LIBRA: Harvesting Idle Resources Safely and Timely in Serverless Clusters

Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li*, Xu Yuan†, Seung-Jong Park

Louisiana State University, SUNY-Binghamton University*, University of Louisiana at Lafayette†
Cloud / HPC

Serverless

From A Serverless Vision for Cloud Computing
by Prof. Ana Klimovic
<table>
<thead>
<tr>
<th></th>
<th>Utilized Resources</th>
<th>Unreserved Resources</th>
<th>Idle Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources in use</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resources reserved but unused</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paid by users</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>Re-assignable by operators</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
</tr>
</tbody>
</table>
Idle Resources

"Which of the following factors contribute to avoidable cloud spend, also known as cloud waste, at your organization?"

- Idle or underused resources: 66%
- Overprovisioning resources: 59%
- Lack of needed skills: 47%
- Manual containerization: 37%
- We do not have any avoidable cloud spend: 6%
User over-provisioning
Computation patterns
Varying input data

Low-priority VMs
SmartHarvest [EuroSys 21]

Memory harvesting VMs
MHVM [ASPLOS 22]

Harvest gSSDs
BlockFlex [OSDI 22]

Function acceleration
Freyr [WWW 22]
Harvesting idle resources timely and safely?

Harvesting for VM-based cloud / HPC

- Long-running
- Always-on
- Workload-agnostic

Serverless computing

- Short-living
- Timeliness
- Varying input sizes
Limitations of Existing Work

- Provider-side works (OFC, Freyr)
  - Hard to generalize to varying input data
  - Ignorance of resource timeliness
- User-side works (COSE, StepConf)
  - Static configuration cannot satisfy dynamic resource demands
Input data sizes
Input data content
Varying latency
Timeliness
Harvesting idle resources to accelerate functions
Timeliness of Resources

Invocation A: Over-provisioned, t1—t3
Invocation B: Under-provisioned, t2—t5
Invocations

1. **Invocation A:** Over-provisioned, $t_1$—$t_3$
2. **Invocation B:** Under-provisioned, $t_2$—$t_5$

**Timeliness of Resources**

- **Occupied**
- **Idle**
- **Released**
- **Reassigned**

**Wall Clock**

```
<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Libra Overview

Profiling
Scheduling
Harvesting & Acceleration
Safeguarding
Step 1: Function Deployment

- Deploy code
- User-defined resource allocations
Step 2: Function Invocation

Invocation #3 for Function k at t

Front End

User-defined Func. k

1. Resource Allocation (user_cpu, user_mem)
2. Codebase

Invo. 1 at t'  Invo. 2 at t'

...
Step 2: Function Invocation

- Function requests arrive

User-defined Func. k

1. Resource Allocation (user_cpu, user_mem)

2. Front End

Invocation #3 for Function k at t

Invo. 1 at t'  Invo. 2 at t'
Step 2: Function Invocation

- Function requests arrive
- Input data of varying sizes
Step 3: Profiling

- Check if current invocation is first-seen
- Profile the function upon its first invocation

Step 3a: Workload Duplicator

- Duplicate the input data to different sizes
- Capture the relationship between input size and resource utilization/execution time
Step 3c:
- Yes, train ML models (RF) predicting future invocations.

Step 3d:
- Or apply Histogram models for conservative estimations.
**Step 3c:**
- Yes, train ML models (RF) predicting future invocations.

**Step 3d:**
- Or apply Histogram models for conservative estimations.
Step 3b:

- If seen, use built models for estimation.

Y: (pre_cpu, pre_mem, dur, type)

Is input size-related?

Analyze Acc. & $R^2$

Profiler

Workload Duplicator

Different input sizes

Pilot run
Step 4: Scheduling

- To harvest
- Or to accelerate?

- Calculate demand coverage
Demand Coverage

An incoming invocation, which demands two extra resource units, $t_3 - t_7$.

Demand coverage =

$$\frac{1 \times (t_5 - t_3) + 2 \times (t_7 - t_5)}{2 \times (t_7 - t_3)}$$
Step 4: Scheduling

- Calculate demand coverage in realtime
- Select the node with the highest score
- Same score, then consider locality

Coverage score =

\[ \alpha \times D_{cpu} + (1 - \alpha) \times D_{memory} \]

Demand coverage of CPU

Demand coverage of Memory
Step 5: Harvesting & Acceleration
Worker Node $m$

Container

Harvested Resource Pools

- CPU
- Mem

5 Harvest or Accelerate?

user_* v.s. pred_*

SafeGuard
Safeguard in Realtime

Event: Reaching the threshold
Worker Node $m$

Container → Harvested Resource Pools

- CPU
- Mem

observed_(cpu, mem, duration)

Model update

ML

Hist.

Harvest or Accelerate?

$\text{user}_* \text{ v.s. } \text{pred}_*$

Updating observations
Implementation

Libra is prototyped on top of Apache OpenWhisk using Scala

<table>
<thead>
<tr>
<th>Profiler</th>
<th>Scheduler</th>
<th>Harvest Pool</th>
<th>Safeguard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>OpenWhisk’s Load Balancer</td>
<td>OpenWhisk’s Invoker</td>
<td>Docker container Linux cgroups</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Evaluation**

**Baselines for harvesting**
- OpenWhisk default
- Freyr [4]

**Baselines for scheduling**
- OpenWhisk default
- Min-Worker-Set [5]
- Join-the-Shortest-Queue
- Round-robin

**Metrics**
- Function response latency
- Resource utilization

**Benchmarks**
- SeBS [6]
- ServerlessBench [7]
- ENSURE [8]

**Traces**
- Azure Functions traces [9]
- 1K+ invocation traces

---

Evaluating harvesting
3 nodes
72 Intel Xeon E5-2670 CPU cores
72 GB memory

Evaluating scheduling
6 nodes
160 Intel Xeon E5-2420 CPU cores
160 GB memory

Evaluating scalability
50 Jetstream nodes
1,200 Intel Xeon E5-2680 CPU cores
1,200 GB memory
Function Response Latency

Libra provides lowest function response latency

- Default: OpenWhisk default
- JSQ: Join-the-Shortest-Queue
- MWS: Min-Worker-Set
- RR: Round-robin
Resource Utilization

(a) Default

(b) Freyr

(c) Libra
Harvesting and Safeguarding

\[
\text{Speedup} := \frac{t^\text{user} - t^*}{t^\text{user}}
\]

(a) Default

(b) Freyr

(c) Libra
Scalability & Overhead

![Graphs showing scalability and overhead metrics](image)

(a) Strong Scaling
(b) Weak Scaling
(c) Sched. Delay
Timeliness-aware resource harvesting & scheduling

Input data size-awareness

Timely safeguard

39%
lower function response latency

3x
higher resource utilization

Libra
Libra Code Repo:
https://github.com/IntelliSys-Lab/Libra-HPDC23

Corresponding Author:
Hanfei Yu <hyu25@lsu.edu>
Hao Wang <haowan25@lsu.edu>