filters. More specifically, we see that the `ConnTable` dominates the usage of these resources to store connection statuses. Because we use hash functions to map packets to both the `ServerTable` and the `ConnTable`, the Hash Bit usage is 4.0%. The VLIW is used for writing values into packets; since P4Mite only writes into packets their destination IPs, VLIW usage is only 2.6%. We are also using only 2.9% exact match input xbar to perform the exact match to get the accelerator state and match the DIP table. The bloom filters also use 2.1% of stats ALU to update the accelerator's state or to store state about a new connection. Finally, P4Mite does not use any TCAM, which is an expensive resource.

F. Discussion

Stateful Applications. While P4Mite is extensively evaluated for stateless applications, other applications (e.g., key-value stores, distributed file systems) may include storing and manipulating state, either locally or as distributed shared variables. Further research could explore the overheads (e.g., memory and data communication demands) of such applications as part of the load balancing decision process.

Other load balancing policies. P4Mite load balancing policies are currently restricted by the information reported by its agents (i.e., binary flags indicating whether an accelerator is underutilized or not). Future work could investigate adding support for more complex policies. For example, collecting multiple statistics from an accelerator (e.g., CPU, memory and network usage) and reporting them to the P4Mite switch could enable policies based on the least utilized resource.

Multiple Accelerators. When compared to state-of-the-art load balancing approaches, the P4Mite improvements ultimately depend on the capabilities of the available accelerators. For example, we could observe up to a 20% latency improvement based on our SmartNIC, which is in line with the results reported in Section II-B. As part of our future work, we consider extensively evaluating P4Mite's performance for balancing traffic across multiple accelerators, including GPUs and programmable SSDs.

VI. Related work

In-network load balancing. Many in-network load balancing solutions have been proposed recently. SilkRoad [3] leverages programmable switches to implement a layer-4 load balancer. Instead of storing the 5-tuple connection information, it uses a hash to overcome switch memory constraints and support millions of concurrent flows. A similar approach is proposed in Loom [37], where the authors deploy multiple bloom filters to compress the connection states. Cheetah [2] takes a different path and stores information about connection states into packet headers rather than inside the switches. In common, none of these approaches have visibility of the accelerators in a server and can only dispatch requests at a per-server granularity.

CrossRSS [38] provides a CPU core-aware stateful LB, where network interface cards (NICs) are used to compute a hash for the core selection. Charon [39], in turn, uses an FPGA-based SmartNIC to select the appropriate server based on their load. In this case, the load balancer receives load updates over packet headers from agents running on the servers. Cui et al. [40] use an ARM-based multi-core SmartNIC to run a lightweight LB. The authors design concurrent connection management mechanisms to cope with the limited LB performance when accessing shared data structures. Finally, Tiara [1] improves the performance of stateful LBs by distributing the task to three types of computing resources: programmable switches, FPGA-based SmartNICs, and server CPUs. Specifically, the switch performs throughput-intensive packet encapsulation/decapsulation while FPGAs/CPU cores take care of the memory-intensive real server selection. Even though these load balancers can assign tasks at a finer granularity (e.g., per CPU core), they rely on the accelerators themselves such as the SmartNICs to perform the load balancing decisions, which wastes precious computing resources.

Task offloading to SmartNICs. Previous work has widely explored offloading tasks to SmartNICs, including distributed transactions [41], distributed file systems [42], binarized neural networks [43] [44], and image classification [45]. These works are complementary to P4Mite and can directly benefit from its load balancing strategies. IPipe [7] proposes an actor-based framework for judiciously offloading distributed applications to SmartNICs. The framework includes an LB that resides inside the NIC and monitors the overall system resource utilization. Similarly to other SmartNIC-based load balancers, IPipe wastes precious computing resources with management rather than processing tasks. Finally, E3 [6] uses a centralized resource controller to efficiently place micro-services among processing nodes (e.g., SmartNICs, server CPUs). Unlike P4Mite, their resource controller is designed to run on general purpose servers and cannot process new requests at line rate.

VII. Conclusion

We introduce an accelerator-aware in-switch load balancer, P4Mite, which distributes the load among CPUs and accelerators with different capacities. As opposed to load balancers that are not resource-aware, P4Mite relies on agents, which monitor available resources, and performs load balancing based on this information. We implemented P4Mite on top of a Tofino switch using P4_16, allocating the load between a server and an SoC SmartNIC. Our experiments show that P4Mite can handle 10-20% more load and can reduce flow latency by up to 50%. P4Mite judiciously balances the load, and does not send requests to wimpy resources whenever powerful ones are free. Moreover, P4Mite can handle a 10x higher rate compared to a SmartNIC-based load balancer, without consuming a significant amount of resources in the switch.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable feedback. This research is supported in part by NSERC under a Discovery Grant and CFI under a JELF grant. Alberto Schaeffer-Filho would like to thank CNPq (grant #311276/2021-0) and FAPERGS (grant #19/2551-0001645-0).

References

[1] C. Zeng, L. Luo, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng *et al.*, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[2] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. M. Jr., P. Papadimitratos, and M. Chiesa, "A high-speed load-balancer design with guaranteed per-connection-consistency," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 667–683.

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

[4] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 125–139.

