SPRIGHT: Extracting the Server from Serverless Computing! High-Performance eBPF-based Event-driven, Shared-Memory Processing

Paper #490, 12 pages body, 16 pages total

ABSTRACT

Serverless computing promises an efficient, low-cost compute capability in cloud environments. However, existing solutions, epitomized by open-source platforms such as Knative, include heavyweight components that undermine this goal of serverless computing. Additionally, such serverless platforms lack dataplane optimizations to achieve efficient, high-performance function chains that facilitate the popular microservices development paradigm. Their use of unnecessarily complex and duplicate capabilities for building function chains severely degrades performance. 'Cold-start' latency is another deterrent.

We describe SPRIGHT, a lightweight, high-performance, responsive serverless framework. SPRIGHT exploits shared memory processing and dramatically improves the scalability of the dataplane by avoiding unnecessary protocol processing and serialization-deserialization overheads. SPRIGHT extensively leverages event-driven processing with the extended Berkeley Packet Filter (eBPF). We creatively use eBPF's socket message mechanism to support shared memory processing, with overheads being strictly load-proportional. Compared to constantly-running, polling-based DPDK, SPRIGHT achieves the same dataplane performance with 10× less CPU usage under realistic workloads. Additionally, eBPF benefits SPRIGHT, by replacing heavyweight serverless components, allowing us to keep functions 'warm' with negligible penalty.

Our preliminary experimental results show that SPRIGHT achieves *an order of magnitude* improvement in throughput and latency compared to Knative, while substantially reducing CPU usage, and obviates the need for 'cold-start'.

1 INTRODUCTION

Serverless computing has grown in popularity because users have to only develop their applications while depending on a cloud service provider to be responsible for managing the underlying operating system and hardware infrastructure. The typical costs borne by the user is only for processing incoming requests and this event-driven consumption of resources is attractive for the user of cloud applications, especially when their demand is intermittent. It does however place the burden on the cloud service provider to provide adequate resources on demand and ensure quality of service requirements are met.

In many cases, existing serverless frameworks are profligate in their resource consumption as they provide the needed functionality by loosely coupling serverless functions and middleware components that each run as a separate container and/or pod[1]. This can be extremely resource intensive, especially when deployed in a limited capacity environment, *e.g.,* edge cloud [53]. There are still a number of shortcomings to be overcome for building a high-performance, resource-efficient and responsive serverless cloud. Some contributors of this overhead are the following.

Use of heavyweight serverless components. In a serverless environment, each function pod has a dedicated sidecar proxy, distinct from its application container. Sidecar proxies help build a inter-function service mesh layer with extensive functionality support, *e.g.,* metrics collection, buffering, which facilitates serverless networking and orchestration. However, the existing sidecar proxy is heavyweight since it is continuously running and incurs excessive overheads, *e.g.,* interrupt and context switching. Moreover, since most serverless frameworks primarily focus on HTTP/REST API [21, 29, 42], additional protocol adaptation is required for specialized use cases, *e.g.,* IoT (Internet-of-Things) with MQTT [15, 69], CoAP [34]. Current design runs protocol adaptation as an individual component, which can result in substantial resource consumption. Having such a heavyweight design may overload serverless environments, especially in resource-limited edge cloud or when handling infrequent workloads (*e.g.,* IoT). Instead, going a step further and invoking code for execution on a completely event-driven basis, without using an individual component, can result in substantial resource savings.

Poor dataplane performance for function chaining. Modern cloud-native architectures decompose the monolithic application into multiple loosely-coupled, chained functions with the help of platform-independent communication techniques, *e.g.,* HTTP/REST API, for the sake of flexibility.

[1] "one-container-per-Pod" is the most common model used by Kubernetes for running a function instance.

But this involves context switching, serialization and deserialization, and data copying overheads. This also relies heavily on the kernel protocol stack to handle the routing and forwarding of network packets to and between function pods, all of which impacts performance. Although function chaining brings flexibility and resiliency to build complex serverless applications, the decoupled nature of building these chains also requires additional components (*e.g.,* message broker such as Apache Kafka [30], to coordinate communication between functions, and a load balancer like Istio [10]). The resulting complex data pipelines adds more network communications for the function chain. All of this contributes to poor dataplane performance (lower throughput, higher latency), potentially compromising service SLAs.

In this paper, we design SPRIGHT, a high-performance, event-driven, and responsive serverless cloud framework that utilizes the shared-memory processing to achieve high-performance communication within a serverless function chain. We base the design of SPRIGHT on Knative [11], a popular open-source serverless framework. Evaluation results are presented for SPRIGHT and compared with Knative under various realistic serverless workloads in a cloud environment. The results with our event-driven shared memory processing, including an event-driven proxy (we call it EPROXY) significantly reduces the high resource utilization in the Knative design, resulting in much lower latency. SPRIGHT overcomes the challenges for existing serverless computing with the following innovations:

(1) We design the SPRIGHT gateway, a per-node component, to facilitate shared memory processing within a serverless function chain. The SPRIGHT gateway provides protocol stack processing by the Linux kernel and distributes the payload to different function chains. It also collects fine-grained Layer 7 metrics to support intelligent autoscaling.

(2) We implement zero-copy message delivery within a serverless function chain by using event-based shared memory communication. Zero-copy message delivery avoids the unnecessarily duplicated in-kernel packet processing between the serverless functions, achieves high-speed, highly scalable packet forwarding within a serverless function chain. Event-based shared memory communication helps reducing CPU usage and alleviate penalties when keeping the function chain as warm.

(3) We design an event-driven proxy (*i.e.,* EPROXY) using the eBPF (extended Berkeley Packet Filter [62]). This effectively replaces the heavyweight sidecar proxy. We support the functions of metrics collection *etc*, with much lower CPU consumption.

(4) We utilize packet redirect function provided by eBPF to further improve packet forwarding performance outside the serverless function chain. Compared to the kernel networking stack, the eBPF-based dataplane exhibits dramatically less latency and lower CPU.

(5) We optimize protocol adaptation by running it as an event-driven component attached to the SPRIGHT gateway, again avoiding unnecessary network communications between components. As the protocol adapter is on the critical packet datapath (execution being triggered by every packet), this optimization can significantly reduce latency.

This work does not raise any ethical issues.

2 BACKGROUND AND CHALLENGES

There are a variety of implementations for function chaining, since there isn't a standard for a general solution architecture for serverless applications. The data pipeline patterns for function chaining of different open-source serverless platforms is slightly different, depending on the messaging model applied, *e.g.,* a publish/subscribe model typically uses a message broker as the intermediate component for coordinating invocations within the function chain, while the request/response model typically employs a front-end proxy to perform invocations within the function chain. We examine the design of a number of proprietary and open-source serverless platforms [6, 12, 23, 24, 52] and developed a common abstract model of the typical data pipeline pattern they use, as shown in Fig. 1.

The data pipeline for function chains use a message routing as follows: ① Clients send messages (requests) to a message broker/front-end proxy through the ingress gateway of the cluster. ② The messages are then queued in the message broker/front-end proxy and registered as an event. ③ Depending on the routing configuration of the function chain, the message is sent to an active pod of the head (first) function in the chain. ④ The function pod is invoked to process the incoming request. When the request message is

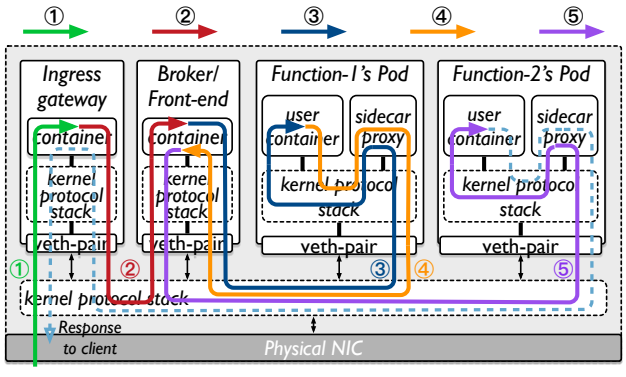

Figure 1: Networking processing involved in a typical serverless function chain setup.

Table 1: Per request Knative overhead auditing of data pipelines for a '1 broker/front-end + 2 functions' chain. Note: we exclude client side overheads

Data Pipeline No.	*External*			*Within chain*				**Total**
	①	②	*total*	③	④	⑤	*total*	
# of copies	1	2	3	4	4	4	12	15
# of context switches	1	2	3	4	4	4	12	15
# of interrupts	3	4	7	6	6	6	18	25
# of protocol processing tasks	1	2	3	3	3	3	9	12
# of serialization	1	1	2	2	2	2	6	8
# of deserialization	0	1	1	2	2	2	6	7

processed by the first function, a response is returned and queued in the message broker/front-end proxy, to be registered as a new event for the next function in the chain. ⑤ The message broker/front-end proxy sends this new event to an active pod for the next function in the chain.

Unfortunately, this data pipeline poses several challenges that are common across the different serverless platforms. The core dataplane components, including the ingress gateway, message broker/front-end proxy, sidecar proxy, *etc*, are usually implemented as individual, constantly-running, loosely coupled components. In addition, for internal calls within the function chain, each involves context switching, serialization/deserialization and protocol processing.

