Speedo: Fast dispatch and orchestration of serverless workflows

Nilanjan Daw
Indian Institute of Technology
Bombay
India
nilanjandaw@cse.iitb.ac.in

Umesh Bellur
Indian Institute of Technology
Bombay
India
umesh@cse.iitb.ac.in

Purushottam Kulkarni
Indian Institute of Technology
Bombay
India
puru@cse.iitb.ac.in

Abstract

Structuring cloud applications as collections of interacting fine-grained microservices makes them scalable and affords the flexibility of hot upgrading parts of the application. The current avatar of serverless computing (FaaS) with its dynamic resource allocation and auto-scaling capabilities make it the deployment model of choice for such applications. FaaS platforms operate with user space dispatchers that receive requests over the network and make a dispatch decision to one of multiple workers (usually a container) distributed in the data center. With the granularity of microservices approaching execution times of a few milliseconds combined with loads approaching tens of thousands of requests a second, having a low dispatch latency of less than one millisecond becomes essential to keep up with line rates. When these microservices are part of a workflow making up an application, the orchestrator that coordinates the sequence in which microservices execute also needs to operate with microsecond latency. Our observations reveal that the most significant component of the dispatch/orchestration latency is the time it takes for the request to traverse into and out of the user space from the network. Motivated by the presence of a multitude of low power cores on today's SmartNICs, one approach to keeping up with these high line rates and the stringent latency expectations is to run both the dispatcher and the orchestrator close to the network on a SmartNIC. Doing so will save valuable cycles spent in transferring requests to and back from the user space. The operating characteristics of short-lived ephemeral state and low CPU burst requirements of FaaS dispatcher/orchestrator make them ideal candidates for offloading from the server to the NIC cores. This also brings other benefit of freeing up the server CPU.

In this paper, we present Speedo— a design for offloading of FaaS dispatch and orchestration services to the SmartNIC from the user space. We implemented Speedo on ASIC based Netronome Agilio SmartNICs and our comprehensive evaluation shows that Speedo brings down the dispatch latency from ~150ms to ~140μs at a load of 10K requests per second.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Networks → Programmable networks**; **In-network processing**.

Keywords: programmable SmartNIC, orchestration, serverless workflows

ACM Reference Format:
Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2021. Speedo: Fast dispatch and orchestration of serverless workflows. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472883.3486982

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '21, November 1–4, 2021, Seattle, WA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486982

1 Introduction

Serverless computing in the form of Function as a Service (FaaS), with its model of elastic auto-provisioning of resources is now the go-to deployment model for Cloud applications (Amazon Lambda [29], Google Cloud Functions [45] and Azure Functions [32]). FaaS platforms ([4], [20], [17]) provision and manage workers (virtual machines, containers, processes, isolates etc.) and *dispatch* invocation triggers to these workers. Since today's applications are typically a workflow of (fine-grained) functions - each function executes on a worker and on completion, hands control back to an *orchestrator* that, in conjunction with the dispatcher, decides which functions and to which workers the next dispatch must happen, to move the workflow forward. FaaS provider actions thus include registration of triggers and function binaries (application deployment by the customer), dynamic and elastic provisioning of resources for workers, (horizontal) auto-scaling workers, monitoring the arrival of triggers and dispatching them to appropriate workers and orchestrating application workflows [9, 35, 56, 63]. Application workflows can be long-running such as Video Analytics [3, 64], Big Data Analytics [5, 37] and Machine Learning

[8, 58] or short millisecond scale middleware services operating between frontends and data stores [22, 35, 41]. Enterprises such as Twitter, Lyft, Uber, Netflix, and Spotify have all deployed applications using the FaaS model that mainly comprise of workflows of fine-grained (tens of) millisecond scale functions [11, 12, 21, 23, 26, 31, 50, 52].

Prior work on FaaS environments has focused primarily on what we term *post-dispatch optimizations* - minimizing the overheads in setting up workers (aka the cold start problem) [2, 17, 18, 33, 51] so as to reduce the end-to-end latency of handling triggers. However, for fine-grained workflows with end to end execution times of a few hundred milliseconds that are invoked at web-scale (thousands of triggers each second), the inefficiencies of the dispatcher and orchestrator come into sharp focus. Workloads such as that of Twitter, which receives about 6000 tweets per second [61] resulting in over 70K function calls per second [39] each with millisecond-scale latency requirements, have run into this wall where the dispatcher/orchestrator latency severely impacts the workflow execution time, and these components fail to scale to the line rate of incoming requests. We term these set of optimizations that reduce the dispatcher and orchestrator overheads as *pre-dispatch optimizations* and is the focus of this paper.

Existing FaaS platforms are architected in one of two ways: (i) A single, centralized request dispatcher and orchestrator with workers distributed in the network. All components execute in user space on the servers. This design allows for workers to be scaled horizontally - however, it is prone to high latency between the dispatcher/ orchestrator and the workers, as well as issues with scaling the dispatcher. In experiments we performed on popular platforms [4, 17, 20], we observed dispatcher latency of a few milliseconds at low load to hundreds of milliseconds at high load. Further, for sequential workflows of 5 (empty) functions, the mean dispatch and orchestration latency varies from hundreds of milliseconds to a few seconds! (ii) The dispatcher, the orchestrator, and workers, are tightly coupled either as all components in a single execution sandbox [2], or via in-lined orchestration of workers embedded in the function execution logic [33]. Again, all components execute in the user space of servers. This architecture trades off application complexity(tight integration of each application with its orchestration needs) for performance. These platforms also perform other optimizations ranging from custom RPC protocols running over OS pipes, shared memories for inter-function data communication, worker concurrency management [33] to employing a hierarchical message bus architecture to reduce dispatch latencies [2].

Ideally, we want the best of both worlds: (i) the freedom to scale workers horizontally and (ii) low latency (in the order of milliseconds) of communication between the dispatcher/orchestrator and the workers. The key to achieving these seemingly incompatible twin goals lies in the following observations: (i) the performance of user space dispatchers is intrinsically linked to server CPU architectures. Even though server grade CPU cores are primed to perform complex computations at extreme speed they are limited in how many requests they can process in parallel (by the number of cores - typically 32 to 64). Since dispatchers cannot scale horizontally, the limits of vertical scalability of the server sets bounds on how far they can scale. However the lightweight nature of their computations suggests they will benefit from a multitude of light weight cores such as those found on SmartNICs. (ii) The time spent in traversing the network stack form the largest component of dispatcher latency, accounting for up to 90% of it. Device pass-through optimizations such as DPDK [14] address this issue, but they need dedicated non-trivial amounts of server CPU resources to work and so will limit the throughput of dispatch. (iii) Most data center servers today are connected via modern NICs that not only offer gigabit speeds but also have a vast array of inexpensive, non-preemptive, low-powered RISC processors that are programmable [15, 30, 49]. These SmartNICs, along with being capable packet processors, can be used to deploy custom compute close to the wire.

With traditional user space request handling designs of FaaS platforms, server CPUs are overloaded with many tasks—network stack processing, load-balancing and dispatch of functions, orchestration of workflows, deployment, and provisioning of resources and workers, and handling request payloads as well as inter-function data transfers in workflows. SmartNICs have proved their utility in taking over activities like TCP offloading, software-defined networking functions, and in-network data caching for key-value stores. [19, 34, 43]. Efforts such as λ-NIC [10] have shown it's also possible to offload function execution in FaaS platforms. We postulate that dispatcher and orchestrator functions of a FaaS platform need to run close to the network and that the fundamental nature of processing involved is ideally suited to the non-preemptive, low power capability of cores provided on ASIC-based SmartNICs. On the memory axis, a dispatcher requires immutable information like worker metadata—addresses, capabilities etc., which can be stored as part of the control plane directives. The orchestrator on the other hand need to maintain state to track progress of a workflow, state that is tied to lifetime of a workflow instance. Since interactive workflows are generally short, the state stored on the orchestrator is minimal and short-lived. Such storage requirements can readily be met by the onboard memory available on today's SmartNICs.

