Libra: Harvesting Idle Resources Safely and Timely in Serverless Clusters

Hanfei Yu
hyu25@lsu.edu
Louisiana State University
Baton Rouge, LA, USA

Christian Fontenot
cfont85@lsu.edu
Louisiana State University
Baton Rouge, LA, USA

Hao Wang
haowang@lsu.edu
Louisiana State University
Baton Rouge, LA, USA

Jian Li
lij@binghamton.edu
SUNY-Binghamton University
Binghamton, NY, USA

Xu Yuan
xu.yuan@louisiana.edu
University of Louisiana at Lafayette
Lafayette, LA, USA

Seung-Jong Park
sjpark@lsu.edu
Louisiana State University
Baton Rouge, LA, USA

ABSTRACT

Serverless computing has been favored by users and infrastructure providers from various industries, including online services and scientific computing. Users enjoy its auto-scaling and ease-of-management, and providers own more control to optimize their service. However, existing serverless platforms still require users to pre-define resource allocations for their functions, leading to frequent misconfiguration by inexperienced users in practice. Besides, functions' varying input data further escalate the gap between their dynamic resource demands and static allocations, leaving functions either over-provisioned or under-provisioned. This paper presents Libra, a safe and timely resource harvesting framework for multi-node serverless clusters. Libra makes precise harvesting decisions to accelerate function invocations with harvested resources and jointly improve resource utilization by profiling dynamic resource demands and availability proactively. Experiments on OpenWhisk clusters with real-world workloads show that Libra reduces response latency by 39% and achieves 3× resource utilization compared to state-of-the-art solutions.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

serverless computing, resource harvesting

ACM Reference Format:
Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2023. Libra: Harvesting Idle Resources Safely and Timely in Serverless Clusters. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3588195.3592996

HPDC '23, June 16–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0155-9/23/06…$15.00
https://doi.org/10.1145/3588195.3592996

1 INTRODUCTION

Motivation. Known as next-generation cloud computing, serverless computing has attracted extensive attention from users and infrastructure providers. Thanks to its event-driven computing, auto-scaling, and cost-efficiency, serverless computing has shipped numerous applications to resilient stateless functions, including video processing [5, 19], scientific computing and HPC [7, 12, 34, 37, 39], and machine learning [3, 11, 44], relieving users from cumbersome infrastructure maintenance and resource management. In addition, serverless computing completely transfers the infrastructure management work to service providers, leaving more flexibility to providers to improve resource utilization and service quality. However, existing serverless platforms still require users to pre-define the resource allocations for each function. Unlike traditional cloud computing running monolithic applications with homogeneous virtual instances, serverless computing decomposes monolithic applications into numerous types of functions[1] and complex dependencies, impeding even experts from configuring each function appropriately. User misconfiguration, unpredictable events, high concurrency, and varying input data jointly make it a non-trivial task to mitigate the gap between the user reserved resources and invocations' dynamic demands [2, 10, 22, 27, 35]. A recent report reveals that most functions invoked on Alibaba Function Compute can only utilize 20–60% of allocated resources, leaving considerable resources idle—*reserved but unused*—during execution [42].

Limitation of State-of-the-arts. Existing studies attempting to reduce idle resources in serverless computing can be classified into two categories: provider-side [29, 49] and user-side solutions [2, 18, 46]. Provider-side solutions utilize idle resources to accelerate functions (*e.g.*, OFC [29] and Freyr [49]). User-side solutions (*e.g.*, Sizeless [18], and StepConf [46]) optimize function resource configurations to improve function efficiency and resource utilization.

However, existing approaches fall short in addressing the following critical challenges raised by inappropriate resource allocation and varying resource demands: *First*, neither user-defined nor estimated function configurations [2, 18, 46, 49] can continuously satisfy invocations' dynamic resource demands, driven by ad-hoc events and varying input data. Even for the same function, input data may vary (*e.g.*, sizes and contents), leading to fluctuating resource consumption and execution time. *Second*, existing

[1] In this paper, a *function* refers to a code package deployed on a serverless platform, and a function *invocation* is a running instance of the code package.

provider-side solutions rely on non-trivial inspections that may violate serverless nature, such as hand-crafted features [29], which can hardly be generalized to various input data and function types. *Finally*, orchestrating multiple worker nodes (or invokers) to serve large-scale invocations and dynamic resource demands with light-weight algorithms is a complicated scheduling problem unsolved by existing methods [9, 48, 49]. Freyr [49] attempts to work around these issues by learning optimal resource harvesting strategies with Deep Reinforcement Learning (DRL) algorithms but still falls short in 1) ignorance of *timeliness*—harvested resources' availability may expire, 2) no support to varying input sizes, and 3) lack of timely release of harvested resources.

Key Insights and Contributions. To address these challenges, this paper proposes Libra, a general provider-side solution that harvests idle resources safely and timely to accelerate large-scale serverless function invocations with varying inputs. Libra makes invocation scheduling decisions with awareness of resource timeliness across multiple worker nodes and accelerates under-provisioned invocations with idle resources carefully harvested from over-provisioned invocations. Each worker maintains a pool tracking idle resources harvested from over-provisioned invocations. We devise a new metric—*demand coverage*—to quantify idle resource volume and timeliness on each node for timeliness-aware scheduling (§6.2). We justify whether an invocation is under-provisioned or over-provisioned by comparing its user-defined resource allocation and dynamic demands.

We summarize Libra's key contributions as follows:

- We develop a profiler that transparently estimates resource demands and execution time of invocations upon varying input data without access to input data content or user code.
- We propose a timeliness-aware and fine-grained resource harvesting mechanism that jointly accelerates invocations and improves resource utilization.
- We design a decentralized sharding scheduler that maximizes cluster-wide resource utilization with timely awareness of idle resources across workers.
- We deploy Libra on real-world clusters and evaluate Libra using extensive experiments with industrial traces and realistic applications. We compare Libra with multiple resource harvesting schemes (*e.g.*, Freyr [49]) and scheduling algorithms (*e.g.*, Min-Worker-Set [50]). Experimental results show that Libra reduces function response latency by 39% and achieves up to 3× resource utilization compared to the state-of-the-art solutions.

Limitations of the proposed approach. We conclude two limitations of Libra's design that can be improved. *First*, Libra's harvesting and acceleration rely on resource demands and timeliness estimated by machine learning (ML) models. Thus, mispredictions are unavoidable due to the bursty, diversity, and fluctuation of serverless workloads. However, Libra implements a safeguard mechanism that effectively mitigates mispredictions and guarantees function execution performance. *Second*, Libra's scheduler greedily serves function invocations to reduce decision complexity, which may result in sub-optimal objectives such as average function response latency. We opt for such a greedy scheduler in Libra to accommodate the sub-second latency requirement of serverless functions.

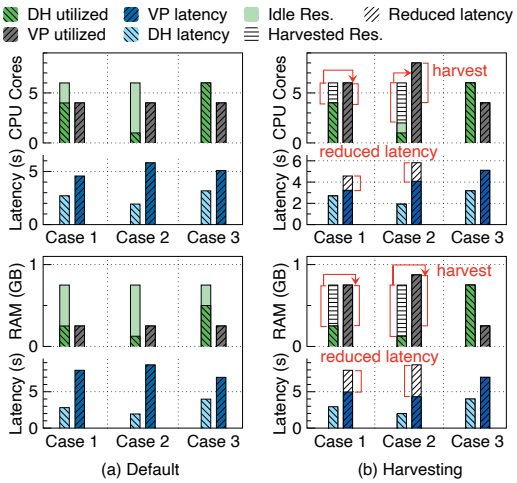

Figure 1: A motivating example of resource harvesting. Input data (DH/VP): Case 1 (4K/video-1), Case 2 (100/video-2), and Case 3 (10K/video-3).

2 BACKGROUND AND MOTIVATION

2.1 Resource Harvesting

Objectives. Cloud users are prone to over-provision their workloads, leaving significant resources (*e.g.*, CPU, memory, disk, and even GPU) idle during running [16, 42]. Resource harvesting techniques are a major solution to utilize such idle resources—*reserved but unused* by users—with safety guarantees [4, 20, 32, 45]. In serverless computing, safety is defined as harvesting resources must not deteriorate function execution performance [29, 49]. For example, a resource allocation strategy is not safe if leading to worse response latency than user-defined resources.

Billing. Harvested resources can be used for various purposes. Infrastructure-as-a-Service (IaaS) cloud platforms package harvested resources as low-priority evictable VMs and resell them to users [20, 32, 45]. Serverless platforms utilize harvested resources to provide transparent optimizations, such as function caching [29] and execution acceleration [49], which are ephemeral free lunch for users but no guarantee.