We quantify the overheads in the representative open source platform Knative, through a systematic auditing performed with a '1 broker/front-end + 2 functions' chain setup based on the current design depicted in Fig. 1. We assume all evaluated components are deployed on the same node and we exclude the overhead on the external client side. We examine the different overheads incurred in the data pipeline processing of one request (from ① to ⑤), including # of copies, # of context switches, *etc* as listed in Table. 1. Due to implementation-specific differences, *e.g.,* running multiple threads on the same CPU core, there may inevitably be additional context switches. Our audit aims to quantify the minimum value of each type of overhead. Based on these observations, we list the following key takeaways:

Takeaway#1: Excessive data copies, context switches, and interrupts.

With the existing Knative framework, each request results in 15 data copies, 15 context switches and 25 interrupts throughout the entire data pipeline. Surprisingly, most of the overhead (80%) comes from networking within the function chain (from ③ to ⑤). Current approaches for serverless function chaining rely on composition of existing networking components to support asynchronous and reliable message exchange between functions, and traffic within the chain has to go through the message broker/front-end proxy each time over the kernel. This inevitably introduces additional data copies, context switches, and interrupts, thus increasing overhead. Furthermore, as the chain becoming more complex, the number of data copies, context switches, and interrupts increase linearly, resulting in very poor scaling.

Takeaway#2: Excessive, duplicate protocol processing.

Protocol processing is another major source of overhead. As seen in Table. 1, networking *within* the function chain accounts for 75% of the total protocol processing overhead, reflecting the problematic design of current serverless function chains. Protocol processing tasks including checksum calculation in software, and complex iptables processing contribute to latency and results in poor scaling (especially as the number of iptables rules increases) [51].

Takeaway#3: Unnecessary serialization/deserialization.

HTTP/REST APIs require additional serialization and deserialization operations to convert application data to transmit or receive from the network, which works in byte streams. These operations incur significant overhead (lowering throughput) and adds latency [67]. Each step in the data pipeline for the function chain (from ③ to ⑤) introduces 2 serialization and 2 deserialization operations. Current designs further amplify this degradation with an excessive number of protocol stack traversals as shown in Table. 1.

Takeaway#4: Individual, constantly-running heavyweight components.

Serverless platforms equip each function pod with an individual, constantly-running sidecar proxy to handle inbound and outbound traffic. The presence of this sidecar proxy introduces a significant amount of overhead. Just going through step ④, the sidecar proxy introduces 2 data copies (50%), 2 context switches (50%) and 2 interrupts (33%). To understand the impact of this overhead on dataplane performance, we evaluate several sidecar proxies, including the Envoy sidecar from Istio [9], Queue proxy from Knative [49], and the OF-watchdog from OpenFaaS [18]. We use these sidecar proxies to work with NGINX [17] as a representative HTTP server function. We also use this NGINX HTTP server function without sidecar proxies as the baseline to quantify the additional overhead introduced by the sidecar proxy. We disable autoscaling and limit ourselves to a single function instance. We use *wrk* [3] as the workload generator and send variable-size HTTP traffic (2% 10KB requests, 98% 100B requests) directly to the function pod (including sidecar). Both *wrk* and the function pod are running on the same node.

Our experimental results are shown in Fig. 2. Equipping a sidecar proxy results in a 3×–7× reduction in throughput, 3×–7× higher latency, and a significant increase (3×–7×) in CPU cycles per request. Even though the overhead varies, it is common across all the evaluated sidecar proxies. Looking deeper, at the CPU overhead breakdown, 50% of CPU cycles are consumed by the kernel stack for the sidecar proxy. This substantial overhead of sidecar proxies undercuts the benefit

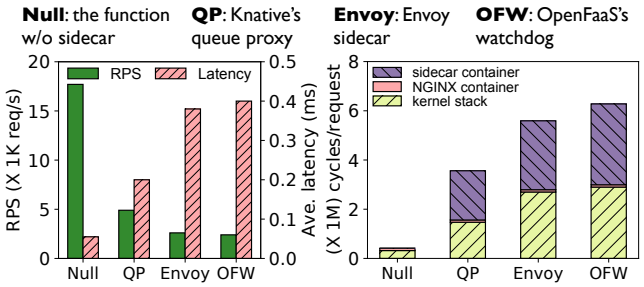

Figure 2: Performance and overhead breakdown of different sidecar proxy implementations.

of serverless computing, and calls for a more lightweight serverless capability to provide the same functionality.

Summary: The goal of serverless computing was to overcome the inefficiencies of 'serverful' computing. However, the excessive overhead in current serverless frameworks shows that the 'server' is still entrenched in serverless computing. Through our auditing, we show that the loosely coupled construction of existing components for serverless computing results in substantial unnecessary processing overhead, possibly discouraging the implementation of microservices as function chains. This poor dataplane design and having individual, constantly-running components in the function chain prompted us to create a more streamlined, responsive serverless framework by considering high-performance shared memory processing and lightweight event-driven based optimizations to help extract the 'server' out of serverless computing.

3 SYSTEM DESIGN OF SPRIGHT

In this section, we start with the overall architecture of SPRIGHT by justifying the design of each component and discussing the benefits it achieves in improving serverless environments. We then talk about each part separately, including the shared memory processing for communication within serverless function chains, the lightweight event-driven proxy (EPROXY), the dataplane acceleration for communication outside the function chain, lightweight protocol adaptation and intelligent autoscaling.

3.1 Overview of SPRIGHT

In this work, we start with open-source Knative as the base platform [11]. Using an innovative combination of event driven processing and shared memory, we support high performance while being very resource-efficient and providing the flexibility to build microservices using serverless function chaining. Importantly, we extensively use eBPF in SPRIGHT for networking and monitoring. eBPF is an in-kernel lightweight virtual machine that can be plugged in/out of the

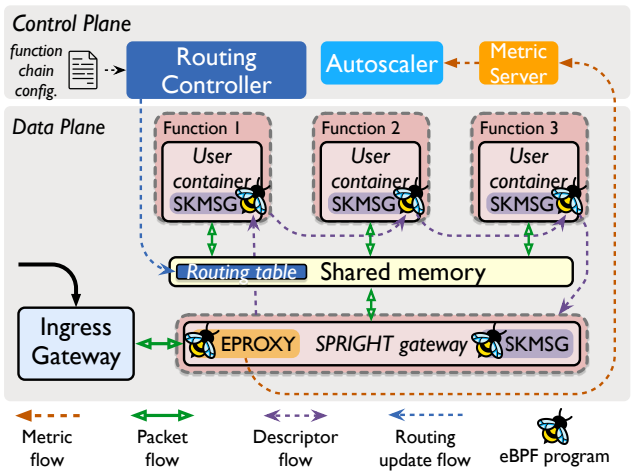

Figure 3: The overall architecture of SPRIGHT

kernel with considerable flexibility, efficiency, and configurability [62]. The execution of eBPF programs is triggered only whenever a new event arrives, thus working naturally with the event-driven serverless environment. By utilizing eBPF, user space components can attach various event-driven programs to kernel hook points (*e.g.*, the network or socket interface). This enables high-speed packet processing [40, 66] and low-overhead metric collection [47, 71]. eBPF achieves its configurability through eBPF maps, a configurable data structure shared between the kernel and user space. With eBPF maps, a more flexible dataplane can be implemented with customized routing. The good features of eBPF help us provide functionality with resource use that is strictly load proportional, a highly desirable toolbox for serverless environments.

The overall architecture of SPRIGHT is shown in Fig. 3. To flexibly manage traffic in and out of the function chain in SPRIGHT and avoid duplicate protocol processing within the chain, we introduce a SPRIGHT gateway. It acts as a reverse proxy for the function chain. Each node is deployed with a SPRIGHT gateway to efficiently manage processing within the local function chain. The SPRIGHT gateway relies on kernel protocol stack for protocol processing and extract the application data (*i.e.*, Layer 7 payload). It intercepts incoming requests to the function chain and copies the payload into a shared memory region. This enables zero-copy processing within the chain, avoiding unnecessary serialization/deserialization, protocol stack processing, and expensive data copies. The SPRIGHT gateway invokes the function chain for requests and processes the results, constructing the HTTP response to external clients.

To eliminate the impact of additional networking components for function chaining, we design Direct Function Routing (DFR). DFR fully exploits the shared memory, and

leverages the configurability provided by eBPF maps. DFR allows dynamic update of routing rules and uses shared memory to pass data directly between functions.

We design a lightweight, event-driven proxy (EPROXY) that uses eBPF to construct the service mesh instead of a continuously-running queue proxy associated with each function instance, as is used by Knative. Thus, we reduce a significant amount of the processing overhead. To accelerate the data path outside the function chain, we utilize XDP/TC hooks in eBPF to forward packets between other serverless dataplane components, *e.g.,* ingress gateway and to/from the chain. An XDP/TC hook processes packets at the early stage of the kernel receive (RX) path before packets enter into the kernel iptables [32, 40], resulting in substantial dataplane performance improvement without dedicated resource consumption, compared to a constantly running queue proxy that depends on the kernel protocol stack.

