Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing

Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, and Chuhao Xu, Shanghai Jiao Tong University; Deze Zeng, China University of Geosciences; Zhuo Song, Tao Ma, and Yong Yang, Alibaba Cloud; Chao Li and Minyi Guo, Shanghai Jiao Tong University
Introduction & Background

- Definition of serverless (FaaS).
- What are advantages and limitations?
Introduction & Background

What is Serverless?

Berkeley’s View: “Serverless = FaaS (Function-as-a-Service) + BaaS (Backend-as-a-Service)”
Introduction & Background

What are the advantages of using serverless model?

- **Infrastructure-as-a-Service**
  - Vertical resource scaling with remained
  - Maintain the underlying environment
  - Pay-as-time, low resource utilization

- **Function-as-a-Service**
  - Auto horizontal scaling without remained
  - Offloaded environment management
  - Pay-as-invocation, high resource utilization
Introduction & Background

The most significant features of serverless computing

**Serverless features**
- Event-driven
- Auto-scaling
- Others (offloaded management, flexible scheduling, pay-as-you-go costing model)

**Serverless benefits**
1/ Containers created on-demand
2/ Fine-grained resource scaling

**Derived challenges**
1/ Containers cold startups
2/ High-density and high-concurrency
Why we should alleviate cold startups?

Functions-invocations follow a Pareto distribution.

- 20% of popular functions occupy 99.6% of overall invocations (observed from Azure trace).

The view from a node:
Cold startup Invocations are less than 1%

The view from tenants:
80% of functions frequently experience cold startups
Introduction & Background

Motivation

• How to alleviate cold startups?
• Does the current method work efficiently?
Motivation

Leveraging prewarmed container to alleviate cold startups:

• **Exclusive size-fixed prewarm pool:**

  *good and stable performance, easy to implement*

  *need to adjust the pool size for each function*

  *many long-term running prewarmed and idle containers consume resources*
Leveraging prewarmed container to alleviate cold startups:

- Template-based shared prewarm pool:
  
  Resource-friendly

  All functions use the same template image, easy to maintain

  Specialization phases introduce unpredictable overhead.
Motivation

The unpredictable overhead of specialization.

- five functions are triggered simultaneously by a caller in eco.
- Concurrent invocations from these functions contend for the prewarmed containers.
Motivation

The unpredictable overhead of specialization.

- High-concurrency invocation
- Conflict with template image
- Prewarm pool breakdown
- Retry with cold startup

Specialization conflict

Pkg_a1: 1.0
Pkg_b_a: 2.0
Pkg_c: 2.0

Pkg_a2: 1.0
Pkg_b_a: 2.0
Pkg_c: 2.2

Pkg conflict
Sub-pkg in b conflict
Version conflict
Motivation

The unpredictable overhead of specialization.

- High-concurrency invocation
- Conflict with template image
- Functions need additional libs
- Prewarm pool break down
- Retry with cold startup
- High loading overhead

- ddns requires to load/install many additional packages in the prewarmed containers
- the package loading is time-consuming, even slower than directly cold startup.
Motivation

Additional trade-offs of template-based prewarm pool.

Warm startup
Apps(20%)

- 2 Req/min
  \[\lambda\]
  Warm startup invocations(>99%)

- 1 Req/min
  \[\lambda\]

Cold startup
Apps(80%)

- 3 Req/h
  \[\lambda\]
  Cold startup invocations(<1%)

- 1 Req/h
  \[\lambda\]

Build templates for
99% invocations
(more cold startups
for 80% cold Apps)

Build templates for
80% cold apps
(more cold startups
for 99% invocations)
Motivation

Exclusive prewarm vs template-based prewarm:

• Exclusive prewarm method:
  to save resource, need to adjust pool size dynamically.
  profiling and predicting -> need to build model for each function
  -> infrequent functions do not have enough trace to train

• Template-based prewarm:
  three unpredictable overhead of specialization
  need to make several trade-offs

The current prewarm method is not efficient due to several inevitable trade-offs. It is beneficial to alleviate cold startups without trapping in the same dilemmas.
Introduction & Background

Motivation

Methodology & Design

• Reusing idle containers
• Build Zygote containers for sharing
• SF-WRS based scheduling policy
Methodology & Design

Cold startup alleviation accelerating - Pagurus

Can we reuse idle containers for functions to avoid cold startups like Pagurus?
Help rather than recycle – idle containers

Feasibility of reusing idle containers

- Serverless platforms use keep-alive strategy to reduce cold startups
- Diurnal pattern wildly exist in many applications
- Containers become idle and recycled 15min later
Methodology & Design

Help rather than recycle – Zygote containers

- The zygote container serve as a safe checkpoint that any function is not invoked.
- Set shared domain and privilege domain.
- Other to-be-helped functions are mounted anonymously.
- Executor invoke functions with non-root users.
Help rather than recycle – scheduling and forking Zygotes

- Identifying idle containers for each function
- Build Zygote image, and replace an idle container with a Zygote
- Fork a Zygote to be a helper container for cold startup functions if it mounted
- Unmount and helper container join in corresponding container pool
Methodology & Design

How to arrange zygote containers for appropriate forking?