Our solution, Speedo, offloads these tasks to the SmartNIC allowing workflow execution requests to flow directly from the NIC to the worker nodes thereby speeding up the dispatch considerably. Speedo is both scalable and ensures low request handling latency since it avoids network stack processing for dispatch. We prototyped Speedo on Netronome Agilio CX SmartNICs [49] and evaluated its performance

using multiple micro-benchmarks and real-world workloads. Our evaluation shows that Speedo significantly improves dispatcher performance, reducing end-to-end latency by three orders of magnitude at the same time improving throughput by 400x for real-world use cases. Our key contributions are:

- A comprehensive quantitative analysis of user space dispatcher designs with popular open source platforms.
- The design and implementation of Speedo that offloads dispatch and orchestration to the NIC.
- Extensive evaluation of the Speedo implementation to demonstrate its efficacy over other designs.

The rest of the paper is structured as follows. We analyse the bottlenecks of user space dispatchers and motivate the need for NIC-based implementations in §2, and then introduce the Speedo architecture in §3. We discuss Speedo's engineering challenges in §4 and performance evaluation in §5. This is followed by a discussion on related work in §6 and concludes in §8.

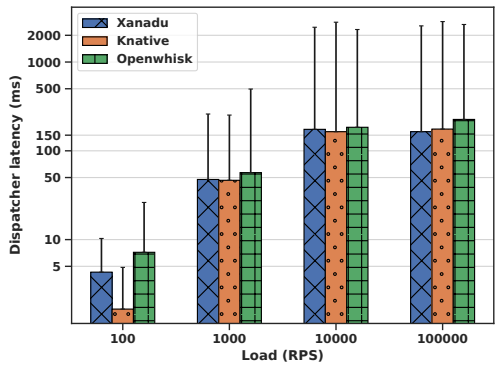
2 Motivation to offload FaaS Platform components on to SmartNICs

In this section, we present a case for offloading FaaS platform components to execute closer to the network on SmartNICs. We do this via comprehensively benchmarking popular open source FaaS platforms with both single function and workflow loads and then further demonstrating that workloads such as those of the dispatcher and orchestrator are particularly well suited to the low power and large number of SmartNIC cores.

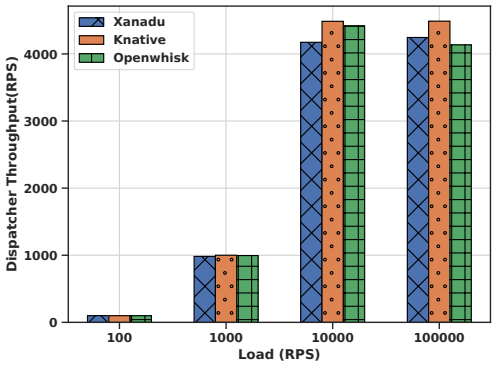
2.1 Dispatcher/Orchestrator Performance in User space

FaaS platforms handle incoming requests for functions (aka triggers) by: (i) pre-processing the request - this includes session termination and authentication, and authorization. (ii) deciding the target workers (nodes) for function execution based on the load on nodes, data availability on the node, the affinity of functions with each other and worker startup costs. We label these activities *dispatch operations* and the component handling this as the `Dispatcher`.

Serverless platforms also support workflows [17, 20, 28, 33] that chain individual functions with control constructs such as linear ordering, broadcast and synchronization barriers. These platforms often feature an explicit `Orchestrator` that coordinates the order in which functions execute based on a deployment time-based control structure (usually a DAG). For such workflows, on completing one function, the orchestrator consults the dependency graph to decide the subsequent set of function(s) to be executed. The functions are then dispatched via the Dispatcher. Note therefore that the orchestrator is temporally stateful since it tracks the current state of the automaton representing the workflow instance it is orchestrating.



(a) Latency



(b) Throughput

Figure 1. Performance of user space dispatchers.

Our investigation into the performance of popular open-source FaaS dispatch systems are divided into two parts— (i) empirical analysis of overheads and resource consumption to dispatch triggers to single functions, and (ii) demonstrating the impact of the cascading nature of the dispatch system (the dispatcher and the orchestrator) overheads in the presence of workflows. We next present each of these in turn.

2.1.1 Single function performance. We benchmarked three open-source FaaS platforms (*Knative, Openwhisk* and *Xanadu*). Each of them implement the dispatcher and the orchestrator as user space processes. We demonstrate the scalability limitations of these platforms and how they fall short of modern (workflow based) application requirements.

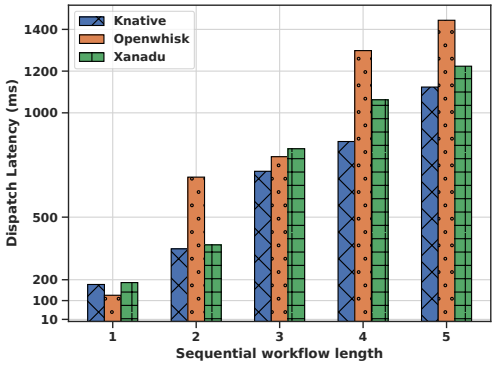
We define *dispatch latency* of a FaaS dispatcher as the aggregate latency incurred to parse the request and dispatch it to a worker. Formally, it is the interval between the trigger arriving at the NIC on which the dispatcher executes to the start of function execution on the chosen worker. The dispatch latency subsumes network processing delay, request queuing delay and network travel time, under warm start conditions (the worker is ready to accept requests and will execute as soon as the request arrives). We deployed `Node.js` based no-op functions and measured the dispatch

Platform	Total dispatch latency	N/W latency	CPU latency	Others
Knative	46.5	42.78	0.09	3.6
Openwhisk	56.7	38.3	0.14	18.25
Xanadu	47.5	44.65	0.98	1.8

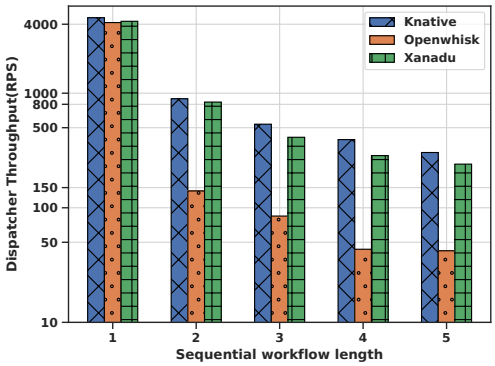
Table 1. Distribution of dispatch latency (in ms) at 1K load

latency and throughput for the three platforms under different load conditions. The experiments were carried out on a x86-64 Linux system with Intel Xeon cores (2.1GHz) and 128GB memory. The FaaS platform executed in a VM and was allocated 48 cores, half of which were allocated to the dispatch system. At low loads, Knative outperforms the other platforms with an average dispatch latency of 1.6ms (Figure 1a).However, increasing the load reveals the issues almost immediately—even at 1000 requests/sec, the dispatch latency jumps to almost 50ms, a value that is far too high to support small functions of the order of tens of milliseconds of execution time. The degradation continues with increase in load—the mean dispatch latency reaches hundreds of milliseconds at 100K requests per second (RPS), the highest mean latency of 225 ms with Openwhisk, completely untenable for a production FaaS platform. These platforms exhibit large tail (99^{th} percentile) latency, more than an order of magnitude larger than their respective mean latency. The 99^{th} percentile latency (Figure 1a) for Knative at 10K RPS and 100K RPS loads, is 2700 milliseconds compared to the mean latency of about 175 milliseconds. Similarly, with Xanadu and Openwhisk the tail latency was more than 2400 ms, at load levels greater than 10K RPS. A deeper look into the subcomponents of this large latency at even moderate loads of 1K RPS suggests that traversing the network stack through the kernel to the user space processes consumes a large percentage of this time (as shown in Table 1). Employing kernel bypass techniques to overcome this merely results in shifting the bottleneck to CPU availability at the server—DPDK is known to be expensive on CPU processing [25].