2.2 The Necessity of Resource Harvesting

A serverless function is usually invoked with varying input data sizes and contents, leading to fluctuating response latency and heterogeneous utilization of different resource types (*e.g.*, CPU and memory) [10, 27, 49]. We explore the opportunities behind such fluctuations and heterogeneity that motivate accurate harvesting and reassignment of individual resource types by running two realistic serverless applications: DH[2] and VP[3] on an Apache OpenWhisk cluster [6]. The resource utilization and response latency of DH is dominated by input sizes, whereas VP's is dominated by data contents. By decoupling CPU and memory allocation [28, 49], we investigate the CPU and memory harvesting individually for DH and VP: when adjusting the CPU (memory) allocation, we fix its memory (CPU) to one GB (eight cores). We simultaneously invoke DH and VP once with different input data of three cases as shown

[2]Dynamic HTML (DH) generates a given number of HTML pages.
[3]Video Processing (VP) generates a GIF from the input video.

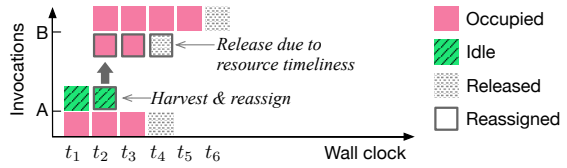

Figure 2: Harvested resources' timeliness.

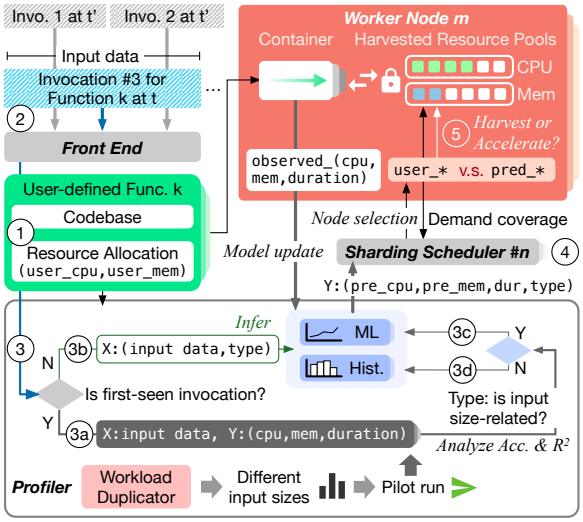

Figure 3: Libra's architecture.

in Fig. 1(a). In Case 1, with an input size of 4K, DH can only utilize four cores (0.25 GB), leaving two cores (0.5 GB) idle, while VP has fully utilized four cores (0.25 GB) with the first input video (video-1). In Case 2, DH can only utilize one core (0.125 GB) with an input size of 100, leaving five cores (0.625 GB) idle, while VP still reaches full utilization with another video (video-2). In Case 3, both DH and VP fully utilize user-allocated CPU cores (memory). Fig. 1(a) illustrates over-provisioned and under-provisioned invocations and their varying response latency due to different input data, which is aligned with the observation in Microsoft Azure Functions [36] and Alibaba Function Compute [42].

However, we can seize the opportunity behind the fluctuation by carefully harvesting the DH invocation's idle resources and accelerating the under-provisioned VP invocation without degrading DH's performance, as the Cases 1 and 2 in Fig. 1(b). Invocations utilize all their resources in Case 3, leaving no idle resources for harvesting. In a highly volatile serverless computing environment, resource harvesting must be proactive and precise to seize the opportunity to accelerate invocations efficiently, timely, and safely.

3 AN OVERVIEW OF LIBRA

3.1 Timeliness of Harvested Resources

Resource harvesting can benefit serverless computing significantly by improving resource utilization and function execution performance. However, such resource reassignment in Fig. 1(b) is not always feasible due to the natural *resource timeliness* in serverless platforms. *Timeliness* indicates that an invocation's resource allocation is available for harvesting only during its execution time. All resources allocated to an invocation will be released to the serverless platform once the invocation is completed or terminated, including harvested resources. Thus, an invocation utilizing harvested resources should follow the resource timeliness and release the harvested resources anytime when the source function invocation is done or terminated.

Fig. 2 demonstrates how an invocation with harvested resources should obey timeliness. We have two function invocations A and B arriving at the platform. Invocation *A* configured with two resource units starts at t_1 but can only utilize one resource unit. Invocation *B* configured with one resource unit arrives at t_2 while having an appetite for two units. To accelerate Invocation *B* without degrading *A*'s performance, we harvest one idle resource unit from *A* and reassign it to *B*. However, Invocation *A* finishes at t_4 and thus releases its two resource units immediately. Though Invocation *B* is not finished, it can no longer utilize the resource unit harvested from *A*. Instead, Invocation *B* continues executing with its own one unit until finishing at t_6.

The awareness of resource timeliness is the pivot to determine *when* to harvest and release resources from *which* invocation, and *how* to reassign the harvested resources for acceleration, making it imperative to estimate function invocations' execution time under different input data. Thus, we develop a profiler to estimate the dynamic execution time and resource demands of individual invocations (see §4).

3.2 Challenges

We design Libra to tackle three key challenges as follows:

How to estimate the resource demands and execution time of invocations with varying input data transparently? Transparent estimation of invocation resource demands and execution time is a must, as user code and input data content are protected from leakage. However, it's challenging to precisely predict the dynamic resource demands and execution time of function invocations under varying input data. We design a profiler that performs one-time offline profiling and online updating for efficient estimation without peeking into user code or input data content (§4).

How to harvest idle resources safely at a fine granularity? Resource harvesting in serverless computing is treading as if on thin ice since even slight over-harvesting easily deteriorates function executions' performance, particularly as resource allocation is fine-grained. We develop a harvest resource pool to manage harvested resources at a fine granularity (§5). Besides, a safeguard mechanism is devised to avoid overly harvesting and protect the performance of harvested functions (§5.2).

How to maximize cluster-wide invocation acceleration and idle resource utilization? Given the large scale of invocations, it is non-trivial to instantly identify workers with fine-grained idle resources of satisfactory volume and availability for the invocations. We design a decentralized sharing scheduler that timely utilizes cluster-wide idle resources to accelerate function executions (§6).

Fig. 3 depicts Libra's architecture with a five-step workflow:
Step ① : Deployment. Developers first upload the codebase of a function *k* to the serverless platform, and specifies a fixed resource allocation, such as, two CPU cores and 1,024 MB memory. User-defined allocation is the upper bound of resources that invocations to the function *k* can utilize.

Step (2): Invocation. The function k is called by per-defined events with varying input data. The *front end* accepts the function invocation and forwards it to the *profiler* for prediction.
Step (3): Profiling. The profiler transparently profiles the incoming function invocations to estimate the actual resource demands and execution duration (§4).
Step (4): Scheduling. A worker node m is selected to execute the incoming invocation based on a resource timeliness-aware scheduling algorithm. To enable large-scale deployment, Libra leverages multiple *decentralized sharding schedulers* that schedule function invocations concurrently (§6).
Step (5): Harvesting or accelerating. The *harvest resource pool* in the selected worker node m performs harvesting or acceleration based on predictions of the current invocation. When the execution completes, Libra collects the actual resource utilization and execution duration to update the corresponding profiling models of function k (§5).

4 PROFILING

4.1 Profiling Workflow

Libra profiles functions offline to characterize the relationships between input data and three metrics: CPU usage, memory usage, and invocation execution time. The profiler is activated upon a function's first invocation. If the function is first-time invoked, Libra serves the very first invocation with user-configured resources (Step (3a) in Fig. 3).

Meanwhile, Libra leverages the input data to produce a dataset for model training using *workload duplicator*. When the dataset is ready, Libra trains ML models offline to predict the above three function metrics. By analyzing the training metrics (*i.e.*, accuracy and R^2 score), we can identify whether the function's input sizes dominate its resource demands and execution time. If the function's demands are input size-related, we use the trained ML models for prediction (Step (3c)); if the function's demands are unrelated to input sizes, we treat future invocations as black boxes and conservatively predict the three metrics using histogram models from historical information (Step (3d)). If the invocation is not called at the first time, and the profiler has already initialized the models for this function, Libra calls either the ML or histogram models to infer the three metrics (Step (3b)).

After gathering three predicted metrics, the profiler forwards the invocation and the prediction result to a worker node for execution. For each function, the profiler builds the models by one-time training and analyzing whether input sizes dominate its demands and execution time. Subsequent invocations can reuse the built models.

