# FaasCache

## Keeping Serverless Computing Alive with Greedy Dual-Caching

Fuerst, Sharma | Indiana University Bloomington

Presented by: T. Pal, P. Kotak, D. Ralisback
University of Washington | Tacoma | Fall 2021

1

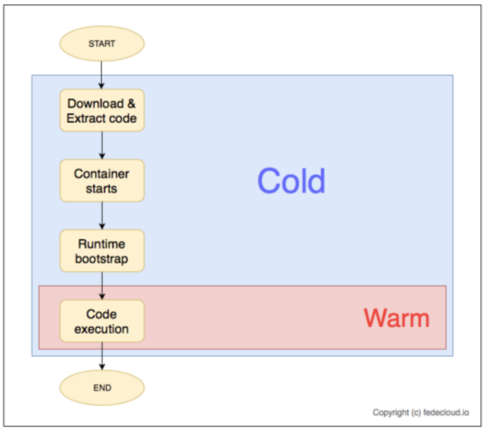
## Outline

- Introduction
- Related work
- Techniques
- Key contributions
- Experimental evaluation
- Authors conclusion
- Critique
- Gap analysis

2

2

# Introduction | Background




Why use FaaS?

- Infinite horizontal scaling
- Pay for what you use
- Scaling is transparent + independent of function implementation

Key Disadvantage: 'Cold-starts'

- Unavoidable overhead of container initialization
- Loss of artifacts / network / caches

Image source: https://medium.com/@danielmanchev/cold-warm-and-hot-start-in-aws-lambda-bc8d64f28575

3

3

# Introduction | Motivation

The problem: Cloud providers' current handling of FaaS cold-starts is inefficient

Why it's a problem

- Consumer : High/unpredictable latency, increased application code complexity
- Provider : Excess resource expenditure → wasted opportunity

Why care (as a researcher)?

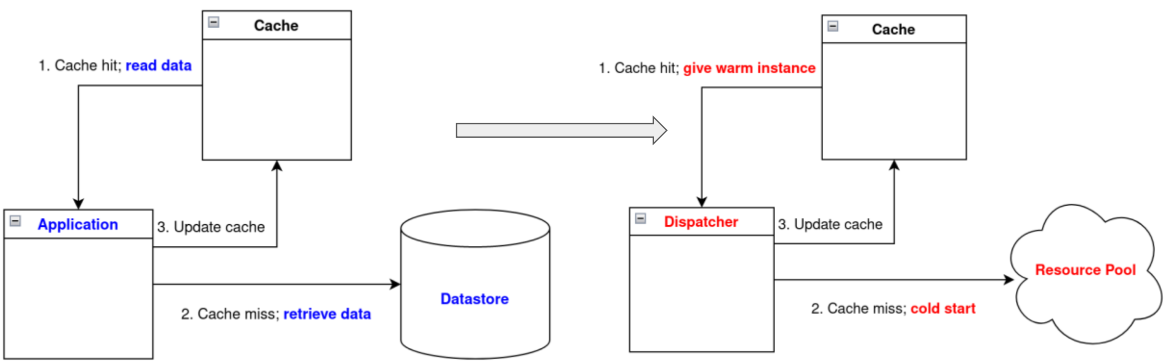
- Researchers overlap with cloud consumers
- Provider cost reduction affects consumer costs
- Environmental -- less energy for same utility
- Expand the set of problems that can leverage FaaS

4

4

Introduction | Hypothesis

- FaaS cold-start should <u>not</u> be treated as a new field of study
- Map results from caching research to the cold start problem.




5

5

# Related Work

<u>Orthogonal + Complementary</u>

- Cold start latency reduction: container startup overheads / lightweight VM's
- Optimizing environment restoration: [Catalyzer] checkpointing/restoring state
- DAG scheduling: allocation based on known workflow.
- Tightening CPU-share bounds: [ENSURE] reduce deprovisioning by increasing colocation.
- Warm pools: Keep containers warm through autoscaling with 'pod migrations'

6

6

# 'Most-Related' Work

- Fixed-time keep-alive + polling: standard approach in industry (bad)
- Time series + predictive allocation: Preemptive allocation from usage patterns
- Primary Motivator: [AZURE] data set

| | |
|---|---|
| Range of function invocation frequency | ~10^8 |
| % functions w/ frequency > 1/min | 81% |
| % functions w/ total latency < 10s | 75% |
| % functions w/ predictable periods | 40% |
| % contribution of most frequent 20% of functions | 99.6% |

Missing considerations

- Surge traffic: [PCPM] Caching doesn't help with surges of utilization, only reuse of existing functions
- Memory overhead: [FAA$T]