Event driven processing can help tremendously in interfacing serverless frameworks, that have a HTTP/REST API, with a variety of application-specific protocols (*e.g.,* for IoT with MQTT [15], CoAP [34]). Current designs use a separate protocol adapter for translation between these protocols. However, since SPRIGHT's shared memory processing directly works on payloads independent of the application layer (L7) protocols, the protocol adapter can ideally run as an internal event-driven component that is part of the SPRIGHT gateway. This way, we achieve a much more streamlined protocol adapter design, using resources strictly on demand.

Although these optimizations are built around the Knative-based environment, our concepts and methodology can also be broadly applied to other serverless platforms.

3.2 Optimizing communication within serverless function chains

3.2.1 Shared memory within a function chain. SPRIGHT allocates a shared memory pool with Linux HugePages for each serverless function chain. Using HugePages can reduce the access overhead of in-memory pages, thus improving the performance of serverless functions when accessing data in the shared memory pool. In addition, the shared memory pool within the function chain supports queueing, to helping sustain traffic bursts.

To enable zero-copy data movement between functions, shared memory processing relies on packet descriptors to pass the location of data in the shared memory pool, which is then accessed by the function. One option for implementation is to use DPDK, which uses a poll mode driver to deliver the packet descriptor through its multi-process support [16]. DPDK has been extensively used to build up high performance dataplane for cloud services [70]. While DPDK allows

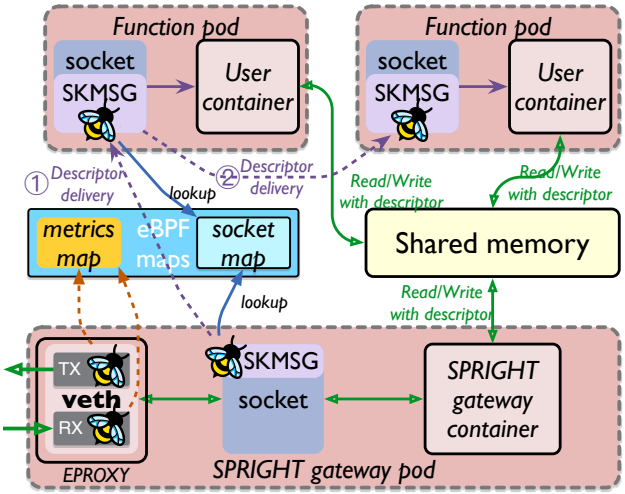

Figure 4: Event-driven design, shared memory mechanism, and EPROXY of SPRIGHT

for fast packet processing and low latency, it continuously consumes significant CPU independent of traffic intensity.

In SPRIGHT, instead of using heavyweight polling-based shared memory processing, we dynamically extend the use of the socket interface at the function pod by attaching an eBPF Socket Message program (SKMSG in Fig. 4) [58]. SKMSG works with eBPF's socket map to enable message redirection between socket interfaces of function pods. The packet descriptor in SKMSG is a small 16-byte message which induces minimal transmission overhead. A packet descriptor contains two fields: the instance ID of next function and a pointer to the data in shared memory. Once the SKMSG receives a packet descriptor, it extracts the instance ID of the next function, which is then used to query the eBPF's socket map to retrieve the target socket interface information (*i.e.,* file descriptor). SPRIGHT's gateway is used to maintain the in-kernel eBPF's socket map (Fig. 4). When a new function pod instance is started-up, the SPRIGHT gateway updates its instance ID and socket interface information into the socket map to support redirection between socket interfaces.

The packet descriptor redirection performed by SKMSG bypasses the kernel protocol stack, incurring minimal latency overhead. SKMSG operates in a purely event-driven manner, avoiding the need to busy-poll packet descriptors and saving CPU resources. Thus communication overhead is entirely load-dependent.

3.2.2 Event-based vs. polling-based shared memory processing. To demystify the most appropriate shared memory processing mechanism in the context of serverless computing, we compare the SPRIGHT's event-based share memory

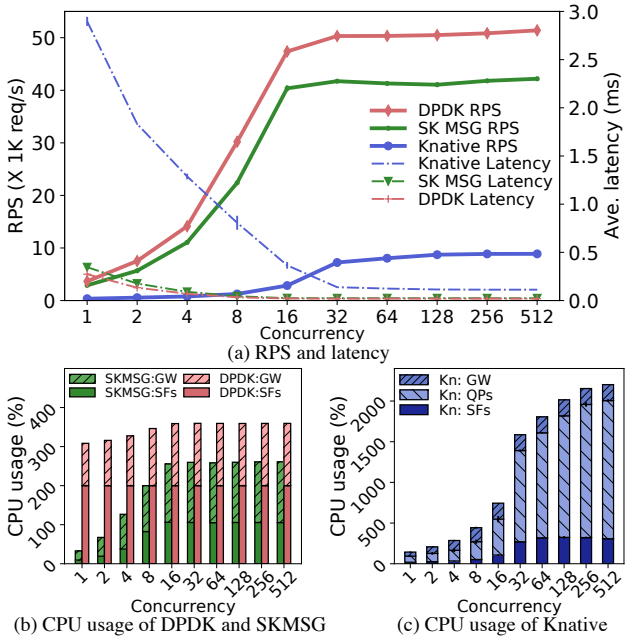

Figure 5: Comparison between polling-based and event-driven shared memory processing with 1 gateway pod and 2 serverless function pods. Kn: Knative; QPs: Queue proxies; SFs: serverless functions; GW: gateway

processing based on SKMSG with polling-based shared memory processing based on DPDK, with a function chain containing 2 function pods. We run a SPRIGHT gateway on the same node and use Apache Benchmark [1] on a second node as the workload generator. We additionally set up a function chain with the base Knative environment and use NGINX as the front-end proxy to coordinate the communication within the chain. Both SPRIGHT gateway and NGINX proxy are configured with two dedicated cores for a fair comparison. Note: We collect the results from 10 repetitions. All results have 99% confidence intervals.

As shown in Fig. 5, with low concurrency, *e.g.*, at 32, SKMSG (0.024ms) shows higher average response delay compared to DPDK (0.02ms), but still with much lower (almost 6×) response delay compared to Knative (0.138ms). In terms of RPS, both DPDK (50.3K) and SKMSG (41.7K) are substantially higher than Knative (7.2K), with a significant 5.7× improvement.

As SKMSG relies on in-kernel eBPF program to deliver packet descriptors, it incurs the overheads for context switching contributing the extra latency. However, SKMSG processing latency is masked when the concurrency increases (≥ 32), with the overlap of the context switching overhead and other processing. Throughput increases rapidly, up to 5× that of Knative. Although, SKMSG has 1.2× lower peak throughput than DPDK, SKMSG being purely event-driven, has a substantially lower CPU usage. Both of those approaches have much lower overhead compared to Knative. With a concurrency is 1, SKMSG consumes 32% CPU, which is 9.6× and 4.5× less than DPDK (308%, or more than 3 CPU cores fully used) and Knative (143%), respectively. When the concurrency increases to 32, SKMSG consumes 259% CPU, which is still less than DPDK (359%). Comparatively, the CPU usage of base Knative increases to a shocking 1585% (more than 15 CPU cores used) at a concurrency of 32. 70% of Knative's CPU is consumed by its queue proxy. Even with increasing concurrency (≥ 32), SKMSG has a consistent and steady savings in CPU compared to the others. Individual, constantly-running components (queue proxy with Knative or DPDK's poll mode using up CPUs) have excessive overhead. More importantly, when there is no traffic, SKMSG consumes negligible CPU resources. We observed that both SKMSG-based SPRIGHT gateway and function pods consume *zero* CPU when there is no traffic, making it possible to keep a function pod 'warm' to overcome the 'cold start' delay. Thus, SKMSG-based shared memory processing is an ideal match for serverless computing, especially for function chains.

3.2.3 Direct Function Routing & Load balancing within a function chain. To optimize the invocations within a function chain we use Direct Function Routing (DFR), which enables the upstream function in the chain to directly invoke/communicate with the downstream function. As shown in Fig. 4, the SPRIGHT gateway only invokes the head function in the chain once (① in Fig. 4). When the first function completes the request message processing (② in Fig. 4), it directly calls the next function without going through the SPRIGHT gateway. The rest of the function invocations in the chain also bypass the SPRIGHT gateway, thus significantly reducing the invocation latency (and overhead) for the function chain.

To manage DFR within the function chain, we introduce a routing controller in SPRIGHT's control plane (Fig. 3). The routing controller configures the routing table based on the user-defined sequence for the function chain. We keep the routing table in shared memory to reduce access latency. To support multiple downstream functions, we use a 'topic' (extracted from the message payload) based publish/subscribe (PUB/SUB) messaging model, and dynamically route requests using the routing table. The message topic and the ID of current function serve as the key to determine the next hop function ID value in the routing table. For load balancing, we select the active pod instance with the maximum residual service capacity and pack its instance ID into the packet descriptor. The invocation is then performed through the SKMSG based on configured instance ID, without going through the SPRIGHT gateway.

