Introduction

- Serverless computing frameworks like AWS Lambda enable users to launch thousands of short-lived tasks with *high elasticity* and *fine-grain resource billing*

- This makes serverless computing appealing for *interactive analytics*
Introduction

- Serverless computing frameworks like AWS Lambda enable users to launch thousands of short-lived tasks with *high elasticity* and *fine-grain resource billing*
- This makes serverless computing appealing for *interactive analytics*
- **The challenge**: tasks (‘lambdas’) need an efficient way to communicate intermediate results

*ephemeral data*
Motivation

- Direct communication between tasks is difficult:
  - Lambdas are short-lived and stateless
  - Users have no control over lambda scheduling
- The natural approach is to share data through a common data store
Motivation

- Direct communication between tasks is difficult:
  - Lambdas are short-lived and stateless
  - Users have no control over lambda scheduling
- The natural approach is to share data through a common data store
- Existing storage systems are not a good fit for ephemeral data: they are designed for long-term storage and/or do not automatically rightsize resources to meet elastic I/O demands at low cost
Motivation

Rightsizing resource allocations to navigate the performance-cost trade-off space is difficult, even for a single application.

Minimum cost resource allocation where application is not bottlenecked on I/O.
Pocket

- An elastic, distributed data store for ephemeral data sharing in serverless analytics

- Key properties:
  - High performance for a wide range of object sizes
  - Automatic, fine-grain scaling to rightsize resource allocation
  - Low cost, pay-what-you-use
System architecture

Controller
monitor & scale cluster

Metadata server(s)
route requests

Storage server

CPU
Net
DRAM
Flash

Storage server

CPU
Net
DRAM
Flash

Storage server

CPU
Net
DRAM
Flash

Storage server

CPU
Net
DRAM
Flash
System architecture

*Job A*
- λ λ λ λ λ λ λ λ λ
- λ λ λ λ λ λ λ λ

*Job B*
- λ λ λ λ λ λ

*Job C*
- λ λ λ λ λ λ λ λ λ λ λ λ

**Controller**
*monitor & scale cluster*

**Metadata server(s)**
*route requests*

**Register job**

**Storage server**
- 0

**Storage server**
- 0.8

**Storage server**
- 0

**Storage server**
- 1
System architecture

**Job A**

**Job B**

**Job C**

Controller  
*monitor & scale cluster*

PUT ‘x’  
1.  
2.  
3.

Metadata server(s)  
*route requests*
System architecture

Job A

Job B

Job C

Controller

monitor & scale cluster

Metadata server(s)

route requests

PUT 'x'

1. 2. 3.

iii. De-register job

The user API supports hints for persisting data or deleting after reading

Storage server 0

Storage server 0.8

Storage server 0

Storage server 1

CPU

Net

DRAM

Flash

CPU

Net

DRAM

Flash

CPU

Net

DRAM

Flash

CPU

Net

DRAM

Flash
Rightsizing for Application Requirements

1. Pocket leverages user hints to decide resource requirements along 3 dimensions

<table>
<thead>
<tr>
<th>Hint</th>
<th>Impact on Throughput</th>
<th>Impact on Capacity</th>
<th>Impact on Tier Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>No knowledge</td>
<td>Use default cluster</td>
<td>Use default cluster</td>
<td>Fill DRAM first, then Flash</td>
</tr>
</tbody>
</table>
Rightsizing for Application Requirements

1. Pocket leverages user hints to decide resource requirements along 3 dimensions

<table>
<thead>
<tr>
<th>Hint</th>
<th>Impact on Throughput</th>
<th>Impact on Capacity</th>
<th>Impact on Tier Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>No knowledge</td>
<td>Use default cluster</td>
<td>Use default cluster</td>
<td>Fill DRAM first, then Flash</td>
</tr>
<tr>
<td>Latency sensitivity</td>
<td>-</td>
<td>-</td>
<td>Flash if not latency sensitive</td>
</tr>
<tr>
<td>Max # Lambdas</td>
<td>Provision based on per-Lambda BW limit</td>
<td>Provision based on per-Lambda BW limit</td>
<td>-</td>
</tr>
<tr>
<td>Capacity C</td>
<td>-</td>
<td>Provision capacity C</td>
<td>-</td>
</tr>
<tr>
<td>Throughput $T$</td>
<td>Provision throughput $T$</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Rightsizing for Application Requirements