4.2 Workload Duplicator

The workload duplicator augments datasets for training and analyzing the profiling model by duplicating input data. Upon the first invocation, the profiler collects the data and inputs it to its workload duplicator, where the data is duplicated uniformly to produce different sizes of data points. After duplicating data points, the profiler creates one invocation with sufficient resource allocation to execute the function for each data point in parallel. Libra analyzes results of all executions and obtains the actual CPU and memory usage, and response latency. The profiler then uses those actual results as ground truth to label each data point in the dataset. Thus, Libra generates a complete dataset for training ML models, where the ML models try to predict the three metrics and use the actual labels to minimize errors. We analyze the training metrics to identify whether input sizes dominate a function's demands and execution time. The ML models are used if the profiled function is input size-related (§4.3.1), otherwise we construct histogram models for input size-unrelated functions (§4.3.2).

4.3 Demand Estimator

4.3.1 For Input Size-related Functions. We formulate the CPU and memory usage prediction as a multi-class classification problem, where each allocation option is a separate class. Given a function invocation, the profiler predicts the actual CPU/memory *usage peak* based on the input data size. We define the usage peak as the highest number of busy cores or the highest amount of memory that can be used during the execution for CPU and memory, respectively. For the execution time prediction, Libra predicts a scalar value based on input data size, which is a common regression problem. Hence, the profiler trains and reserves three ML models per function, including two classification models and one regression model. Theoretically, any prediction model can work for the profiler. After examining different models, we opt for Random Forest (RF) regarding the prediction performance (§8.6).

The ML models cannot always capture a strong correlation between data size and CPU/memory peak and execution time. As opposed to data size, the execution performance of many functions is dominated by data content [10, 27]. Libra distinguishes between input size-related and unrelated functions by analyzing the training metrics of ML models (§8.6). Since serverless providers generally have no access to the secured data content, we treat new invocations as black boxes and predict the three metrics using histogram models constructed online from historical information.

4.3.2 For Input Size-unrelated Functions. Upon Libra identifying a function being input size-unrelated at its first invocation, we start to collect CPU/memory peak and execution time of the subsequent invocations online. Due to lacking knowledge of a black-boxed function, Libra requires a profiling window for collecting enough historical information to construct a distribution. During the profiling window, Libra serves the invocations with maximum allocation to inspect the actual CPU/memory peak and execution times. Libra builds three histogram models for each function based on the information collected during the window and continuously updates the models after serving new invocations. The histogram data structure tracks the distribution of CPU peak, memory peak, and execution time for each function. To estimate the three metrics for future invocations of a profiled function, we calculate a tail percentile on CPU/memory peak distribution and a head percentile on execution time distribution from the histogram models for conservatively harvesting or acceleration. To exclude outliers, we follow the industrial convention [36] and use the 99^{th}- and 5^{th}- percentiles to estimate CPU/memory peak and execution time, respectively.

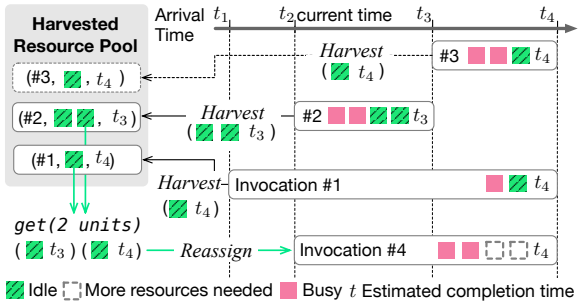

Figure 4: The harvest resource pool for tracking resources.

5 RESOURCE HARVESTING

5.1 Harvest Resource Pools

Libra maintains a *harvest resource pool* to track idle resources harvested from over-provisioned function invocations and record the priority of harvested resources for reassignment. Fig. 4 illustrates the harvest resource pool's tracking mechanism for idle resources. Each tracking object represents a harvested function invocation using a tuple of three elements: (`invo_id`, `hvst_resource_vol`, `priority`). The `invo_id` tracks the source function invocation of harvested resources. The `hvst_resource_vol` denotes how many resources (*e.g.*, two CPU cores or 128 MB memory) are harvested, and the `priority` indicates the order to utilize. We use the absolute timestamp at the completion of execution as the priority, where the timestamp equals to the sum of the current timestamp and predicted execution duration.

In Fig. 4, Invocation 1 arrives at the platform at t_1 with user-defined two resource units. Libra's profiler predicts that Invocation 1 can only utilize one unit (rectangle in red), leaving another unit idle (rectangle in green). Libra harvests the idle unit, takes the estimated completion timestamp of Invocation 1 as the priority, and issues a `put` operation to track the idle unit in the pool. The same occurs to Invocations 2 and 3 upon their arrivals. At the same time of Invocation 2's arrival, Invocation 4 arrives at the platform simultaneously, which has an appetite for four units but is only configured with two. At t_2, the pool has three available harvested resource units: one from Invocation 1 completing at t_4 and two from Invocation 2 completing at t_3. Note that Invocation 3 has not arrived at the platform yet, thus there are only Invocations 1 and 2 in the pool at t_2. Libra issues a `get` operation to borrow two idle units from the pool to accelerate Invocation 4. The harvest resource pool then provides one unit from Invocation 1 and one unit from Invocation 2, as Libra prioritizes harvested resources that can potentially be utilized longer. The reassignment takes effect immediately when Invocation 4 starts execution.

The harvest resource pool's features are as follows:

Essential operations. The harvest resource pool has two essential operations inherited from a standard hash map: `put` and `get`. Libra puts a harvested resource into the pool by specifying the invocation ID, the resource volume that is harvested, and the priority. To accelerate invocations, Libra gets harvested resources from the pool in a best-effort manner by specifying the desired volume.

Priority. Libra assigns a *priority* to each object in the pool for tracking, where objects with larger priorities are utilized earlier to accelerate other function invocations. We set the *estimated completion timestamp* of the harvested invocation as the resource's priority, where Libra estimates the completion timestamp of executing each function invocation using the predicted execution time from the profiler. Intuitively, we design Libra to prioritize harvested resources that potentially stay longer in the pool for function acceleration.

Preemptive release. When the harvested function invocation completes its execution, the harvested resources are no longer valid for use in acceleration. Therefore, Libra must adopt a *preemptive release* operation to free up the harvested resources in realtime either from the harvest resource pool or other invocations that are currently utilizing them. Libra can accelerate one function invocation using harvested resources from multiple invocations with varying timeliness. With harvested resource tracking, Libra can precisely release harvested resources in different contexts and avoid violating resource validity.

Re-harvesting. A typical accelerated function invocation possesses two kinds of resources: resources owned by itself and resources harvested from other invocations. When an accelerated invocation completes the execution, the resources owned by itself are released. However, the harvested resources are still available for accelerating other function invocations. Libra re-harvests the resources and stores them in the harvest resource pool for reassignment. When re-harvesting the resources, we set the priorities to be the initial estimated completion time for re-entering the pool.

Concurrency. Harvested resources can only be accessed by one function invocation at a time, and thus the pool must maintain the same view for all function invocations. Our harvest resource pool achieves atomic resource operations with mutex exclusion.

Mitigating Out-of-Memory (OOM). We use several methods to jointly mitigate the OOM problem during memory harvesting. *First*, we set a lower bound of memory for each function so that Libra needs to reserve minimum memory for harvesting invocations. *Second*, we use a safeguard (§4) for every container to detect the memory usage bulk and try to release harvested memory back in advance. *Finally*, we stop harvesting memory for functions that frequently get safeguard triggered due to OOM problems and retreat to user-defined memory allocation.

5.2 Safeguard

Libra estimates the actual resource demands of an incoming function invocation using ML and histogram models. However, models' potential misprediction might lead to performance degradation—function invocations' performance degrades due to resource over-harvesting. Libra adopts a *safeguard* mechanism to avoid such performance degradation. When detecting an approximated performance drop from a function invocation with resource harvested, Libra immediately returns all harvested resources to the invocation using the preemptive release operation.

When receiving an incoming function invocation, Libra executes the function using a container. During the execution, Libra continuously monitors the container's resource usage in a monitor window (*e.g.*, 100 ms). If the usage approaches a certain threshold (*e.g.*, 80%), Libra immediately returns all resources harvested from the invocation (either cached in the harvest resource pool or being used by other function invocations) using preemptive release.

6 FUNCTION SCHEDULING

6.1 Complexity of Timeliness-aware Scheduling

Libra orchestrates multiple worker nodes and utilizes harvested resources of each worker node to accelerate bursty and highly concurrent function invocations. The function scheduling problem for Libra to fully utilize harvested resources across multiple worker nodes can be simplified and reduced to the dynamic bin-packing problem [13], which is NP-hard in general. It is more challenging to find an optimal solution in the case of online setting [8]. The scheduling algorithm must be low-latency, lightweight, and timely to accommodate the burstiness, high concurrency, and short lifetime of function invocations. Besides, to enable efficient resource timeliness-aware scheduling, the algorithm should be able to process fine-grained harvested resource information from a large scale of worker nodes efficiently. Hence, we opt for a heuristic algorithm that greedily assigns a function to a worker node with the maximum availability in both resource volume and timeliness.

