Serverless Computing
Problems and Solutions

Yiying Zhang
Outline

- Managing states: Pocket
- Resource efficiency: Scad
- Discussion on Serverless Computing
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 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
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
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

Job A
λ λ λ λ λ λ λ λ λ λ

Job B
λ λ λ λ λ λ λ

Job C
i. Register job

Metadata server(s)
request routing

ii. Allocate & assign resources for job

Storage server
CPU
Net
HDD

Storage server
CPU
Net
Flash

Storage server
CPU
Net
DRAM

Storage server
CPU
Net
DRAM
Using Pocket

iii. Deregister job

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

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 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

GET/PUT API accepts hints about job attributes and data lifetime
Assigning Resources to Jobs

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

1. Register job

Controller
app-driven resource allocation & scaling

Metadata server(s)
request routing

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

Controller
app-driven resource allocation & scaling

i. Register job

ii. Allocate & assign resources for job

Job A
λ λ λ λ λ λ

Job Weight Map

metadata server(s)
request routing

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

online bin-packing algorithm

Storage server A
CPU
Net
HDD

Storage server B
CPU
Net
Flash

Storage server C
CPU
Net
DRAM

Storage server D
CPU
Net
DRAM

1. Job A:
   - Server C → 0.4
   - Server D → 0.6

2. Job B:
   - Server A → 0.2
   - Server B → 0.3
   - Server C → 0.5
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**
Application Performance with Pocket

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

S3 does not provide sufficient throughput.

<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>
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:
- Ephemeral capacity: 100 GB
- Latency sensitive: False
- Aggregate peak throughput: 100 Gb/s
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)
What about resources?
Problem 1: Resource Limits

- Execution time limits
- Per-function memory and CPU limits
- Maximum parallel instances
- Exceeding limits would result to run-time failure

- Solution: increase cap?
Problem 2: Lack resource elasticity

• Most FaaS offerings have fixed CPU:memory ratio

• and use one function size for all invocations and the entire duration of an invocation
Problem 3: User burden

• Programmers need to manually “split” their applications into smaller sizes to fit function limits

• Users need to (statically) decide the size of each function
Root Cause?

- **Function** as a Service
  - Functions are fix-sized boxes
  - that are pre-defined by cloud providers
Solution?

- **Resource** as a Service (RaaS), or **Resource-Centric Serverless Computing**
  - executing user applications according to their resource features

Today's FaaS-based Serverless Computing

Proposed RaaS-based Serverless Computing

Manually Written Function DAG

Manually Decided Function Size

Fixed Functions:
- Wastes Resources
- Potential Failure

Resource-based Execution:
- Tight Resource Packing
- No Execution Limit

Automatically Generated Resource Graph

Automatic Sizing

User Program
Resource Graph

- **Node:**
  - a compute (code piece) or data (objects) unit
  - of a certain resource feature

- **Edge**
  - Directed: triggering relationship
  - Undirected: communicating (accessing) relationship
User Annotation and Compiler Support

• Users annotate where to split code and what data structures to separate out

• Scad compiler generates code pieces and JSON representing resource graph

```
import numpy as np

def main():
    @remote
    a = np.array(n)
    @split
    for i in range(n):
        sum += a[i]

app.py
```

```
...
'comp1': {
    'type': compute,
    'code': app_1.py
    'trigger': 'comp2'
    'comm': 'mem1'
},
'mem1': {
    'type': memory
    'comm': [comp1, comp2]
},
'comp2': {
    'type': compute,
    'code': app_2.py
    'comm': 'mem1'
}
app.json
```

```
from scad import rarray
def main():
    a = rarray(Channel('mem1'), size=n)
    return a.meta()

app_1.py
```

```
from remoteArrayLib import rarray
def main(meta):
    a = rarray(Channel('mem1'), meta)
    for i in range(n):
        sum += a.get(i)

app_2.py
```

```
from scadRaw import read, write
class rarray():
    def get(self, i):
        return read(self.channel,
                    self.get_remote_offset(i),
                    self.dtype.size)
remoteArrayLib/rarray.py
```
How to Execute Resource Graphs?

- A node in a resource graph can be of arbitrary size and execution time
- Hard to pack them tightly on servers (multi-dimensional bin-packing)
Hardware Resource Disaggregation:

Breaking monolithic servers into distributed, network-attached hardware components
Executing Resource Graph

Virtual
- Comp1
- Comp2
- Mem1
- Mem2

Materialize

Physical
- Comp1-Phy1
- Comp1-Phy2
- Comp2-Phy1
- Mem1-Phy1
- Mem2-Phy1
- Mem1-Phy2
- Mem2-Phy2

Compute Pool
- Comp1-Phy1
- Comp1-Phy2

Memory Pool
- Mem1-Phy1
- Mem2-Phy1
- Mem1-Phy2
- Mem2-Phy2
Whole Flow

Step 1: Annotated User Programs (High-Level Interface)

Step 2: Generated Component Graph (Low-Level Interface)

Step 3: Materialization

Step 3: Aggregation

Step 4: Disaggregation
System Architecture

- Global Scheduler
- Rack Scheduler
- Reliable Messaging
- Triggering Events
- User Code Metadata Telemetry
- Global Monitor

Nodes:
- Node 1: Compute Component, Language Runtime, Scad Runtime, Sandbox, Network Virtualization, Compute Controller, Executor
- Node 2: Compute Comp1, Mem Comp, Compute Comp2, Language Runtime, Scad Runtime, Sandbox, Net Virt, Memory Controller, Compute Controller, Executor
- Node 3: Phys Mem, RDMA Region, Network Virtualization, Executor, Memory Controller, Sandbox
- Node 4: Phys Mem, RDMA Region, Memory Controller, Sandbox

Components:
- Physical Element
- RDMA Network
- Virtual Memory Component
- Executor
- Memory Controller
- Compute Controller
Results
Discussion

• Final thoughts on serverless?