Corresponding to the latency degradation, we find the throughput also fails to scale with load on any of these platforms albeit for a different reason. Figure 1b reveals that all three platforms successfully scale throughput to match load only until ~4K RPS, beyond which their throughput saturates. At 10K requests, Knative which performs the best among the three, could only serve ~44% of all requests, with Xanadu and Openwhisk managing to serve 42% and 41% requests, respectively. At higher input rates, the throughput drops even further, down to ~4% at 100K RPS. Further investigation revealed that the root cause of this saturation in throughput is the single threaded event loop in Knative and Xanadu. Even as these platforms scale to handle the



(a) Latency



(b) Throughput

Figure 2. Performance of user space orchestrators.

dispatch having a pool of threads for doing the actual dispatch, the event loop that picks triggers off the incoming queue and handing them to worker threads becomes a bottleneck. While Openwhisk is architected differently we find that Openwhisk quickly saturates CPU cores to handle the increased load placing a vertical scalability limit on throughput. The higher CPU consumption of Openwhisk is due to their use of CouchDB for state management.

To summarize, we observe that *the performance of user space dispatcher degrades dramatically with load. Further, since our only option is to vertically scale these dispatchers, we are limited in how far we can go with such designs.*

2.1.2 Workflow Performance. Next, we benchmark workflows of FaaS functions [17, 33] to understand the additional impacts of orchestration. Since the orchestrator is tightly knit with the dispatcher, the dispatch latency includes the overhead involved in orchestrating the functions of a workflow and and dispatching them.

For our investigation, we deployed linear chains of length varying from 1 up to 5 functions. A chain in an ordered sequence of functions, with a function to be executed when it's predecessor finishes execution. We leverage each platform's

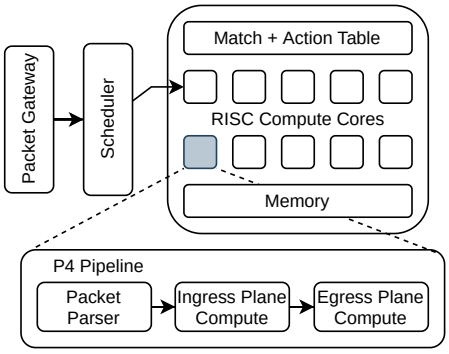


Figure 3. Architecture of an ASIC based SmartNIC.

native workflow orchestration system to deploy the workflows. Figure 2 shows the dispatch saturation throughput and the corresponding dispatch latency. For single function chains, the throughput and latency are identical to those obtained by triggering singular functions, with the orchestrator adding a negligible overhead. The dispatch latency remains at ~170ms across all platforms when the load and throughput is in the range of 4K RPS. However, this quickly changes as we increase the size of the chain. Function chains of length 5 incurred a cumulative dispatch latency of 1.1 seconds to 1.4 second and a dramatic decrease in the throughput—Knative, Xanadu and Openwhisk only managing a maximum of 303 RPS, 240 RPS and 42 RPS respectively. This is because each workflow trigger results in multiple internal requests, with a linear sequence of n functions, triggering at least $2n + 1$ network stack traversals which quickly overwhelms the dispatch system capacity. In addition, the orchestrator takes up additional time and CPU cycles in deciding the next function(s) to be dispatched (adding about 300 μs to 400 μs per invocation) and worsens the situation when compared with single function dispatch.

2.2 Why use Multicore programmable SmartNICs

We now turn our attention to how we can overcome the issues we have outlined using a resource that is now commonly deployed on data center servers—i.e., a SmartNIC. SmartNICs typically have a cluster of discrete compute cores, that can be programmed for general compute tasks over and above the packet processing capability they possess [15, 30, 42, 54]. SmartNIC designs feature FPGAs [42, 54], or are ASIC-based [49] or are SoC based [15, 30], and provide different trade offs of performance and ease of programmability and functionality. In this work, we use the ASIC based Netronome Agilio SmartNICs [49]. ASIC based SmartNICs (Figure 3) are built using a custom hardware design, containing specialized hardware for packet processing tasks like parsing, load balancing, cryptographic computations. To improve programmability of such NICs, vendors also embed clusters of programmable multi-threaded non-preemptive RISC processors, generally called micro-engine clusters. All processor threads in a micro-engine cluster run the same logic, supported by a hierarchical set of memory banks. The Netronome Agilio also supports the P4 [6, 7] programming language, to program the network data plane via vendor provided SDKs, while the control planes are mutated via user space programs. Based on the outcome of the data plane processing, packets received by the NIC can either be sent to the server via a DMA channel, or can be sent out to the network through the outbound NIC channel. The ASIC SmartNICs have extreme parallel computing capabilities (running up to 300+ parallel threads) and low packet processing latencies.

2.2.1 Offloading the dispatch system to the NIC.

It is clear that the work done by a dispatcher is tightly coupled with packet processing—processing packet payloads, creating packets for replication, handling reliability and coordinating ordered delivery of unicast and multicast packets. Further, the dispatcher and orchestrator operate a state machine of event-action sequences that are based on small set of primitives—packet type based dispatch logic, coordinating function invocation based on progress in workflow, generating appropriate response and request packets etc.

Our hypothesis is that given the programmiblity and massive parallel packet processing capabilities, SmartNICs are ideally suited to overcome the shortcomings of slow and expensive server CPU based user space dispatcher designs. The custom programming can be exploited to build semantics of the dispatcher and the orchestrator onto the SmartNIC, and packet processing close to the network can be leveraged for significantly lower latency for dispatcher actions [10, 43]. Our solution, Speedo, is built on this hypothesis and aims at achieving significant scalability improvements compared to CPU based user based dispatchers.

3 Speedo architecture

The design of Speedo is guided by the principle of minimizing CPU-based processing of FaaS dispatchers and performing as many of its actions as possible on the SmartNIC to achieve extremely low dispatch latencies and high throughput.

Speedo breaks up the processing of the FaaS dispatch system in to two main categories — request handling and resource monitoring and provisioning. Request handling consist of parsing incoming requests, deciding which exact function to execute via the orchestrator and dispatching the function for execution to a worker node. These actions form the bulk of the processing load of a FaaS dispatch system and are ideal candidates for being offloaded for processing on the SmartNICs. Speedo offloads the request handling pipeline and provides a fast path of dispatching incoming requests.

The other important category of work of FaaS dispatching systems is that of resource monitoring and provisioning. Execution sandboxes for functions have to be provisioned,

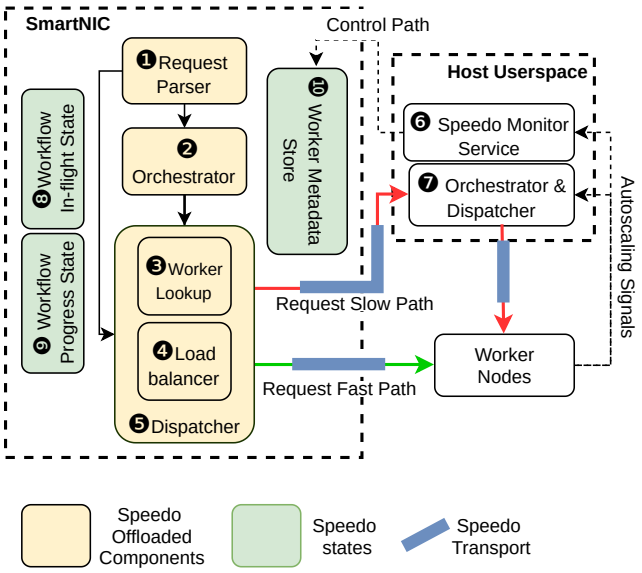


Figure 4. Speedo architecture and components.

e.g., on cold starts when a function is triggered for the first time. This includes reserving resources, setting up execution environments and loading relevant binaries for standard libraries and/or operating systems etc. Further, monitoring of all provisioned entities is important for dynamic and proactive vertical and horizontal scaling decisions. Since these actions of monitoring, provisioning and scaling operate at much lower rates as compared to requests for service and since the semantics and data collection vector themselves need to flexible, Speedo operates these components in user space on server CPUs.

Figure 4 shows the architecture and main components of Speedo. The three main components of the dispatching system— request parser, the dispatcher and the orchestrator execute on the SmartNIC, and the Speedo monitoring service and resource provisioning services execute in the user space. Requests serviced by the SmartNIC are on the fast path, while those requiring intervention of the user space components are on the slow path.