6.2 Demand Coverage

The resource availability lies in two dimensions: volume and timeliness. Thus, we use the product of resource volume and available duration to indicate the resource availability (*e.g.*, 2 CPU cores × 10 seconds). We devise a new metric, *demand coverage*, which is defined as the ratio of how much of the invocation's resource demands can be satisfied by a worker node's harvested resources. We monitor the status of harvest resource pools in each worker node and calculate a per-node demand coverage ratio for the invocation. Fig. 5 shows an example to calculate demand coverage, where a harvest resource pool tracks idle resource collections *a*–*e* harvested from different invocations, and an incoming invocation demands two extra resource units from t_3 to t_7. We count the entire *d* from t_3 to t_5 and only part of *e* from t_5 to t_7 when calculating the demand coverage for the invocation. We compute a separate CPU and memory demand coverage ratio for each harvest resource pool. Finally, we calculate a weighted demand coverage $D := \alpha \times D_c + (1-\alpha) \times D_m$, where D_c and D_m denote CPU and memory demand coverage, and $\alpha \in [0, 1]$ is the weight that controls the contribution of each type of demand coverage. In general, we define $\alpha > 0.5$ to represent that harvested idle CPU cores are more precious than memory.

6.3 Algorithm Design

The scheduler first classifies incoming function invocations into two categories based on their user-defined resources (*i.e.*, `user_cpu` and `user_mem`) and actual resource demands estimated by the profiler (*i.e.*, `pred_cpu` and `pred_mem`):

- **Non-accelerable invocations**, which cannot be accelerated with any extra resources since their user-defined resources can fully cover or beyond their actual demands.
- **Accelerable invocations**, which can be accelerated by extra resources (CPU or memory) since their actual demands are beyond their user-defined resources.

For non-accelerable invocations, the scheduler assigns a function's invocations to the same worker node via a hashing algorithm, which reduces the cold-starts[4] of invocations by reusing containers on the same worker node. If the worker node does not have enough available resources, the hashing algorithm will be executed again to locate another available worker node, to which this function's upcoming invocations will be assigned.

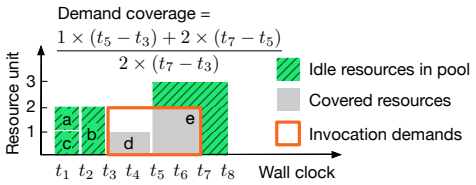

Figure 5: An example of demand coverage calculation.

For accelerable invocations, the scheduler first selects a set of worker nodes with sufficient available resources to satisfy the invocations' user-defined resource demands. Then, the scheduler calculates demand coverage ratios of CPU and memory for each selected worker node. Finally, the scheduler greedily searches for a worker node with the maximum weighted demand coverage ratio to execute the invocation.

6.4 Decentralized Sharding Scheduler

To serve large-scale concurrent function invocations, the design of Libra's scheduler must be low-latency (*i.e.*, sub-second) and efficient. A centralized scheduler that handles invocations one by one is impractical in production, which can easily pile up queued invocations and become the bottleneck when a serverless cluster scales to hundreds of worker nodes. We employ *decentralized sharding schedulers* in Libra to extend scalability for large serverless clusters. Libra manages a set of schedulers that distribute function invocations to multiple worker nodes. The capacity (*i.e.*, CPU and memory) of a node is evenly sharded among all schedulers, meaning that each scheduler has access to a slice of every worker node. For example, if a node with 32 CPU cores and 32 GB is sharded between four schedulers, each scheduler controls access to 8 cores and 8 GB on this node. Though the capacity of each node is horizontally sharded, every scheduler can observe the same demand coverage for a node as a whole, *i.e.*, concurrent schedulers can accurately capture the locality of harvested resources. The idea behind such decentralized sharding is simple—schedulers no longer need to share any data for synchronization. Synchronizing shared states among multiple schedulers is a costly operation for scheduling invocations in serverless platforms. Querying every node when scheduling an invocation is also impractical in large-scale clusters. Thus, Libra embeds the status information of harvest resource pool in nodes' health ping messages. The piggyback trick significantly reduces the scheduling overhead—instead of querying every node, schedulers only need to query their local data to compute the coverage for selecting a node.

7 IMPLEMENTING LIBRA

Libra provides a general resource management service for functions in serverless platforms. For concreteness, we describe its implementation in the context of Apache OpenWhisk [6], and the changes needed in OpenWhisk. We implement Libra using 2K lines

[4]Initialization delay of creating a container and installing dependencies for function execution.

of Python code for the profiler and user client, and 6K lines of Scala code for OpenWhisk built-in components.

Frontend. OpenWhisk only allows users to define the memory limit of their functions and allocates CPU power proportionally based on memory. To decouple CPU and memory, we add a CPU argument and enable the front end to take user-input CPU and memory configurations.

Profiler. We implement the profiler using Python. The workload duplicator uses the `multiprocessing` library for parallel executing functions and analyzing results. We build and train the profiler's ML models using the `scikit-learn` library [31].

Function scheduler. We implement Libra's timeliness-aware scheduling algorithm and decentralized sharding schedulers based on OpenWhisk's built-in load balancer. We embed the status information of harvest resource pool in the invoker's health ping message. The controller periodically receives the pool status information, which is further used to compute resource coverage when serving an invocation.

Harvest resource pool. OpenWhisk runs a container pool on each worker node to manage containers independently. We implement the harvest resource pool in OpenWhisk's container pool module. We use the immutable hash map from Scala standard library to implement the atomic access control for the harvest resource pool.

Preemptive release. We use `docker-update` API from the Docker library to implement the preemptive release operation. The API can update the CPU and memory configuration for multiple containers in real time [17]. Similar to OpenWhisk's native Docker interfaces, we wrap the `docker-update` using Scala code so that Libra can call it asynchronously.

Safeguard. We implement the safeguard as a daemon process running inside OpenWhisk container runtimes. When a container receives an invocation and starts execution, it first activates the safeguarding process. If the resource usage exceeds the safeguard threshold while executing the code, the daemon process immediately sends a request back to OpenWhisk's container pool. The container pool calls a preemptive release operation to return the harvested resources. After the execution is done, the container deactivates the daemon process. We employ Linux `cgroups` tools [25] to monitor containers' CPU and memory utilization at runtime.

8 EVALUATION

We implement a prototype of Libra using 2K lines of Python code on the profiler and user client, and 6K lines of Scala code on other components in OpenWhisk [6]. We deploy and evaluate Libra on three clusters. Libra will be open-sourced after review.

8.1 Evaluation Metrics

We use two metrics to evaluate Libra's performance: *function response latency* and *system resource utilization*.

Function response latency is the end-to-end response time from invoking the function until receiving the execution result, dictating serverless service user experience. Specifically, we use

$$speedup := \frac{t^{user} - t^{libra}}{t^{user}} \tag{1}$$

as a unified metric to quantize the performance of how Libra improves a function invocation, where t^{user} indicates the response latency of a function invocation executed with the user-defined resource allocation r_u, and t^{libra} indicates the latency of an invocation executed by Libra. The speedup shows how Libra affects the performance of a function invocation. Intuitively, a positive speedup indicates the invocation is accelerated, a negative speedup indicates the invocation is slowed down (*e.g.*, due to inappropriate resource harvesting), and a zero speedup indicates that the invocation preserves its performance.

Table 1: Characterizations of serverless applications used in OpenWhisk evaluation. (UL: Uploader, TN: Thumbnailer, CP: Compression, DV: DNA Visualization, DH: Dynamic HTML, VP: Video Processing, IR: Image Recognition, GP: Graph Pagerank, GM: Graph Minimum-spanning-tree, GB: Graph Breadth-first-search.)

Input Size	Func.	Description
Related	UL	Upload input files to storage
	TN	Thumbnail input images
	CP	Compress input files
	DV	Visualize input DNA sequence files
	DH	Generate HTMLs from input templates
Unrelated	VP	Generate GIF of an input video
	IR	Recognize an input image
	GP	Pagerank a randomly generated graph
	GM	MST on a randomly generated graph
	GB	BFS on a randomly generated graph

System resource utilization measures how efficient the serverless computing platform can utilize the hardware resources, which is calculated as

$$sys\_util := \frac{utilized\_resources}{available\_resources}, \tag{2}$$

where *utilized_resources* and *available_resources* indicate the resources utilized by function invocations and the total available resources for users, respectively.

8.2 Experiment Setup

8.2.1 ***Testbeds***. We evaluate Libra on three OpenWhisk clusters:

Single-node cluster: The single-node cluster has three nodes, including one client for invoking functions, one controller that hosts OpenWhisk components, and one worker with 72 Intel Xeon E5-2670 CPU cores and 72 GB of memory for executing functions.