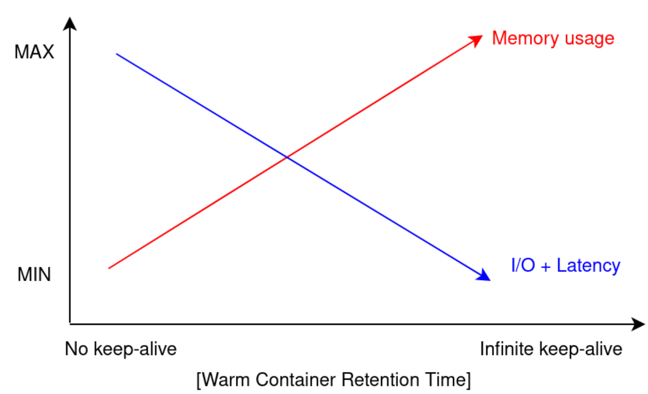


# Techniques | Keep-alive Policies

- Scope: Optimize total latency at the server level.
- Design trade-offs? (graph)

Idea

- Greedy-Dual-Size-Frequency caching
- Only evict when a new container doesn't fit
- Favor:
  - small containers
  - frequent use
  - high init costs



# Techniques | Keep-alive Policies (cont.)

Evict container with lowest priority based on

priority = clock + (frequency x cost) / size

- clock: Shared by all containers. Increments after each eviction.
- frequency: the number of times the function has been invoked
- cost: cold start time of the function
- size: memory usage[1] of this container

Alternatives:

- Simplifies to LRU, LFU for param subsets
- Landlord algorithm is also possible.

[1]could also be the magnitude of an n-D "resource footprint vector"

9

9

# Techniques | Server Provisioning Policies

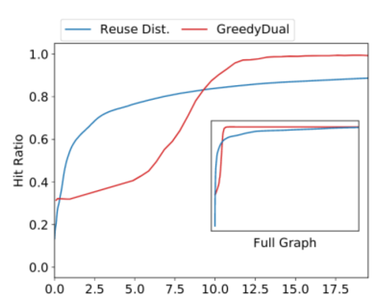
Scope: determining the size and capacity of the servers for handling FaaS workloads

Design trade-offs? Analogous to server-level

Static Approach

- Choose minimum cache size that achieves some success metric
- E.g.
  - Cache Hit Ratio > threshold
  - Optimize marginal utility

$$\text{Hit-ratio}(c) = \sum_{x=0}^{c} P(\text{Reuse-distance} = x)$$




10

10

# Techniques | Server Provisioning Policies

Shortcomings of Static Approach

- The caching analogy crumbles for concurrent executions (caches consider unique sets of objects)
- Cache-hit-ratio is poisoned (to some degree) by concurrency. How to contend with this?

---

Dynamic Auto-scaling Policy (Periodically reoptimize VM memory size)

Calculate the ideal cache size with recent metrics

- Assume there is an ideal miss rate
- Compute →
- Invert result to determine cache size

$$\mathrm{HR}(c') = 1 - m = 1 - h\frac{\bar{\lambda}}{\lambda}$$

11

11

# Key Contributions

- Equivalence between concepts: FaaS keep-alive = object caching
  - Rigorous/extensive body of work to leverage for a new problem
- A specific family of keep-alive policies based on Greedy-Dual caching
  - Cold start overhead reduction: 3x
  - Application latency reduction: 6x
  - Requests served per host: 2x
- A static resource allocation policy based on cache hit ratio, and
- an elastic policy based on maintaining an ideal hit ratio
  - Reduces average server size by 30%

12

12

# Experimental Evaluation

- Experiments conducted
  - Experimental evaluation of the caching based keep-alive and provisioning techniques was conducted by using function workload traces and serverless benchmarks
- Experimental design
  - Trace samples from the Azure Function trace
    - Three trace samples
      - Rare
      - Representative
      - Random

| Trace | Num Invocations | Reqs per sec | Avg. IAT |
|---|---|---|---|
| Representative | 1,348,162 | 190 /s | 5.4 ms |
| Rare | 202,121 | 30 /s | 36 ms |
| Random | 4,291,250 | 600 /s | 1.8 ms |

  - A single server with 250 GB RAM and 48-core Intel Xeon Platinum 2.10 GHz CPUs is used for running all functions.



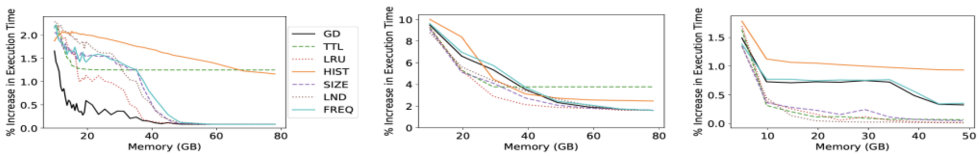


# Experimental Evaluation

- Methods used
  - Trace-Driven Keep-Alive Evaluation
    - It uses the Azure function traces to evaluate different keep-alive policies in the discrete event simulator.
  - OpenWhisk Evaluation
    - Evaluating the performance of the FaasCache system on real functions.