3.1 The Dispatcher

The Dispatcher (Figure 4 ⑤) is on the critical path of every request and is the single most important component of Speedo. The dispatcher receives requests either directly from the Request parser module in case of an single function invocation or coordinates with the Orchestrator for workflows to schedule individual requests pertaining to a workflow. The dispatcher is composed of two modules, the Worker Lookup (Figure 4 ③) and the Load balancer (Figure 4 ④).

(i) **The worker lookup** module is responsible for identifying workers capable of executing a particular request. It uses the Worker Metadata Store (Figure 4 ⑩) as a repository for backend worker metadata, which includes data like the workers' network address, their identifiers, isolation engines and function runtime. On request arrival, the lookup module uses the request signature to identify worker subsets. A lookup failure at this stage indicates unavailability of workers (cold start). The SmartNIC short-circuits the request to the user space dispatcher, which is then responsible for resource deployment and further processing of the request — the *Slow Path* for handling requests (the red connector line in Figure 4).

The slow path suffers from user space dispatch latencies, however, our design choice is motivated by the observation that under cold start conditions, post-dispatch latencies dominate end-to-end latencies. Secondly, the decision to handle cold starts at the user space significantly reduces the NIC dispatcher complexity and metadata storage requirements, and eases the process of changes to the monitoring and provisioning decisions.

(ii) **The load balancer** is the second part of the Speedo dispatcher and is responsible for selecting one of the many worker nodes available for execution of a function. Currently, Speedo uses a randomized allocation of functions to workers to minimize the state storage requirements in the SmartNIC and to speed up dispatch decisions. We leave exploration of more complex worker selection algorithms as future work. Once a worker is selected, the request is routed to a worker completing the request handling *Fast Path* (green connector line in Figure 4).

3.2 The Orchestrator

The Orchestrator (Figure 4 ⑤) is the second unit of the Speedo dispatch system and is responsible for coordinating developer specified schemas (DAG-based workflows) based on accompanying trigger requests. With Speedo we assume that each request along with the function trigger and data payload also embeds the workflow

The process of disseminating the workflow schema with the request obviates the need for workflow related metadata storage on the NIC. The orchestrator however needs to store ephemeral workflow state onboard the NIC to track its progress. The Speedo orchestrator uses two important NIC-resident data structures, the Workflow Progress State (Figure 4 ⑧) and the Workflow In-flight state (Figure 4 ⑨) for this purpose.

3.3 The Speedo Transport

In Speedo , the primary communication channel between the clients, Speedo dispatch system, the worker nodes and external services is via Remote Procedural Calls (RPCs). These are essentially single packet trigger requests, with the first few bytes consisting of Speedo invocation metadata, including the orchestration schema (if any) while the rest is request payload for a maximum of 8KB packets. The use of single

packet RPCs is driven by the rationale that serverless payloads are small. [55] shows that 80% of all requests have payloads of less than 12KB. Payload sizes further reduce to less 1KB for 97% of microservice requests [40].

Triggers for Speedo workflows arrive via HTTP while the internal communication between the dispatch system and the worker nodes is over UDP. The small payload sizes coupled with the high reliability of current data center networks warrants the usage of low overhead connectionless protocols(UDP). This eases protocol handling and resource requirements at the NIC.

4 Engineering Speedo

We implemented Speedo on P4-enabled Netronome Agilio SmartNICs [49] and BMv2 [13] based software switches. While our choice of the NIC is driven by the availability of a large number of on board parallel processors (48 processor cores, each with 8 threads contexts) and 2 GB on board memory, our BMv2 implementation ensures that Speedo can be deployed on any P4-16 compliant SmartNIC. Speedo is integrated with Xanadu [17], offloading most of Xanadu's components to the NIC and leaving the cold start handling and resource management to execute on the user space. The choice of the Agilio SmartNIC and the Xanadu platform are based on ease of implementation and sufficient features to showcase the offloading benefits. The Speedo monitoring service to track auto-scaling requirements and health of worker nodes is implemented in Python and executes along with the user space Xanadu dispatcher. Additionally, Speedo run time libraries, which can be integrated with clients or gateways for conversion from Xanadu-compatible schemas to custom Speedo header formats and request batching are available in C++ and Node.js to ensure easy portability for existing applications.

4.1 Speedo Dispatcher

The Netronome Agilio SmartNIC is capable of executing custom P4 code on the RISC processors. Each processor has up to 8 non-preemptible run-to-completion thread contexts, making them suitable for short burst characteristic of packet processing loads. Speedo's dispatch mechanism, developed in P4-16, is designed such that each core on the SmartNIC executes the Speedo dispatcher. Once a request is scheduled to a context, which was received either from an external client or the Speedo Orchestrator, the request is parsed by the P4 parser engine. The dispatcher uses the onboard Match Action Table (Worker Metadata Store) to find a target workers' subset based on the function signature. The dispatcher then consults the loadbalancer to select a worker. Based on the location of the target, the dispatcher either pushes the request to the server via the dedicated DMA channel between the NIC and the server or sends it out to a remote worker via the egress gateway.

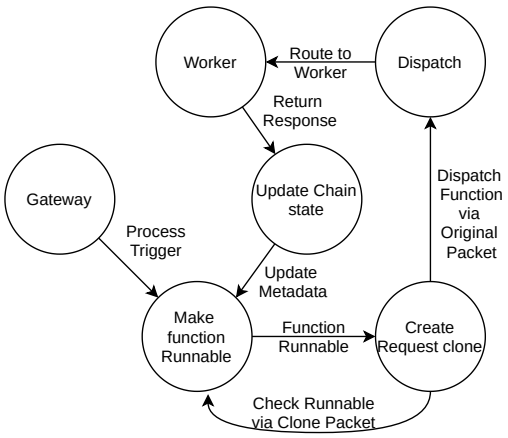


Figure 5. Sequence of operations to handle the multicast primitive.

4.2 Speedo Orchestrator

To realize general purpose workflows, the Speedo orchestrator supports four types of invocation patterns [17], (i) one-to-one (`1:1`) unicast, (ii) one-to-many (`1:m`) dynamic multicast, (iii) many-to-one (`m:1`) dependency barrier and (iv) many to many (`m:n`) junction.

4.2.1 Handling Multicast in Workflows. Multicast and Junction invocations require dynamic multicast support—the multicast group membership is dynamically determined based on the state of the workflow for the instance being orchestrated. Supporting dynamic multicast on a P4 based NIC has two issues. Firstly, P4's support for multicast is limited to pre-defined multicast groups, whose membership is set via the control plane. This fails to provide the degree of flexibility required to tune multicast group for each invocation. Secondly, as of P4-16, there is no support for iterators in the P4 language. The lack of iterator support prevents generation of multiple multicast targets requests. Speedo solves both of these issues using a **repeated cloning** technique where the Speedo orchestrator generates a Egress-to-Ingress (E2I) packet clone for the first runnable function it detects. The E2I clone then produces a second clone to service the next runnable function in the workflow, while the original packet is dispatched. This process is repeated until all the runnable functions have been serviced (refer to state machine representation of repeated cloning in Figure 5). Speedo effectively uses the P4 cloning pipeline repeatedly as a substitute for iterators producing multiple packet clones each with a distinct target to simulate a dynamic multicast.

4.2.2 Handling Barrier Functions. An important requirement for the general purpose orchestrator is *barrier* functions. Barrier functions are hold-and-wait functions that execute only after its dependent functions have completed. Speedo handles this by storing the execution state of a workflow on NIC memory on a Speedo data structure termed

Workflow Progress State (WPS). The WPS is a bitmap of function states, with a separate instance of the bitmap for each workflow invocation. For each response received for function requests triggered as part of a workflow orchestration, the Speedo orchestrator updates the WPS, whose value is then used to mark functions runnable. Functions whose dependency conditions are not met based on the WPS value are not triggered, thus effectively providing support for barrier functions.