[5] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 523–535.

[6] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: Energy-efficient microservices on smartnic-accelerated servers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 363–378.

[7] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartnics using ipipe," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, 2019, p. 318–333.

[8] Intel, *Explore the Power of Intel® Programmable Ethernet Switch Products*, Intel, 2021 (accessed April 15, 2021), https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, p. 87–95, 2014.

[10] "P4mite," https://github.com/PINetDalhousie/p4mite, 2022.

[11] J. Nider and A. S. Fedorova, "The last cpu," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, p. 1–8.

[12] "Google AutoML. [n.d.] AutoML: Train high-quality custom machine learning models with minimal effort and machine learning expertise," https://cloud.google.com/automl/.

[13] "Microsoft brainwave. [n.d.]. brainwave: a deep learning platform for real-time ai serving in the cloud," https://www.microsoft.com/en-us/research/project/project-brainwave/.

[14] Mellanox, *BlueField SmartNIC for Ethernet High Performance Ethernet Network Adapter Cards*, Intel, 2021 (accessed April 15, 2021), https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[15] J. Do, I. L. Picoli, D. Lomet, and P. Bonnet, "Better database cost/performance via batched i/o on programmable ssd," *The VLDB Journal*, pp. 403–424, 2021.

[16] S. Wang, C. Lou, R. Chen, and H. Chen, "Fast and concurrent RDF queries using RDMA-assistedGPU graph exploration," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 651–664.

[17] "Intel. [n.d.]. intelÂő visual compute accelerator (intelÂő VCA) product brief," https://www.intel.com/content/www/us/en/homepage.html.

[18] M. Tork, L. Maudlej, and M. Silberstein, "Lynx: A smartnic-driven accelerator-centric architecture for network servers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 117–131.

[19] H. T. Johansson, A. Furufors, and P. Klenze, "Fakernet–small and fast fpga-based tcp and udp communication," *arXiv preprint arXiv:2003.12527*, 2020.

[20] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline model toolkit: A practical tool for architectural and program analysis," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds., 2015, pp. 129–148.

[21] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Computing Surveys (CSUR)*, pp. 1–36, 2021.

[22] M. D. Mitzenmacher and A. Sinclair, "The power of two choices in randomized load balancing," Ph.D. dissertation, 1996.

[23] L. A. Barroso, U. Hölzle, and P. Ranganathan, "The datacenter as a computer: Designing warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, pp. i–189, 2018.

[24] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, and B. Wang, "HiveD: Sharing a GPU cluster for deep learning with guarantees," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 515–532.

[25] S. Choi, M. Shahbaz, B. Prabhakar, and M. Rosenblum, "λ-nic: Interactive serverless compute on programmable smartnics," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 67–77.

[26] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[27] "Braodcom. [n.d.]. PCI Express switches," https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches.

[28] "PCIE Switches — Arrow Electronics," https://www.arrow.com/en/categories/electronic-switches/pci/pci-express-switches.

[29] "Mellanox. [n.d.].ConnectX®-5 EN Card," https://network.nvidia.com/files/doc-2020/pb-connectx-5-en-card.pdf.

[30] "TCP/IP offload overview," https://docs.microsoft.com/en-us/windows-hardware/drivers/network/tcp-ip-offload.

[31] Broadcom, *Stingray PS225*, Broadcom, 2021 (accessed April 15, 2021), https://docs.broadcom.com/doc/PS225-PB.

[32] S. Rajagopalan, "An overview of layer 4 and layer 7 load balancing," *Computer Networks, Big Data and IoT*, pp. 663–672, 2021.

[33] S. Cohen and Y. Matias, "Spectral bloom filters," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 241–252.

[34] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[35] C. Hopps *et al.*, "Analysis of an equal-cost multi-path algorithm," RFC 2992, Internet Engineering Task Force, Tech. Rep., 2000.

[36] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "Wcmp: Weighted cost multipathing for improved fairness in data centers," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[37] "Loom: Switch-based cloud load balancer with compressed states," 2021.

[38] T. Barbette, M. Chiesa, G. Q. Maguire, and D. Kostić, "Stateless cpu-aware datacenter load-balancing," in *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, 2020, p. 548–549.

[39] C. Rizzi, Z. Yao, Y. Desmouceaux, M. Townsley, and T. H. Clausen, "Charon: Load-aware load-balancing in p4," 2021.

[40] T. Cui, W. Zhang, K. Zhang, and A. Krishnamurthy, *Offloading Load Balancers onto SmartNICs*, 2021, p. 56–62.

[41] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy, "Xenic: Smartnic-accelerated distributed transactions," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, 2021, pp. 740–755.

[42] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 756–771.

[43] G. Siracusano and R. Bifulco, "In-network neural networks," *arXiv preprint arXiv:1801.05731*, 2018.

[44] G. Siracusano, S. Galea, D. Sanvito, M. Malekzadeh, H. Haddadi, G. Antichi, and R. Bifulco, "Running neural networks on the nic," *arXiv preprint arXiv:2009.02353*, 2020.

[45] H. Siddique, M. Neves, C. Kuzniar, and I. Haque, "Towards network-accelerated ml-based distributed computer vision systems," 2021.