A Transparent Auto-Scaling Cache for Serverless Applications

Francisco Romero§
Gohar Irfan Chaudhry†, Íñigo Goiri†, Pragna Gopa‡, Paul Batum†, Neeraja Yadwadkar§, Rodrigo Fonseca†, Christos Kozyrakis§, and Ricardo Bianchini†

§Stanford University †Microsoft Research ‡Microsoft Azure

February 11, 2021
Serverless computing

- Short-lived
- Per-function resource allocation with fine-grained billing
- Offered as a service (Function-as-a-Service, FaaS)
  - An *application* is a collection of logically-related functions
Data transfer and sharing in serverless today

• Resources can be reclaimed at any moment → stateless applications

• **Challenges:**
  • How to persist state for future invocations?
  • How to communicate *between* functions?

• **Solution:** read/write state from/to a common data store

  **Storage clusters**  
  (Pocket – OSDI’18, Locus – NSDI’19, Redis)

  **Stateful FaaS Platforms**  
  (Faasm – ATC’20, Cloudburst – VLDB’20)

  **Commercial object stores**  
  (Azure Blob Storage, Amazon S3, Google Cloud Storage)

  **Functions as ephemeral storage**  
  (InfiniCache – FAST’20)
Data transfer and sharing in serverless today

What are the performance implications?
Data loading/transfer hurts performance

- **Native**: local VM, all data in local storage; time to load PyTorch not included (700MB, ~400ms to load)
- **FaaS**: data communication through commercial object storage

Time dominated by data movement to/from remote storage
Data access characterization

14 days of logs for 855 applications with ~44 million data accesses

• **Data size**
  - Blob sizes range from a few bytes to almost 2GB
  - **Takeaway:** large variety in object sizes

• **Data accesses and reuse**
  - 11% access more than one blob per invocation
  - ~12% access the same blob across all invocations
  - **Takeaway:** large variety in application working set and data reuse

• **Temporal access pattern**
  - Many accesses are bursty
  - **Takeaway:** need to support frequently and rarely invoked applications and objects
Insights

• Single cache for all apps and external storage resources is wasteful
  • Support both frequently and rarely invoked applications
  • Tied to application

• Scaling according to the computational load is insufficient
  • Overall cache size (data reuse pattern)
  • Bandwidth to remote storage (large objects)

• User-managed resources and custom APIs go against FaaS abstraction
  • Transparent to applications
The serverless cache

• Transparent caching layer that is tied to each application
  • Resources are loaded/unloaded with application functions (no storage clusters)
  • Pre-warms frequently-accessed objects
  • Scales for cache size and bandwidth to remote storage

• Improves performance by up to 92% compared to existing FaaS offerings and serverless storage systems
  • Eliminates the cost incurred by provisioning additional storage resources
Talk outline

• Architecture overview
• How does the cache make data accesses?
• How does the cache pre-warm frequently accesses objects?
• How does the cache scale for cache size and BW to remote storage?
• Evaluation
Architecture overview

Remote Storage

Frontend
Scale Controller

VM/Container
FaaS Runtime
Application Function
Cache
Member daemon Load daemon

VM/Container
FaaS Runtime
Application Function
Cache
Member daemon Load daemon
Cache instances

- Part of the FaaS runtime
  - Shared memory used for storing and communicating data
  - Enables transparency
- Each cache instance loaded/unloaded with each application instance
  - Tied to each application
Membership and Load daemon

- Membership daemon identifies available instances and determine object ownership
  - Uses consistent hashing
- Load daemon pre-warms cache prior to first application query
Frontend and Scale Controller

- Frontend load-balances requests across running instances
- Scale Controller adds and removes instances
  - Metrics provided by FaaS runtime
Data accesses

Local Cache Hit

Remote Cache Hit

Remote Cache Miss

Local Cache Miss
Pre-warming the serverless cache

**Goal:** Pre-warm cache prior to first application query
- Important for infrequently invoked applications

**Challenges:**
- Deciding *when* to pre-warm data
  - Hybrid histogram policy
- Deciding *what* data to pre-warm
Deciding what data to pre-warm

• Collect metadata during unloading
  • Size of objects, version, and access types (e.g., local hit)

• Load under two conditions
  • Cache hits greater than a threshold
  • Object is accessed more than once across merged metadata