Multi-node cluster: A six-node cluster includes one client node, one controller node, and four worker nodes. Each worker node provides 32 Intel Xeon E5-2420 CPU cores and 32 GB memory for executing functions.

Jetstream cluster: A 50-node cluster using Jetstream [40, 43], which is a cloud computing environment for scientific research. Each node has 24 Intel Xeon E5-2680 CPU cores and 24 GB memory for executing functions.

8.2.2 ***Workloads***. We sample eleven function invocation trace sets from Azure Functions traces [36] for evaluation:

- One **single** trace set for single-node cluster evaluation. The single set consists of 165 function invocations.
- Ten **multi** trace sets for multi-node cluster evaluation. The ten multi sets consist of in total 1,050 function invocations with invocation frequency increasing from 10 to 300 request per minute (RPM). Note that 95% of the functions running

on Azure Functions have 60 RPM or less [36]. We assume 300 RPM is sufficiently high for realistic serverless traces.

We employ a real-world serverless benchmark suite, SeBS [14], to conduct a realistic evaluation. Table 1 characterizes ten applications from the SeBS benchmark suite. All functions are implemented in Python. We set the initial resource configuration of each function according to the default settings from the suites. Since the SeBS benchmark suite does not provide input datasets, we collect data points from real-world datasets as input data for invoking the ten functions. Specifically, we randomly sample 100 pictures from the CIFAR-100 dataset [26] for TN and IR. We randomly sample 100 videos from the YouTube-8M dataset [1] for UL, CP, and VP. We use genome sequences of Bacillus subtilis from NCBI dataset [47] for DV. We randomly sample 100 different graphs from igraph [15] for GP, GM, and GB as input data.

8.2.3 ***Libra's settings.*** We implement ML models using two RF classifiers and one regressor with scikit-learn library [31], and histogram models with NumPy [24] in the profiler. Profiler's workload duplicator scales and duplicates the input data in a uniform distribution with a maximum of 100 times. We use the created dataset to train three initial RF models in Libra's profiler for prediction. The models capture function-specific patterns such as resource usage and execution time. Hence, the function (or application) code is the same for evaluation and development since models are built per function. However, the training data (*e.g.*, function-specific patterns for training) for model development and testing data (*e.g.*, function-specific patterns during evaluation) for evaluation are different. We split the datasets into a 7:3 ratio for training and testing. All the testing data (patterns) are not exposed during the model development phase. The ten functions are configured with eight CPU cores and 1,024 MB memory in offline profiling, which is the maximum allocation for each function in our experimental environment. We set the safeguard threshold to be 0.8 and the demand coverage weight to be 0.9 for Libra in our evaluation.

8.3 Effectiveness of Libra's Harvesting

We compare Libra with two existing resource managers for serverless platforms and three variants of itself on the single-node cluster: 1) *Default*, the default resource management in OpenWhisk (also in existing serverless platforms) that allocates user-defined resources to functions. The resource allocation stays fixed during individual function executions, and all invocations of the same function receive a fixed amount of resources. 2) *Freyr*, a state-of-the-art serverless resource management platform that uses DRL to harvest idle resources and accelerate function executions [49]. We implemented Freyr based on its open-sourced code repository and trained the models following the algorithms described in its paper using the same workloads in our evaluation. 3) Libra-NS (No Safeguard). A variant of Libra without safeguard mechanism. We turn off the safeguard daemon when Libra executes function invocations. 4) Libra-NP (No Profiler). A variant of Libra without profiler. This variant does not have a profiler to predict three metrics (*i.e.*, CPU usage peak, memory usage peak, and execution time). Instead, it uses a moving window to determine three metrics. Every function has a moving window for monitoring invocation history. The moving window keeps track of n latest invocations and takes the maximum

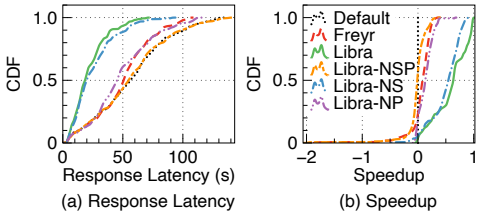

Figure 6: CDFs of comparing Libra with two existing serverless platforms and three variant.

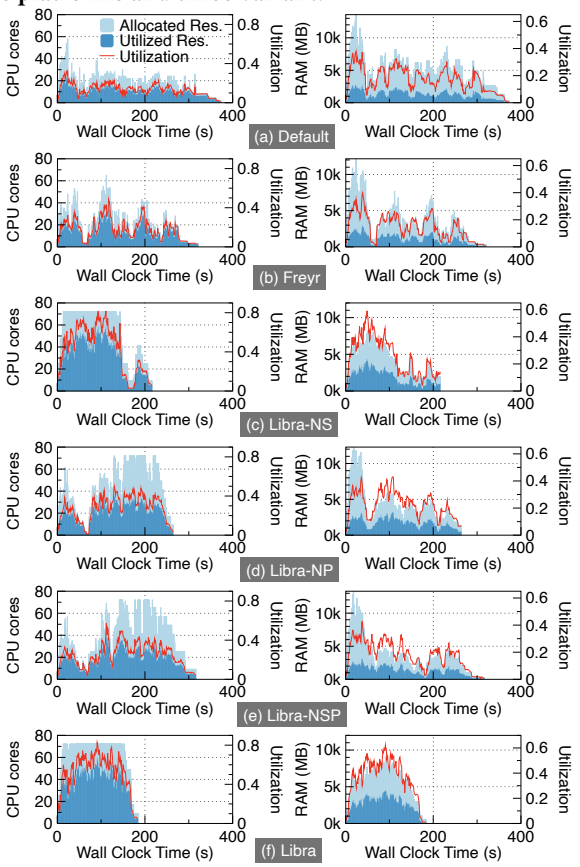

Figure 7: CPU and memory utilization of six platforms through the experiment timeline.

CPU usage peak, memory usage peak, and execution time as the decision for the next incoming invocation. We set the window size to five for each function in the experiment. 5) *Libra-NSP* (no Safeguard and Profiler), a variant of Libra without safeguard mechanism and the profiler. We run six platforms using the same single trace set and present the results averaged over five times of experiments.

8.3.1 ***Comparisons with State-of-the-arts.*** We first compare Libra with two existing resource managers for serverless platforms, default OpenWhisk and Freyr, to evaluate the performance.

Response latency. Fig. 6(a) shows the CDF of function response latency of three platforms. Libra outperforms the other two platforms because of carefully harvesting and accelerating function invocations with resource timeliness awareness. Libra reduces the 99th-percentile of the same workload by 50% and 39% compared to OpenWhisk default and Freyr, respectively.

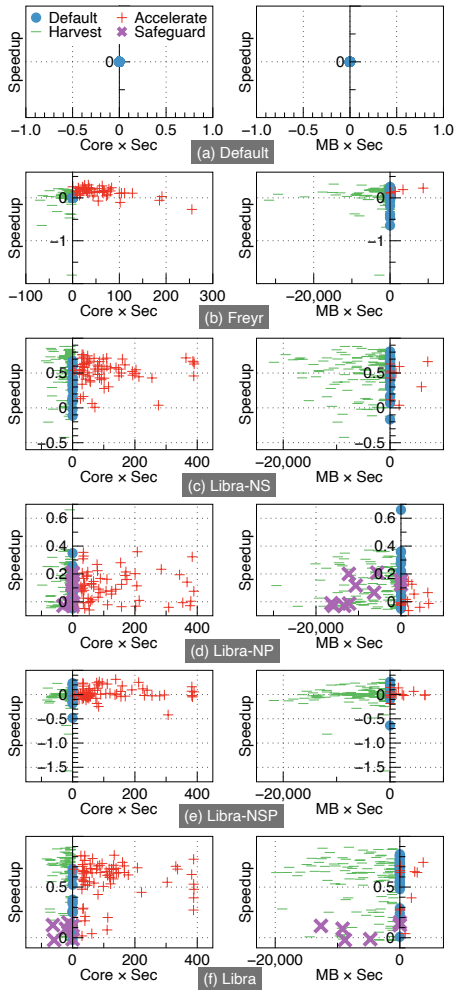

Figure 8: Performance of individual invocations processed by six platforms. Default (•): invocations with user-requested allocation. Accelerate (+): invocations accelerated by supplementary allocation. Harvest (−): invocations with resource harvested. Safeguard (×): safeguarded invocations.

Speedup and performance degradation. Fig. 6(b) shows the CDF of execution speedup of three platforms. Libra outperforms the other two by providing faster function invocation executions without significantly degrading performance. Invocations processed by OpenWhisk default have no speedup or degradation due to default is the baseline. Libra degrades invocation performance with 2% at worst regarding response latency, whereas Freyr suffers at worst 180% performance degradation.

