λ-NIC: Interactive Serverless Compute on Programmable SmartNICs

Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum
Stanford University

ABSTRACT

There is a growing interest in serverless compute, a cloud computing model that automates infrastructure resource-allocation and management while billing customers only for the resources they use. Workloads like stream processing benefit from high elasticity and fine-grain pricing of these serverless frameworks. However, so far, limited concurrency and high latency of server CPUs prohibit many interactive workloads (*e.g.*, web servers and database clients) from taking advantage of serverless compute to achieve high performance.

In this paper, we argue that server CPUs are ill-suited to run serverless workloads (*i.e.*, lambdas) and present λ-NIC, an open-source framework, that runs interactive workloads directly on a SmartNIC; more specifically an ASIC-based NIC that consists of a dense grid of Network Processing Unit (NPU) cores. λ-NIC leverages SmartNIC's proximity to the network and a vast array of NPU cores to simultaneously run thousands of lambdas on a single NIC with strict tail-latency guarantees. To ease development and deployment of lambdas, λ-NIC exposes an event-based programming abstraction, *Match+Lambda*, and a machine model that allows developers to compose and execute lambdas on SmartNICs easily. Our evaluation shows that λ-NIC achieves up to 880x and 736x improvements in workloads' response latency and throughput, respectively, while significantly reducing host CPU and memory usage.

1 INTRODUCTION

Serverless compute is emerging as an attractive cloud computing model that lets developers focus only on the core applications—building the workloads as small, fine-grained custom programs (*i.e.*, lambdas)—without having to worry about the infrastructure they run on. Cloud providers dynamically provision, deploy, patch, and monitor the infrastructure and its resources (*e.g.*, compute, storage, memory, and network) for these workloads; with tenants only paying for the resources they consume at millisecond increments. Serverless compute lowers the barrier to entry, especially, for organizations lacking expertise, manpower, and budget to efficiently manage the underlying infrastructure resources.

Today, all major cloud vendors offer some form of serverless frameworks, such as Amazon Lambda [10], Google Cloud Functions [21], and Microsoft Azure Functions [24], along with open-source versions like OpenFaaS [55] and OpenWhisk [6]. These frameworks rely on server virtualization (*i.e.*, virtual machines (VMs) [3]) and container technologies (*i.e.*, Docker [79]) to execute and scale tenants' workloads. These technologies are designed to maximize utilization of the providers' physical infrastructure, while presenting each tenant with its own view of a completely isolated machine. However, in serverless compute, where server management is hidden from tenants, these virtualization technologies become redundant, unnecessarily bloating the code-size of serverless workloads, and causing processing delays (of hundreds of milliseconds) and memory overheads (of tens of megabytes) [91]. The increased overheads also limit the concurrent execution (less than hundred or so) of these workloads on a single server, hence, raising the overall cost of running such workloads in a data center.

The cloud-computing industry is now realizing these issues and some providers, such as Google and CloudFlare, have already started developing alternative frameworks (like Isolate [42]) that remove these technology layers (*e.g.*, containers) and run serverless workloads directly on a bare-metal server [42]. However, these bare-metal alternatives are inherently limited by the design restrictions of CPU-based architectures, which exacerbate when running at the scale of cloud data centers [65]. CPUs are designed to process sequence of instructions (*i.e.*, a function) blazingly fast. They are not designed to run thousands of such small, discrete functions in parallel—a typical server CPU has in the order of 4 to 28 cores that can run up to 56 simultaneous threads [22]. Each serverless function interrupts a CPU core to store the state (*e.g.*, registers and memory) of the currently running function and load itself with the new one, resulting in wasting tens of milliseconds worth of CPU cycles per context switch (such wasted cycles increase the overall costs for the cloud providers [56]). Thus, with ever increasing network speeds—100/400G NICs are on the horizon—these overheads quickly add up, limiting throughput and leading to long-tail latency in the order of 100s of milliseconds [70].

Recently, public cloud providers are deploying SmartNICs in an attempt to reduce load on host CPUs [56]. So far, these attempts have been limited to offloading ad-hoc application tasks (like TCP offload, VXLAN tunneling, overlay networking, or some partial computation) to accelerate network processing of the hosts [75, 76, 87]. However, modern

SmartNICs, more specifically ASIC-based NICs, consist of hundreds of RISC processors (*i.e.*, NPUs) [11], each with their local instruction store and memory. These SmartNICs are more flexible and can run many discrete functions in parallel at high speed and low latency—unlike GPUs and FPGAs, which are optimized to accelerate specific workloads [39, 56, 89],

Serverless workloads by design are small, presenting unique opportunities for SmartNICs to accelerate them, while also achieving strict tail-latency guarantees. However, the main shortcomings of using SmartNICs come from their programming complexity. Programming SmartNICs is a formidable undertaking that requires intimate familiarity with NIC's system and resource architecture (*e.g.*, memory hierarchy, and multi-core parallelism and pipelining). Developers need to carefully partition, schedule, and arbitrate these resources to maximize performance of their applications, which is a characteristic that is counter to the motivation behind serverless compute (*i.e.*, where developers are unaware of the architectural details of the underlying infrastructure). Furthermore, each application has to explicitly handle packet processing as there is no notion of a network stack in SmartNICs.

In this paper, we present λ-NIC, a framework for running interactive serverless workloads entirely on SmartNICs. λ-NIC supports a new programming abstraction (Match+Lambda) along with a machine model—an extension of P4's match-action abstraction with more sophisticated actions—and helps address the shortcomings of SmartNICs in five key ways. First, users provide their lambdas, which λ-NIC compiles and then, at runtime, selects to execute by matching on the header of the incoming requests' packets. Second, users program their lambdas assuming a flat memory model; λ-NIC analyzes the memory-access patterns (*i.e.*, read, write, or both) and sizes, and optimally maps these lambdas across different memory hierarchies of the NICs while ensuring that memory accesses are isolated. Third, λ-NIC infers which packet headers are used by each lambda and automatically generates the corresponding parser for the headers, thus eliminating the need for manually specifying packet-processing logic within these lambdas. Fourth, instead of partitioning and scheduling a single lambda across multiple NPUs, λ-NIC assumes a run-to-completion (RTC) model exploiting the fact that lambdas are small and can run within a single NPU. The vast array of NPU cores and short service times of lambdas further mitigate head-of-line-blocking and performance issues that lead to high tail latency. Lastly, serverless functions mostly communicate using independent, mutually-exclusive request-response pairs and do not need the strict in-order, streaming delivery semantic provided by TCP [72]. λ-NIC, therefore, employs a weakly-consistent delivery semantic [47, 72], alongside RDMA [93], for communication between serverless workloads—processing requests directly within the NIC cores without involving the host CPU. In summary, we make the following contributions:

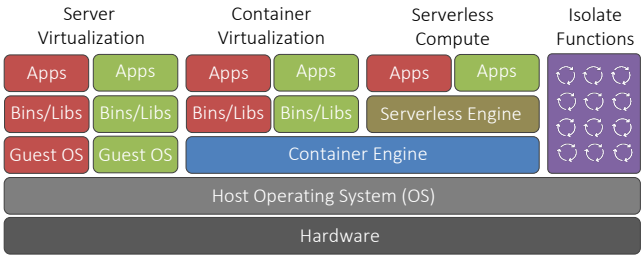

Figure 1: Overview of cloud-computing frameworks and how they partition compute across various layers.

- We introduce a new abstraction for our λ-NIC framework, called Match+Lambda (§4.1), and a machine model (§4.2) to easily and effectively code and execute lambdas on modern SmartNICs.
- We develop an open-source implementation of λ-NIC using P4-enabled Netronome SmartNICs (§5), and implement methodologies to optimize lambdas to efficiently utilize the SmartNIC resources (§5.1).
- We evaluate λ-NIC and show up to two orders of magnitude improvement in latency and throughput compared to existing serverless compute frameworks (§6).

We begin with a background on the current state-of-the-art in cloud-computing frameworks and SmartNICs (§2) followed by the challenges and motivations behind λ-NIC (§3). We then describe the overall architecture of λ-NIC (§4) with an extensive evaluation of the system (§6). Finally, we conclude by discussing λ-NIC's limitations and future works (§7), and comparisons with related implementations (§8).

