InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

Ao Wang*, Jingyuan Zhang*, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, Yue Cheng

* These authors contributed equally to this work
Web applications are storage-intensive
Web applications – heterogeneous I/O
Case study: IBM Docker registry workloads

• IBM Cloud container registry service across 75 days during 2017
• Selected data centers: Dallas & London
Case study: IBM Docker registry workloads

• Object size distribution
• Large object reuse patterns
• Storage footprint
Case study: IBM Docker registry workloads

• Object size distribution

• Large object reuse patterns

• Storage footprint

Extreme variability in object sizes:

➢ Object sizes span over 9 orders of magnitude

➢ 20% of objects > 10MB
Case study: IBM Docker registry workloads

• Object size distribution

• Large object reuse patterns

• Storage footprint

Caching large objects is beneficial:

➢ > 30% large object (>10MB) access 10+ times

➢ Around 45% of them got reused within 1 hour
Case study: IBM Docker registry workloads

• Object size distribution

• Large object reuse patterns

• Storage footprint

Extreme tension between small and large objects:

➢ Large objects (>10MB) occupy 95% storage footprint
Existing cloud storage solutions

Both dimensions: the lower the better

Price ($/GB/hour)

Performance (latency)

Cheap $

Expensive $$$

Fast

Slow
Large objects managed by cloud object stores

Both dimensions: the lower the better

Object stores are cheap but too slow

AWS S3: $0.023 per GB per month

Performance (latency)

Fast

Slow

Price ($/GB/hour)

Cheap $

Expensive $$$
Small objects accelerated by in-memory caches

Both dimensions: the lower the better

Object stores are cheap but too slow

In-memory caches are fast but too expensive

AWS ElastiCache: $0.016 per GB per hour
• Caching both small and large objects is challenging
• Existing solutions are either too slow or expensive
• Caching both small and large objects is challenging
• Existing solutions are either too slow or expensive

How can we achieve the best of both worlds?
• Caching both small and large objects is challenging
• Existing solutions are either too slow or expensive

Requires rethinking about a new cloud cache/storage model that achieves both cost effectiveness and high-performance!
InfiniCache: A cost-effective and high-performance in-memory caching solution atop Serverless Computing platform

- **Insight #1**: Serverless functions’ \(<\text{CPU, Mem}>\) resources are pay-per-use
- **Insight #2**: Serverless providers offer “free” function caching for tenants
InfiniCache: A cost-effective and high-performance in-memory caching solution atop Serverless Computing platform

- **Insight #1:** Serverless functions’ `<CPU, Mem>` resources are pay-per-use → Cost-effectiveness
- **Insight #2:** Serverless providers offer “free” function caching for tenants → High-performance
A primer on Serverless Computing

- Serverless computing enables cloud tenants to launch short-lived tasks (i.e., Lambda functions) with high elasticity and fine-grained resource billing.
A primer on Serverless Computing

- Serverless computing enables cloud tenants to launch short-lived tasks (i.e., Lambda functions) with **high elasticity** and **fine-grained resource billing**

- Function: basic unit of deployment. Application consists of multiple serverless functions
A primer on Serverless Computing

• Serverless computing enables cloud tenants to launch short-lived tasks (i.e., Lambda functions) with high elasticity and fine-grained resource billing

• Function: basic unit of deployment. Application consists of multiple serverless functions

• Popular use cases: Backend APIs, data processing…
Serverless Computing is desirable

• Pay-per-use pricing model
  • AWS Lambda: $0.2 per 1M invocations
    $0.00001667 for every GB-sec
Serverless Computing is desirable

- Pay-per-use pricing model
  - AWS Lambda: $0.2 per 1M invocations
    $0.00001667 for every GB-sec

- Short-term function caching
  - Provider caches triggered functions in memory without charging tenants
Serverless Computing is desirable

- Pay-per-use pricing model
  - AWS Lambda: $0.2 per 1M invocations
    $0.00001667 for every GB-sec

- Short-term function caching
  - Provider caches triggered functions in memory without charging tenants

**Goal**: Exploit the serverless computing model to build a cost-effective, high-performance in-memory cache
Challenges: to build a memory cache with serverless functions

• A strawman proposal
  • Directly cache the objects in serverless functions’ memory?