1. Pocket leverages user hints to decide resource requirements along 3 dimensions
2. Use online bin-packing algorithm to assign resources (represented as a *weightmask*)

<table>
<thead>
<tr>
<th>Hint</th>
<th>Impact on Throughput</th>
<th>Impact on Capacity</th>
<th>Impact on Tier Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>No knowledge</td>
<td>Use default cluster</td>
<td>Use default cluster</td>
<td>Fill DRAM first, then Flash</td>
</tr>
<tr>
<td>Latency sensitivity</td>
<td>-</td>
<td>-</td>
<td>Flash if not latency sensitive</td>
</tr>
<tr>
<td>Max # Lambdas</td>
<td>Provision based on per-Lambda BW limit</td>
<td>Provision based on per-Lambda BW limit</td>
<td>-</td>
</tr>
<tr>
<td>Capacity C</td>
<td>-</td>
<td>Provision capacity C</td>
<td>-</td>
</tr>
<tr>
<td>Throughput $T$</td>
<td>Provision throughput $T$</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Rightsizing Cluster Resources

- Pocket’s controller monitors overall cluster usage (storage capacity, CPU, and network bandwidth)
  - Scale up: if any resource type is overutilized (i.e. >80%)
  - Scale down: if all resource types are underutilized (i.e. <60%)

- Load balance with data steering:
  - Use weightmask to steer data onto new nodes and away from nodes preparing to leave the cluster
  - Ephemeral data soon expires and is garbage collected, no data migration
Implementation

- Pocket’s storage and metadata server implementation is based on the **Apache Crail** distributed storage system
- We use **ReFlex** for the Flash storage tier
- Pocket runs the storage and metadata servers in containers, orchestrated using **Kubernetes**
Evaluation

- We deploy Pocket on Amazon Web Services (AWS) EC2

<table>
<thead>
<tr>
<th>Component</th>
<th>Instance Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pocket Controller / Metadata server</td>
<td>m5.xlarge</td>
</tr>
<tr>
<td>DRAM server</td>
<td>r4.2xlarge</td>
</tr>
<tr>
<td>NVMe Flash server</td>
<td>i3.2xlarge</td>
</tr>
</tbody>
</table>

- Serverless applications on AWS Lambda:
  - MapReduce sort, video analytics, parallel software build
Latency and Throughput

1KB request latency from 1 lambda

- **S3**: 25819 (PUT), 12102 (GET)
- **Redis**: 232 (PUT), 230 (GET)
- **Pocket-DRAM**: 508 (PUT), 414 (GET)
- **Pocket-Flash**: 572 (PUT), 466 (GET)

Low latency

1MB request throughput from 100 lambdas

- **S3**: 1063 (PUT), 1275 (GET)
- **Redis**: 698 (PUT), 1102 (GET)
- **Pocket-DRAM (per core)**: 968 (PUT), 1007 (GET)
- **Pocket-Flash (per core)**: 996 (PUT), 1016 (GET)

High throughput
End-to-end Performance for 100GB sort

Pocket achieves similar I/O performance to Redis at lower cost (Flash vs. DRAM)
Leveraging Hints for Resource Allocation

Provision based on per-\(\lambda\) throughput limit

Use Flash if app is not latency sensitive

Provision only the GB and GB/s the app needs
Rightsizing with Multiple Jobs
Conclusion

- Pocket is a distributed data store that enables data sharing for serverless analytics with:
  - High performance
  - Automatic, fine-grain scaling
  - Cost effective resource allocation & data placement across multiple storage tiers