2 BACKGROUND

We now discuss the latest advancements in cloud-computing frameworks and programmable SmartNICs, which are the core building blocks of λ-NIC.

2.1 Cloud Computing Frameworks

Figure 1 illustrates four different cloud-computing frameworks in use today. *Server virtualization* is the foremost technology underneath cloud computing that allows a bare-metal server to host multiple VMs each with their independent, isolated environments containing separate Operating Systems (OS), libraries, and applications. It arrived at a time when advances in hardware made it difficult for a single application to efficiently consume the entire bare-metal server resources, and having multiple applications co-existing on a single server raised various issues (such as isolation and contention for resources). However, as trends shifted from monoliths to building applications as microservices [58]—to

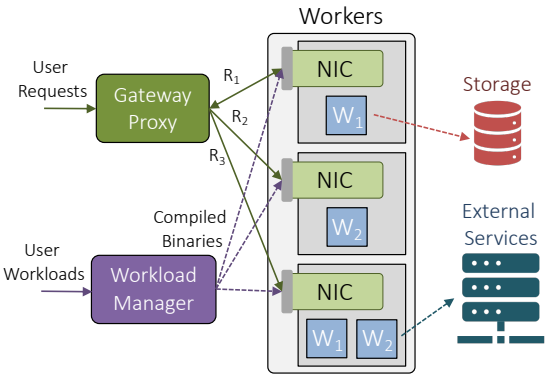

Figure 2: An overview of a general serverless compute framework that executes users' workloads and serves requests to such workloads. (R_i represents the ith request to workload W_i).

increase manageability, resiliency, and scalability—the overheads of having a separate OS for each microservice were no longer negligible. This gave rise to *container virtualization* [79], which is a way to isolate applications' binaries and libraries while sharing an OS.

Still, with growing complexity and scale of cloud workloads, it became daunting for many users to provision and manage infrastructure resources with tasks requiring fine-grain allocation of resources under changing workload demands. Early solutions settled on over-provisioning these resources, incurring added cost for idle resources. More recently, *serverless compute* [59] has emerged as a favorable compute model that alleviates such operational and financial burdens from the users by letting them specify only the workloads, their memory and timing constraints, and when and which events (*e.g.*, API calls, database updates, incoming data streams) to trigger them. In response, cloud providers independently provision infrastructure resources, deploying a set of containers on-demand to serve workloads' requests. These containers are quickly taken down once the workloads complete and users are charged only for the time a container is executed. Serverless workloads are therefore short-lived with strict compute-time and memory limits (up to 15 minutes and 3 GB, respectively, for Amazon Lambda [14]).

Serverless frameworks. Figure 2 depicts a typical serverless compute framework. Workload manager compiles users' workloads to executable binaries, which along with their data dependencies are stored in a global storage (*e.g.*, Amazon S3 [7], Google Cloud [31], or Microsoft Azure Storage [46]). Gateway proxies users' requests (or events) to appropriate workloads, which typically run as containers on a set of dynamically provisioned servers, called worker nodes. Upon completion, results are written back to the storage or forwarded to other workloads for further execution.

Serverless compute frameworks embed users' workloads within containers managed by orchestration engines (*e.g.*, Docker) running atop an OS, which provide memory, compute, and file-system isolation using OS-based mechanisms (*e.g.*, cgroups [78] and namespaces [25]). Each container maintains its own libraries and binaries, and communicates with other containers using an overlay network that is set up using virtual switches (like Open vSwitch (OVS) [86]).

However, running workloads as containers incur additional processing and networking overheads. As an alternative, *Isolate functions* [42] run workloads directly on the bare-metal server itself (Figure 1), while providing all the benefits of a typical serverless framework (*e.g.*, resource isolation between workloads). Although still in early development, these Isolate functions are showing promising results: up to 3x improvement in request latency while consuming 90% less memory than containers with faster startup times.

Serverless workloads. Today, serverless frameworks find applications in two types of use cases [13, 29, 96]: (1) Running API backends that serve interactive applications, such as returning static or dynamic content, or key-value responses based on users' requests. (2) Processing changes in data stores at real-time, such as cropping a newly uploaded image or running a custom operation on a newly added database entry.

The complexity of lambda functions ranges from running simple arithmetic operations to making machine-learning decisions that are generally short-lived. A bare-metal server can house thousands of such lambda functions; however, doing so causes CPUs to constantly context switch between these functions. These, along with other communication overheads, make it difficult for services (using lambda functions) to meet their tail-latency service-level objectives (SLOs) [42].

To mitigate these issues, efforts are underway to move computation down to SmartNICs; however, unlike λ-NIC, the focus is mostly on offloading either network processing or some small portion of applications to these NICs [75, 76, 87].

2.2 Programmable SmartNICs

In addition to handling basic networking tasks, SmartNICs can offload more general tasks that a CPU normally handles (*e.g.*, checksum, TCP offload, and more). Based on their architecture and processing capabilities, these SmartNICs come in three different types: FPGA-, ASIC-, and SoC-based [56]. Table 1 summarizes the main differences.

Alongside other issues (*e.g.*, steep development cost and power consumption [56]), FPGA-based NICs [1] are dominated by the on-chip interconnect overhead [50] that significantly limits the lookup tables (LUTs) and memory (*e.g.*, SRAM) resources available for executing lambda functions—today's large FPGAs can barely support a small number of

	FPGA-based SmartNICs	ASIC-based SmartNICs	SoC-based SmartNICs
Programmability	Hard	Limited	Easy
Performance	10+ cores, low latency	200+ cores, low latency	50+ cores, high latency
Development cost	High	Medium	Low

Table 1: A comparison of various types of SmartNICs.

processing cores (< 10 or so). SoC-based NICs (*e.g.*, Mellanox Bluefield [16] and Broadcom Stingray [15]) are easier to program as they run a Linux-like OS on embedded cores (like ARM); however, similar to server CPUs, they are susceptible to high tail latency due to context switch and network stack overheads. Therefore, it's questionable that these SoC-based NICs can support speeds higher than 100 Gb+ with low latency [56].

Due to these limitations of FPGA- and SoC-based NICs, we opted for the ASIC-based NICs when designing λ-NIC. These NICs consist of an ASIC that can sustain traffic rates of 100 Gbps+; contain hundreds of non-cache-coherent multi-threaded RISC cores (*e.g.*, NPU, ARM, or RISC-V), operating at GHz speeds, along with specialized hardware functions (*e.g.*, lookup, load balancing, queuing, and more); and are capable of running embarrassingly parallel workloads with low latency. Furthermore, recent advances in the design of these SmartNICs (*e.g.*, Netronome Agilio [11] and Marvell LiquidIO [23]) make it easier for users to customize the NIC's data-plane logic (*e.g.*, parse, match, and action) using, for example, P4 [43] and Micro-C [27] programs, thus exposing dataflow and C-like abstractions a typical programmer is familiar with, without the need for an OS.

In §6, we demonstrate how the unique characteristics of serverless workloads (*i.e.*, short-lived with strict compute and memory limits) make ASIC-based SmartNICs a viable execution platform to accelerate lambda functions.

3 MOTIVATION & CHALLENGES

The key motivation behind λ-NIC is to accelerate interactive serverless workloads by: (1) mitigating excessive compute and network virtualization overheads and inefficiencies of the modern server architecture to achieve low and bounded tail latencies for lambda functions, and (2) exploiting the right domain-specific architecture (*i.e.*, ASIC-based SmartNICs) to sustain high throughput while reducing CPU load and cost-per-watt in cloud data centers.

Low latency and high throughput serverless functions. The key tenet of serverless compute is that it establishes a clear demarcation between users and infrastructure providers; users only specify programs (or functions) that the providers efficiently execute on their infrastructure. Yet, all modern serverless frameworks are based on technologies (*i.e.*, VMs and containers) that were designed to give users explicit control over the underlying infrastructure from the get-go. This control—in the form of compute and network (physical and overlay) virtualization—adds a significant overhead to serverless functions. For interactive serverless workloads, with strict tail latency SLOs, eliminating such computational and networking overheads is becoming crucial [35].

