OFC: An Opportunistic Caching System for FaaS Platforms

Djob Mvondo
Mathieu Bacou
Kevin Nguetchouang, Lucien Ngale
Stéphane Pouget
Josiane Kouam
Renaud Lachaize
Jinho Hwang
Tim Wood
Daniel Hagimont
Noël De Palma
Bernabé Batchakui
Alain Tchana

Univ. Grenoble Alpes, ENS Lyon
Télécom SudParis, IP Paris
ENSP Yaoundé
ENS Lyon
Inria
Univ. Grenoble Alpes
Facebook
The George Washington University
University of Toulouse
Univ. Grenoble Alpes
ENSP Yaoundé
ENS Lyon, Inria
Context: Function-as-a-Service

- Cloud-native applications
  - Built as collections of (chains of) functions
  - Rely on platform-provided back-end servers (serverless)
  - Mostly stateless by design
Extract-Transform-Load pattern

1. Extract (E) data from remote persistent storage (object store...)
2. Transform (T) by performing some computation (blur image...)
3. Load (L) result to remote persistent storage

Function-as-a-Service architecture in a serverless cloud.

Performance issue: latency

- Storage access is a big issue with ETL
- Problem of data locality
  - Out-of-infrastructure remote storage
  - Even worse for pipelines

Faas performance issues in latency of function invocation, and concerns of our work.
Related work

Caching, caching, and caching …

- Cloudburst\textsuperscript{a}
- Infinicache\textsuperscript{b}
- Pocket\textsuperscript{c}
- ....

Existing works either require function modification or extra-resources (memory) to provision the cache layer

Solution: caching in the FaaS age

- Avoid remote storage with in-memory caching
- FaaS characteristics: very short latency, very elastic
- New challenges in the FaaS context:
  - How to provision memory for the cache?
  - How to make caching scale?
  - How to provide caching to functions?
OFC: Opportunistic FaaS Cache

Opportunistic

Function-as-a-Service

Cache

The three pillars of OFC.
OFC: Opportunistic FaaS Cache

Model Trainer

Controller

predictor

RC Coor

proxy

rcLib

RC M

Invoker

Sizer

cacheAgent

proxy

rcLib

RC M

Invoker

Sizer

cacheAgent

func-monitor

func-monitor

Base FaaS platform

Caching system

Memory reuse (Machine learning)
Unused reserved memory

1. Over-provisioning by tenants to absorb workload variation:
   - 50% of functions reserve ≥512MB
   - 50% of functions use ≤29MB

2. Keep-alive policy: keep functions warm to reduce latency:
   - 81% invoked once per min. or less
   - Functions kept warm 10~20min (OpenWhisk, AWS Lambda)

Timeline of a function sandbox illustrating wasted memory.

Predicting wasted memory

- How much memory is available to the cache?
  - Complex relation with data, parameters

- Use machine learning!
  - White-box functions
    - Parameters, inputs...
  - High invocation rate
    - Quick dataset gathering

![Graph showing relation between memory usage and function invocation parameters and input.](image-url)
Learning memory usage, and more

- **Constraints of the FaaS:**
  - Learn and update models
    - Maintain training dataset
  - Learn from unknown features: bounds, sets of values?
    - Cannot compute from features
  - **Prediction speed:** on the critical path of the invocation
    - Predict in less than 1ms

- **Classification instead of regression**
  - Predict among 16MB intervals

- **Decision trees:** J48 (C4.5)
  - 92.7% accuracy for exact-or-over predictions
  - Model *accurate enough* for 95% of functions in less than 8h of lifetime
  - 13x faster at 99% than RandomForest
    - While being just as accurate

- **ML also used to predict caching benefits**
  - Keep only useful data in cache
OFC: Opportunistic FaaS Cache

- Opportunistic
- Function-as-a-Service
- Cache

The three pillars of OFC.
OFC caching mechanisms overview

- OFC leverages RAMCloud\(^a\)
  - Distributed
  - In-memory
  - Fault tolerant

- RAMCloud can store objects up to 8MB. We updated this to 10MB.

---

OFC caching mechanisms overview

- On each invoker node:
  - **RC M**: RAMCloud cache master
  - **CacheAgent**: cache autoscaling
    - Scale the cache memory up/down
    - Monitor the cache pressure
    - Perform Garbage Collection
The three pillars of OFC:

- **Opportunistic**
- **Function-as-a-Service**
- **Cache**

Unused reserved memory is gathered into an in-memory scalable system, which is a transparent efficient reliable cache.
OFC caching mechanisms overview

- A **proxy** transparently intercepts function calls to storage nodes.
  - Runtime interception
  - Routes request to cache API (**rcLib**)
OFC caching mechanisms overview

- RAMCloud library rcLib:
  - Persist data on the local cache
  - Ensure consistency with remote storage

- To ensure consistency with OFC, on storage node, a webhook checks for queries the cache for incoming read requests.

Data persistence and consistency with remote storage.
OFC evaluation results

Does OFC improve serverless functions latencies?

- Single functions
- Multi-stage functions

Five scenarios

1) Redis
2) OFC Local Hit (LH)
3) OFC Remote Hit
4) Miss (M)
5) Default (Swift)

<table>
<thead>
<tr>
<th>Memory</th>
<th>512 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Ubuntu 16.04.7 LTS</td>
</tr>
<tr>
<td>CPUs</td>
<td>2 Intel Xeon E5-2698v4 CPUs (20 cores/CPU)</td>
</tr>
<tr>
<td>Disk</td>
<td>480 GB SSD</td>
</tr>
<tr>
<td>Network</td>
<td>Intel Ethernet 10G 2P X520 Adapter</td>
</tr>
</tbody>
</table>
OFC evaluation results

- Single functions:

OFC overcomes Swift by up to 82%
OFC evaluation results

- Multi-stage functions

OFC overcomes Swift by up to 60%
OFC: Conclusion

• OFC leverages ML and RAMCloud
  - Opportunistic caching layer for serverless functions
• OFC does not require function modification
  - Direct benefit for existing functions

• OFC ensures consistency between the platform’s cache and the remote storage
• OFC achieves major latency improvements
  - Up to 82% for single functions
  - Up to 60% for multi-stage functions

Checkout OFC source code at https://gitlab.com/lenapster/faascache/