• No data availability guarantee

• Banned inbound network

• Limited per-function resources
Challenges: to build a memory cache with serverless functions

• A strawman proposal
  • Directly cache the objects in serverless functions’ memory?

• **No** data availability guarantee

• Banned inbound network

• Limited per-function resources
Challenges: to build a memory cache with serverless functions

• A strawman proposal
  • Directly cache the objects in serverless functions’ memory?

• No data availability guarantee

• **Banned** inbound network

• Limited per-function resources

⚠ Serverless functions cannot run as a server
Challenges: to build a memory cache with serverless functions

• A strawman proposal
  • Directly cache the objects in serverless functions’ memory?

• No data availability guarantee

• Banned inbound network

• Limited per-function resources

⚠ Memory up to 3 GB
⚠ CPU up to 2 cores
Our contribution: InfiniCache

- The first in-memory cache system built atop serverless functions
- InfiniCache achieves high data availability by leveraging erasure coding and delta-sync periodic data backup across functions
- InfiniCache achieves high performance by utilizing the aggregated network bandwidth of multiple functions in parallel
- InfiniCache achieves similar performance to AWS ElastiCache, while improving the cost-effectiveness by 31—96X
Outline

• InfiniCache Design

• Evaluation

• Conclusion
InfiniCache bird’s eye view

Application

EC encoder/decoder

InfiniCache client library

Request routing

InfiniCache proxy server

Lambda management

Lambda cache pool
InfiniCache: PUT path

Application

EC encoder

InfiniCache client library

Request routing

InfiniCache proxy

Lambda cache pool
InfiniCache: PUT path

Application

InfiniCache client library

Request routing

InfiniCache proxy

Lambda cache pool
InfiniCache: PUT path

1. Object is split and encoded into k+r chunks

InfiniCache client library

EC encoder

d1  d2  p1

InfiniCache proxy

Request routing

Lambda cache pool

\[ k = 2, r = 1 \]
InfiniCache: PUT path

1. Object split and encode into $k+r$ chunks
2. Object chunks are sent to the proxy in parallel
InfiniCache: PUT path

1. Object split and encode into k+r chunks
2. Object chunks are sent to the proxy in parallel
3. Proxy invoke Lambda cache nodes

EC encoder

k = 2, r = 1

Request routing

Invocation path

Lambda cache pool

InfiniCache client library

InfiniCache proxy
InfiniCache: PUT path

1. Object split and encode into k+r chunks
2. Object chunks are sent to the proxy in parallel
3. Proxy invoke Lambda cache nodes
4. Proxy streams object chunks to Lambda cache nodes

EC encoder

Request routing

InfiniCache client library

InfiniCache proxy

Lambda cache pool

Application

Data path

k = 2, r = 1

1. Object split and encode into k+r chunks
2. Object chunks are sent to the proxy in parallel
3. Proxy invoke Lambda cache nodes
4. Proxy streams object chunks to Lambda cache nodes
InfiniCache: GET path

Application

InfiniCache client library

EC decoder

Request routing

InfiniCache proxy

Lambda cache pool

d1  d2  p1
InfiniCache: GET path

1. Client sends GET request

Application

InfiniCache client library

InfiniCache proxy

Lambda cache pool

1. Client sends GET request

EC decoder

Request routing

d1

d2

p1
InfiniCache: GET path

1. Client sends GET request
2. Proxy invokes associated Lambda cache nodes
InfiniCache: GET path

1. Client sends GET request
2. Proxy invokes associated Lambda cache nodes
3. Lambda cache nodes transfer object chunks to proxy