System utilization and workload completion time. Fig. 7 presents the CPU and memory utilization of three platforms through the experiment timeline. Libra utilizes CPU and memory more efficiently than the other two platforms. Specifically, Libra achieves 3.82×/2.09× and 2.93×/2.48× average CPU/memory utilization compared to OpenWhisk default and Freyr, respectively. Correspondingly, Libra completes the workload 51% and 43% faster.

Harvesting and acceleration. Fig. 8 shows the resource reassignment details of all invocations processed by three platforms. We use a product of reassigned resources and occupied time to present the two-dimensional impact of resource reallocation on each invocation. OpenWhisk default has no resource adjustment during workload processing. Freyr achieves poor performance on both harvesting and acceleration without awareness of harvested resource timeliness. Libra offers careful harvesting and better acceleration performance with higher speedups for invocations.

8.3.2 ***Ablation Study***. We then perform an ablation study to examine the effectiveness of two key components in Libra: the profiler and safeguard. We compare Libra with three variants: Libra-NS, Libra-NP, and Libra-NSP.

Response latency. Fig. 6(a) shows the function response latency CDF of four Libra variants. Libra outperforms the other variants due to being fully equipped with profiler and safeguard. Libra reduces the 99th-percentile of the same workload by 15%, 30%, and 34% compared to Libra-NS, Libra-NP, and Libra-NSP, respectively.

Speedup and performance degradation. Fig. 6(b) shows the execution speedup CDF of four variants. Libra outperforms the other three variants by providing faster function invocation executions without significantly degrading harvested invocations' performance. Libra and Libra-NP degrade execution performance 2% and 6% at worst regarding response latency. Compared to Libra and Libra-NP, Libra-NS and Libra-NSP suffer at worst 42% and 197% performance degradation.

System utilization and workload completion time. Fig. 7 shows the CPU and memory utilization of four Libra variants through the experiment timeline. Libra utilizes CPU and memory more efficiently than the other three variants. Libra achieves 1.21×/1.40×, 1.84×/1.60×, and 2.05×/2× average CPU/memory utilization compared to Libra-NS, Libra-NP, and Libra-NSP, respectively. It completes the workload 17%, 30%, and 42% faster.

Harvesting and acceleration. Fig. 8 shows the performance of all invocations processed by four variants. Libra and Libra-NS provides more precise harvesting and faster execution acceleration (higher speedups) with profiler's accurate predictions, whereas Libra-NP and Libra-NSP accelerates function invocations less (lower speedups). Note that due to profiler's predictions, Libra has less safeguarded invocations than Libra-NP. Libra and Libra-NP have some invocations protected by the safeguard daemon, resulting in limited performance degradation. In contrast, invocations handled by Libra-NS and Libra-NSP can experience serious performance degradation without safeguard protection.

8.4 Effectiveness of Libra's Scheduling

We then deploy Libra on the multi-node cluster to evaluate the effectiveness of its scheduling algorithm. We compare Libra's scheduling algorithm with four baselines: 1) *Default*. The default scheduling algorithm inside OpenWhisk. OpenWhisk controller calculates a unique hash key for each function and always schedules invocations under the same function to the same node. 2) *Round Robin (RR)*. A classic yet popular load balancing algorithm that distributes the load by sending successive requests to different invokers in a cyclical manner. 3) *Join-the-Shortest-Queue (JSQ)* [23]. A well-known load balancing algorithm that effectively reduces queueing time and resource contention by sending incoming invocation to the node with the least pending jobs. 4) *Min-Worker-Set (MWS)* [50]. A state-of-the-art scheduling algorithm dynamically schedules invocations

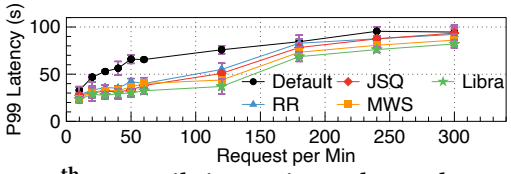

Figure 9: 99th-percentile invocation end-to-end response latency of five scheduling algorithms.

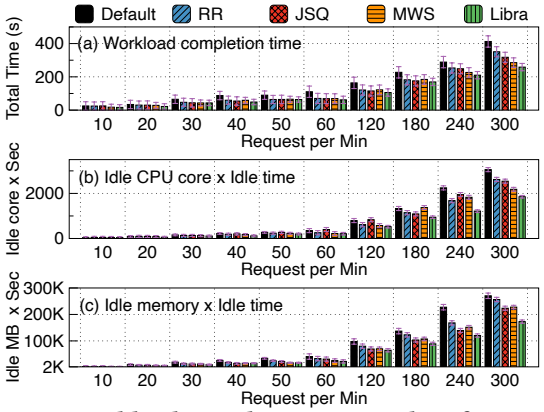

Figure 10: Workload completion time and performance of utilizing harvested resources of five scheduling algorithms.

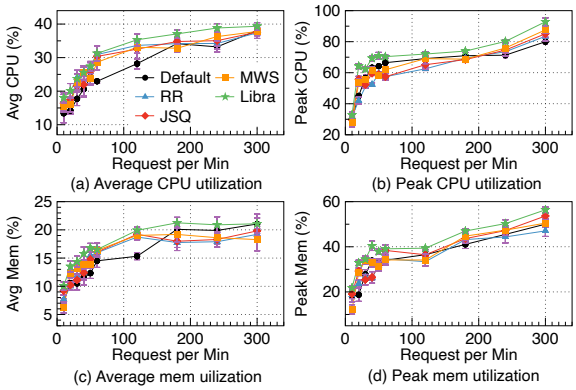

Figure 11: Average/peak CPU and memory utilization.

to the node with the least resource pressure. We enable the cluster with Libra's function harvesting and acceleration when evaluating all five algorithms for a fair comparison on scheduling. We evaluate five algorithms by running the ten multi trace sets sequentially and report the results averaged over five times of experiments.

P99 response latency. Libra consistently achieves the lowest P99 latency for all traces (Fig. 9).

System utilization. Fig. 11 reports the average/peak CPU and memory utilization of five scheduling algorithms. Libra generally maintains the highest CPU and memory utilization among all baselines for ten trace sets.

Workload completion time. We define workload as a collection of multiple incoming function invocations, and workload completion time as the time from invoking the first invocation until the last invocation completes. From Fig. 10(a), Libra outperforms the other four algorithms by completing workloads faster.

Idle time of harvested resources. Our evaluation keeps track of every harvested resource's entry time and leave time in harvest resource pools. We define idle time as the time when harvested

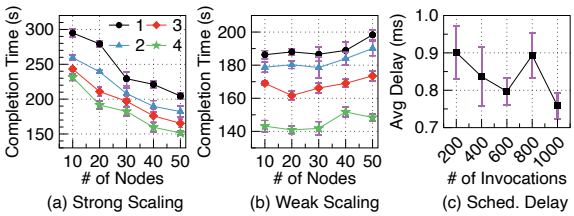

Figure 12: Libra's scalability and scheduling overhead.

resources staying in the pool without any invocations actually utilizing them. We then sum up the values of all the product of harvested resources and their idle time to indicate the performance of how a scheduling algorithm utilizes the harvested resources. Intuitively, a lower value indicates a better utilization of harvested resources for a scheduling system. Fig. 10(b) and (c) show that Libra generally maintains the lowest value among all algorithms for all traces. It makes the best use of harvested resources.

8.5 Scalability

We study the *strong scaling* and *weak scaling* of decentralized sharding schedulers in Libra using the Jetstream cluster. Strong scaling evaluates Libra's performance on increasing worker nodes when given a fixed total number of function invocations; and weak scaling evaluates the performance on increasing worker nodes when the average number of invocations distributed to each node is fixed. We gradually increase the number of Libra's schedulers from one to four to examine the effectiveness of increasing concurrent schedulers. Note that one scheduler is exactly a centralized version. We use the same workload that consists of ten real-world functions used in previous experiments to measure the scalability of Libra. Each function has the number of invocations evenly divided from the total number. We keep the initial resource configuration the same as the previous experimental setup. We enable Libra's harvesting and timeliness-aware scheduling to evaluate the scalability in realistic scenarios.

Strong scaling. We launch 1000 concurrent invocations where each function is invoked 100 times simultaneously. Fig. 12(a) shows the performance of strong scaling when Libra gradually increases the number of schedulers from one to four. The completion time of workload decreases with the number of schedulers increasing.

Weak scaling. We set the average number of invocations assigned to each worker as 20 and evaluate the weak scaling of Libra, which means that 200 concurrent invocations for 10 nodes and 1000 for 50 nodes. Fig. 12(b) shows the trend of completion time does not rise significantly when workload intensity increases.