3.3 Event-driven proxy (EPROXY)

In Knative, the queue proxy runs as an additional container in a function pod distinct from the user container. It buffers incoming requests before forwarding them to the user container, to handle traffic bursts and maintain throughput. The queue proxy is also responsible for collecting metrics for the pod (*e.g.,* request rate, concurrency level, response time) and exposing them to a metrics server to facilitate control plane decision-making, *e.g.,* autoscaling. However, this design has several drawbacks we described earlier. We overcome these with our lightweight, event-driven eBPF based EPROXY, replacing the queue proxy.

The goal of EPROXY is to achieve functionality comparable to that of the queue proxy, but with lower overhead. We do not need the queueing capability in the EPROXY as the shared memory within the function chain already provides that queueing. Thus, SPRIGHT still provides the same functionality to improve concurrency and handle traffic bursts as with a queue proxy. But eliminating the additional queuing stage helps reduce request delays.

To collect the required metrics for the Knative control plane, we attach eBPF-based monitor programs to the function pods, as shown in Fig. 4. In addition, we assign a 'metrics map' in the eBPF Maps that serves as a local metrics storage for each node. When a new request or response occurs, the monitor programs are triggered to collect and update the metrics to the metrics map. The SPRIGHT gateway has a built-in metrics agent, responsible for reading the metrics map periodically, providing the latest metrics to the metrics server. We further extend the SPRIGHT gateway with internal event-driven metrics collection capabilities as an enhancement of EPROXY to provide fine-grained L7 metrics. Since the EPROXY is only triggered when there are incoming requests, there is no CPU overhead when it is idle.

Kubernetes natively supports function pod health checks via a *kubelet*, as an indispensable process for pod management running on each physical node. We directly leverage *kubelet's* health check capability to proactively collect this pod health information and expose it to the control plane for load balancing decisions. Thus, we can dispense with Knative's queue proxy doing a health check to check on function pods, using HTTP probing.

3.4 eBPF-based dataplane acceleration for external communication

We exploit eBPF's XDP/TC hooks to accelerate the communication by the function chain in SPRIGHT to external components. We develop an eBPF forwarding program and attach it to the XDP/TC hook that is positioned on the receive (RX) path of the network interface, including the host-side veth

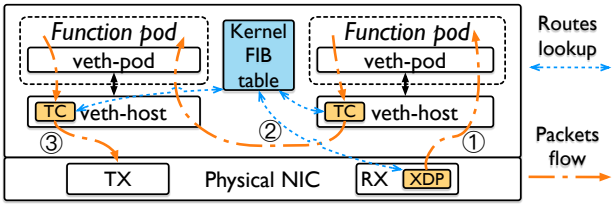

Figure 6: Dataplane acceleration based on eBPF XDP/TC hooks

of the pod (*i.e., veth-host*[2]) and the physical NIC, as shown in Fig. 6. eBPF offers packet redirect features (*i.e.,* 'XDP_REDIRECT' and 'TC_ACT_REDIRECT') that support passing raw frames between the virtual network interfaces, or to and from the physical NIC without going through the kernel protocol stack [63]. This helps save CPU cycles consumed by iptables. The eBPF forwarding program has two functions: 1) Look up the kernel FIB (Forwarding Information Base) table to find the destination network interface based on the FIB parameters [5] of the received packet, including the IP 5-tuple, index of source interface, *etc.* 2) Forward the raw packet frame to the target (*veth-host* or NIC) interface via 'XDP_REDIRECT' or 'TC_ACT_REDIRECT'. The communication could be either in the same node or across different nodes, supported by an eBPF-based dataplane via the eBPF forwarding program. An XDP program at the physical NIC processes all inbound packets received by the NIC. It redirects the packet to the *veth-host* of the destination function pod after a routing table lookup (① in Fig. 6). The TC program at the *veth-host* handles the outbound packet from the function pod. Depending on the destination of the packet, the TC program may take different routes. If the destination of the packet is to another function pod (*e.g.,* traffic between ingress gateway pod and SPRIGHT gateway pod) on the same node, the TC program directly passes the packet to the *veth-host* of the destination function pod via 'TC_ACT_REDIRECT' (② in Fig. 6). If the destination function pod is on another node, the TC program redirects the packet to the NIC (③ in Fig. 6). Our evaluation in Appendix.B shows that XDP/TC redirection helps achieve a 1.3× improvement in throughput and a 20% reduction in latency under peak load.

3.5 Event-driven protocol adaptation

To run a protocol adapter as an internal, lightweight event-driven component, we predefine 'protocol adaptation hook points' on the packet datapath inside the SPRIGHT gateway, just before the gateway sends messages to the function pod. The protocol adaptation hook is a function call entry point

[2]A function pod is connected to the host through a pair of veths, *i.e.,* the host-side veth and pod-side veth.

that can be invoked to execute customized protocol adaptation programs that are attached. Once an application-specific message arrives at the hook point, the protocol adaptation program is triggered and executed. With internal event-driven execution, invocations are integrated into the same component without introducing extra context switching and networking overhead. Our design supports attaching the protocol adapter program at runtime by exploiting dynamic code injection [8]. Different programs are pre-compiled into a dynamic library and can be loaded to, or unloaded from, the hook point at runtime according to the protocol adapter's requirements. This facilitates compatibility when handling traffic specific to each distinct protocol. In addition, dynamic loading of the program helps reduce startup time compared to initializing a separate protocol adapter function pod.

Our adapter works seamlessly with stateless protocol adaptation, *e.g.,* HTTP, since the management of the transport layer (L4) connections is offloaded to the SPRIGHT gateway. However, some adaptation scenarios with stateful protocols, *e.g.,* MQTT, requires an additional L7 connection establishment before exchanging messages [15]. To retain stateless protocol adaptation, we use the SPRIGHT gateway to handle the L7 connection establishment rather than the internal protocol adapter. Then, the SPRIGHT gateway passes the received application messages to the event-driven protocol adapter, which extracts the payload from the application message and delivers it to shared memory. To improve interoperability and compatibility with current serverless platforms, our adapter is designed to be compatible with the CloudEvent specification [7], an event data format definition widely adopted by serverless platforms [12, 22].

3.6 Intelligent autoscaling

In the control plane, the autoscaler scrapes metrics from the metrics server to determine the load intensity, based on which serverless function instances are automatically scaled up or down to serve requests on demand. Knative's autoscaler depends primarily on the users to specify the requested resources for their functions based on a single metric, and is unaware of the complexity of function chains. To improve on Knative, several approaches, *e.g.,* Mu [53], GRAF [56], *etc.* have been proposed, to more effectively scale cloud resources for serverless applications. SPRIGHT can be used in conjunction with these advanced autoscalers.

3.7 Overhead auditing (contd.): SPRIGHT

We now perform an audit of overhead for SPRIGHT, following the same methodology used before in §2 and compare it against the base design depicted in Fig. 1. As can be seen in Table. 2, SPRIGHT significantly reduces overheads for processing within the function chain. With shared memory processing, SPRIGHT achieves **0** data copies, **0** additional protocol processing, and **no** serialization/deserialization overheads within the chain. Although the use of SKMSG generates context switches and interrupts, which do add latency for processing, the total number of context switches and interrupts for SPRIGHT is still much less than that of the base Knative design (repeated in the last column of Table. 2). In addition, the results in Fig. 5 show that the context switches and interrupts introduced by SKMSG have limited impact on the performance with concurrent processing of just a few sessions. The event-based shared memory processing brings substantial reduction of resource usage, more than compensating for any of the added context switches and interrupts.

Table 2: Per request data pipeline overhead for SPRIGHT for a '1 broker/front-end + 2 functions' chain (excluding client side overhead). Note-2: SPRIGHT uses DFR: so there is no route ④ and ⑤ (Fig. 1). ④: SPRIGHT's direct route from function-1's pod to function-2's pod.