4.3 Speedo event loop

Our investigation into the bottlenecks in user space dispatchers has shown the main event loop to be one of the main reasons for the low throughput and large latency. While the requests are serviced asynchronously in parallel via a service thread pool, the event loop is responsible for reading the requests and handing them over to a thread in the thread pool. While the thread pool can scale, our hypothesis is that the event loop fails to scale. The Netronome Agilio SmartNIC provides a work-conserving ASIC based packet scheduler, designed to uniformly distribute packets over the available processors. In Speedo the event loop is realized by the hardware scheduler (shown in Figure 3). This provides a shared queue abstraction similar to user space dispatchers with very high throughput, allowing the Speedo dispatcher to scale with minimal queuing delays. Once a request arrives at the NIC gateway and joins the request queue, the scheduler picks a free dispatch thread and assigns the request to be processed. The request is processed when the thread is executed on one of the RISC cores.

4.4 Speedo Monitoring Service

The Speedo monitoring service (SMS) is a user space control plane component residing along with the user space dispatch system. Its primary task is to update the Worker Metadata Store on the NIC, which the Speedo dispatch system uses for dispatch and routing purposes. The SMS listens on the message exchanges between the user space dispatcher and worker nodes during cold start handling and autoscaling (up and down) to update the worker related metadata on the NIC using Apache Thrift based RPC calls.

4.5 Speedo runtime library

Speedo uses a custom message format and a custom transport protocol to exchange messages between entities of the Speedo system. Incoming requests/triggers are converted via a Speedo run time library into a Speedo interpretable format. The library functions to do request encoding can either be a part of the client or be a part of a gateway on the server side. Further, each worker node use this library to intercept and decode incoming requests from the Speedo dispatcher.

4.5.1 Auto scaling Support. The Speedo dispatcher routes requests directly to the worker nodes, bypassing the user space dispatcher. This prevents user space autoscalers that rely on metrics such as concurrency or open request count to scale workers. Towards enabling the user space dispatcher to proactively react to changes in load and current capacity, the Speedo run time executing on worker nodes publishes usage metrics via Apache Kafka. We modified the Xanadu autoscalar to subscribe and receive the published usage metrics on the Kafka channel, and use them for scaling decisions. Subsequently, scaling worker nodes up and down results in change of state at the SmartNIC and impacts future NIC dispatching decisions.

5 Evaluation

We now present evaluations of Speedo to demonstrate its efficacy in reducing dispatch and orchestration latency and improving throughput of serverless workflows. Our base case is the Xanadu FaaS platform [17] to compare Speedo against. Our evaluation seeks to answer the following questions:

(i) What is the improvement in latency and throughput with Speedo's offloaded design for single function dispatch? (§5.1)
(ii) How do external factors such as conventional network processing influence the performance of the NIC based dispatcher? (§5.3)
(iii) How does the dispatcher's performance scale with resource availability? (§5.3)
(iv) How does offloading the dispatcher to the NIC affect performance of applications executing on the server? (§5.3 and §5.2)
(v) How does the Speedo dispatcher perform with real-world applications? (§5.4)

Setup: We setup Speedo on an x86-64 Linux server (kernel v4.15.0) with a 16 Intel Xeon core (2.6GHz) processors and 64GB memory. The system had a Netronome Agilio CX 2x10GbE SmartNIC with 48 RISC cores (8 threads each) running at 633MHz and 2GB onboard memory. We ran the client, user space dispatcher and the function workers on this machine in separate VMs. The base case, Xanadu[1], was run on a machine with 64 Intel Xeon Cores (2.1GHz) and 128GB memory also running Linux (Kernel v4.15.0). The dispatch system and the worker nodes were setup to run on the same machine in separate VMs such that all components had sufficient resources to prevent any resource bottlenecks. We use `wrk2` [62] to generate load from a separate VM on the same machine to minimize network related latency.

Workloads: To comprehensively cover the workload options, we divide the deployed benchmarks into: (i) Synthetic micro benchmarks and (ii) Real world macro benchmark applications. We further subdivide the former into single function micro benchmarks and workflows that are either linear or have a combination of multicast and junctions. Since

[1]Web link: https://git.cse.iitb.ac.in/synerg/xanadu

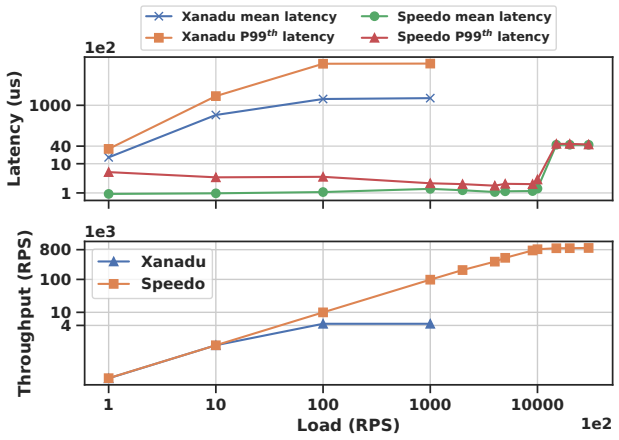


Figure 6. Performance comparison of Speedo with Xanadu.

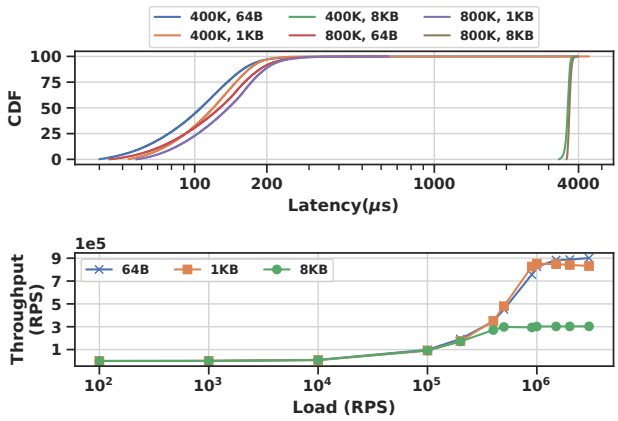


Figure 7. Effect of payload size on Speedo performance. Legend encoded as (load, payload size).

the actual processing done by the functions are of no real consequence, we restrict the functions to an empty Node.js function, which when invoked immediately returns with an empty string. For the real-world macro benchmarks, we consider two workflows inspired from the SocialNetwork and the MediaService workflow from Deathstarbench [21]. We further detail the macrobenchmarks in §5.4.

Parameters of interest: Our key parameter of interest is load on the system, to study and compare scalability of Speedo. The load is expressed in terms of function/triggers requests per second that reach the system. The other parameter that captures load is the payload size of each incoming requests—the payload holds workflow description and the input data for the triggered function. Since NICs are also tasked with performing regular operations for user space application, the other parameters of interest is the level in interference with Speedo. The computing resource availability on the NIC for Speedo and the non-FaaS platform packets processed by the NIC are used a measure of interference.

Metrics of goodness: Our chief interest is in observing the dispatch latency and throughput with and without orchestration. Whenever otherwise mentioned latency means the dispatch latency. The dispatch latency in case of workflows also subsumes the orchestration latency along with the aggregate dispatch latency over all the functions in the control flow path. In case of workflows we also define an end-to-end (E2E) latency which is the total round trip time taken in receiving a response after a request is sent out. Throughput in each case means the end-to-end throughput of the system.

5.1 Speedo Dispatch (of standalone functions)

To measure the effectiveness of the Speedo dispatcher, we invoke a micro benchmark consisting of requests to a simplest workflow— a standalone function. We gradually increase load and we measure the mean and tail latency (99^{th} percentile latency) and throughput of the platform under test.

Latency: Figure 6 shows the performance of the respective platforms at different load levels. We observe that, as expected, Xanadu's user space dispatcher shows a large latency of about ~4 ms even at low loads. In comparison Speedo shows an almost 50x reduction in mean latency, keeping it down to 92.4 μs. We also observe, as the load is increased, the latency for the user space dispatcher increases sub-linearly until its throughput saturates. Speedo's dispatcher latency on the other hand, remains almost constant, showing only about 16% increase in mean latency as we increased load from 100 RPS to 10K RPS compared to an almost 4000% increment in case of Xanadu within the same range. Speedo only shows a spike in latency as it approaches its saturation point at ~800K RPS. Even at this extreme end, its mean latency remains at 4.6 ms, just 7% above the lowest latency observed with the user space dispatcher. Further, the tail latency of the user space dispatcher at saturation was 25 times higher the mean latency (2.3 s as compared to mean latency), whereas the corresponding values for Speedo were much lower 350 μs, three orders of magnitude less than Xanadu.