The modern server architecture (with CPUs and GPUs) further adds to these overheads. CPUs are Von Neumann machines designed to efficiently execute a long sequence of instructions (or a function). However, they perform poorly when executing a large number of small serverless functions where significant time is wasted context switching between functions. Similarly, GPUs are Single-Instruction-Multiple-Data (SIMD) machines that serve as look-aside accelerators [41] in a typical server, controlled by the primary CPU. Although, in recent years, these GPUs have shown orders of magnitude improvements in accelerating machine-learning workloads [51, 98] (which by nature are long-running, batch jobs), they perform poorly for low-latency, interactive tasks (like serverless functions). Even with technologies (such as GPUDirect RDMA [19] and RDMA-over-Converged-Ethernet (RoCE) [5]) that can bypass a CPU and push data directly into the GPU or main memory, the requests for serverless functions still have to traverse a NIC—adding non-negligible delays in the order of sub-microseconds.

λ-NIC eliminates both these virtualization and architectural overheads by running serverless workloads directly on the vast array of NIC-resident RISC cores.

A domain-specific processor for serverless functions. Till now, cloud providers have almost always relied on newer, faster CPUs to improve applications' performance. More recently, they have started looking into other fine-grain, domain-specific processors, like look-aside or bump-in-the-wire accelerators (GPUs or FPGAs [56]). This is because, with Moore's Law slowing down [64], CPUs today are no-longer a viable solution to meet ever-rising performance demands of customer workloads in a cost-efficient way—as has been demonstrated by both GPUs (for improving machine-learning training and inference throughput [51]) and FPGAs (for accelerating search indexes [89] and host networking [56]). We believe that ASIC-based SmartNICs present the same opportunity for accelerating serverless workloads with orders of magnitude improvement in performance-per-watt at one-tenth of the hardware cost [30, 85, 95], compared to server CPUs and GPUs [56].

3.1 Key Challenges

The embarrassingly-parallel and independent nature of serverless workloads take away much of the complexities that arise when synchronizing state between functions [75], making them an ideal candidate for SmartNICs with hundreds of cores. Still, executing them on these NICs is not a panacea and comes with its own unique challenges:

a. Programming SmartNICs. Due to their non-cache-coherent design, programming ASIC-based SmartNICs has always been considered hard [49]; non-coherency requires developers to program each NPU core separately, forcing them to manually handle synchronization between individual functions. With serverless functions, however, this is no longer an issue as functions do not share state and can run independently. Still, the lack of an OS layer in these NICs—though useful in reducing unwanted processing—puts the onerous of mapping and placing these functions, across various clusters of cores and memory hierarchy, on the developers; requiring them to have low-level knowledge of the NIC architecture, firmware, and specialized languages it supports. To take this burden away from developers, we need a high-level abstraction and a framework that can automatically and efficiently compile, optimize, and deploy serverless programs across a collection of these SmartNICs.

b. Offloading serverless workloads. NPUs are optimized for packet forwarding, and they typically do not support features (*e.g.*, floating-point operations, dynamic-memory allocation, recursion, and reliable transport) that a general-purpose CPU supports [60, 67]. A serverless framework, therefore, must be able to compile workloads that rely on these features by, for example, transforming programs with floating-point operations to fixed-point arithmetic [40], dynamic-memory allocations to explicit memory calls, recursions to iterations, as well as employing other forms of reliable (or weakly-consistent) delivery protocols (*e.g.*, RoCEv2 [5], Lightweight Transport Layer [47], or R2P2 [72]). To achieve high throughput and low latency, emerging workloads are also lowering their dependency on these features. For example, deep-learning training and inference is shown to perform well with lower, fixed bit-width integers [40, 54]. Moreover, serverless request-response (RPC) pairs are mostly independent and mutually-exclusive, and do not need TCP's strict, reliable, and in-order streaming delivery of messages [72].

c. Ensuring security under multi-tenancy. Lambdas run alongside other workloads (*e.g.*, microservices) in a data center and share infrastructure resources. When using SmartNICs: (1) a serverless framework should ensure that lambdas running on the NICs do not interfere with each other or degrade the network performance between the NIC and host CPUs that are running traditional workloads. (2) The framework should reserve ample SmartNIC resources (*i.e.*, cores and memory) for basic NIC operations (*e.g.*, TCP/IP offload and checksums) while maximally consuming remaining resources for serverless functions. (3) Serverless functions should execute in their own isolated sandboxes and the framework should restrict them from accessing each others' working set. (4) Lastly, the framework should be robust against security attacks (*e.g.*, DDoS) both from outside actors and malicious tenants.

4 λ-NIC OVERVIEW

λ-NIC adds a new backend to existing serverless frameworks with its own programming abstraction, called Match+Lambda (§4.1), and the accompanying machine model (§4.2) that makes it easier to program and deploy lambdas directly on a SmartNIC.

4.1 Match+Lambda Abstraction

λ-NIC implements a Match+Lambda programming abstraction that extends the traditional Match+Action Table (MAT) abstraction [44] with more complicated actions (lambdas).

Programming lambdas. In λ-NIC, users provide one or more lambdas written in a restricted C-like language, called Micro-C [27].[1] Listing 1 shows the signature of the top-level function, which each lambda must begin with, having two arguments: `headers` and `match_data`. The number and structure of all the supported headers (*i.e.*, the `EXTRACTED_HEADERS_T` data structure) and function parameters (*i.e.*, `MATCH_DATA_T`), in λ-NIC, are defined a-priori. The lambdas operate directly on these parameters and headers without having to parse packets, which is done at the parse stage (Figure 3). Furthermore, these functions can have both local objects as well as global objects that persist state across runs.

```
1 int function_name(EXTRACTED_HEADERS_T *headers,
      MATCH_DATA_T *match_data)
2 {
3   // local/global memory and objects.
4   return return_value;
5 }
```

Listing 1: Signature of the top-level function in Micro-C for the Match+Lambda abstraction.

Listing 2 shows a real-world example of a lambda running as a web server. The function reads the server address (Line 6) from the `headers` variable. It then copies the requested web content from the memory into the header location pointed by the server address (Line 8), before returning.

[1] We use Micro-C as it is the native language of the SmartNICs we have for the evaluation. The Micro-C language can support a large class of serverless functions (§3.1); however, λ-NIC is not just limited to this language and can work with more feature-rich languages supported by other SmartNICs.

```
#define MEM_PER_LAMBDA 20
uint8_t memory[MEM_PER_LAMBDA * 3];
int web_server(EXTRACTED_HEADERS_T *headers,
    MATCH_DATA_T *match_data)
{
  serverHdr_T *serverHdr =
    hdr_get_serverHdr(headers);
  memcpy(serverHdr->address, memory,
    MEM_PER_LAMBDA);
  return RETURN_FORWARD;
}
```

Listing 2: An example of a web-server lambda.

Expressing match. The user further specifies the corresponding P4 code[2] for the match stage (Listing 3). During compilation, the workload manager assigns unique identifiers (IDs) to each of these lambdas, shares this mapping with the gateway, and populates the ID variables (*e.g.*, `WEB_SERVER_ID`, `OTHER_LAMBDA_ID`) in the P4 code. For each incoming request, the gateway inserts the ID of the destined lambda as a new header. The match stage of a λ-NIC (as defined in the P4 code), checks the ID listed in the new header and calls the matching lambda (implemented as an extern in P4 [33]) or sends the packet to the host OS, in cases where no matching ID is found.

```
control ingress {
  if (valid(lambda_hdr)) {
    if (lambda_hdr.wId == WEB_SERVER_ID) {
      apply(web_server);
      apply(return_web_server_results);
    } else if (lambda_hdr.wId == OTHER_LAMBDA_ID)
    {
      apply(other_lambda);
      apply(return_other_lambda_results);
    }
  } else { apply(send_pkt_to_host); }
}
```

Listing 3: Snippet of a P4 code for the match stage.

In the end, the workload manager pairs the lambdas (Micro-C code) and match stage (P4 code) into a single Match+Lambda program, and prepends it with a generic P4 packet-parsing logic. It then compiles and transforms this program into a format that the target SmartNIC can execute (§5), while ensuring fair allocation of resources and isolation between lambda workloads.