Data Pipeline No.	*External*			*Within chain*			**Total**	**Total of Kn**
	①	②	*total*	③	④	*total*		
# of copies	1	2	3	0	0	0	3	15
# of context switches	1	2	3	2	2	4	7	15
# of interrupts	3	4	7	2	2	4	11	25
# of proto. processing tasks	1	2	3	0	0	0	3	12
# of serialization	1	1	2	0	0	0	2	8
# of deserialization	0	1	1	0	0	0	1	7

4 EVALUATION & ANALYSIS

4.1 Serverless Workloads Setup

To examine the improvement of SPRIGHT and its components, we consider several typical serverless scenarios, including (1) a popular online shopping boutique, (2) An IoT environment of motion detectors, (3) a more complex processing of image detection & charging for an automated parking garage. For each scenario, we setup a function chain to execute the serverless application (Fig. 7). The details of the setup for each scenario are as follows:

1. Online Boutique: 'Online Boutique' is an open source benchmark of microservices consisting of 10 different functions [19]. Based on the online boutique implementation provided in [19], we measured the average execution time of each function, which is then used as the CPU service time of the functions implemented in our SPRIGHT setup (Appendix.C). We use Locust [14] as the load generator and use the default workload provided in [19] to implement a realistic web-based shopping application's request pattern. The default workload utilizes a total of 6 different sequences of function chains (Appendix.C). Since the online boutique employs a front-end proxy to coordinate the invocations within the chain, we use NGINX as the front-end proxy for the base

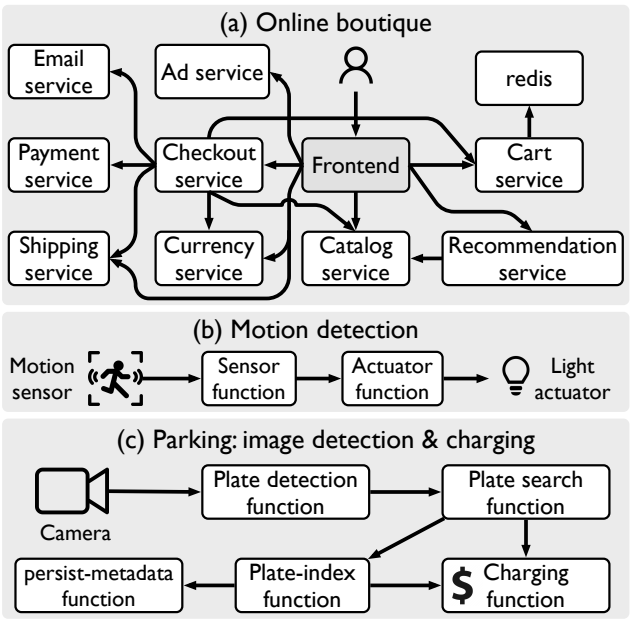

Figure 7: Serverless function chains setup

Knative setup. SPRIGHT gateway serves as the front-end proxy for our SPRIGHT setup.

2. IoT - Indoor motion detection for automated lighting: Indoor motion detection requires tracking a sequence of events utilizing multiple sensors. The simple function chain contains 2 functions (Fig. 7 (b)). Motion sensors going 'on' triggers an actuator function to turn on the light. The light may be automatically turned off after a period of no activity. We consider the MERL motion detector dataset [68]. We use a traffic generator developed in Python to send motion events based on the timestamps in the dataset. The CPU service time of the sensor function and actuator function are both set at 1ms. For the base Knative setup, we use NGINX to coordinate the communication within the function chain.

3. Parking: image detection & charging: This application takes snapshots of each parking spot as input for visual occupancy (of parking spots) detection in parking lots. It detects the vehicle's license plate and determines whether the plate metadata is stored in the database through a plate search function. If it is not stored, a 'persist-metadata' function is invoked to store the plate metadata in the database. Finally, it charges parking fees based on the license plate's metadata. We consider the *CNRPark+EXT* image dataset collected from a parking lot with 164 parking spaces [27]. We use the same load generator used for IoT workload to send snapshot images (150×150 pixels, ~ 3KB each) through HTTP/REST API call. Every 240 second interval, 164 snapshots are sent to the function chain. We use NGINX to coordinate the message exchanges within the chain. We use VGG-16 as the image detection algorithm and the CPU service time of image detection function is set to 435ms [36]. The CPU service times of other functions are listed in Appendix.C.

We use these serverless applications to quantify the performance gain brought by each of SPRIGHT's optimization. We evaluate it based on a number of different metrics, including CPU usage, RPS, response time, and to understand in detail, show the time series and CDF, when appropriate.

4.2 Experiment setup

The components of testbed is built on top of a base Knative platform include 1) Knative serving components (v0.22.0) [13]; 2) Knative eventing components (v0.22.0) [12]; 3) Kubernetes components (v1.19.0), including API server, placement engine, etcd, *etc* [2]. We consider Calico CNI (Native routing mode) [64] as the underlying networking solution except the communication within the function chain of SPRIGHT. We run the experiments on the NSF Chameleon Cloud with two nodes [45]. Each node has a 64-core Intel Cascade Lake CPU@2.8 GHz, 192GB memory, and a 10Gb NIC. We use Ubuntu 20.04 with kernel version 5.15. We configure the concurrency of both Knative and SPRIGHT function as 32. The concurrency level of a function pod determines the # of requests that can be processed in parallel at each time.

4.3 Performance with Realistic Workloads

4.3.1 Comparing SPRIGHT and Knative. We now compare SPRIGHT with Knative, for all the different function chains (*i.e.,* `Ch-1` to `Ch-6`) of the online boutique application. To evaluate Knative, DPDK, and `SKMSG`, we configure different concurrency levels (*i.e.,* # of concurrent users) of requests from the Locust load generator. We select two concurrency levels 4K and 12K to show here. To get to the 4K concurrency, we set the spawn rate of concurrency of 100/sec. The spawn rate controls the # of concurrency steps to increase per second. Above 4K, Knative's performance is highly variable over time, indicating overload (especially very high tail response times). At 12K, SPRIGHT's `SKMSG` alternative consumes about 75% CPU utilization in the core executing the

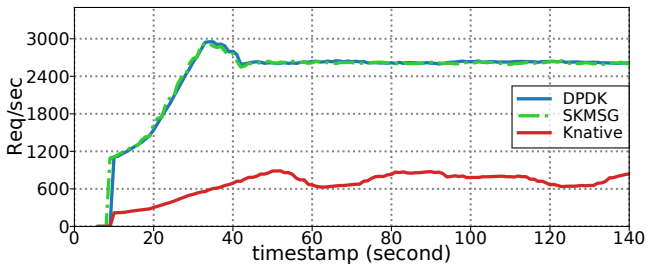

Figure 8: RPS for online boutique workload: base Knative at 4K & {DPDK, `SKMSG`} at 12K concurrency.

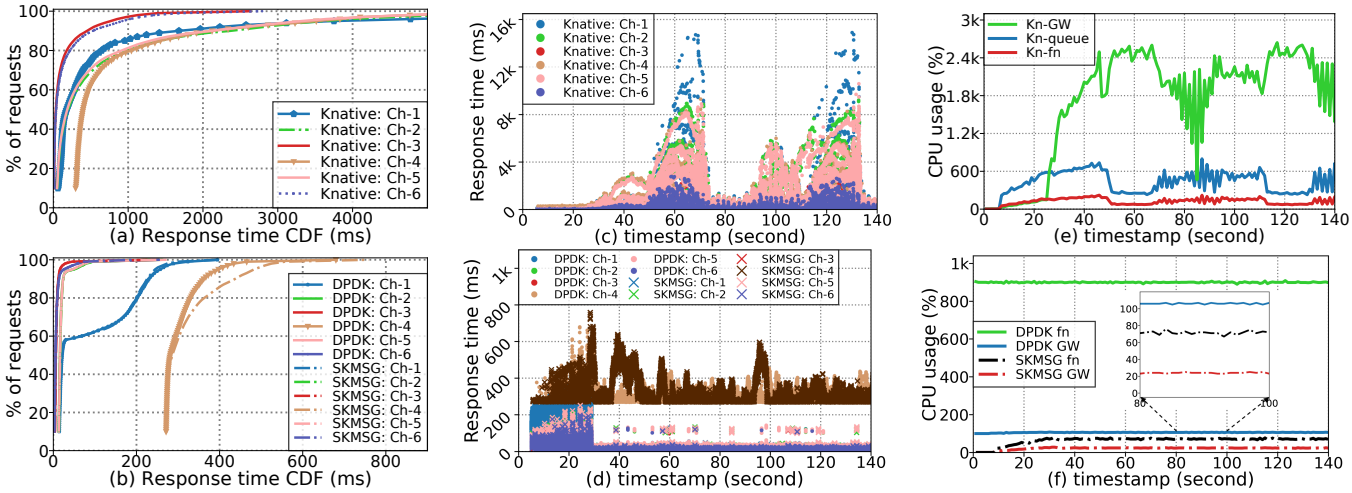

Figure 9: Online boutique service performance. Top row: Base Knative, 4K concurrency. Bottom row: {DPDK, SKMSG }, 12K concurrency. (Left) Response time CDF for 6 different function chains; (Middle) Time series of response times of function chains; (Right) Times series of CPU usage for gateway, function chains, and queue proxy (Knative).

functions, and we sought to demonstrate stable performance at that point. For 12K concurrency, we set the spawn rate of concurrency to 500/sec.

At 4K concurrency, Knative's gateway begins to be overloaded. From 10s to 50s (Fig. 8), as the concurrency level of load generator is ramping up to 4K, the requests/sec (RPS) increases to (~890 req/sec). The Knative gateway begins to overload (at 50s in Fig. 8) due to interrupt processing of incoming requests and from the functions in the chain. This slows down request processing, leading to the reduction in RPS observed (beyond 50s in Fig. 8). The closed loop nature of the workload generation and request processing results in the RPS, resource utilization and response times going through cycles of overload (occurring again at 113s to 132s). Fig. 9 (c) further demonstrates the overload of Knative. For requests sent between 50s and 72s, the response time increases significantly, contributed by large queueing at the gateway. The resulting large tail latency as shown in Fig. 9 (a) with a 95%ile of 3000ms, measured across all the functions of the Boutique service. With a concurrency of 4K (from 50s onwards), the entire Knative setup consumes ~30 CPU cores, which is 46% of the total CPU available on the physical node.

