Serverless Computing
Overview

Yiying Zhang
Outline

- Pocket
- Some real application examples of using serverless
- Issues of today’s serverless computing and potential mitigations
- Breakout room 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.
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  \[\Rightarrow\]  Result
In traditional analytics…

- Ephemeral data is exchanged directly between tasks

mapper₀ mapper₁ mapper₂ mapper₃

reducer₀ reducer₁
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

mapper₀ mapper₁ mapper₂ mapper₃ ? reducer₀ reducer₁
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

Job A
λ λ λ λ λ λ λ
λ λ λ λ λ λ λ

Job B
λ λ λ λ λ

Job C

Controller
app-driven resource allocation & scaling

i. Register job

ii. Allocate & assign resources for job

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

Job A

Job B

Job C

iii. Deregister job

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

PUT ‘x’
Assigning Resources to Jobs

Controller
- app-driven resource allocation & scaling

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

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

i. Register job

Storage server

CPU
Net
HDD

Storage server

CPU
Net
Flash

Storage server

CPU
Net
DRAM

Storage server

CPU
Net
DRAM
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 Weight Map

Metadata server(s)
request routing

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>Instance Type</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.

![Graph showing average time per Lambda (s) for S3, Redis, and Pocket.]

- S3 does not provide sufficient throughput.

<table>
<thead>
<tr>
<th>MapReduce sort job hints</th>
<th></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>
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

<table>
<thead>
<tr>
<th>MapReduce sort job hints</th>
<th></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>
Application Storage Cost with Pocket

- 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.

<table>
<thead>
<tr>
<th>Job hints</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
ExCamera: Video encoding in real-time

"Apply this awesome filter to my video."

(slide from https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi)
Video Encoding/Compression

Exploit the temporal redundancy in adjacent images.

- Store the first image on its entirety: a key frame.
- For other images, only store a "diff" with nearby images: an interframe.

4K video @15Mbps: key frame ~1 MB; an interframe is ~25 KB
Video Encoding/Compression

Key idea:
- split video in very small chunks
- encode chunks in parallel
- stitch the encoded chunks
<table>
<thead>
<tr>
<th>Video Quality and Encoding</th>
<th>Time (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpxenc Single-Threaded</td>
<td>453</td>
</tr>
<tr>
<td>vpxenc Multi-Threaded</td>
<td>149</td>
</tr>
<tr>
<td>YouTube (H.264)</td>
<td>37</td>
</tr>
<tr>
<td>ExCamera[6, 16]</td>
<td>2.6</td>
</tr>
</tbody>
</table>

(slid from https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi)
| Real-time video compression (ExCamera) | On-the-fly video encoding | Object store too slow to support fine-grained communication; functions too coarse grained for tasks. | Function-to-function communication to avoid object store; a function executes more than one task. | 60x faster, 6x cheaper versus VM instances. |
MapReduce (an analytic framework)

| MapReduce | Big data processing (Sort 100TB) | Shuffle doesn’t scale due to object stores latency and IOPS limits | Small storage with low-latency, high IOPS to speed-up shuffle. | Sorted 100 TB 1% faster than VM instances, costs 15% more. |
| Databases (Serverless SQLite) | Primary state for applications (OLTP) | Lack of shared memory, object store has high latency, lack of support for inbound connectivity. | Shared file system can work if write needs are low. | 3x higher cost per transaction than published TPC-C benchmarks. Reads scale to match but writes do not. |
What’s missing?

- No Inter-function communication
  - No State Sharing (a.k.a “Hey I am done with computation”)
  - No Data Transferring (Mapper send data to Reducer)
  - All distributed task that requires cooperation won’t work well (MapReduce, Distributed Model Training, ..)
How is shuffle a challenge?

