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

Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, K. K. Ramakrishnan
University of California, Riverside

Visit us at: https://kknetsyslab.cs.ucr.edu/
Serverless Computing

What is Serverless computing?

• Paradigm for development and deployment of cloud applications to ease burden on users
  • Function as a service (FaaS): Users only provide application function code
  • Remove need for traditional always-on server components
  • Provisioning and managing the infrastructure becomes the cloud providers’ job

➢ Reduce user cost and complexity, and greatly improve service scalability and availability

• Challenges with serverless computing
  • Less focus on optimizing for high-performance, resource-efficiency, or being responsive
  • Need better support for both low latency processing and low resource consumption

SPRIGHT: achieve high-performance, resource-efficient serverless function chains through shared memory and event-driven processing
Serverless Computing
An abstract functional view of a serverless cloud:

Control plane
- Placement engine
- Autoscaler
- Metric Server

Data plane
- Ingress gateway
- Message Broker/Front-end proxy
- Function pods
  - Sidecar proxy
  - User container
- Execute functions

Requests
- TLS termination; auth.; ...
- Function chaining

Scale functions based on load
- Provide metrics for control plane activities
- Execute functions
- Find a node for placing a function
- Metrics collection, queuing, …
Auditing the Overheads of Serverless Computing (1)

Processing involved in a typical serverless function chain setup: network protocol, copies, interrupts, context switches etc. abound

Ingress gateway: Intercept external requests; TLS Termination, authentication, etc

![Diagram of a data pipeline showing different components and their interactions.]

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of copies</td>
<td># of ctxt switches</td>
<td># of irqs</td>
</tr>
<tr>
<td>①</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>②</td>
<td>③ total</td>
<td>④ total</td>
<td>⑤ total</td>
</tr>
</tbody>
</table>

- Physical NIC
- Userspace
- Kernel space
- External client ➔ Ingress gateway
Auditing the Overheads of Serverless Computing (2)

Broker/front-end: an intermediate component for coordinating invocations within the function chain

**Broker**: 1 copy, 1 context switches, 2 interrupts, 1 protocol processing, and 1 deserialization

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>①</td>
<td>②</td>
<td>total</td>
</tr>
<tr>
<td># of copies</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of ctxt switches</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of irqs</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of serialization</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td># of deserialization</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Auditing the Overheads of Serverless Computing (3)

**Total:** 4 copies, 4 context switches, 6 interrupts, 3 protocol processing, 2 serializations, and 2 deserializations

**Sidecar:** 2 copies, 2 context switches, 2 interrupts, 1 protocol processing, 1 serialization, and 1 deserialization

![Diagram showing the process and overheads in serverless computing](image)

### Data Pipeline No.

<table>
<thead>
<tr>
<th></th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># of copies</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of ctx switches</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of irqs</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of serialization</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td># of deserialization</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Auditing the Overheads of Serverless Computing (4)

**Total**: 4 copies, 4 context switches, 6 interrupts, 3 protocol processing, 2 serializations, and 2 deserializations

**Sidecar**: 2 copies, 2 context switches, 2 interrupts, 1 protocol processing, 1 serialization, and 1 deserialization

---

### Data Pipeline

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>①</td>
<td>②</td>
<td>total</td>
</tr>
<tr>
<td># of copies</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of ctxt switches</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of irqs</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of serialization</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td># of deserialization</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Auditing the Overheads of Serverless Computing (5)

**Total:** 4 copies, 4 context switches, 6 interrupts, 3 protocol processing, 2 serializations, and 2 deserializations

**Sidecar:** 2 copies, 2 context switches, 2 interrupts, 1 protocol processing, 1 serialization, and 1 deserialization

---

**Diagram:**

1. **Userspace**
   - Ingress gateway container
   - Broker/ Front-end container
   - Function-1’s Pod (user container, sidecar proxy, kernel protocol stack)
   - Function-2’s Pod (user container, sidecar proxy, kernel protocol stack)

2. **Kernel space**
   - Physical NIC
   - veth-pair
   - kernel protocol stack

3. **External client**

4. **Processing in function chain complete; return a response**

---

**Data Pipeline**

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>① ② total</td>
<td>③ ④ ⑤ total</td>
<td></td>
</tr>
<tr>
<td># of copies</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of ctxt switches</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of irqs</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td># of serialization</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td># of deserialization</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Auditing the Overheads of Serverless Computing**