— SF-WRS (Similarity Filtered Weighted Random Sampling)

• Select to-be-helped functions:

  based on the similarity of functions’ packages (cosine)

  set similarity as 0 if conflict exist

  WRS makes to-be-helped functions more likely to be repacked if it has more cold startups

Filter to-be-helped candidates

(pkg: a,b,c,d) (pkg: b,c,e) (pkg: a,b,c) (pkg: a,e,f)

\( \cos = 0.577 \quad \text{Cold startup times: 8} \)

\( \cos = 0.866 \quad \text{Cold startup times: 2} \)

\( \cos = 0.288 \)

\( P(\text{Repack}) = 0.8 \)

\( P(\text{Repack}) = 0.2 \)
Introduction & Background
Motivation
Methodology & Design
Evaluation
Evaluation

Evaluation setups:

• Baselines:

  *OpenWhisk with AWS application samples and Azure trace day07.*

• Software and hardware setup:

<table>
<thead>
<tr>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node</strong></td>
</tr>
<tr>
<td>CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz</td>
</tr>
<tr>
<td>Cores: 8, DRAM: 16GB, Disk: 100GB SSD (3000 IOPS)</td>
</tr>
<tr>
<td><strong>Software</strong></td>
</tr>
<tr>
<td>Operating system: Linux with kernel 4.15.7, Docker: 20.10.6</td>
</tr>
<tr>
<td>Nginx version: nginx/1.10.3, Database: Couchdb:3.1.1</td>
</tr>
<tr>
<td>runc version: 1.0.0-rc93, containerd version: 1.4.4</td>
</tr>
<tr>
<td><strong>Container</strong></td>
</tr>
<tr>
<td>Container runtime: Python-3.7.0, Linux with kernel 4.15.7</td>
</tr>
<tr>
<td>Resource limit and Lifetime: 1-core with 256MB, 600s</td>
</tr>
<tr>
<td>Function container limit: 10 for each function on each node</td>
</tr>
<tr>
<td>prewarm pool size in OpenWhisk: 2 on each node</td>
</tr>
<tr>
<td><strong>Benchmarks (38 functions in 10 AWS Lambda best practice applications)</strong></td>
</tr>
<tr>
<td>serverless-ecommerce-platform (eco), etl-orchestration (etl)</td>
</tr>
<tr>
<td>cost-explorer-report (rep), serverless-tokenization (tok)</td>
</tr>
<tr>
<td>transcribe-comprehend-podcast (pod), serverless-chatbot (bot)</td>
</tr>
<tr>
<td>serverless-shopping-cart (cart), refarch-fileprocessing (file)</td>
</tr>
<tr>
<td>finding-missing-persons-using-rekognition (rek), ddns</td>
</tr>
</tbody>
</table>
Evaluation

Key improvements in Azure trace:

- **84.6%**
- **Alleviate most functions’ cold startups, 73.4% of functions no longer need cold startups**

- **20ms**
- **Reduce cold startup response latency to 16ms if it need additional packages**

- **p95 latency**
- **Lower 95%-ile latency, especially for mid-popular functions**
Introduction & Background
Motivation
Rationale & Design
Evaluation
Conclusion
Conclusion

Summary:

• Resource-friendly and security-ensured Zygote design.
  • Shared domain and privilege domain.
• Replacing idle containers as Zygote containers for inter-function sharing.
  • Reusing others’ Zygote containers to alleviate cold startups.
• SF-WRS based Zygote arrangement and scheduling.
  • Calculate cosine distance as similarity to improve sharing efficiency

Another related track presentation:

RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing Introduces how to enable high-density and high-concurrency startup
Thanks!

Q&A

Zijun Li, lzjx1122@sjtu.edu.cn;
Linsong Guo, gls1196@sjtu.edu.cn;
Quan Chen, chen-quan@cs.sjtu.edu.cn;
Jiagan Chen, chengjiagan@sjtu.edu.cn;
Chuhao Xu, barrin@sjtu.edu.cn;
Deze Zeng, deze@cug.edu.cn;
Zhuo Song, songzhuo.sz@alibaba-inc.com;
Tao Ma, boyu.mt@alibaba-inc.com;
Yong Yang, zhiche.yy@alibaba-inc.com;
Chao Li, lichao@cs.sjtu.edu.cn;
Minyi Guo, guo-my@cs.sjtu.edu.cn;