4.2 Abstract Machine Model

In λ-NIC, users write their Match+Lambda workloads against an abstract machine model (Figure 3). In this model: (1) lambdas are independent programs that do not share state and are isolated from each other; only a matching rule can invoke these functions. (2) The match stage serves as a scheduler (analogous to the OS networking stack) that forwards packets to the matching lambdas or the host OS. Finally, (3) a parser handles packet operations (like header identification), and lambdas operate directly on the parsed headers.

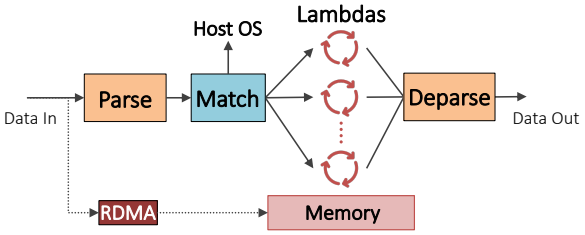

Figure 3: λ-NIC's abstract machine model.

These properties of Match+Lambda machine model make it easier for software developers to express serverless functions by separating out the parsing and matching logic, as well as for hardware designers to efficiently support the model on their target SmartNICs (§5 demonstrates one such implementation using the Netronome SmartNICs). Thus, the abstract machine model enables unique optimizations that lets serverless workloads run as lambdas in parallel without any interference from each other.

4.2.1 Design Characteristics.

The abstract machine model has the following three design characteristics:

D1: Run-to-completion execution. λ-NIC executes each lambda to completion. The machine model exposes a dense array of discrete, non-coherent processing threads (Figure 3), having their own instruction and data store in the memory. The lambdas execute in the context of a given thread, maximally utilizing the resources of that thread only (*e.g.*, CPU and memory). Given the short service times and strict memory footprints of serverless functions, modern SmartNICs hold ample resources per thread to execute these lambdas [27].

Having a large number of parallel threads, further mitigate issues related to head-of-line-blocking where lambdas wait behind other lambadas to finish or context switch, as in the case of server CPUs. These issues severely affect the performance of lambdas at the tail and require more complicated scheduling policies (like preemption [68] or core allocation [35]). With λ-NIC, however, this is not the case as lambdas—even at the tail—can run to completion without degradation in performance (§6).

Moreover, the highly-parallel nature and run-to-completion characteristic of the machine model ensure strong ***performance isolation*** between different lambdas running as separate threads, and λ-NIC implements weighted-fair-queuing (WFQ) [84] to route requests between these threads. We

[2]We use P4 as it is the most widely used data-plane language [43].

leave it as future work to explore more sophisticated resource-allocation mechanisms (*e.g.*, DRF [61]) to further improve the performance of λ-NIC.

D2: Flat memory access. The abstract machine model lets users write lambdas assuming a flat (virtual) memory address space. All objects (local and global) on the thread stack are allocated from within that address space. This has the advantage that users do not have to worry about the complex memories, and their structure and hierarchies, present in modern SmartNICs [27]. Each of these memories come with their own performance benefits and are necessary to reach high speeds in these NICs; however, having so makes it the responsibility of the programmers to efficiently utilize these memories. λ-NIC's machine model takes this onerous away by exposing a single, uniform memory to the user.

When deploying to a particular SmartNIC, the compiler (or the workload manager) can take into account NIC's specifications and can perform target-specific optimizations to effectively utilize its memory resources (§5.1). The users can also provide pragmas—specifying which objects are read more frequently—to guide the compiler in allocating objects to memories based on their access needs; it can place small or hot objects to core-local memories, and large or less frequently used ones in external, shared memories.

Furthermore, having a virtual memory space per lambda can let the compiler enforce policies for ***data isolation***, since virtual spaces do not interfere and are isolated from each other. The compiler can insert static and dynamic assertions [36] to ensure that a lambda does not access the physical memory of other lambdas on the target SmartNIC.

D3: Network transport. In λ-NIC, the primary mode of communication between the gateway, lambdas, and external services (*e.g.*, storage) is via Remote Procedure Calls (RPCs). These RPCs are small, typically single-packet, request-response messages [8, 72]. The parser, in the abstract machine model (Figure 3), decomposes these messages into headers and forwards them to the match stage, which further directs them to a matching lambda (by looking up the lambda ID associated with each message). Multi-packet RPCs, depending upon their size, can either be processed directly by the parse and match stage or pushed into the memory over RDMA (*i.e.*, RoCEv2 [5]). (In the latter case, an event RPC triggers the lambda to start reading data from the desired memory location.)

Already, modern datacenter applications (like Amazon DynamoDB [12] and Deep Learning [40, 54]) are choosing to go away with the strict, reliable, and in-order guarantees provided by TCP, which are far stronger and computationally intensive than what applications need. Instead, these applications are being designed to work with weaker guarantees to achieve low tail latency [72]. λ-NIC exploits these facts

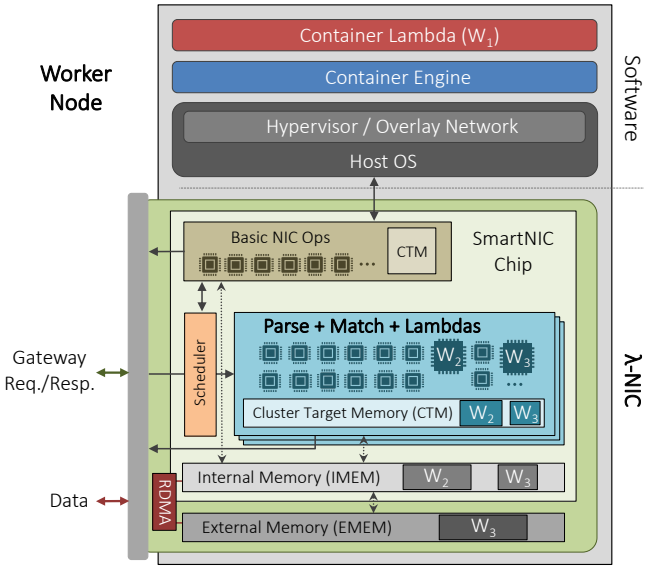

Figure 4: Architecture of a λ-NIC worker node on a P4-enabled Netronome NIC.

and assumes a weakly-consistent delivery semantic for RPCs that are processed by the parse and match stage. A sender (the gateway or external services) tracks the outgoing RPCs to lambdas, and is responsible for resending a message in case of timeouts or packet drops. λ-NIC, on the other hand, performs packet reordering at the SmartNIC for multi-packet RPCs.[3]

5 IMPLEMENTATION

In this section, we present an implementation of λ-NIC on P4-enabled Netronome SmartNICs (Figure 4). These NICs contain hundreds of RISC cores grouped together into islands. Each core has its own instruction and local memory, as well as a Cluster Target Memory (CTM) [27] per island, and is capable of executing multiple threads, concurrently. There are also on-chip internal memories (IMEMs) and an external memory (EMEM) shared between all islands and their cores; and a dedicated scheduler unit that directs incoming packets to cores. The architecture of Netronome SmartNICs therefore has the necessary elements to efficiently implement λ-NIC's Match+Lambda abstract machine model: cores can execute lambdas to completion (D1), data can reside in different memories (*e.g.*, local, CTM, or more) based on their usage patterns (D2), and the scheduler can direct RPCs to lambdas (D3).

A more programmable scheduler (*e.g.*, RMT/PIFO-based [37, 94]) can execute the parse and match stage of the machine model directly, with cores only processing the lambda logic.

[3] We measured that Netronome SmartNICs can reorder four 100 B packets using 120 instructions, which is only 1.3% of the instructions used by our benchmark lambdas (§6.4).

However, the scheduler inside the Netronome NICs we used for our evaluation is not programmable; it is work-conserving and uniformly distributes incoming traffic to all cores. We therefore execute all three stages (parse, match, and lambdas) together inside a core, with every core running the same Match+Lambda program.[4,5] For single-packet messages, the scheduler directs an incoming packet to a core at random, which after parsing selects the matching lambda using the ID embedded in the packet. The multi-packet messages are committed to memory via RDMA, and lambdas read directly from the desired location.

5.1 Target-Specific Optimizations

Next, we discuss optimizations, to improve the execution time and binary size of Match+Lambda programs (§6.4), that arise as a result of our design choices and architectural constraints of Netronome NICs.