**Key takeaways: Excessive overhead within the function chain**

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

**Takeaway#2:** Excessive, duplicate protocol processing for communication within the function chain

**Takeaway#3:** Unnecessary serialization/deserialization.

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

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>① ② total</td>
<td>③ ④ ⑤ total</td>
<td></td>
</tr>
<tr>
<td># of copies</td>
<td>1 2 3</td>
<td>4 4 4</td>
<td>12 15</td>
</tr>
<tr>
<td># of ctxt switches</td>
<td>1 2 3</td>
<td>4 4 4</td>
<td>12 15</td>
</tr>
<tr>
<td># of irqs</td>
<td>3 4 7</td>
<td>6 6 6</td>
<td>18 25</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>1 2 3</td>
<td>3 3 3</td>
<td>9 12</td>
</tr>
<tr>
<td># of serialization</td>
<td>0 1 2</td>
<td>2 2 2</td>
<td>6 8</td>
</tr>
<tr>
<td># of deserialization</td>
<td>1 1 1</td>
<td>2 2 2</td>
<td>6 7</td>
</tr>
</tbody>
</table>
What is SPRIGHT?
eBPF-based event-driven capability + Shared memory processing

**Optimization#1: Event-driven proxy**
Replacing individual, constantly-running sidecar

**Optimization#2: Shared memory processing**
Reduce data movement overhead

**Optimization#3: Direct Function Routing (DFR)**
Simplify inter-function invocations
SPRIGHT: Lightweight Serverless Function Chains

Optimization#1: eBPF-based Event-driven proxy (EPROXY and SPROXY)

- In-kernel eBPF-based “sidecar”
  - **EPROXY**: @Veth of SPRIGHT GW pod
    - Monitoring; iptables acceleration
  - **SPROXY**: Sidecar being injected at the socket level
    - Monitoring; Security; Routing

- Purely event-driven
  - No CPU overhead when there are no requests

- All in the kernel
  - Avoid extra user-kernel boundary crossings
SPRIGHT: Lightweight Serverless Function Chains

Optimization#2: Shared memory processing

- How to handle protocol processing?
  - **SPRIGHT Gateway**: Entry-point of a function chain
    - **Consolidate** kernel protocol processing
    - Move payload into shared memory
SPRIGHT: Lightweight Serverless Function Chains

Optimization#2: Shared memory processing

- How to handle protocol processing?
  - **SPRIGHT Gateway**: Entry-point of a function chain
    - **Consolidate** kernel protocol processing
    - Move payload into shared memory
  - **NO** copy, protocol processing, serialization, …
  - Packet descriptor delivery: eBPF's socket message
    - reside in Event-driven proxy (SPROXY)
    - Socket-to-socket data transfer; Routing using eBPF's socket map
    - Strictly load-proportional compared to polling-based packet descriptor delivery (DPDK RTE Ring)
SPRIGHT: Lightweight Serverless Function Chains

Optimization#3: Direct Function Routing

• Having the broker/front-end perform invocations between functions is unnecessary
  • Routing overhead

• DFR optimizes invocations within a function chain
  • The upstream function in the chain directly invokes the downstream function: bypass the gateway
  • DFR rules in eBPF’s socket map

• DFR can reduce end-to-end latency of the function chain and improve the scalability
  • Eliminate an extra hop on the datapath
  • Benefit increases as the chain scales
# Overhead auditing: Knative vs. SPRIGHT

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>① ② ③ ④ ⑤</td>
<td>① ② ③ ④ ⑤</td>
<td></td>
</tr>
<tr>
<td># of copies</td>
<td>1 2 3</td>
<td>4 4 4 12</td>
<td>15</td>
</tr>
<tr>
<td># of ctxt switches</td>
<td>1 2 3</td>
<td>4 4 4 12</td>
<td>15</td>
</tr>
<tr>
<td># of irqs</td>
<td>3 4 7</td>
<td>6 6 6 18</td>
<td>25</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>1 2 3</td>
<td>3 3 3 9</td>
<td>12</td>
</tr>
<tr>
<td># of serialization</td>
<td>0 1 2</td>
<td>2 2 2 6</td>
<td>8</td>
</tr>
<tr>
<td># of deserialization</td>
<td>1 1 1</td>
<td>2 2 2 6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>① ② total</td>
<td>③ ④ total</td>
<td>⑤ total</td>
</tr>
<tr>
<td># of copies</td>
<td>1 2 3</td>
<td>0 0 0</td>
<td>3</td>
</tr>
<tr>
<td># of ctxt switches</td>
<td>1 2 3</td>
<td>2 2 4</td>
<td>7</td>
</tr>
<tr>
<td># of irqs</td>
<td>3 4 7</td>
<td>2 2 4</td>
<td>11</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>1 2 3</td>
<td>0 0 0</td>
<td>3</td>
</tr>
<tr>
<td># of serialization</td>
<td>0 1 2</td>
<td>0 0 0</td>
<td>2</td>
</tr>
<tr>
<td># of deserialization</td>
<td>1 1 1</td>
<td>0 0 0</td>
<td>1</td>
</tr>
</tbody>
</table>