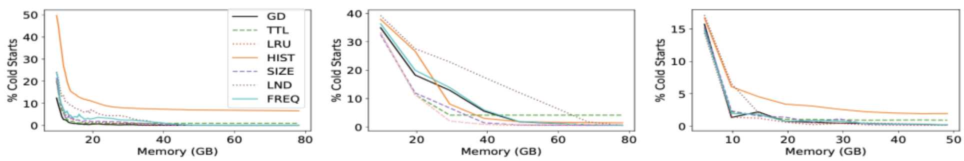
# Key Experimental Results

- Figure 1 shows increase in execution time due to cold-starts for different workloads derived from the Azure function trace.




(a) Representative functions. (b) Rare functions. (c) Random sampling.

- Figure 2 shows fraction of cold starts is lower with caching-based keep-alive.



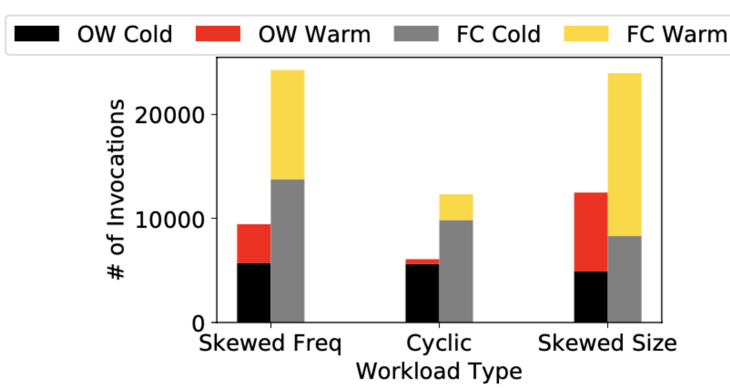

(a) Representative functions. (b) Rare functions. (c) Random sampling.

15



# Key Experimental Results

- Workload type versus the number of invocations is shown in figure 3




16

## Authors' Conclusions

- Function keep-alive and object caching are equivalent problems
  - Far-reaching implications in cloud resources management policies
  - Future research should be viewed through this lens
- Specifically, Greedy-Dual (considering frequency and memory size) is a good heuristic
- Tradeoff between memory utilization and cold-start overheads can be analyzed with hit-ratio curves
- FaasCache - an OpenWhisk-based framework, implements Greedy-Dual caching-based techniques and produce positive results

17

17

## Critique: Strengths

- primary strengths
  - Performance: Reduce cold-start overheads by 3×, improve application-latency by 6×, and reduce system load to serve 2× more requests
  - Cost-effective : To some extent. Greedy-Dual algorithm's eviction policy is based on size and frequency of the object
  - Scalability : Supports diverse FaaS workload and server resources are adjustable using dynamic vertical-scaling policy
- In general, new approaches that don't provide at least a 10% performance improvement are not very significant depending on the problem. An order of magnitude (10x) improvement is preferred.
  - Improvements are not OOM, but the framework of thought seems significant

18

18

## Critique: Weaknesses

- Weaknesses:
  - Favors superusers
  - It demands for huge infrastructure
  - This could be things such as complexity/effort of applying the approach, or it's usability.
  - Requires adoption by cloud providers to reap benefits
  - All results are empirical. It would be interesting to see more theory developed around this
  - Deals with small datasets
  - In research, domain agnostic solutions can have broader impacts and importance than one-off solutions for a specific use case.
  - Not enough information about security or fault tolerant characteristics
  - Not fully dynamic. It depends on the past traffic intensity(invocations per second)
  - Not useful for the concurrent execution of functions

19

19

## Critique: Evaluation

- Authors have not talked about fault tolerance and the security of this method
- Narrow scoped experiment
- Not enough information available for reproducing tests
- In this paper, authors have discussed the GDSF impact on co-located application, cluster-level implementation but this discussion lacks proofs.

20

# Gap Analysis

- This work warrants
  - A rigorous definition of the mapping between the two problem spaces
  - A better understanding of the differences between unique objects and concurrent functions
- Assumptions
  - Users must know an ideal miss ratio
  - Prior knowledge required to predict the EtE workload
  - Memory is the only important factor.
- Future work
  - Find better / more specific eviction heuristics (or learn them) for particular workloads
  - Reconcile difference between hit ratio curve and actual curve (caused by concurrency)
  - Combine with orthogonal related works section
  - Separate init from function code for predictive loading.
  - the tradeoff between function and other colocated application

21

21

Question break.

22

22