Elastic Ephemeral Storage for Serverless Analytics

Ana Klimovic*, Yawen Wang*, Patrick Stuedi+, Animesh Trivedi+, Jonas Pfefferle+, Christos Kozyrakis*

*Stanford University, +IBM Research

OSDI 2018
Serverless Computing

- Serverless computing enables users to launch short-lived tasks with high elasticity and fine-grain resource billing
Serverless Computing

- Serverless computing enables users to launch short-lived tasks with high elasticity and fine-grain resource billing.
- Serverless computing is increasingly used for interactive analytics.

- PyWren (SoCC’17)
- ExCamera (NSDI’17)
- gg: The Stanford Builder
- Amazon Aurora Serverless
Serverless Computing

- Serverless computing enables users to launch short-lived tasks with high elasticity and fine-grain resource billing
- Serverless computing is increasingly used for interactive analytics
  - Exploit massive parallelism with large number of serverless tasks

User query & input data → Result
The Challenge: Data Sharing

- Analytics jobs involve multiple stages of execution.
- Serverless tasks need an efficient way to communicate *intermediate data* between different stages of execution.

*ephemeral data*

User query & input data → Result
In traditional analytics…

- Ephemeral data is exchanged directly between tasks

mapper\textsubscript{0} \hspace{1cm} reducer\textsubscript{0}
mapper\textsubscript{1} \hspace{1cm} reducer\textsubscript{1}
mapper\textsubscript{2}
mapper\textsubscript{3}
In traditional analytics...

- Ephemeral data is exchanged directly between tasks
In serverless analytics...

- Direct communication between serverless tasks is difficult:
  - Tasks are short-lived and stateless

mapper_0
mapper_1  mapper_2  mapper_3
?

reducer_0  reducer_1
In serverless analytics…

- The natural approach for sharing ephemeral data is through a common data store
In serverless analytics…

- The natural approach for sharing ephemeral data is through a common data store

mapper_0
mapper_1
mapper_2
mapper_3

reducer_0
reducer_1
Requirements for Ephemeral Storage

1. High performance for a wide range of object sizes
2. Cost efficiency, i.e., fine-grain, pay-what-you-use resource billing

Requirements for Ephemeral Storage

1. High performance for a wide range of object sizes
2. Cost efficiency, i.e., fine-grain, pay-what-you-use resource billing
   • Example of performance-cost tradeoff for a serverless video analytics job with different ephemeral data store configurations

Finding the Pareto optimal resource allocation is non-trivial…and gets harder with multiple jobs.
Requirements for Ephemeral Storage

1. High performance for a wide range of object sizes
2. Cost efficiency, i.e., fine-grain, pay-what-you-use resource billing
3. Fault-tolerance

Existing cloud storage systems do not meet the elasticity, performance and cost demands of serverless analytics jobs.
Pocket

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

- Pocket achieves high performance and cost efficiency by:
  - Leveraging multiple storage technologies
  - Rightsizing resource allocations for applications
  - Autoscaling storage resources in the cluster based on usage

- Pocket achieves similar performance to Redis, an in-memory key value store, while saving ~60% in cost for various serverless analytics jobs
Pocket Design

Controller
app-driven resource allocation & scaling

Metadata server(s)
request routing

Storage server

CPU
Net
HDD

Storage server

CPU
Net
Flash

Storage server

CPU
Net
DRAM

Storage server

CPU
Net
DRAM
Using Pocket

**Controller**

*app-driven resource allocation & scaling*

**Storage server**

- CPU
- Net
- HDD

**Storage server**

- CPU
- Net
- Flash

**Storage server**

- CPU
- Net
- DRAM

**Storage server**

- CPU
- Net
- DRAM

**Metadata server(s)**

*request routing*

**Job A**

- λ λ λ λ λ λ λ
- λ λ λ λ λ λ λ

**Job B**

- λ λ λ λ λ
- λ λ λ λ

**Job C**

- i. Register job
- ii. Allocate & assign resources for job
Using Pocket

 iii. Deregister job

 GET/PUT API accepts hints about job attributes and data lifetime

 Metadata server(s)
 request routing

 Controller
 app-driven resource allocation & scaling

 Job A
 λ λ λ λ λ λ λ
 λ λ λ λ λ λ λ

 Job B
 λ λ λ λ λ λ λ

 Job C
 λ λ λ λ λ λ λ λ
 λ λ λ λ λ λ λ λ
 λ λ λ λ λ λ λ λ λ λ
 λ λ λ λ λ λ λ λ λ λ λ λ

 Storage server
 CPU
 Net
 HDD

 Storage server
 CPU
 Net
 Flash

 Storage server
 CPU
 Net
 DRAM

 Storage server
 CPU
 Net
 DRAM
Assigning Resources to Jobs

Optional hints about job:
- Latency sensitivity
- Maximum # of concurrent tasks
- Total ephemeral data capacity
- Peak aggregate bandwidth required