Application

EC decoder

InfiniCache client library

Request routing

InfiniCache proxy

Data path

Lambda cache pool
InfiniCache: GET path

1. Client sends GET request

2. Proxy invokes associated Lambda cache nodes

3. Lambda cache nodes transfer object chunks to proxy
   - **First-d optimization**: Proxy drops straggler Lambda

\[ k = 2, r = 1 \]

\( d_2 \) is straggling…
**InfiniCache: GET path**

1. Client sends GET request
2. Proxy invokes associated Lambda cache nodes
3. Lambda cache nodes transfer object chunks to proxy
4. Proxy streams $k$ chunks in parallel to client

**Diagram:**
- **Client** sends GET request to **InfiniCache client library**.
- **InfiniCache proxy** invokes Lambda cache nodes.
- Lambda cache nodes transfer object chunks to **InfiniCache proxy**.
- **Proxy** streams $k = 2$ chunks to **client**.
  - $k = 2$, $r = 1$
  - $d2$ is straggling...

**Data path:**
- $d1$, $p1$, $d1$, $p1$
InfiniCache: GET path

1. Client sends GET request
2. Proxy invokes associated Lambda cache nodes
3. Lambda cache nodes transfer object chunks to proxy
4. Proxy streams $k$ chunks in parallel to client
5. Client library decodes $k$ chunks

Application

InfiniCache client library

InfiniCache proxy

$\lambda$

Lambda cache pool

$X$

EC decoder

Request routing

$d_1$

$p_1$

Data path

$k = 2$ chunks

$k = 2, r = 1$

d2 is straggling...
Maximizing data availability

• Erasure-coding

• Periodic warm-up

• Periodic delta-sync backup
Maximizing data availability

• Erasure-coding

• Periodic warm-up

• Periodic delta-sync backup
Maximizing data availability: Periodic warm-up

AWS Lambda reclaiming policy
Maximizing data availability: Periodic warm-up

AWS Lambda reclaiming policy
Maximizing data availability: Periodic warm-up

AWS Lambda reclaiming policy

- **Shorter** triggering interval will **lower** the function reclaiming rate.

1 min interval significantly reduce function reclaiming rate.
Maximizing data availability: Periodic warm-up

1. Lambda nodes are cached by AWS when not running
   • AWS may reclaim cold Lambda functions after they are idling for a period
Maximizing data availability: Periodic warm-up

1. Lambda nodes are cached by AWS when not running
   • AWS may reclaim cold Lambda functions after they are idling for a period

2. Proxy periodically invokes sleeping Lambda cache nodes to extend their lifespan
Maximizing data availability: Periodic backup
Maximizing data availability: Periodic backup
Maximizing data availability: Periodic backup

1. Proxy periodically sends out backup commands to Lambda cache nodes
Maximizing data availability: Periodic backup

1. Proxy periodically sends out backup commands to Lambda cache nodes

2. Lambda node performs delta-sync with its peer replica
   - Source Lambda propagates delta-update to destination Lambda
Seamless failover

Proxy

Function deployment

: Primary

: Backup
Maximizing data availability: Seamless failover

1. Proxy invokes a Lambda cache node with a GET request
Maximizing data availability: Seamless failover

1. Proxy invokes a Lambda cache node with a GET request

2. Primary Lambda gets reclaimed
Maximizing data availability: Seamless failover

1. Proxy invokes a Lambda cache node with a GET request

2. Primary Lambda gets reclaimed

3. The invocation request gets seamlessly redirected to the backup Lambda
Maximizing data availability: Seamless failover

1. Proxy invokes a Lambda cache node with a GET request

2. Source Lambda gets reclaimed

3. The invocation request gets seamlessly redirected to the backup Lambda
   - Failover gets **automatically** done and the backup becomes the primary
   - By exploiting the **auto-scaling** feature of AWS Lambda
Outline

• InfiniCache Design

• Evaluation

• Conclusion
Experimental setup

• InfiniCache
  • 400 1.5GB Lambda cache nodes
  • Client running on one c5n.4xlarge EC2 VM
  • Warm-up interval: 1 minute; backup interval: 5 minutes
  • Under one AWS VPC