Throughput: For the throughput, we observe Speedo exhibiting tremendous performance gains over Xanadu. While Xanadu's dispatcher could only scale to 4.5K RPS, Speedo scales to 800K RPS without any significant increase in dispatch latency, a **177x** improvement. The saturation throughput for Speedo (around 900K RPS) is due to the deployed NIC reaching its operational limits.

Effect of payload size. We evaluate Speedo's performance under various payload sizes. Increasing the payload size increases the per-packet computation load, so that Speedo has to spend more time parsing and routing the packets,

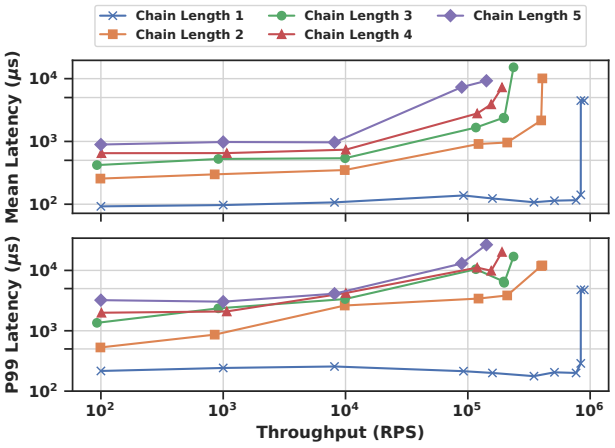


Figure 8. Performance of Speedo with workflows.

stressing its request queues. Data from public service providers [55] and previous work [40] establish that a significant number (~60%) of serverless requests carry less than 8KB payloads, with the payload requirements going down further for micro-services to 1KB for up to 97% of all requests. We vary the payloads based on these guidelines and choose three payload sizes of 64B, 1KB and 8KB. Figure 7 shows the latency CDF for each of the payloads at 400K RPS and 800K RPS load. We observe a 11% increase in mean latency as payload size was increased from 64B to 1KB at 400K RPS, and a 16% increase in mean latency at 800K RPS for the same payload range with negligible decrease in throughput (8% at saturation). Mean latency however sharply increases at payload sizes of 8KB. A 65% drop in saturation throughput accompanies this compared to 64B payloads. This is because the NIC has to perform far more computation per request to route them, increasing both the queuing and computational delay. However, even at 800K RPS load the dispatch time is 16.2% lower than fastest dispatch time observed on Xanadu.

5.2 Evaluating Speedo performance with workflows

Workflows or function chains cause the impacts of dispatch latency to cascade and aggregate across the chain of functions triggered by a single user request. Orchestration logic adds to this cascading effect at each stage of the workflow. We now present the performance of the Speedo dispatcher in the presence of workflows and compare it with its user space counterpart for multiple workflow configurations.

Linear Sequence: We deploy function chains with no-op functions, increasing its length from 1 to 5 and measure the aggregate orchestration and dispatch latency in these tests. Figure 8 illustrates the results. Performance of single function chains is close to that of single function dispatch, with a 2% increase in mean latency compared to that of single function dispatch and a 6% drop in throughput at very high loads. Thus the orchestrator itself adds little to the dispatch overhead.

Load (RPS)	Fan out 4	Fan out 8	Fan out 12	E2E
100K	11.9	19.7	40.2	1442
200K	15.7	22.1	48.3	2130
400K	315	1496	1614	2489

Table 2. Performance of repeated cloning. (Latency in μs).

Note that even though the overall throughput drops linearly with increase in chain length, Speedo outperforms user space dispatchers by a wide margin (for chains of length 5 we observe 590x improvement compared to Xanadu).

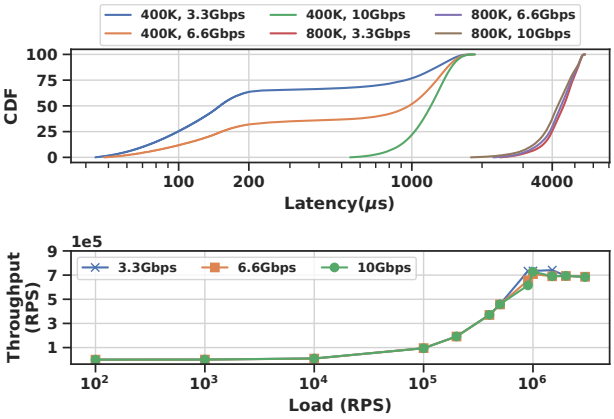
An increase in chain length does result in larger latency, however since Speedo ensures individual dispatch latency is small, the aggregate dispatch latency for the workflow remains tolerable. For example, at a chain length of 5, Xanadu had a mean dispatch latency ~1200 ms at the saturation throughput of 240 RPS. At similar throughput Speedo improves mean dispatch latency by more than three orders of magnitude keeping the average latency down to ~900 μs and tail latency of 3.2 ms

Impact of Multicast: We now turn our attention to the orchestrator's repeated cloning engine and evaluate its performance for workflows containing multicast controls with different fan outs. We deploy three simple workflows where a trigger causes a multicast (fan outs of 4, 8 and 12) after which a barrier function follows. The workflow ends with a final function after the barrier thus creating a three step workflow. Table 2 shows the end-to-end (E2E) latency and the average branch creation time at different load conditions. At loads of 100K to 200K RPS, individual request cloning latency was around 3 μs to 4 μs. Since the cloning pipeline creates request clones sequentially, this leads to an aggregate cloning time of about 11 μs for fan outs of 4 at 100K RPS, going to 40 μs at fan out of 12. We however observed an increase in cloning time as the NIC reached saturation, with single request clones taking about 80 μs at 400K RPS for fan out of 4, which further increases with fan out. This is due to increased queuing delays, as the cloned requests need to spend more time waiting for processor access.

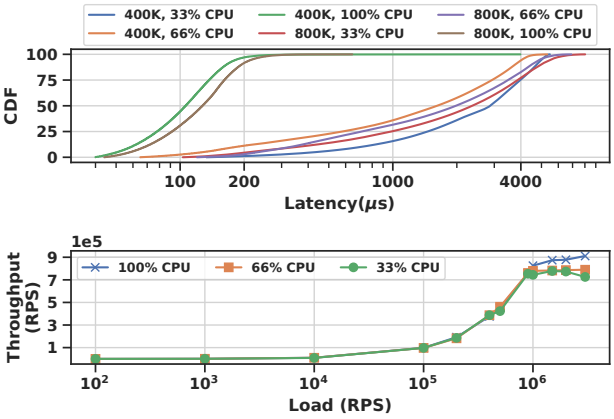
In terms of end to end throughput and latency, we did not observe any appreciable degradation in performance, with both the parameters staying within 10% of that of linear chains of similar length.

5.3 Impact of interference on performance

Next, we look at how well Speedo performs when it has to contend with interference from external sources. We consider two separate scenarios to evaluate Speedo's performance, (i) high network packet rates the NIC has to deal with in parallel to the dispatch system (network contention) and (ii) other offloaded applications running on the NIC cores,



(a) Effect of network contention.



(b) Effect of CPU contention.

Figure 9. Performance of Speedo under different operating conditions. Legend encoded as (load, influence factor).

thereby restricting the cores available to Speedo (compute contention).

Network contention. We evaluate Speedo's performance under network contention, where a parallel stream of network packets are flowing through the SmartNIC, consumed at the server. We use `iperf3` as a network load generator and send a parallel data stream for network contention. Figure 9a shows the latency CDF and throughput of Speedo as we increase the network contentions from 3.3Gbps to 10Gbps. We observe a 5x to 7x increase in mean latency at loads of 400K RPS (64B payload). The saturation throughput drops by about 22% to 700K RPS compared to no contention. This is because the dispatch requests now have to contend for compute cores alongside network packets, leading to queuing delays and requests drops.