Lambda coalescing. As multiple lambdas run on a single core, the workload manager runs program analysis (*i.e.*, dead-code elimination and code motion [92]) to remove duplicate logic (*e.g.*, for modifying similar headers or generating packets) and move it into shared libraries as helper functions.

Match reduction. The workload manager can further reduce the number of tables (*e.g.*, for route-management) defined in the match stage of a Match+Lambda program. Each new lambda consists of both a parse and a match stage; the workload manager can compose these stages with the logic already running on the core, removing the unused headers and duplicate match fields from the final code. Furthermore, the P4 tables are converted into if-else sequences, which the NIC core can execute more efficiently. Transforming tables into if-else sequences also helps reduce the total number of instructions of the final binary, running on a core.

Memory stratification. Based on the access patterns, the workload manager can choose the most efficient memory for an object at compile time. It can also look at the object size or hints from the user (as pragmas) to decide whether to put the object in a local memory, CTM, IMEM or EMEM.

6 EVALUATION

We now compare the performance of λ-NIC with bare-metal and container backends both in isolation, when executing a single lambda (§6.3.1), and in a shared setting, when running multiple lambdas together (§6.3.2). We also evaluate the impact of λ-NIC on resource utilization, startup times, as well as the effectiveness of the λ-NIC compiler to optimize lambda program size when running on the NIC cores (§6.4).

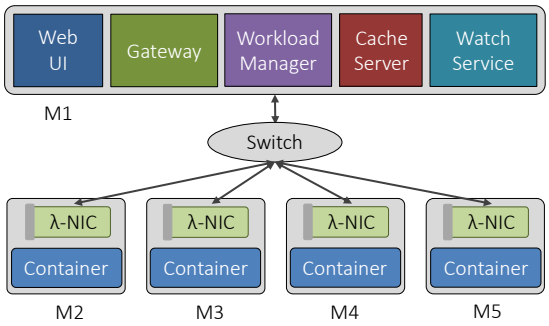

Figure 5: An overview of the testbed having 4 worker nodes and a master node all connected to a 10 G switch.

6.1 Test Methodology

6.1.1 Baseline framework. To evaluate λ-NIC against existing serverless backends, we select OpenFaaS [55] as our baseline serverless compute framework, which closely resembles the architecture depicted in Figure 2. We choose OpenFaaS for its simplicity, ease of deployment, extensive feature set, and greater adoption by the community.[6] It is written in Golang [88] and includes: (1) a Web UI, (2) an autoscaler to scale lambdas as demands change, (3) a Prometheus [45] based monitoring engine to analyze system state and (4) a gateway with a NAT [97] to proxy users' requests to the appropriate lambdas. Each of these components and lambdas run as Docker [79] containers, managed via Kubernetes [90] or Docker Swarm [4].

Adding a bare-metal backend. For evaluating emerging runtimes like Isolate [42], we add support for a bare-metal backend to OpenFaaS. It is implemented as a Python service[7] that runs on a bare-metal server as a standalone process, launching lambdas as new threads to serve users' requests. The service relies on a Raft-based [83] distributed key-value store, called etcd [20], to sync lambda-related states (*e.g.*, number of active lambdas, their placement and load balancing policies) with the gateway to correctly proxy requests. Our goal, using the bare-metal backends, is to analyze how performance of lambdas improves in the absence of the container processing stack.

Introducing λ-NIC extensions. We built λ-NIC as an extension to our baseline framework, inheriting all of OpenFaaS's core features with additional support for running

[4] The other approach is to pipeline these stages and run them on separate cores; we intend to look into this as future work.

[5] At present, the binary running on SmartNICs must be swapped with a new one each time, resulting in downtimes. However, this constraint is expected to disappear in the next-generation NICs (§7).

[6] OpenFaaS is the most favorable open-source serverless compute framework with ~12,000 stars on GitHub [18].

[7] We found the performance of the Python service similar to a comparable C implementation except for the one-time startup cost of the backend. We, therefore, used the Python service for its ease of integration with OpenFaaS.

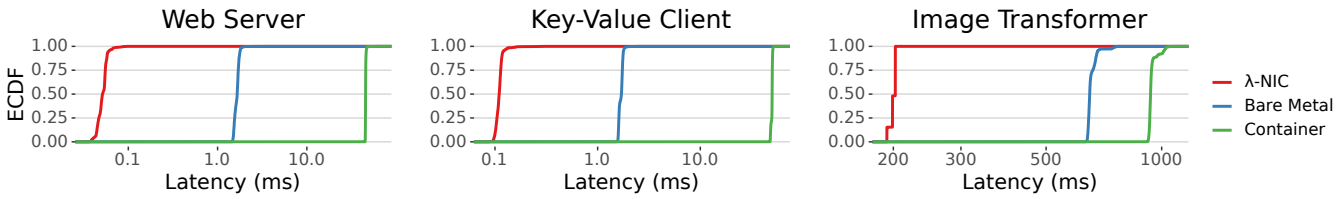

Figure 6: ECDF of latencies, in log scale, when executing a single workload instance in isolation.

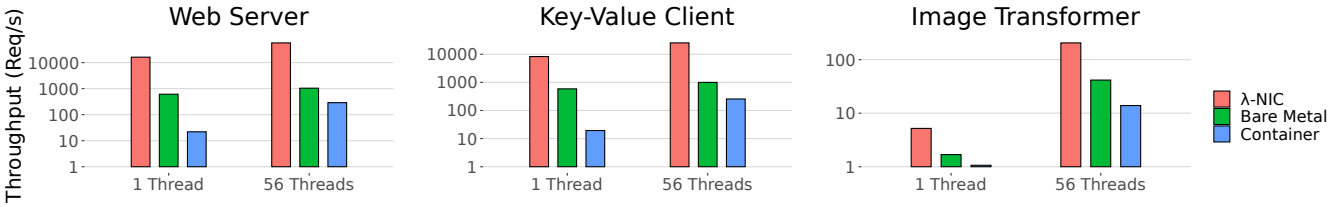

Figure 7: Average throughput, in log scale, when executing a single workload instance in isolation. (Standard deviation is negligible across all runs.)

lambdas on P4-enabled Netronome SmartNICs. With these extensions, the baseline framework can simultaneously deploy lambdas to containers, bare-metal, and SmartNIC backends. We also augment etcd to share state and manage λ-NIC deployments across multiple worker nodes.

6.1.2 Testbed setup. Our evaluation testbed consists of a cluster of five servers (Figure 5) housing two Intel Xeon Gold 5117 processors with 14 physical cores, running at 2.0 GHz with 32 GiB DDR4 2666 MT/s Dual-Ranked RAM and 120 GiB SATA SSD. One of the servers (M1) act as a master node running: Kubernetes services, gateway, workload manager, memcached server [2], the web interface, and the monitoring engine. M1 comes with a Broadcom 57412 2x10 Gb and 2x1 Gb Quad-Port NIC, which is used for management traffic. Other servers (M2–5) are worker nodes equipped with a Netronome Agilio CX 2x10 Gb SmartNIC [11] having 56 RISC cores (8 threads and 16 K instructions per core) running at 633 MHz with 2 GiB of on-board RAM. All servers connect to an Arista DCS-7124S switch over a 10 Gbps link. The backends communicate over an overlay network using Kubernetes' calico [17] networking plugin for high performance switching and policy management across nodes inside a Kubernetes cluster.

6.2 Benchmark Workloads

We evaluate λ-NIC on three different types of interactive lambdas (*i.e.*, a web server, a key-value client, and an image transformer), each reflecting a popular use case [13, 29, 96].

a. Web server. A common usage pattern for lambdas is to serve web contents [13], such as text or HTML pages, similar to traditional web servers (like nginx [32]). These workloads are typically self-contained and do not need information from external sources (*e.g.*, data stores) to service a request. For our experiments, we wrote a lambda that returns text responses based on the incoming requests.

b. Key-value client. Next, we consider workloads with external dependencies, needing information from remote services. These workloads query users' data from external storage, *e.g.*, databases or key-value stores (such as memcached [2]), do customization on the retrieved data, and finally send the processed data back to the user. Moreover, these workloads generate extensive intra-data center requests and typically have strict tail-latency requirement to meet user service-level objectives (SLOs). To evaluate, we implement lambdas acting as key-value clients that generate write (SET) and read (GET) requests to a memcached server.

c. Image transformer. Finally, we evaluate workloads that involve real-time, interactive processing of large datasets (*i.e.*, image processing or stream processing) [48, 100], where the datasets span multiple packets and must be stored in memory (*i.e.*, DRAM). These workloads perform customization to the requested datasets, and either return a response to the user immediately or store results back to the memory for further processing [9]. For our evaluation, we consider lambdas that transform RGBA images to grayscale.