1. Naming the functions can already be a trouble. They are ephemeral and can’t be self-identified.

2. Way more message need to transfer.

\[(\text{# of VM})^2 \gg \text{(# of VM \times # of task)}^2\]

\[\Rightarrow (\text{# of task})^2\text{ message need to be sent!}\]

The number of message can range from 10 to 100, which yields a magnitude of 100 to 10000 more message for same task. Woah.
Problems exposed in this experiment

1. Communication is tough, and it is slow.
Problems exposed in this experiment

2. Even with third party messaging service, like SQS, the communication is slow. And could be very expensive.

3. Task that require frequent access to storage service is a nightmare. It’s an inherited problem of stateless computation.
What’s missing?

- **No good way to manage states**
  - Slow to access cloud storage (e.g., S3) for storing states
  - Active research area
  - [Pocket: Elastic Ephemeral Storage for Serverless Analytics](#)
What’s missing?

- **Inflexible Resource Configuration**
  - Sometimes we can’t even fit a machine learning model into a serverless instance.

*Quote from AWS Lambda Docs: “Choose an amount between 128 MB and 3,008 MB in 64-MB increments. Lambda allocates CPU power linearly in proportion to the amount of memory”*
Generally unpredictable performance

- **Lower startup latency comparing to VM-based instances. No booting up OS is needed.**
  
  But the slow start up of VM-based instance can be amortized in the long run.

- **In cloud function, the cold start latency includes:**
  
  (1) Start a cloud function
  
  (2) Initialize env such as loading the libraries,
  
  (3) Application-specific initialization in the user code, such as pull the model from storage server, initialize the entire logging system and message system, etc.
  
  (2) and (3) can simply dwarf the performance of (1)!

And you simply **lose the control of the type of hardware** you got allocated. You do not have guarantee of hardware as you had in EC2.
Security challenges

1. Scheduling randomization and physical isolation.
   a. Hardware level side-channel / Rowhammer attacks inside the cloud
   b. Solution: Ephemeral functions already mitigate this problem.

2. Fine-grained security context.
   a. Sensitive information during function execution.
   b. Solution: Use capability-based access control to cooperate with other functions.
   c. Still hard for things such as non-equivocation and revocation of capability.

3. Oblivious serverless computing
   a. Functions might leak access pattern and timing information through communication.
   b. Solution: Use oblivious algorithms. But has high-overhead.
   c. **Oblivious algo**: An algorithm whose behavior, by design, is independent of some property that influences a typical algorithm for the same problem.
Other limitation

- No fine-grained billing in respect to resource used. Still billed by usage time.

- Limited runtime. Up to 900 seconds.
What’s suitable and what’s not

Serverless is **good** for:
- short-running
- stateless
- event-driven

  - Microservices
  - Mobile Backends
  - Bots, ML Inferencing
  - IoT
  - Modest Stream Processing
  - Service integration

Serverless is **not good** for:
- long-running
- stateful
- number crunching

  - Databases
  - Deep Learning Training
  - Heavy-Duty Stream Analytics
  - Spark/Hadoop Analytics
  - Numerical Simulation
  - Video Streaming
Fallacies and Pitfalls

Using AWS lambda is more expensive than AWS EC2.

- Not quite. You will save operational expense. You will also be charged by resources used instead of reservation time for instances.

Serverless computing can have unpredictable costs.

- Bucket based pricing like your mobile phone data plan.

It’s easy to port among serverless computing providers.

- Not really. Since you are also using BaaS service, it creates stickiness of service. And it cause vendor lock-in.

Cloud functions cannot handle low latency application needing predictable performance.

- We can pre-warm the cloud functions and shorten the startup time. More predictable.
Open questions: Legacy Code?

The economical value of existing code represents a huge investment of countless hours of developers coding and fixing software.

One of the most important problems may be to what degree existing legacy code can be automatically or semi-automatically decomposed into smaller-granularity pieces to take advantage of these new economics?
Open questions: New Programming Models?

- Tools
- Deployment
- Monitoring and debugging
- Short-lived functions, scaling to large invocations,
- Looking for problems is like finding needles in ever growing haystack?
- Serverless IDEs?
- Decompose micro-service into FaaS?
Thoughts?

The authors claim serverless will skyrocket and bring EC2 type of thing into history.