Compared to Knative, DPDK and SKMSG both have stable RPS throughout the experiment for concurrency levels from 4K to 12K. At 4K, The 95%ile latency of DPDK and SKMSG is 160ms, 19× less than Knative (3000ms), while utilizing far less CPU. Although DPDK constantly consumes CPU cycles when idle, even at maximum load, it consumes only 10 total CPU cores, which is 3× less than Knative (similar to Fig. 5). This again validates the benefits of shared memory processing, saving CPU resources by avoiding the various needless processing overheads with Knative discussed previously in §2. SKMSG further reduces the CPU usage dramatically by using purely event-driven processing compared to DPDK. With 4K concurrency, SKMSG consumes only 0.42 CPU cores, including the gateway and all the functions, getting comparable performance (throughput, response time) to DPDK.

We further increase the concurrency level of the load generator to 12K for DPDK and SKMSG, increasing the utilization while still maintaining low tail response times. Both DPDK and SKMSG maintain a stable RPS of ~2600 req/sec (Fig. 8), which is 3× higher than the highest stable RPS achieved with Knative. Even with complex chains, diverse functions and higher CPU service times for some of the functions, the performance benefits of SPRIGHT are dramatic. SKMSG uses much less CPU resources compared to DPDK, even as the load increases. At 12K concurrency, SKMSG consumes only ~0.94 CPU cores, which is 10× less than DPDK, showing the benefit of eBPF-based event driven processing.

With SKMSG generating context switches and interrupts for descriptor delivery (Table. 2), there is additional latency in SKMSG's shared memory processing. SKMSG is slightly worse than DPDK in terms of tail latency. For the 4th function chain (*i.e.,* Ch-4) which has the Checkout service function with a higher CPU service time (260ms), SKMSG shows worse tail latency compared to DPDK, in Fig. 9 (b). The additional delay for SKMSG's descriptor delivery, adds to the transient queueing and hence longer tail latency. However, as we said in §3.2.2, the impact of this additional latency introduced by SKMSG is quite limited. Further, the higher CPU service time of the functions dwarfs the extra latency introduced by SKMSG in relative terms. Importantly, the SKMSG throughput (RPS) is very close to DPDK.

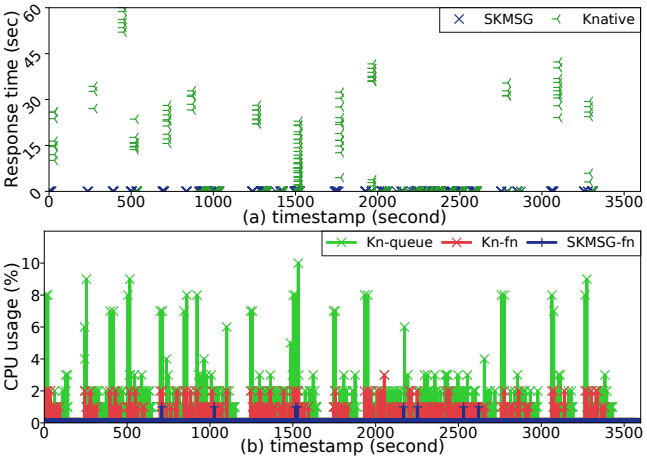

Figure 10: Time series of response time, and CPU utilization for motion detection workload - 1-hour long experiments.

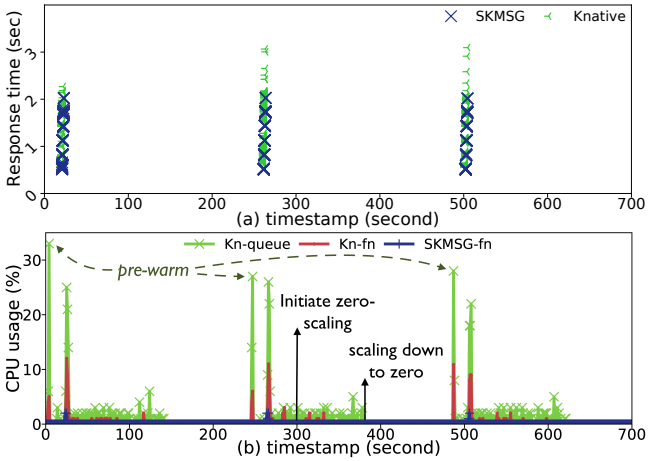

Figure 11: Parking: image detection & charging performance. (a) Time series of response times of function chains; (b) Times series of aggregated CPU usage for function chains, and queue proxy (Knative).

4.3.2 Bypassing the impact of cold start and zero scaling. We set up an experiment with zero scaling enabled in Knative to study the impact of cold start. Without incoming requests, Knative scales functions down to zero to save resources and reduce costs. We set the 'grace period' for scaling down to zero as 30 seconds. In contrast, we keep functions in SPRIGHT 'warm' by having a minimum number of active function pods, knowing that our purely event-driven processing will not consume CPU resources when idle. We use the motion detection workload to study the impact of cold start, because of the intermittent nature of such IoT traffic.

Fig. 10 (a) clearly shows the impact of cold start in Knative, with large response times that possibly render the motion detection application ineffective, and severely violate SLOs. E.g., starting at 1500s, a number of motion events occur one after another (inter-arrival time of a few seconds) that are sent to the currently zero scaled function chain. The first motion event that arrives at the gateway is queued and triggers the instantiation of the functions. Since a serverless function pod take some time to start, subsequent requests have to be queued. The cascading effect during the cold start of the entire function chain further degrades the response time [56], resulting in a long tail latency going up to 60s. Once the function is active, Knative has a reasonably small response time when there are consecutive incoming events (*e.g.,* before the grace period terminates between 2000s and 2500s), that keep the functions 'warm'.

In contrast, SPRIGHT shows consistently low response times over the entire runtime of the workload since there is always an active pod to serve the request without leaving requests waiting in the queue (we can sidestep going down to zero-scale). More importantly, although SPRIGHT keeps one (or more) function warm, the event-driven nature of SPRIGHT leads to negligible CPU consumption when there is no traffic. In fact, with Knative, the higher resource usage of the queue proxy under load more than offsets any benefit of Knative's zero-scaling. E.g., in Fig. 10 (b), the spikes in the CPU usage for the queue proxy (*e.g.,* at 1500s), even when handling small traffic is quite wasteful, and is eminently avoidable with SPRIGHT's event-driven design.

Since the 'Parking: image detection & charging' workload has a distinct periodic arrival pattern (*e.g.,* monitoring and billing every 4 minutes), we configure a 'pre-warm' phase for Knative functions 20 seconds before the next burst is scheduled to arrive. 'Pre-warming' helps avoid the penalty of the cold start delay of serverless functions, while trading-off a small amount of the resource savings of shutting down the pods in serverless computing with zero-scaling [59]. However, as observed in Fig. 11 (b), the CPU usage for each function instantiation at the pre-warming stage in fact exceeds the CPU usage consumed by request processing (*i.e.,* observe the CPU usage spike for the pre-warming and the function execution 20 seconds later). Thus, while zero-scaling reduces CPU usage if the idle period is long, there is a CPU cost for frequent creation/destruction of functions that has to be considered. Knative also is quite inefficient for scaling functions down to zero. When there is no traffic for a grace period of 30s (*e.g.,* 270s to 300s in Fig. 11 (b)), Knative begins scaling down the functions to zero. But, functions remain in a 'terminating' state until 380s without being really terminated or releasing CPU resources. Thus, the scaling down process lasts as long as 80s, during which all the Knative queue proxies and functions are consuming CPU resources, which is unnecessary and wasteful.

For comparison, SKMSG consumes only a small amount of CPU throughout the entire period, in fact with slightly

lower (about 16%) response times (both average and 95%, Fig. 11 (a)). Overall, without resorting to zero-scaling, SKMSG saves up to 41% CPU cycles in this 700s experiment, almost doubling system capacity compared to Knative.

5 RELATED WORK

In recent years, a number of serverless platforms have been launched, *e.g.,* AWS Lambda [28], IBM Cloud Functions [43], Apache OpenWhisk [4], OpenFaaS [20], Knative [11], *etc*, to support cloud-resident applications. Work on understanding the performance impact of commercial or open-source serverless platforms [31, 49], have guided us on the design of SPRIGHT. Li *et al.* [49] showed that the overhead of the ingress gateway reduced the throughput by 13%, compared to the performance of function invocation using the 'direct call' mode (*i.e.,* the client directly invokes the function instance, bypassing the ingress gateway). Priscilla *et al.* [31] studied the suitability of different serverless function startup modes (*i.e.,* cold and warm) for supporting IoT applications, indicating that cold start can have significant resource-saving benefits, but can impact response time. This prompts us to examine the resource consumption and overheads of each component carefully.