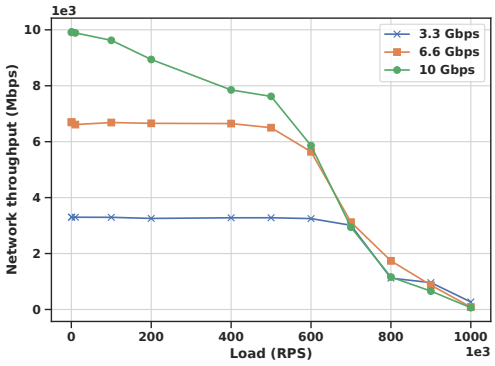


Figure 10. Effect of Speedo dispatch on a parallel data stream.

Figure 10 shows the throughput of the network stream as observed at the `iperf3` server when contending with the dispatcher. A minimal drop in throughput is seen for the 3.3Gbps and 6.6Gbps streams until 500K load on the dispatcher —however, increasing dispatcher load to 900K and above causes a dramatic decrease in throughput.

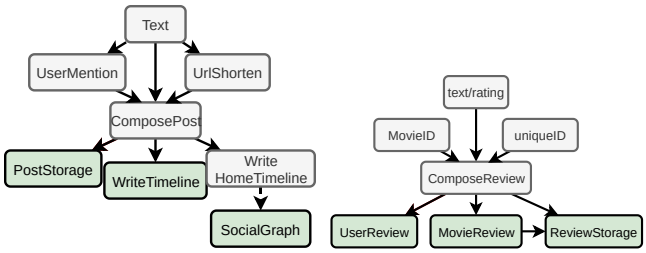
Compute contention. Finally, we look into the effects of CPU contention on the NIC and how the dispatch design scales. Figure 9b shows the effect on the latency and throughput of Speedo as we restrict the CPU cores available to the dispatcher to one-third of the total strength and gradually increase it. At 33% NIC cores allocated to the dispatcher, we observe a 20% drop in throughput compared to the base case of all cores running the dispatcher. We observe an elongated tail for latency, with the mean latency increasing to ~2.4 ms and 99^{th} percentile latency to 6.6ms with dispatcher loads at 800K RPS.

We make two observations on Speedo's behavior under compute contention, (1) the drop in throughput is only about 20%, operating close to its saturation point, even when CPU availability is reduced by 66% (albeit with longer dispatch latency) (2) the dispatcher scales as more processors are made available, reducing mean latency by ~26x.

5.4 Case Studies

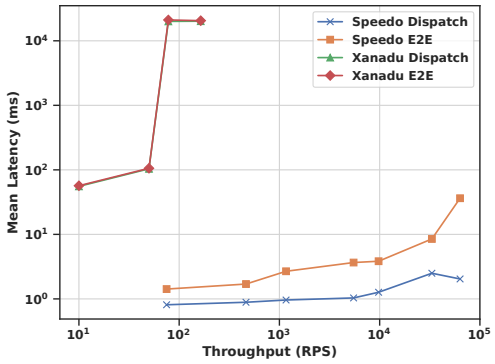
Having shown the effectiveness of Speedo on synthetic micro-benchmarks, we now present two real applications as case studies to benchmark Speedo against.

The applications are architected as workflows of microservices and include: (a) *SocialNetwork* and (b) *MediaService* inspired from DeathStarBench [21]. We evaluate the ComposePost pipeline from the SocialNetwork workflow and the MovieReview pipeline from the MediaService workflow. We adapted the workflows to use Speedo's transport protocol and replaced the Memcached and MongoDB backends with in-memory datastores. We also modified the workflow's to use Speedo's orchestrator to coordinate the workflows. In

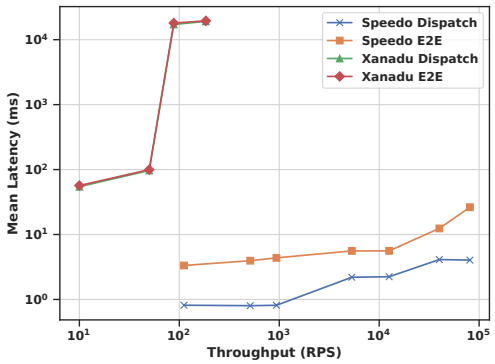


(a) SocialNetwork ComposePost. **(b)** MediaService MovieReview.

Figure 11. Macrobenchmark workloads. Functions shaded in green require storage services.



(a) Social Network



(b) Movie Review

Figure 12. Performance of macro-benchmarks with Speedo.

each case, we report the dispatch and end-to-end latency for both Xanadu as well as Speedo .

SocialNetwork: The ComposePost pipeline consists of a range of interacting services, including services to upload a text post, update the user's timeline and social graphs (shown in Figure 11a). Figure 12a shows the results of our evaluation. While Xanadu saturates ~160 RPS, Speedo easily dispatches over 60K RPS, without its dispatch latency degrading by much. At the same time, at comparable loads, Speedo is 150x faster compared to Xanadu. At higher loads, we observe the E2E latency goes up considerably, while the dispatch latency remains comparatively stable, indicating that the server that supports the workers is saturated. We, however, could not test Speedo in a multi-host system due to a lack of hardware.

MediaService: The MediaService pipeline, shown in Figure 11b, allows users to add reviews and movie ratings that are then stored durably. The result of our experiment is shown in Figure 12b. We observe similar profiles to that of SocialNetwork. Speedo vastly outperforms Xanadu, dispatching up to 80K RPS. The higher throughput, compared to SocialNetwork, is due to lower per-request computation on the server. Dispatch latency at all times remained below 5ms, compared to Xanadu which showed dispatch latency of over 190ms.

6 Related Work

In this section, we present prior work and highlight our contributions in light of existing research. We discuss research into two areas— that of improvements made to FaaS platforms and work done on leveraging SmartNIC capabilities.

6.1 Serverless Optimizations

Issues with making serverless computing performant and scaleable have generated a host of approaches in the recent past [1, 2, 17, 19, 51, 55, 58]. The bulk of this research has gone into solving post-dispatch bottlenecks such as the cold start problem [1, 18, 48, 57]. However, for interactive services that operate at millisecond scales, problems such as cold starts are not relevant. Such services require extremely low latency communication [47, 60]. While some work has gone into optimizing the communication layer [2, 33, 38, 55, 59], most often, these have been to optimize the data transport layer [38, 46, 55, 59]. Microservices, however, typically feature extremely small payloads [40] ranging from a few bytes to a few KBs. Thus having an efficient data transport layer does not necessarily help such services. What is really needed are optimizations in the dispatch system. SAND [2] uses a hierarchical memory bus to route triggers amongst functions. Faasm [58] showcases a similar trigger transport mechanism. However, SAND requires all members of a workflow to share the same container while Faasm runs workflows within the same process. Speedo does not impose any such restrictions on the deployment of workflows. Nightcore [33] uses communication optimizations such as OS pipes between functions of workflows scheduled on the same server to reduce latency. Further, most of these platforms lack a dedicated orchestrator and require functions to coordinate amongst themselves— requiring functions to be cognizant of workflow protocols and increasing development complexity.

6.2 SmartNIC offloading

SmartNICs have become ubiquitous in the data center. Offloading complex workloads to the NIC is an active area of research; solutions such as Microsoft's ClickNP [19, 42]

has been used to accelerate datacenter tasks. Researchers are exploring ways to bring computation like real-time analytics, network function virtualisations, key-value stores closer to the network [24, 44]. Solutions such as ipipe [43], λ-NIC [10], Floem [53] have tried to offload general-purpose workloads to the NIC. However, compute cores on the NIC are far simpler than server-grade CPUs and often comes with expressibility limits. Offloading workloads without taking these into consideration can thus be counterproductive. Speedo offloads the serverless dispatch system, which has a compute profile closely resembling that of network packet processing, making it ideal for offloading. Dagger [40] accelerates RPCs on FPGA based NICs. However, they do not handle workflow coordination at the NIC. There have also been some efforts to provide services entirely at the NIC [16, 36] along with in-network caches such as [34] which we believe, along with Speedo, will provide an ideal substrate for workflow based microservices.