<table>
<thead>
<tr>
<th>Object name</th>
<th>Object size</th>
<th>Access types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notebook state</td>
<td>100KB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cache hit: 3</td>
</tr>
<tr>
<td>DataFrame A</td>
<td>100MB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td>DataFrame B</td>
<td>100MB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td>DataFrame C</td>
<td>100MB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cache hit: 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object name</th>
<th>Object size</th>
<th>Access types</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML model</td>
<td>50MB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td>Image A</td>
<td>70KB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td>DataFrame A</td>
<td>100MB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td>DataFrame B</td>
<td>100MB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td>DataFrame C</td>
<td>100MB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cache hit: 3</td>
</tr>
<tr>
<td>ML model</td>
<td>50MB</td>
<td>Cache misses: 1</td>
</tr>
<tr>
<td>Image B</td>
<td>70KB</td>
<td>Cache misses: 1</td>
</tr>
</tbody>
</table>
Scaling the serverless cache

In addition to compute scaling, the serverless cache scales:

• **Cache size**: based on the data reuse pattern
• **Bandwidth to remote storage**: based on the object size
Cache size scaling

**Goal:** scale to match the application’s working set size

**Mechanism:**

- Track the number of evictions of each locally-cached object
- **Scale out:** any object evicted more than once since the last controller query
- **No action:** no object evicted more than once, but substantial cache access traffic
- **Scale in:** number of accesses is below threshold or minimal
Bandwidth to remote storage scaling

**Goal**: partition large object downloads across multiple instances
- Create higher *cumulative* BW to remote storage
- Exploit higher BW *between* instances

**Mechanism**: Estimate data transfer latency as number of instances increases
- Select number of instances with minimal data transfer latency
**Bandwidth to remote storage scaling**

**Goal**: partition large object downloads across multiple instances
- Create higher *cumulative* BW to remote storage
- Exploit higher BW *between* instances

**Mechanism**: Estimate data transfer latency as number of instances increases
- Select number of instances with minimal data transfer latency

![Diagram showing the partitioning of large object downloads across multiple instances with an estimate of latency (3s) and a note (4.5s if instances need to be loaded)]
Evaluation – baselines and testbed

Baselines
- **Native** – Large, local VM
  - Backed by local SSD
- **Vanilla** – Commercial FaaS offering
  - Backed by remote storage
- InfiniCache (FAST’20)
- Cloudburst’s caching layer (VLDB’20)
- Pocket’s DRAM tier (OSDI’18)
- *Redis service* – A commercial Redis service

Testbed
- Each application instance is a single VM:
  - 8vCPU, 28GiB
  - 500MB/s network bandwidth between instances, 90MB/s to remote storage
Evaluation – applications

Jupyter notebook

Machine learning pipeline

<table>
<thead>
<tr>
<th>Index</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>test</td>
<td>test0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>test</td>
<td>test1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>test</td>
<td>test2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>test</td>
<td>test3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>test</td>
<td>test4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10485755</td>
<td>10485755</td>
<td>test</td>
<td>10485755</td>
</tr>
<tr>
<td>10485756</td>
<td>10485756</td>
<td>test</td>
<td>10485756</td>
</tr>
<tr>
<td>10485757</td>
<td>10485757</td>
<td>test</td>
<td>10485757</td>
</tr>
<tr>
<td>10485758</td>
<td>10485758</td>
<td>test</td>
<td>10485758</td>
</tr>
<tr>
<td>10485759</td>
<td>10485759</td>
<td>test</td>
<td>10485759</td>
</tr>
</tbody>
</table>

10485760 rows x 3 columns

Blob/cache-triggered

HTTP-triggered

Bounding box model (37MB)

Car recog. model (5MB)

People recog. model (98MB)
Comparing the serverless cache to existing systems

- **LH**: local hit, **LM**: local miss, **RH**: remote hit, **RM**: remote miss
- **Native IM**: DataFrame loaded in-memory, **Native RS**: DataFrame fetched from remote storage
- **CB**: Cloudburst’s caching layer, **IC**: InfiniCache

- Improves performance by accessing data in local and remote instances
- **50% to 99.999%** cheaper than baselines with separate servers
Is the serverless cache pre-warming effective?

- **Cold-start**: FaaS runtime and app code *not* loaded, data *not* pre-warmed
- **Hybrid hist**: FaaS runtime and app code loaded, data *not* pre-warmed
- **Hybrid hist + pre-warm**: FaaS runtime and app code loaded, data pre-warmed

Pre-warming frequently-accessed objects improves performance
Conclusion

• Transparent caching layer that is tied to each application
• Improves performance and cost compared to existing FaaS offerings and serverless storage systems
• Enables new class of *interactive* serverless applications

Questions?
faromero@stanford.edu