Several past works have examined the inefficiency and overheads that exist in Linux networking, including data copies and context switching [35, 46, 48, 54]. The overhead of protocol processing [57], and serialization-deserialization [44, 67] directly impacts networking performance, which applies to the container-based serverless function, including function chains. A variety of optimizations have been proposed to improve the network performance for different application scenarios, which can be complementary to current Linux networking (*e.g.,* XDP [40], AF_XDP in OVS [65]) or bypass kernel-based networking (*e.g.,* NetVM for NFV [41]). Our work combines the advantages of kernel bypass zero-copy networking where essential for serverless function chains, and leveraging eBPF-based event-driven processing.

There are multiple proposals to optimize different aspects of serverless frameworks, *e.g.,* runtime overhead reduction [25, 26, 39, 55], intelligent resource provisioning and traffic management [53, 60]. Further, [56], [33], [61] aim to optimize resource allocation and deployment of serverless functions on the basis of a chain, which improves the efficiency and flexibility for building microservices using serverless function chaining. However, they do not focus on optimizing the dataplane, which as we show has a significant impact.

'Cold start' in serverless: The cold start latency of serverless functions detracts from their being an ideal framework for building microservices. Fu *et al.* [37] propose a startup latency optimization specifically for Kubernetes-based environments by placing pods on nodes that have container image dependencies locally to avoid the latency of pulling images. However, their 95%ile startup latency after optimization is still around 23s, which still severely impacts the QoS. In addition, start-up (either cold start or pre-warm [59]) adds additional costs as we have observed, making optimizations built around cold start less desirable. A policy of 'keep-warm' of pods has been an alternative to mitigate the cold start latency in serverless [50]. They can achieve 85% improvement of the tail (99%ile) latency. Although, [50] considerably improves the SLOs, it is built on top of Knative with heavyweight components (*e.g.,* queue proxy) and the result is excessive resource usage. Fuerst *et al.* [38] consider greedy-dual caching to determine which functions should be kept as warm. By factoring in several key indicators of a function, *e.g.,* memory footprint, invocation frequency *etc.* , they can prioritize functions to be kept warm, thus limiting memory consumption to keep a minimum number of warm functions and achieve SLOs. Since SPRIGHT primarily contributes to control the CPU usage, [38] can be a good complement to SPRIGHT to reduce memory utilization.

6 CONCLUSIONS

SPRIGHT demonstrated the effectiveness of event-driven capability for reducing resource usage in serverless cloud environments. With extensive use of eBPF-based event-driven capability in conjunction with high-performance shared memory processing, SPRIGHT achieves up to 3× throughput improvement, 19× latency reduction and 31× CPU usage savings than Knative serving a complex web workload. Compared to an environment using DPDK for providing shared memory and zero-copy delivery, SPRIGHT achieves competitive throughput and latency, while consuming 10× less CPU resources. Additionally, for intermittent request arrivals typical of IoT applications, SPRIGHT still improves the average latency by 16%, compared to Knative using 'pre-warmed' functions, while reducing CPU cycles by 41%. This makes it feasible for SPRIGHT to support several 'warm' functions at a minimum overhead (as CPU usage is load-proportional), side-stepping the 'cold-start' latency problem. Across several typical serverless workloads, SPRIGHT shows higher dataplane performance, while reducing the inefficiencies in current open-source serverless environments, thus getting us closer to meeting the promise of serverless computing.

We recognize the need for isolation between serverless functions in a shared and untrusted cloud environment, especially with the use of SPRIGHT's shared memory processing. We are implementing function-chain-level separation by restricting access of a private shared memory pool to only trusted functions of that chain. The SPRIGHT gateway further provides traffic isolation by routing requests into the private shared memory pool based on the destination function chain. We expect to report on experiments to demonstrate its efficacy shortly.

REFERENCES

[1] 2021. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. (2021). [online].

[2] 2021. Kubernetes Components. kubernetes.io/docs/concepts/overview/components/. (2021). [online].

[3] 2021. wrk - a HTTP benchmarking tool. github.com/wg/wrk. (2021). [online].

[4] 2022. Apache OpenWhisk. openwhisk.apache.org/. (2022). [online].

[5] 2022. BPF-HELPERS - list of eBPF helper functions. man7.org/linux/man-pages/man7/bpf-helpers.7.html. (2022). [online].

[6] 2022. Chaining OpenFaaS functions. ericstoekl.github.io/faas/developer/chaining_functions/. (2022). [online].

[7] 2022. CloudEvents Spec. github.com/cloudevents/spec. (2022). [online].

[8] 2022. Dynamically Loaded (DL) Libraries. tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html. (2022). [online].

[9] 2022. Istio Architecture. istio.io/latest/docs/ops/deployment/architecture/. (2022). [online].

[10] 2022. Istio Traffic Management. istio.io/latest/docs/concepts/traffic-management/. (2022). [online].

[11] 2022. Knative. knative.dev. (2022). [online].

[12] 2022. Knative Eventing. knative.dev/docs/eventing/. (2022). [online].

[13] 2022. Knative Serving. knative.dev/docs/serving/. (2022). [online].

[14] 2022. Locust: An open source load testing tool. locust.io/. (2022). [online].

[15] 2022. MQTT Version 5.0. docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. (2022). [online].

[16] 2022. Multi-process Support of DPDK. doc.dpdk.org/guides/prog_guide/multi_proc_support.html. (2022). [online].

[17] 2022. NGINX. www.nginx.com/. (2022). [online].

[18] 2022. of-watchdog. github.com/openfaas/of-watchdog. (2022). [online].

[19] 2022. Online Boutique by Google. github.com/GoogleCloudPlatform/microservices-demo. (2022). [online].

[20] 2022. OpenFaaS. www.openfaas.com/. (2022). [online].

[21] 2022. OpenFaaS API Gateway / Portal. docs.openfaas.com/architecture/gateway/. (2022). [online].

[22] 2022. OpenFaaS Triggers. docs.openfaas.com/reference/triggers/. (2022). [online].

[23] 2022. OpenWhisk - Creating action sequences. github.com/apache/openwhisk/blob/master/docs/actions.md#creating-action-sequences. (2022). [online].

[24] 2022. OpenWhisk Composer. github.com/apache/openwhisk-composer. (2022). [online].

[25] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[26] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[27] Giuseppe Amato, Fabio Carrara, Fabrizio Falchi, Claudio Gennaro, and Claudio Vairo. 2016. Car parking occupancy detection using smart camera networks and deep learning. In *Computers and Communication (ISCC), 2016 IEEE Symposium on*. IEEE, 1212–1217.

[28] Amazon Web Services, Inc. 2022. AWS Lambda. aws.amazon.com/lambda/. (2022). [online].

[29] Amazon Web Services, Inc. 2022. AWS Serverless API. docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-resource-api.html. (2022). [online].

[30] Apache Software Foundation. 2022. APACHE KAFKA. kafka.apache.org/. (2022). [online].

[31] Priscilla Benedetti, Mauro Femminella, Gianluca Reali, and Kris Steenhaut. 2021. Experimental Analysis of the Application of Serverless Computing to IoT Platforms. *Sensors* 21, 3 (2021), 928.

[32] Matteo Bertrone, Sebastiano Miano, Jianwen Pi, Fulvio Risso, and Massimo Tumolo. 2018. Toward an eBPF-based clone of iptables. *Netdev'18* (2018).

[33] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. *Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3472883.3486992

[34] Carsten Bormann, Angelo P Castellani, and Zach Shelby. 2012. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing* 16, 2 (2012), 62–67.

[35] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888

[36] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. ECML: Improving Efficiency of Machine Learning in Edge Clouds. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. 1–6. https://doi.org/10.1109/CloudNet51028.2020.9335804

[37] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. 2020. Fast and Efficient Container Startup at the Edge via Dependency Scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association. https://www.usenix.org/conference/hotedge20/presentation/fu

[38] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757

[39] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge. In *Proceedings of the 21st International Middleware Conference (Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 265–279. https://doi.org/10.1145/3423211.3425680

[40] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[41] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 445–458. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

[42] IBM. 2022. Creating serverless REST APIs. cloud.ibm.com/docs/openwhisk?topic=openwhisk-apigateway. (2022). [online].

[43] IBM. 2022. IBM Cloud Functions. cloud.ibm.com/functions/. (2022). [online].

[44] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. *SIGARCH Comput. Archit. News* 43, 3S (jun 2015), 158–169. https://doi.org/10.1145/2872887.2750392

[45] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[46] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. 2021. Parallelizing Packet Processing in Container Overlay Networks. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 261–276. https://doi.org/10.1145/3447786.3456241

[47] Joshua Levin and Theophilus A. Benson. 2020. ViperProbe: Rethinking Microservice Observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. 1–8. https://doi.org/10.1109/CloudNet51028.2020.9335808

[48] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the Cost of Context Switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS '07)*. Association for Computing Machinery, New York, NY, USA, 2–es. https://doi.org/10.1145/1281700.1281702

[49] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. 2019. Understanding Open Source Serverless Platforms: Design Considerations and Performance. In *Proceedings of the 5th International Workshop on Serverless Computing (WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/3366623.3368139

[50] Ping-Min Lin and Alex Glikson. 2019. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. (2019). arXiv:cs.DC/1903.12221

[51] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a Faster and Scalable Iptables. *SIGCOMM Comput. Commun. Rev.* 49, 3 (nov 2019), 2–17. https://doi.org/10.1145/3371927.3371929

[52] Microsoft. 2022. Azure - Function chaining in Durable Functions. docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-sequence?tabs=csharp. (2022). [online].