Scheduling overhead. We define the scheduling overhead of an invocation as the time when a scheduler picks it up until sending it to a node. We measure the scheduling overhead averaged over concurrent invocations using the 50-node cluster with four schedulers. Fig. 12(c) shows the average overhead with workload intensity increasing from 200 to 1000. Libra maintains the overhead under 1 ms consistently.

8.6 Profiler's Model Analysis

Metrics. We adopt two metrics, *accuracy* and R^2 *score*, to evaluate the performance of multi-classification and regression models, respectively. For classification problems, accuracy measures the

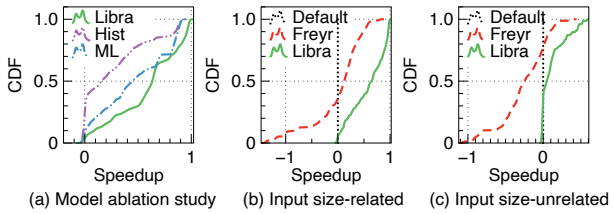

Figure 13: Model ablation study and input size sensitivity.

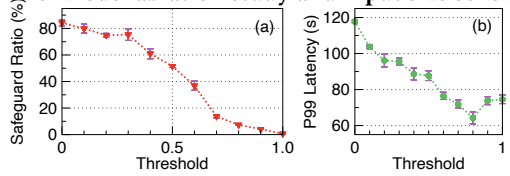

Figure 14: Sensitivity analysis of safeguard thresholds.

fraction of predictions from models that are correct. A higher accuracy generally means a better model. For regression problems, R^2 score measures the proportion of the variation in the dependent variable that is predictable from the independent variable. An R^2 score at 1.0 indicates two variables are perfectly correlated.

Models. We examine four popular ML models for each task. For CPU/memory usage and execution time prediction, we evaluate Logistic/Linear Regression (LR), Support Vector Machines (SVM), Neural Networks (NN), and Random Forest (RF). All models are tuned with hyperparameter searching from scikit-learn library [31]. We use the collected ten functions and realistic datasets in §8.2 to evaluate four models. The datasets are split in a 7:3 ratio for training and testing.

Prediction performance. Table 2 reports the accuracy and R^2 score of four models for each workload function. RF outperforms the other three models for all tasks regarding average accuracy and R^2 score. Specifically, for CPU usage prediction, RF's accuracy is 6%, 8%, and 2% higher than LR, SVM, and NN, respectively. Four models show less variance for memory usage prediction, where RF's accuracy is 1%, 1%, and 2% higher than LR, SVM, and NN, respectively. For execution time prediction, RF's R^2 score is significantly higher than others and the closest to 1.0.

Input size-related and unrelated functions. Input size-related and unrelated functions show a significant difference. We can draw a boundary between the two types by specifying a threshold on the accuracy and R^2 score. For example, we may use a 0.9 accuracy and a 0.9 R^2 score as indicators to distinguish whether a function is input size-related.

Model ablation study. Fig. 13(a) presents the speedup CDF of Libra and two variants, Libra with only histogram models (Hist) and Libra with only ML models (ML), when running the workload on the single-node cluster. Libra outperforms the two variants for handling workloads that include both types of functions.

Prediction overhead. In our experiments, Libra's predictions introduce an average prediction overhead of less than 2 ms. The overhead is negligible compared to the execution time of most serverless functions, as 75% of functions on Azure Functions execute for at least 1 second [36].

Training time. The offline training to initialize a model takes less than 120 ms, while the online training on an existing model takes less than 1 ms. Both offline training and online training incur trivial overhead compared to the execution times of most functions.

Table 2: Comparison on ten functions with four different machine learning models. Metrics include: CPU usage prediction accuracy/Memory usage prediction accuracy/R^2 score of execution time prediction.

Func.	LR	SVM	NN	RF
UL	0.76/**1.0**/**0.93**	0.70/**1.0**/0.91	0.84/0.97/0.53	**0.85**/**1.0**/0.92
TN	0.91/0.92/0.92	0.94/0.96/0.77	0.97/0.96/0.22	**0.98**/**0.97**/**0.93**
CP	0.90/1.0/0.94	0.87/1.0/0.89	0.92/0.98/0.35	**0.95**/**1.0**/**0.97**
DV	**1.0**/0.99/0.89	**1.0**/0.98/-5.65	0.99/0.97/-0.36	**1.0**/**1.0**/**0.95**
DH	0.89/**1.0**/0.65	0.87/0.97/-475	0.93/**1.0**/-21	**0.95**/0.99/**0.92**
Avg.	0.89/0.98/0.86	0.87/0.98/-95	0.93/0.97/-4.14	**0.95**/**0.99**/**0.94**
VP	0.45/0.20/-2.29	0.52/0.52/-34	**0.62**/**0.67**/-15	0.58/0.63/**-0.14**
IR	0.47/0.42/-5.99	0.43/0.49/-254	0.49/0.57/-88	**0.65**/**0.59**/**-4.40**
GP	0.51/0.47/-0.07	0.53/0.59/-0.13	0.56/0.65/-0.08	**0.61**/**0.69**/**-0.06**
GM	0.52/0.46/-0.05	0.61/0.49/-0.05	0.58/0.46/-0.20	**0.62**/**0.50**/**-0.04**
GB	0.41/0.56/-0.02	**0.47**/0.46/-0.04	0.43/0.63/-0.07	**0.47**/**0.66**/**-0.01**
Avg.	0.47/0.42/-1.68	0.51/0.51/-57	0.53/0.60/-20	**0.59**/**0.61**/**-0.93**

8.7 Input Size Sensitivity

We investigate Libra's sensitivity of input sizes via the three types of workloads: hybrid, input size-related, and input size-unrelated. The hybrid workload includes all ten functions in Table 1, the input size-related workload consists of five functions (UL, TN, CP, DV, and DH), and the input size-unrelated workload consists of the other five (VP, IR, GP, GM, and GB). All functions' initial configurations follow the same settings of experiments in §8.3.

Fig. 6(b), Fig. 13(b), and Fig. 13(c) show the speedup CDF of the OpenWhisk default, Freyr, and Libra when running the hybrid, input size-related, and input size-unrelated workloads, respectively. Libra offers the best performance running the input size-related workload, which accelerates 99th-percentile of invocations by 94% and 58% over default and Freyr, respectively. For the hybrid workload, Libra accelerates 99th-percentile of invocations less by 50% and 39% over default and Freyr, respectively. Libra provides the least performance gain for the input size-unrelated workload, which still improves by 13% and 12% over default and Freyr, respectively. The more input size-related functions in a workload, the lower is Libra's performance. Nevertheless, Libra still improves input size-unrelated functions' performance.

8.8 Parameter Sensitivity Analysis

Safeguard's threshold. We set the default threshold value to be 0.8 in Libra's safeguard, allowing invocations to trigger safeguard just before detecting a full utilization. We conduct a sensitivity check by running Libra with the single trace set on the single-node cluster. Fig. 14(a) shows the ratio of invocations that safeguarded by Libra with the threshold increasing from 0 to 1 in the step of 0.1. Fig. 14(b) shows the P99 function response latency of each run. The safeguarded invocation ratio drops with threshold increasing as Libra gradually harvests idle resources wildly. Due to degraded performance and limited safeguarding, Libra performs worse for thresholds beyond 0.8.

Demand coverage weight. We set the default weight to be 0.9 in Libra's demand coverage calculation, letting CPU demand coverage contribute much larger than memory coverage. We conduct a sensitivity check on the weight by running Libra using the multi trace set with 120 RPM on the multi-node cluster. Fig. 16(a) shows

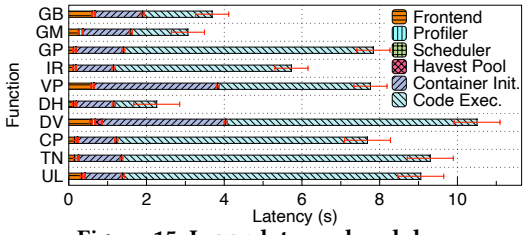

Figure 15: Libra latency breakdown.

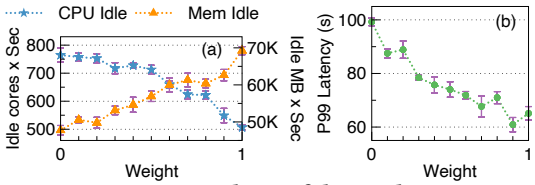

Figure 16: Sensitivity analysis of demand coverage weight.

the CPU/memory idle values with weight increasing from 0 to 1 in the step of 0.1. Fig. 16(b) shows the P99 latency of each run. When the weight increases, CPU coverage gradually contributes more to the total weighted coverage, and memory coverage contributes less. This results in CPU idle value decreasing and memory idle value increasing. Libra with 0.9 coverage weight achieves the lowest P99 function response latency.