7 Future Work

7.1 Handling large payloads

Speedo is optimized for fast dispatch of workflows of millisecond scale microservices that have small payloads (in the order of tens to hundreds of bytes). Large payload functions usually rely on indirect payload access through external storage such as Amazon S3 [27] (and separately exploit data caching based optimizations for quick access). The dispatch mechanism is concerned with setup of function execution environments. For small payload functions it provides data transport and in other cases can be used to encapsulate metadata to access external data stores. Data caching based optimizations and management are currently outside of Speedo's realm of considerations, and can be interesting complimentary solutions.

7.2 Handling application-layer network protocols

Speedo relies on UDP as its transport layer for communication between the dispatcher and the worker nodes. This reduces operational overhead without loss of QoS on datacenter networks. However, while the scope of this work is to optimize the serverless dispatch system and not the general network stack protocols we believe that our solution can co-exist and leverage other efforts which focus on partially or completely offloading the network stack onto SmartNICs. Extending Speedo to work with TCP and application layer protocols such as HTTP can generalize its applicability and usage.

7.3 Dynamic offloading of Speedo dispatch system

Section 5.3 of our paper talks about the effects of interference from external sources. We show that under heavy network and compute contentions, offloading the dispatch system can lead to performance degradation for both the userspace applications as well as the dispatch system. An adaptive system that can dynamically switch the dispatch functionality between the NIC and host is in the works.

7.4 Extending Speedo's feature set

The Speedo offload mechanism shows considerable performance gains compared to its userspace peers. However, the performance gains comes with certain limitations. The dispatch system is developed in P4, that is a not a Turing-complete language. This coupled with the limited on-board resources places some restrictions on what can be offloaded to the NIC. To overcome these challenges we are also investigating the performance of offloading the dispatch system on off-path ARM based SmartNICs such as the Mellanox Bluefield DPU.

8 Conclusion

Millisecond scale interactive microservices bring unique opportunities and challenges with their high throughput and low latency constraints. We delved into the challenge of dispatching requests at sub-milliseconds and presented Speedo, a NIC offloaded, application agnostic dispatch system. Speedo provides extreme low latency request dispatch by exploiting SmartNIC hardware architectures and by simplifying network stack processing by eliminating multiple network stack traversals. Comparison with Xanadu, shows Speedo provide orders of magnitude improvements both in terms of latency and throughput.

Acknowledgments

We thank the anonymous reviewers, our shepherd Karla Saur, and our colleague Rinku Shah for their helpful comments. This work was supported in part by a VMware research grant (17DON003).

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. 419–434.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 Usenix Annual Technical Conference (USENIX ATC '18)*. 923–935.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. 263–274.

[4] The Knative Authors. 2020. Knative. https://knative.dev/. Accessed: 2020-04-22.

[5] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the Faas Track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. 41–54.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[7] Mihai Budiu. 2017. Programming networks with P4. https://blogs.vmware.com/research/2017/04/07/programming-networks-p4/. Accessed: 2021-04-18.

[8] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. 13–24.

[9] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The rise of serverless computing. *Commun. ACM* 62, 12 (2019), 44–54.

[10] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2019. λ-NIC: Interactive Serverless Compute on Programmable SmartNICs. *arXiv preprint arXiv:1909.11958* (2019).

[11] Jeremy Cloud. 2021. Decomposing Twitter: Adventures in Service-Oriented Architecture. https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture. Accessed: 2021-04-18.

[12] Adrian Cockcroft. 2021. Evolution of Microservices - Craft Conference. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference. Accessed: 2021-04-18.

[13] P4 Consortium. 2018. Behavioral Model (BMv2). https://github.com/p4lang/behavioral-model. Accessed: 2021-04-18.

[14] Intel Corporation. 2021. Data Plane Development Kit. https://software.intel.com/content/www/us/en/develop/topics/networking/dpdk.html. Accessed: 2021-04-18.

[15] NVIDIA Corporation. 2021. NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-in/networking/products/data-processing-unit/. Accessed: 2021-04-18.

[16] 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)*. 1–7.

[17] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference (Middleware '20)*. 356–370.

[18] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 467–481.

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*. 51–66.

[20] The Apache Software Foundation. 2020. Apache OpenWhisk. https://openwhisk.apache.org/. Accessed: 2020-04-12.

[21] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 3–18.

[22] Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Barnard Metzler. 2020. Serverless end game: Disaggregation enabling transparency. *arXiv preprint arXiv:2006.01251* (2020).

[23] Adam Gluck. 2021. Introducing Domain-Oriented Microservice Architecture. https://eng.uber.com/microservice-architecture/. Accessed: 2021-04-18.

[24] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. 2020. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20)*. 681–693.

[25] Dave Hunt. 2018. Recent Power Management Enhancements in DPDK. https://www.dpdk.org/wp-content/uploads/sites/35/2018/10/pm-01-DPDK_Summit18_PowerManagement.pdf. Accessed: 2021-04-18.

[26] Scott M. Fulton III. 2021. What Led Amazon to its Own Microservices Architecture. https://thenewstack.io/led-amazon-microservices-architecture/. Accessed: 2021-04-18.

[27] Amazon Web Services Inc. 2020. Amazon S3. https://aws.amazon.com/s3/. Accessed: 2020-04-11.

[28] Amazon Web Services Inc. 2020. Amazon States Language. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html. Accessed: 2020-04-12.

[29] Amazon Web Services Inc. 2020. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2020-04-11.

[30] Broadcom Inc. 2021. Broadcom Stingray Smartnic. https://docs.broadcom.com/docs/5880X-UG30X. Accessed: 2021-04-18.

[31] Lyft Inc. 2021. Lyft Engineering. https://eng.lyft.com/tagged/microservices. Accessed: 2021-04-18.

[32] Microsoft Inc. 2020. Azure Functions. https://azure.microsoft.com/en-in/services/functions/. Accessed: 2020-04-11.

[33] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. 152–166.

[34] 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)*. 121–136.

[35] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[36] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the 2nd ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '16)*. 1–12.

[37] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding ephemeral storage for serverless analytics. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*. 789–794.

[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. 427–444.

[39] Raffi Krikorian. 2021. Twitter by Numbers. https://www.slideshare.net/raffikrikorian/twitter-by-the-numbers/20-REST_API_XMLJSON_API_over. Accessed: 2021-04-18.

[40] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. 36–51.

[41] Collin Lee and John Ousterhout. 2019. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. 149–154.

[42] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. 1–14.

[43] 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)*. 318–333.

[44] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*. 363–378.

[45] Google LLC. 2020. Google Cloud Functions. https://cloud.google.com/functions. Accessed: 2020-04-11.

[46] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*. USENIX Association. https://www.usenix.org/conference/atc21/presentation/mahgoub

[47] Amirhossein Mirhosseini, Brendan L West, Geoffrey W Blake, and Thomas F Wenisch. 2020. Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*. IEEE, 207–219.

[48] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '19)*.

[49] Netronome. 2021. About Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/. Accessed: 2021-04-18.

[50] Cao Duc Nguyen. 2021. A Design Analysis of Cloud-based Microservices Architecture at Netflix. https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f. Accessed: 2021-04-18.

[51] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 Usenix Annual Technical Conference (USENIX ATC '18)*. 57–70.

[52] Gonzalo P. 2021. Microservices Architecture at Spotify. https://medium.com/codebase/microservices-architecture-at-spotify-beac905e9622. Accessed: 2021-04-18.

[53] 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 *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. 663–679.

[54] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA '14)*. IEEE, 13–24.

[55] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. *arXiv preprint arXiv:2104.13869* (2021).

[56] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*. 1063–1075.

[57] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. 205–218.

[58] Simon Shillaker and Peter Pietzuch. 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. 419–433.

[59] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592* (2020).

[60] Akshitha Sriraman and Thomas F Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. 177–194.

[61] Internet Live Stats. 2021. Twitter Usage Statistics. https://www.internetlivestats.com/twitter-statistics/. Accessed: 2021-04-18.

[62] Gil Tene. 2012. wrk2. https://github.com/giltene/wrk2. Accessed: 2021-04-18.

[63] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*. 133–146.

[64] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. 2019. Video processing with serverless computing: a measurement study. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '19)*. 61–66.