- 0 data copies, 0 protocol processing, 0 serialization/deserialization overheads **within the chain**

*The event-based shared memory processing brings substantial reduction of overheads for communication within the serverless function chain*
Evaluation: across multiple serverless workloads

1. **Online Boutique from Google [1]**
   - **Intense** web traffic
   - 10 functions
   - 6 different sequences of function chains

2. **Motion detection [2]**
   - **Intermittent** IoT traffic (a burst every few seconds)
   - 2 simple functions

3. **Parking: image detection & charging [3]**
   - **Intermittent & Periodic** IoT traffic (once every 240 seconds)
   - 5 functions with heterogeneous CPU service time for each (from 1ms to 435ms)

---

Performance with Online Boutique

S-SPRIGHT vs. D-SPRIGHT vs. Knative mode vs. gRPC mode (no sidecar)

**Throughput:**
- Both D- and S-SPRIGHT maintain a stable RPS of $\sim 5500$ req/sec $\Rightarrow (6 \times$ more than Knative)
- Without sidecar and front-end proxy, gRPC is slightly better than Knative, but its throughput is much lower than SPRIGHT ($5 \times$ lower)

**Latency:**
- Knative shows clear overload behavior: high tail latency
- SPRIGHT’s shared memory processing reduces communication overhead within function chains, achieving lower latency than Knative and gRPC, even at much higher traffic load

*Note:* for simplicity, we only show the latency results of a representative function chain in online boutique. More results can be found in the paper.
Performance with Online Boutique

S-SPRIGHT vs. D-SPRIGHT vs. Knative mode vs. gRPC mode (no sidecar)

**Resource efficiency:**

- entire Knative setup (including the gateway and queue proxies, which are constantly running): \( \sim 26 \) CPU cores (out of 40)
- Entire gRPC setup (only functions, no gateway and sidecars): \( \sim 36 \) CPU cores (out of 40)
- D-SPRIGHT consumes 11 cores (one core for Gateway, 10 cores for functions)
- S-SPRIGHT consumes in total only \( \sim 3 \) CPU cores
Performance with Online Boutique

S-SPRIGHT vs. D-SPRIGHT vs. Knative mode vs. gRPC mode (no sidecar)

Resource efficiency:

- entire Knative setup (including the gateway and queue proxies, which are constantly running): $\sim$26 CPU cores (out of 40)
- Entire gRPC setup (only functions, no gateway and sidecars): $\sim$36 CPU cores (out of 40)
- D-SPRIGHT consumes **11 cores** (one core for Gateway, 10 cores for functions)
- S-SPRIGHT consumes in total only $\sim$3 CPU cores

The event-driven socket message mechanism makes our design’s shared memory processing more resource-efficient
Evaluation: across multiple serverless workloads

1. Online Boutique from Google [1]
   - Intense web traffic
   - 10 functions
   - 6 different sequences of function chains

2. Motion detection [2]
   - Intermittent IoT traffic (a burst every few seconds)
   - 2 simple functions

   - Intermittent & Periodic IoT traffic (once every 240 seconds)
   - 5 functions with heterogeneous CPU service time for each (from 1ms to 435ms)

---

Our event-driven design sidesteps the need for cold start by keeping functions warm at minimum cost.

Our design’s event-driven features make it more efficient even if we keep functions warm compared to ‘pre-warming’ Knative functions.
Shared memory considered harmful?

Our Solution: Security domain

- **Trust model**: functions within a chain trust each other, functions in different chains may not
- SPRIGHT constructs a security domain for each function chain with:
  - a private shared memory pool for each chain
    - DPDK’s multi-process support
    - Use different *shared data file prefix* to separate memory pool
  - inter-function packet descriptor filtering with the SPROXY
    - Use SPROXY to construct packet descriptor filtering between functions
    - Restrict unauthorized access to the shared memory
Conclusion

SPRIGHT: event-driven + shared memory processing = load-proportional, high-performance

Using shared memory processing to optimize the data pipeline of current serverless function chains

Using event-driven processing to improve resource efficiency of current serverless function design

• When serving an intense web workload (online boutique):
  • 6x throughput improvement, 70x tail latency reduction and 30x CPU usage savings over Knative
• When serving intermittent IoT workload (motion detection; parking):
  • Better than Knative even with ‘pre-warmed’ functions; side-stepping the ‘cold-start’

Find SPRIGHT at: https://github.com/ucr-serverless/spright