• Production workloads
  • The first 50 hours of the Dallas datacenter traces from IBM Docker registry workloads
  • All objects: including small and large objects
  • Large object only: objects > 10MB
Cost effectiveness of InfiniCache

AWS ElastiCache

- One cache.r5.24xlarge with 600GB memory
- $10.368 per hour
Cost effectiveness of InfiniCache

Workload setup
- All objects
- Large object only
  - Object larger than 10MB

<table>
<thead>
<tr>
<th>Workload</th>
<th>ElastiCache</th>
<th>IC (all objects)</th>
<th>IC (large only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>$518.40</td>
<td>$20.52</td>
<td>$16.51</td>
</tr>
</tbody>
</table>
Cost effectiveness of InfiniCache

- **$518.40**
- **$20.52**
- **$16.51**

Workload setup
- All objects
- Large object only
  - Object larger than 10MB

ElastiCache
IC (all objects)
IC (large only)
Cost effectiveness of InfiniCache

Workload setup
- All objects
- Large object only
  - Object larger than 10MB
- Large object w/o backup

$518.40
- ElastiCache
- IC (all objects)
- IC (large only)
- IC (large no backup)

96x
• $20.52
• $16.51
• $5.41
Cost effectiveness of InfiniCache

<table>
<thead>
<tr>
<th>Workload</th>
<th>ElastiCache</th>
<th>InfiniCache</th>
<th>InfiniCache w/o backup</th>
</tr>
</thead>
<tbody>
<tr>
<td>All objects</td>
<td>67.9%</td>
<td>64.7%</td>
<td>---</td>
</tr>
<tr>
<td>Large object only</td>
<td>65.9%</td>
<td>63.6%</td>
<td>56.1%</td>
</tr>
</tbody>
</table>

Workload setup
- All objects
- Large object only
  - Object larger than 10MB
- Large object w/o backup
Cost effectiveness of InfiniCache

Workload setup
- All objects
- Large object only
  - Object larger than 10MB
- Large object w/o backup

<table>
<thead>
<tr>
<th>Workload</th>
<th>ElastiCache</th>
<th>InfiniCache</th>
<th>InfiniCache w/o backup</th>
</tr>
</thead>
<tbody>
<tr>
<td>All objects</td>
<td>67.9%</td>
<td>64.7%</td>
<td>---</td>
</tr>
<tr>
<td>Large object only</td>
<td>65.9%</td>
<td>63.6%</td>
<td>56.1%</td>
</tr>
</tbody>
</table>

Hit ratio and $$ cost tradeoff

- $518.40 for ElastiCache
- $20.52 for IC (all objects)
- $16.51 for IC (large only)
- $5.41 for IC (large no backup)
Cost effectiveness of InfiniCache

InfiniCache is 31 – 96x cheaper than ElastiCache because tenant does not pay when Lambdas are not running

Workload setup
- All objects
- Large object only
  - Object larger than 10MB
  - Large object w/o backup
Performance of InfiniCache

All objects

Latency (sec)

Large objects only

Latency (sec)
Performance of InfiniCache

- All objects

- Large objects only

> 100 times improvement
Performance of InfiniCache

![Normalized latency chart comparing ElastiCache, InfiniCache, and AWS S3 across different object sizes in MB.](chart.png)
Performance of InfiniCache

Lambda invocation overhead (~13ms) dominates when fetching small objects
Performance of InfiniCache

InfiniCache achieves same or higher performance than ElastiCache for large objects
Conclusion

• InfiniCache is the **first** in-memory cache system built atop a serverless computing platform (**AWS**  

• InfiniCache synthesizes a series of techniques to achieve **high performance** while maintaining **good data availability**

• InfiniCache improves the cost-effectiveness by **31-96x** compared to **AWS ElastiCache**
Thank you!

• Contact: Ao Wang – awang24@gmu.edu,
  Jingyuan Zhang – jzhang33@gmu.edu

• https://github.com/mason-leap-lab/infinicache

University of Nevada, Reno