6.3 System Performance

We now discuss how λ-NIC performs, both in terms of latency and throughput, compared to the bare-metal and container backends. We evaluate two cases: (1) when there is only a single lambda running on a backend in isolation, and (2) when there are multiple lambdas running, all contending for the shared resources (*i.e.*, processing cores and memory).

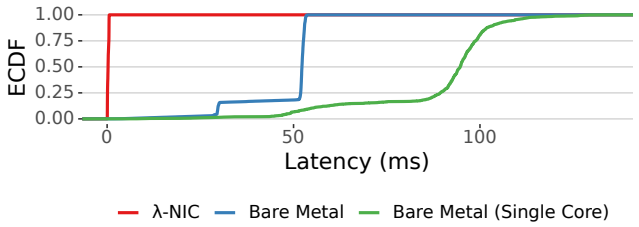

Figure 8: CDF of latencies when running three distinct web server lambdas concurrently.

6.3.1 Performance in Isolation. We first look at the latency and throughput of a lambda in isolation.

Latency. We measure the latency of each backend, which is the time it takes for a gateway to send a request to a node running a single thread of the pre-loaded (or warm) lambda and receive a response back (Figure 6). For web server and key-value client lambdas, λ-NIC outperforms containers by 880x and bare-metal by 30x in average latency—completing requests in under 100 ns—while still achieving 5x to 3x improvements for the data-intensive image-transformer lambda. Improvement are more visible at the tail (*i.e.*, 99th-percentile) where λ-NIC achieves 5x to 24x better tail latency than bare-metal for the three benchmark lambdas. For the key-value client lambdas, λ-NIC even improves upon reported latencies in a highly-optimized cloud-scale data center [82] by three orders of magnitude. Furthermore, container and bare-metal exhibit longer tail latency, specifically, for short-lived web server and key-value lambdas. This is likely the artifact of miscellaneous software overheads (*e.g.*, context switching, cache management, and network stack).

Throughput. We see similar improvements in throughput, measured as request serviced per second, for the three lambdas when running on λ-NIC. We carryout two separate experiments: (1) closed-loop testing with sender generating each request one after the other, and (2) parallel testing with 56 requests—the maximum number of threads that can run simultaneously on our testbed server CPU—to stress the backends under concurrent load. λ-NIC outperforms both container and bare-metal backends (Figure 7), servicing requests 27x to 736x faster than the two backends for the web server and key-value client lambdas, and 5x to 15x faster for the image-transformer lambda.

6.3.2 Performance Under Resource Contention. In a real setting, a serverless backend will typically run multiple lambdas at the same time. These workloads will contend for their fair share of resources (*i.e.*, CPU and memory), leading to added delays due to, for example, context switching of lambdas and movement of data to and from CPU and memory.

	λ-NIC	Bare Metal	
		56 Threads	1 Thread
Throughput (Req/s)	58,000	950	520

Table 2: Average throughput when running three distinct web server lambdas concurrently. (Standard deviation is negligible across runs.)

Effects of context switching. In the previous experiment, we measured the performance of each backend when running a single lambda in isolation. Now, we evaluate a set up with three distinct web-server lambdas running on a single backend at once. We generate requests for each of these workloads in a round-robin fashion, causing the processor to context switch between lambdas when servicing each incoming request. Figure 8 shows the context-switching overhead on latency for λ-NIC as well as the bare-metal backend (with 1 and 56 threads) when executing the warm (pre-loaded) web-server lambdas. With multiple lambdas running concurrently, the bare-metal backend suffers even higher latency (178x to 330x) compared to λ-NIC. Moreover, λ-NIC completes requests 55x to 100x faster than the bare-metal backend, with both single and 56 threads (Table 2)—a difference of at least 9% compared to running a single lambda in isolation (§6.3.1), whereas λ-NIC shows no significant change. The performance of containers was a lot worse than even the bare-metal backend (not shown). These experiments demonstrate that, unlike container and bare-metal backends (with server CPUs), λ-NIC is not susceptible to context switching and performs better under resource contention by virtue of a vast array of on-chip NPU cores and the lack of operating system and container software.

6.4 Other Metrics

Resource utilization. We also compare the memory and CPU usage at the host and the SmartNIC when running a single data-intensive image-transformer instance in isolation. Table 3 shows the additional resources utilized by each backend when servicing 56 concurrent requests. Containers have the largest memory footprint and consume an order of magnitude more host memory and CPU cycles than the bare-metal backend. On the other hand, as expected, λ-NIC's impact on the host memory and CPU is negligible, and it consumes roughly the same amount of NIC memory for the image-transformer workload as the bare-metal backend on the host.

Startup times. Next, we measure the startup time; the total time a backend takes to download the image-transformer binary and its dependencies and start serving requests. To compare startup times we use: (1) Lambda binary size (*i.e.*, SmartNIC's compiled firmware, Python library packaged

	λ-NIC	Bare Metal	Container
Host CPU (Avg. %)	+0.1	+9.2	+13.7
Host Memory (MiB)	0	+62.5	+219.5
NIC Memory (MiB)	+63.2	0	0

Table 3: Additional resources utilized by each serverless backend for the image-transformer workload.

	λ-NIC	Bare Metal	Container
Workload Size (MiB)	11.0	17.0	153.0
Startup Time (s)	19.8	5.0	31.7

Table 4: Factors affecting startup times.

using setuptools and Wheel [34], and the docker container). (2) Boot-up time of the lambda, from launching the system to responding to a user request. λ-NIC's image-transformer binary is 13x smaller in size than a container image, and is comparable to a bare-metal binary (Table 4). In addition, it takes the image-transformer lambda 38% less time on λ-NIC to service the first request compared to a container. On the bare-metal backend, the image-transformer binary starts up in under 5 seconds (4x faster than λ-NIC); however, in our evaluation, we do not consider any framework overheads (like Isolate [42]), which will likely lead to higher startup times. In summary, while slow start is a known issue in serverless compute, λ-NIC keeps the additional delay over bare-metal backends 2x less than the container overhead.

Optimizer effectiveness. We now report the results of our compiler optimizations. The number of instructions in the naïve implementation—consisting of two key-value clients, a web server, and an image transformer lambda—is gradually reduced by applying the following target-specific optimizations (§5.1). First, we perform *lambda coalescing* for the two distinct key-value clients. We coalesce these lambdas, as they contain equivalent logic to generate a new packet to query memcached, which we can combine and reuse. We further coalesce the web server and image-transformer lambdas, having a pattern of response that does not query external services. Hence, we combine their reply logic. Next, we apply *match reduction*. The naïve implementation adds a separate table for managing routes for each lambda. We combine these tables into one, and use individual parameter values (defined as P4 metadata) for route management. Finally, we do *memory stratification* to place variables into appropriate memories based on their sizes. For example, the image variable within the image-transformer lambda is mapped to IMEM, whereas the web server results are mapped to CTM inside the island. These optimizations bring the total number of instructions of the final binary down to 8,050 (a reduction

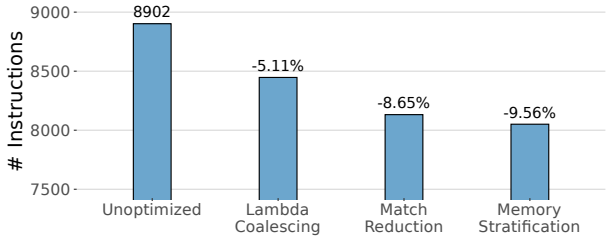

Figure 9: Effectiveness of λ-NIC's target-specific optimizations in reducing the code size.

of 9.56% from the naïve implementation); hence, improving latency by 6.3 μs (on average) [52] or letting additional lambdas to fit within the program-size constraints of the Netronome SmartNIC.