[53] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G. Kulkarni, Dan Li, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2021. *Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-Constrained Edge Clouds*. Association for Computing Machinery, New York, NY, USA, 168–181. https://doi.org/10.1145/3472883.3487014

[54] Jeffrey C Mogul and KK Ramakrishnan. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems* 15, 3 (1997), 217–252.

[55] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[56] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: A Graph Neural Network Based Proactive Resource Allocation Framework for SLO-Oriented Microservices. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '21)*. Association for Computing Machinery, New York, NY, USA, 154–167. https://doi.org/10.1145/3485983.3494866

[57] Shixiong Qi, Sameer G. Kulkarni, and K. K. Ramakrishnan. 2021. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 656–671. https://doi.org/10.1109/TNSM.2020.3047545

[58] Red Hat, Inc. 2022. Understanding the eBPF networking features in RHEL. access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/assembly_understanding-the-ebpf-features-in-rhel_configuring-and-managing-networking. (2022). [online].

[59] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[60] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. *Atoll: A Scalable Low-Latency Serverless Platform*. Association for Computing Machinery, New York, NY, USA, 138–152. https://doi.org/10.1145/3472883.3486981

[61] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 311–327. https://doi.org/10.1145/3419111.3421306

[62] The Linux Foundation. 2022. eBPF. ebpf.io/. (2022). [online].

[63] Tigera, Inc. 2022. eBPF XDP: The Basics and a Quick Tutorial. www.tigera.io/learn/guides/ebpf/ebpf-xdp/. (2022). [online].

[64] Tigera, Inc. 2022. Project Calico. www.tigera.io/project-calico/. (2022). [online].

[65] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the Open VSwitch Dataplane Ten Years Later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 245–257. https://doi.org/10.1145/3452296.3472914

[66] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (feb 2020), 36 pages. https://doi.org/10.1145/3371038

[67] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. 2021. Zerializer: Towards Zero-Copy Serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/3458336.3465283

[68] Christopher R. Wren, Yuri A. Ivanov, Darren Leigh, and Jonathan Westhues. 2007. The MERL Motion Detector Dataset. In *Proceedings of the 2007 Workshop on Massive Datasets (MD '07)*. Association for Computing Machinery, New York, NY, USA, 10–14. https://doi.org/10.1145/1352922.1352926

[69] Muneer Bani Yassein, Mohammed Q Shatnawi, Shadi Aljwarneh, and Razan Al-Hatmi. 2017. Internet of Things: Survey and open issues of MQTT protocol. In *2017 international conference on engineering & MIS (ICEMIS)*. IEEE, 1–6.

[70] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM.

[71] Jianer Zhou, Xinyi Qiu, Zhenyu Li, Gareth Tyson, Qing Li, Jingpu Duan, and Yi Wang. 2021. Antelope: A Framework for Dynamic Selection

of Congestion Control Algorithms. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. 1–11.

A A CASE STUDY OF MQTT PROTOCOL ADAPTER

The MQTT adapter translates the MQTT 'PUBLISH' message (*i.e.,* the message type used by MQTT-based components to forward the payload to subscriber functions) in the payload by removing the application layer header. Once the adapter intercepts an MQTT 'PUBLISH' message, it extracts the MQTT-related metadata (*e.g.,* topic) and the payload. The payload is stored in shared memory. The MQTT-related metadata (*e.g.,* topic) is used for making the appropriate DFR routing decision based on the subscribing function's information. In this way, serverless functions in SPRIGHT can transparently handle MQTT requests without changing any application logic and still support the stateful processing needed for MQTT. Our adapter design can be easily extended to support other application-specific protocols, *e.g.,* CoAP.

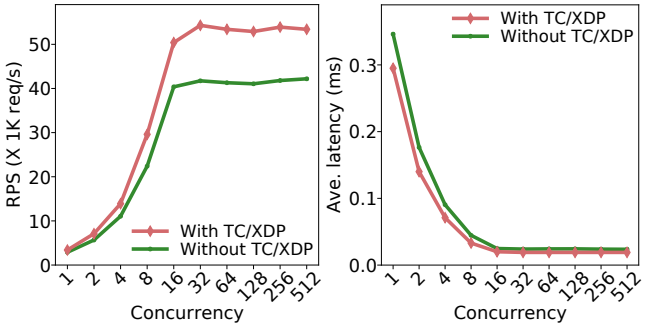

Figure 12: Performance impact of TC/XDP redirect: RPS and latency

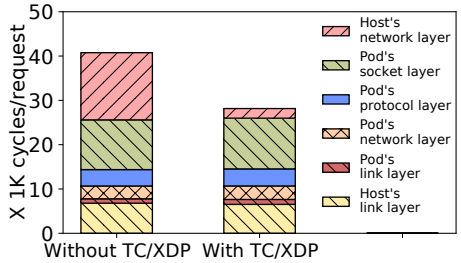

Figure 13: CPU overhead breakdown of receiver side kernel stacks: with/without TC/XDP redirect acceleration.

B IMPROVEMENT WITH DATAPLANE ACCELERATION BASED ON EBPF'S XDP/TC HOOKS

To understand the improvement of eBPF's packet redirection features, including 'XDP_REDIRECT' and 'TC_ACT_REDIRECT', we evaluate the networking performance of SPRIGHT when the eBPF's packet redirection is enabled. We reuse the experiment setup of SKMSG described in §3.2.2. We further breakdown the CPU cycles of the kernel stack processing to accurately quantify the CPU cycles saved by eBPF-based packet redirection.

Fig. 12 compares the RPS and response latency performance of SPRIGHT with eBPF's packet redirection feature enabled and disabled. With a concurrency of 32, SPRIGHT with TC/XDP redirection enabled has a 1.3× improvement in RPS compared to SPRIGHT without TC/XDP redirection. Since TC/XDP redirection transfers raw packets between network devices (*i.e.,* veth and NIC), the overhead spent in kernel iptables can be saved, thus in turn benefiting throughput. The response latency of SPRIGHT with TC/XDP redirection is 19μs for a concurrency of 32, which is 5μs less than SPRIGHT without TC/XDP redirection (24μs). The overhead and response latency savings remain as the concurrency increases, allowing SPRIGHT to maintain a peak RPS of 53K when TC/XDP redirection is enabled.

We further break down the CPU cycles spent on processing a request (at 32 concurrency) based on where it is spent, including host's kernel stack and the pod's kernel stack, as shown in Fig. 13. The client side overhead is excluded. In the case of SPRIGHT without TC/XDP redirection, about 15.2K CPU cycles are spent on the host's kernel networking layer for iptables processing. Whereas with SPRIGHT with TC/XDP redirection, since iptables in host's kernel stack is skipped, only 2.1K CPU cycles are consumed by the host's kernel networking layer, resulting in 86% CPU cycles saving for each request. This clearly demonstrates the benefits of eBPF's TC/XDP redirection. Bypassing the host's kernel networking stack and associated iptables processing can save considerable CPU usage and thus benefit SPRIGHT's dataplane performance for communication outside the function chain. This option, however, means the loss of full-featured iptables network policy support, which may not be required in all cases (e.g., for users only requiring higher dataplane performance).

C ADDITIONAL EXPERIMENT DETAILS

Table 3: CPU service time of functions in online boutique

No.	*Functions*	*CPU service time (ms)*
①	Product Catalog Service	0.6
②	Recommendation Service	2.5
③	Checkout Service	260
④	Currency Service	0.9
⑤	Ad Service	1.1
⑥	Email Service	0.5
⑦	Payment Service	0.44
⑧	Shipping Service	0.2
⑨	Cart Service	1.2

Table 4: Sequence of different function chains in online boutique

	Sequence of the function chain
Ch-1	④, ①, ⑨, ④, ④, ④, ④, ④, ④, ④, ④, ⑤
Ch-2	④, ⑨, ②, ①, ①, ①, ①, ①, ⑧, ④, ①, ④
Ch-3	①, ⑨
Ch-4	③, ①, ⑦, ⑧, ⑥, ②, ①, ①, ①, ①, ①, ④
Ch-5	①, ④, ⑨, ④, ②, ①, ①, ①, ①, ①, ⑤
Ch-6	④

Table 5: CPU service time of functions in parking: image detection & charging

No.	*Functions*	*CPU service time (ms)*
①	Plate detection	435
②	Plate search	20
③	Plate index	1
④	Charging	50
⑤	Persist metadata	10

Table 6: Sequence of different function chains in parking: image detection & charging

	Sequence of the function chain
Ch-1	①, ②, ③, ⑤, ④
Ch-2	①, ②, ④