Controller
app-driven resource allocation & scaling

Storage server
- CPU
- Net
- HDD

Storage server
- CPU
- Net
- Flash

Storage server
- CPU
- Net
- DRAM

Storage server
- CPU
- Net
- DRAM

1. Throughput allocation
2. Capacity allocation
3. Choice of storage tier(s)
Assigning Resources to Jobs

1. Throughput allocation
2. Capacity allocation
3. Choice of storage tier(s)

Controller
app-driven resource allocation & scaling

i. Register job

ii. Allocate & assign resources for job

Job A

Allocate & assign resources for job

Job Weight Map

online bin-packing algorithm

Job A:

Server C → 0.4
Server D → 0.6

Job B:

Server A → 0.2
Server B → 0.3
Server C → 0.5

Storage server A

CPU

Net

HDD

Storage server B

CPU

Net

Flash

Storage server C

CPU

Net

DRAM

Storage server D

CPU

Net

DRAM
Autoscaling the Pocket Cluster

- **Goal**: scale cluster resources dynamically based on resource usage

- **Mechanisms**:
  - Monitor CPU, network bandwidth, and storage capacity utilization
  - Add/remove storage & metadata nodes to keep utilization within range
  - Steer data for incoming jobs to active nodes
  - Drain inactive nodes as jobs terminate

- **Avoid migrating data**
Implementation

- Pocket’s metadata and storage server implementation is based on the Apache Crail distributed storage system [1]
- We use ReFlex for the Flash storage tier [2]
- Pocket runs the storage and metadata servers in containers, orchestrated using Kubernetes [3]

Pocket Evaluation

- We deploy Pocket on Amazon EC2

<table>
<thead>
<tr>
<th>Component</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>m5.xlarge</td>
</tr>
<tr>
<td>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>
<tr>
<td>SATA/SAS SSD server</td>
<td>i2.2xlarge</td>
</tr>
<tr>
<td>HDD server</td>
<td>h1.2xlarge</td>
</tr>
</tbody>
</table>

- We use AWS Lambda as our serverless platform
- **Applications**: MapReduce sort, video analytics, distributed compilation
Application Performance with Pocket

- Compare Pocket to S3 and Redis, which are commonly used today.

<table>
<thead>
<tr>
<th>MapReduce sort job hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ephemeral capacity</td>
</tr>
<tr>
<td>Latency sensitive</td>
</tr>
<tr>
<td>Aggregate peak throughput</td>
</tr>
</tbody>
</table>

S3 does not provide sufficient throughput.
Application Performance with Pocket

- Compare Pocket to S3 and Redis, which are commonly used today.

Pocket achieves similar performance to Redis but uses NVMe Flash.

MapReduce sort job hints:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ephemeral capacity</td>
<td>100 GB</td>
</tr>
<tr>
<td>Latency sensitive</td>
<td>False</td>
</tr>
<tr>
<td>Aggregate peak throughput</td>
<td>100 Gb/s</td>
</tr>
</tbody>
</table>
Pocket leverages job attribute hints for cost-effective resource allocation and amortizes VM costs across multiple jobs, offering a pay-what-you-use model.

Pocket reduces cost by ~60% compared to Redis for all 3 jobs.
Autoscaling the Pocket Cluster

Job hints

<table>
<thead>
<tr>
<th>Job hints</th>
<th>Job1: Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency sensitive</td>
<td>False</td>
</tr>
<tr>
<td>Ephemeral data capacity</td>
<td>10 GB</td>
</tr>
<tr>
<td>Aggregate throughput</td>
<td>3 GB/s</td>
</tr>
</tbody>
</table>
Autoscaling the Pocket Cluster

The controller elastically scales resources to meet the requirements of multiple jobs.

### Job hints

<table>
<thead>
<tr>
<th></th>
<th>Job1: Sort</th>
<th>Job2: Video analytics</th>
<th>Job3: Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency sensitive</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Ephemeral data capacity</td>
<td>10 GB</td>
<td>6 GB</td>
<td>10 GB</td>
</tr>
<tr>
<td>Aggregate throughput</td>
<td>3 GB/s</td>
<td>2.5 GB/s</td>
<td>3 GB/s</td>
</tr>
</tbody>
</table>
Conclusion

- Pocket is a distributed ephemeral storage system that:
  - Leverages multiple storage technologies
  - Rightsizes resource allocations for applications
  - Autoscales storage cluster resources based on usage

- We designed Pocket for ephemeral data sharing in serverless analytics. More generally, Pocket is an elastic, distributed `/tmp`.

[www.github.com/stanford-mast/pocket](http://www.github.com/stanford-mast/pocket)