7 DISCUSSION

Choice of hardware. λ-NIC is not just limited to NPU-based SmartNICs. In fact, the λ-NIC's abstract machine model can run on other SmartNICs (with varying benefits) having more general-purpose processors: either FPGA- or SoC-based SmartNICs, or some form of ASIC with ARM cores [26]. These alternatives can further extend the processing capabilities of λ-NIC, providing support for more features (such as floating point operations, deeper instruction store, and dynamic memory allocation) to run more complicated workloads.

Hot swapping workloads. For each new lambda, λ-NIC needs to recompile and swap the firmware with the one currently running on the SmartNIC. Present versions of Netronome SmartNICs do not support hot swapping or hitless updates [27], resulting in downtimes each time a new firmware is loaded. Hitless updates are not technically challenging as devices like FPGAs [28] and programmable switch fabrics [44] already support it (using partial reconfiguration and versioning techniques). We believe this limitation will go away in the future versions of SmartNICs as well, allowing λ-NIC to load new lambdas with causing downtimes.

Accelerating other forms of workloads. In this work, we primarily focused on offloading interactive serverless workloads to SmartNICs. However, λ-NIC can accelerate other parts of the serverless framework as well. For example, the gateway is a proxy that routes users' requests to λ-NIC's worker nodes (Figure 2); its performance is therefore crucial to the end-to-end behavior of the system. λ-NIC can provide strict bounds on tail latency and throughput, by running the gateway directly on a SmartNIC. Certain types of data stores (like key-value stores [2]) can also benefit from λ-NIC. Their restricted compute pattern [66] lends itself nicely to run on λ-NIC's Match+Lambda machine model. Likewise, existing large and long-running workloads (like code compilation and

video processing) have shown to perform better when broken down into small serverless functions [57]; thus, making λ-NIC a suitable target for these workloads. We believe that these trends will inspire more workloads to migrate to λ-NIC in the future, with serverless frameworks automatically determining which backend to execute the workload on.

Security and reliable transport. The serverless framework (*i.e.*, gateway, workload manager, and worker nodes) typically run within a trusted domain of a provider or a tenant, such that any malicious attempt to trigger the lambdas will be blocked by the gateway. In addition, each lambda operates within its own memory region on the NIC, restricting them from accessing each others data. λ-NIC enforces this policy using compile-time assertions; in the future, we plan to explore enforcing assertions at runtime for dynamically-allocated memory (once it becomes available in the upcoming SmartNICs). For transport, λ-NIC relies on RDMA for multi-packet messages. However, with recent focus on terminating the entire transport layer on the NIC [38, 81], λ-NIC's serverless functions can instead operate on complete messages rather than individual packets.

8 RELATED WORK

NIC offloading technologies. Offloads for network features, such as L3/L4 checksum computation, large send and receive offload (LSO, LRO) [74], and Receive Side Scaling (RSS) [77] have been around for decades. More recent work, however, is looking into offloading more complex, application-related functions to modern programmable, SmartNICs [56]. For example, Microsoft is using FPGA-based SamrtNICs to offload hypervisor switching tasks [56], which were previously handled by CPU-based software switches (like Open vSwitch (OVS) [86]). HyperLoop [71] provides methods for accelerating replicated storage operations using RDMA on NICs. λ-NIC can assist these works by providing a framework to easily deploy general compute on a cluster of nodes hosting SmartNICs. Both Floem [87] and iPipe [75] provide a framework to enable easier development of NIC-assisted applications. However, these frameworks can offload only a portion of these applications to a NIC as a bump-in-the-wire, and need CPUs to do the remaining processing. In contrast, λ-NIC runs complete workloads on the NIC, mitigating the effects of any CPU-related overheads.

In-network computing. Orthogonal to NIC offloading technologies, there is a recent focus on moving various application tasks inside the network. P4 [43] and RMT [44] have provided the initial building blocks: a data-plane language and an architecture for programmable network devices, which enabled developers to run various applications in-network. For example, SilkRoad [80] and HULA [69] present methods for offloading load balancers, NetCache [66] implements a key-value store, and NetPaxos [53] runs the Paxos [73] consensus algorithm inside switches. Tokusashi et al. [99] further demonstrate that in-network computing not only improves performance but also is more power efficient. λ-NIC, alongside these networking devices, can provide a more programmable environment for accelerated application processing. In fact, SmartNICs have more memory and less-restricted programming model, which can help alleviate the limitations present in these switches.

Improving serverless compute. Serverless compute is a relatively new idea and many of its details are not yet disclosed by the cloud providers. Thus, most of the recent work focuses on reverse engineering existing frameworks to study their internals and to educate the public. For example, OpenLambda [63] provides an open-source serverless compute framework that closely resembles the ones deployed by the cloud providers. Glikson et al.[62] proposed another framework with support for edge deployments. λ-NIC complements these efforts by presenting a high-performance, open-source serverless compute framework for testing and developing lambdas on SmartNICs.

9 CONCLUSION

Server CPUs are not the right architecture for serverless compute. The particular characteristics of the severless workloads (*i.e.*, fine-grain functions with short service times, and memory), as well as the slowdown of Moore's Law and Denard Scaling, demand a radically different architecture for accelerating serverless functions (or lambdas): an architecture that can execute multiple of these lambdas in parallel with minimal contention and context-switching overhead. In this paper, we present λ-NIC, a serverless framework along with an abstraction, Match+Lambda, and a machine model for executing thousands of lambdas on an ASIC-based SmartNIC. These SmartNICs host a vast array of RISC cores with their own instruction store and local memory, and can execute embarrassingly parallel workloads with very low latency. λ-NIC's Match+Lambda abstraction hides the architectural complexities of these SmartNICs and efficiently compiles, optimizes, and deploys serverless functions on these NICs—improving average latency by 880x and throughput by 736x. With new and emerging developments in both SmartNICs and serverless frameworks, we believe offloading lambdas to SmartNICs will become a common practice, inspiring even more complex real-world workloads to run on λ-NIC.

REFERENCES

[1] 2004. Field-programmable gate array. https://en.wikipedia.org/wiki/Field-programmable_gate_array.
[2] 2007. Memcached. https://en.wikipedia.org/wiki/Memcached.
[3] 2009. Virtualization. https://en.wikipedia.org/wiki/Virtualization.

[4] 2014. Docker Swarm. https://github.com/docker/swarm.

[5] 2014. RoCEv2. https://cw.infinibandta.org/document/dl/7781.

[6] 2016. Apache OpenWhisk. https://openwhisk.apache.org/documentation.html.

[7] 2016. *AWS Storage Services Overview.* Technical Report.

[8] 2016. *SPDY: An experimental protocol for a faster web.* Technical Report.

[9] 2017. Resize Images on the Fly with Amazon S3, AWS Lambda, and Amazon API Gateway. https://aws.amazon.com/blogs/compute/resize-images-on-the-fly-with-amazon-s3-aws-lambda-and-amazon-api-gateway/.

[10] 2017. *Serverless Architectures with AWS Lambda.* Technical Report. Amazon Web Services.

[11] 2018. Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx/.

[12] 2018. Amazon DynamoDB Documentation. https://docs.aws.amazon.com/dynamodb/index.html.

[13] 2018. *AWS Lambda Developer Guide.* Technical Report. 137–139 pages.

[14] 2018. AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html.

[15] 2018. *BCM5880X SmartNIC Solution.* Technical Report.

[16] 2018. BlueField SmartNIC Ethernet. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic.

[17] 2018. Calico for Kubernetes. https://docs.projectcalico.org/v2.0/getting-started/kubernetes/.

[18] 2018. A Comparison of Serverless Frameworks for Kubernetes: OpenFaas, OpenWhisk, Fission, Kubeless and more. https://winderresearch.com/a-comparison-of-serverless-frameworks-for-kubernetes-openfaas-openwhisk-fission-kubeless-and-more/.

[19] 2018. Developing a Linux Kernel Module using GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[20] 2018. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://coreos.com/etcd/.

[21] 2018. Google Cloud Functions. https://cloud.google.com/functions/.

[22] 2018. *Intel Xeon Processor Scalable Family.* Technical Report.

[23] 2018. LiquidIO II 10/25GbE Adapter family. https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics/index.jsp.

[24] 2018. Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[25] 2018. Namepsaces - overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html.

[26] 2018. Netronome Agilio FX ASIC+SoC SmartNICs. https://www.netronome.com/products/agilio-fx/.

[27] 2018. *Programming Netronome Agilio SmartNICs.* Technical Report. Netronome Systems Inc.

[28] 2018. *Vivado Design Suite User Guide: Partial Reconfiguration.* Technical Report.

[29] 2018. What you can build with Cloud Functions. https://cloud.google.com/functions/use-cases/.

[30] 2019. Colfax Direct: Netronome NICs. https://colfaxdirect.com/store/pc/showsearchresults.asp?pageStyle=M&resultCnt=10&keyword=Netronome.

[31] 2019. Google Cloud Storage. https://cloud.google.com/storage/.

[32] 2019. NGINX: High Performance Load Balancer, Web Server, and Reverse Proxy. https://www.nginx.com/.

[33] 2019. P4 Language Specifications. https://p4.org/specs/.

[34] 2019. Pythoon Wheels. https://pythonwheels.com/.

[35] 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[36] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[37] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA.

[38] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA.

[39] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. 2015. Comparative Study of Caffe, Neon, Theano, and Torch for Deep Learning. *CoRR* abs/1511.06435 (2015). arXiv:1511.06435 http://arxiv.org/abs/1511.06435

[40] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. 2018. Scalable Methods for 8-bit Training of Neural Networks. In *NeurIPS*.

[41] John Biebelhausen. 2017. The Era of Smart Networks. http://www.mellanox.com/blog/2017/11/the-era-of-smart-networks-innova2/.

[42] Zack Bloom. 2018. Cloud Computing without Containers. https://blog.cloudflare.com/cloud-computing-without-containers/.

[43] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[44] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 99–110. http://doi.acm.org/10.1145/2486001.2486011

[45] Brian Brazil. 2016. Introduction to Prometheus: An Approach to Whitebox Monitoring. https://www.grafanacon.org/2016/presentations/brian-brazil-anintroductiontoprometheus.pdf.

[46] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 143–157. https://doi.org/10.1145/2043556.2043571

[47] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 7, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195647

[48] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of*

Data (SIGMOD '16). ACM, New York, NY, USA, 1087–1098. https://doi.org/10.1145/2882903.2904441

[49] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. 2005. Shangri-La: Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 224–236. https://doi.org/10.1145/1065010.1065038

[50] Xiaolei Chen and Yajun Ha. 2010. The Optimization of Interconnection Networks in FPGAs. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum fÃ1/4r Informatik.

[51] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. 2013. Deep Learning with COTS HPC Systems. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13)*. JMLR.org, III–1337–III–1345. http://dl.acm.org/citation.cfm?id=3042817.3043086

[52] Patrick Crowley, Marc E. Fluczynski, Jean-Loup Baer, and Brian N. Bershad. 2000. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proceedings of the 14th International Conference on Supercomputing (ICS '00)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/335231.335237

[53] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2774993.2774999

[54] Tim Dettmers. 2016. 8-bit Approximations for Parallelism in Deep Learning. In *ICLR*.

[55] Alex Ellis. 2016. OpenFaaS: Serverless Functions Made Simple for Docker and Kubernetes. https://www.openfaas.com/.

[56] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, Berkeley, CA, USA, 51–64. http://dl.acm.org/citation.cfm?id=3307441.3307446

[57] Sadjad Fouladi, Dan Iter, Shuvo Chatterjee, Christos Kozyrakis Matei Zaharia, and Keith Winstein. 2017. A Thunk to Remember: make-j1000 (and other jobs) on functions-as-a-service infrastructure. (2017).

[58] Martin Fowler. 2014. Microservices. https://martinfowler.com/articles/microservices.html.

[59] Martin Fowler. 2016. Serverless Architectures. https://martinfowler.com/articles/serverless.html.

[60] Lal George and Matthias Blume. 2003. Taming the IXP Network Processor. In *ACM PLDI*.

[61] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 323–336. http://dl.acm.org/citation.cfm?id=1972457.1972490

[62] Alex Glikson, Stefan Nastic, and Schahram Dustdar. 2017. Deviceless Edge Computing: Extending Serverless Computing to the Edge of the Network. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR '17)*. ACM, New York, NY, USA, Article 28, 1 pages. https://doi.org/10.1145/3078468.3078497

[63] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[64] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[65] Tomas Hruby, Teodor Crivat, Herbert Bos, and Andrew S. Tanenbaum. 2014. On Sockets and System Calls: Minimizing Context Switches for the Socket API. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*. USENIX Association, Broomfield, CO. https://www.usenix.org/conference/trios14/technical-sessions/presentation/hruby

[66] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[67] Peder Jungck, Ralph Duncan, and Dwight Mulcahy. 2011. *packetC Programming*. Springer.

[68] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazieres, and Christos Kozyrakis. 2019. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 345–360.

[69] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research (SOSR '16)*. ACM, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/2890955.2890968

[70] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. *SIGPLAN Not.* 51, 4 (March 2016), 67–81. https://doi.org/10.1145/2954679.2872367

[71] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 297–312. https://doi.org/10.1145/3230543.3230572

[72] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs First-class Datacenter Citizens. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Berkeley, CA, USA, 863–879. http://dl.acm.org/citation.cfm?id=3358807.3358881

[73] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[74] Konstantin Lepikhov. 2018. Large Send Offload (LSO). https://wiki.geant.org/pages/releaseview.action?pageId=103712345.

[75] 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 (SIGCOMM '19)*. ACM, New York, NY,

USA, 318–333. https://doi.org/10.1145/3341302.3342079

[76] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *USENIX ATC*.

[77] Duncan MacMichael. 2017. Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[78] Paul Menage. 2006. CGroups - Linux Control Groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[79] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[80] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. 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 (SIGCOMM '17)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/3098822.3098824

[81] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 221–235. https://doi.org/10.1145/3230543.3230564

[82] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[83] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

[84] A. K. Parekh and R. G. Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking* 1, 3 (June 1993), 344–357. https://doi.org/10.1109/90.234856

[85] Edwin Peer. 2017. Mapping P4 to SmartNICs. https://p4.org/assets/p4_d2_2017_nfp_architecture.pdf.

[86] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 117–130. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[87] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *USENIX OSDI*.

[88] Rob Pike. 2012. Go at Google: Language Design in the Service of Software Engineering. https://talks.golang.org/2012/splash.article.

[89] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[90] David K. Rensin. 2015. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472. All pages. http://www.oreilly.com/webops-perf/free/kubernetes.csp

[91] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi. 2017. Performance comparison between container-based and VM-based services. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. 185–190. https://doi.org/10.1109/ICIN.2017.7899408

[92] Muhammad Shahbaz and Nick Feamster. 2015. The Case for an Intermediate Representation for Programmable Data Planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/2774993.2775000

[93] Cris Simpson. 2002. *Overview of Amazon Web Services*. Technical Report. T10: Technical Committee on SCSI Storage Interfaces.

[94] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 44–57. https://doi.org/10.1145/2934872.2934899

[95] Ripduman Sohan, Andrew C. Rice, Andrew W. Moore, and Kieran Mansley. 2010. Characterizing 10 Gbps network interface energy consumption. *IEEE Local Computer Network Conference* (2010), 268–271.

[96] Praveen Kumar Sreeram. 2017. *Azure Serverless Computing Cookbook*. Technical Report. Microsoft Corporation.

[97] P. Srisuresh and M. Holdrege. 1999. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 2663. RFC Editor. 1–29 pages. http://www.rfc-editor.org/rfc/rfc2663.txt

[98] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. 2011. OptiML: An Implicitly Parallel Domain-specific Language for Machine Learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning (ICML'11)*. Omnipress, USA, 609–616. http://dl.acm.org/citation.cfm?id=3104482.3104559

[99] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/3302424.3303979

[100] Paul Viola and Michael J. Jones. 2004. Robust Real-Time Face Detection. *Int. J. Comput. Vision* 57, 2 (May 2004), 137–154. https://doi.org/10.1023/B:VISI.0000013087.49260.fb