8.9 Latency Breakdown

Fig. 15 shows the latency breakdowns of ten functions in evaluation. We run the ten functions in the same setting as multi-node experiments. Libra's components incur negligible overhead compared to the container initialization time and function execution time.

8.10 Overheads of Libra's Components

We measured the overheads of Libra's individual components when running the same workloads from previous experiments on the multi-node cluster (§8.2.1). The profiler incurs 1.0 core CPU overhead and 268 MB memory overhead; the scheduler incurs 0.6 core CPU overhead and 134 MB memory overhead; the harvest resource pool incurs 0.2 core CPU overhead and 46 MB memory overhead. Running the whole Libra incurs less than 3% (5%) CPU (memory) overheads compared to the volume of harvested resources, which are negligible. The overheads will be even lower if Libra handles fewer function invocations.

9 RELATED WORK

Resource harvesting. A few studies proposed to harvest VM's idle resources due to users' static and inaccurate resource allocation. SmartHarvest [45] proposed a VM resource harvesting algorithm using online learning and offers a new low-priority VM service using harvested resources. Zhang *et al.* proposed an MWS scheduling algorithm that uses Harvest VMs to serve serverless computing [50]. In contrast, Libra harvests idle resources from function invocations in fine granularity and uses the harvested resources to accelerate function executions. Freyr is the closest work to Libra, yet being outperformed due to three key differences: *First*, Freyr is not aware of resource timeliness. Freyr can estimate the volume of resources that a function invocation needs, but when it harvests and reallocates resources from other functions, it ignores whether the harvested resources would be available throughout the whole execution. *Second*, Freyr's prediction does not explicitly capture input data size. Freyr uses a DRL agent to predict function invocation demands, where the observed states lack of input size information. *Third*, Freyr's safeguard is not timely safe. Unlike Libra's preemptive release, which returns harvested resources at function runtime, Freyr only resumes the resource allocation to the user-defined value for the next invocation, leaving the current invocation suffering from mis-prediction.

Caching. Faa$T [33] transparently scales caches based on data access patterns to speed up function invocations. Instead of data caching, Libra focuses on harvesting idle CPU and memory resources to accelerate function executions. OFC [29] is the closest work to Libra. OFC needs to manually craft features for different data types and request access to input data content for memory allocation estimation. However, Libra generalizes to different data types and requires no access to input content by optimizing function invocations as black boxes. OFC only harvests memory, whereas Libra jointly harvests CPU and memory and can be easily extended to other resource types.

User-side function configuration. Many studies optimized single serverless function resource configuration from the user side such as [2, 18, 27, 30, 46]. Libra is a provider-side solution that provides transparent resource reallocation from the view of providers instead of tuning user configuration.

Function invocation scheduling. Many existing studies aimed to improve function execution performance [21, 38, 41, 48] and resource utilization [9, 10, 50] by designing novel scheduling algorithms for serverless computing. Existing scheduling solutions for serverless computing consider no harvested resources. While no existing scheduling solutions incorporate resource harvesting, we show that Libra's scheduling outperforms state-of-the-arts.

10 CONCLUSION

This paper proposed Libra, a new provider-side serverless computing technique that accelerates function invocations by harvesting idle resources safely and timely. Libra achieves both low function response latency and high resource utilization on real clusters with realistic workloads. Experimental results showed that Libra reduces function response latency by 39% and achieves 3× resource utilization compared to state-of-the-art solutions.

11 ACKNOWLEDGMENTS

We thank the anonymous HPDC shepherd and reviewers for their valuable input. The work of H. Yu, C. Fontenot, and H. Wang was supported by NSF CRII-OAC-2153502 and the AWS Cloud Credit for Research program. The work of J. Li was supported by NSF CRII-CNS-NeTS-2104880, RINGS-2148309, ARO W911NF-23-1-0072, and DOE DE-EE0009341. The work of X. Yuan was supported by NSF 1763620, 1948374, and 2146447. The work of S. Park was supported by NSF 2120248. This work used JetStream at IU through allocation CIS220024 from the Extreme Science and Engineering Discovery Environment (XSEDE), which was supported by NSF grant number 1548562. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. 2016. Youtube-8m: A Large-scale Video Classification Benchmark. *arXiv preprint arXiv:1609.08675* (2016).

[2] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring Serverless Functions using Statistical Learning. In *Proc. of the 2020 IEEE Conference on Computer Communications (INFOCOM)*.

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms With Adaptive Batching. In *Proc. of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–15.

[4] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[5] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)*.

[6] Apache. 2018. Apache OpenWhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org. [Online; accessed 1-May-2018].

[7] Arda Aytekin and Mikael Johansson. 2019. Harnessing the Power of Serverless Runtimes for Large-Scale Optimization. *arXiv preprint arXiv:1901.03161* (2019).

[8] Yossi Azar and Danny Vainstein. 2019. Tight Bounds for Clairvoyant Dynamic Bin Packing. *ACM Transactions on Parallel Computing (TOPC)* (2019).

[9] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan Narayanaswamy. 2021. FirePlace: Placing Firecraker Virtual Machines with Hindsight Imitation. *Proc. of Machine Learning and Systems (MLSys)* 3 (2021).

[10] Vivek M Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. 2022. Cypress: Input Size-sensitive Container Provisioning and Request Scheduling for Serverless Platforms. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*.

[11] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)*.

[12] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. FuncX: A Federated Function Serving Fabric for Science. In *Proc. of The 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 65–76.

[13] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. 1983. Dynamic Bin Packing. *SIAM J. Comput.* (1983).

[14] Marcin Copik et al. 2020. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. *arXiv preprint arXiv:2012.14132* (2020).

[15] Gabor Csardi, Tamas Nepusz, et al. 2006. The igraph Software Package for Complex Network Research. *InterJournal, complex systems* (2006).

[16] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. *ACM SIGPLAN Notices* (2014).

[17] Docker. 2021. Docker Update API. https://docs.docker.com/engine/reference/commandline/update/. [Online].

[18] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the Optimal Size of Serverless Functions. In *Proc. of the 22nd International Middleware Conference (MIDDLEWARE)*.

[19] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proc. of USENIX NSDI*.

[20] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-harvesting VMs in Cloud Platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[21] Alexander Fuerst and Prateek Sharma. 2022. Locality-aware Load-Balancing For Serverless Clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[22] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T Kandemir, and Chita R Das. 2020. Fifer: Tackling Underutilization in the Serverless Era. In *Proc. the 21st International Middleware Conference (Middleware)*.

[23] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. 2007. Analysis of Join-the-Shortest-Queue Routing for Web Server Farms. *Performance Evaluation* (2007).

[24] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array Programming with NumPy. *Nature* (2020).

[25] Heo, Tejun. 2021. Control Group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt. [Online; accessed 1-April-2022].

[26] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning Multiple Layers of Features from Tiny Images. (2009).

[27] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proc. of the 2021 USENIX Annual Technical Conference (USENIX ATC)*.

[28] Bilal Muhammad, Canini Marco, Fonseca Rodrigo, and Rodrigues Rodrigo. 2023. With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

[29] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. 2021. OFC: an Opportunistic Caching System for FaaS Platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*.

[30] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: A Graph Neural Network Based Proactive Resource Allocation Framework for SLO-Oriented Microservices. In *Proc. of the 17th International Conference on emerging Networking EXperiments and Technologies (CONEXT)*.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research (JMLR)* (2011).

[32] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. 2022. BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[33] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[34] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2022. Mashup: Making Serverless Computing Useful for HPC Workflows via Hybrid Execution. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[35] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proc. of IEEE/ACM MICRO*.

[36] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proc. of USENIX ATC*.

[37] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless Linear Algebra. In *Proc. of the 11th ACM Symposium on Cloud Computing (SoCC)*.

[38] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)*. 138–152.

[39] Tyler J Skluzacek, Ryan Wong, Zhuozhao Li, Ryan Chard, Kyle Chard, and Ian Foster. 2021. A Serverless Framework for Distributed Bulk Metadata Extraction. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[40] Craig A Stewart, Timothy M Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. 2015. Jetstream: a Self-provisioned, Scalable Science and Engineering Cloud Environment. In *Proc. of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure (XSEDE)*.

[41] Amoghavarsha Suresh and Anshul Gandhi. 2021. ServerMore: Opportunistic Execution of Serverless Functions in the Cloud. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)*.

[42] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. 2022. Owl: Performance-aware Scheduling for Resource-efficient Function-as-a-Service Cloud. In *Proc. of the 13th ACM Symposium on Cloud Computing (SoCC)*.

[43] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. 2014. XSEDE: Accelerating Scientific Discovery. *Computing in Science & Engineering* (2014).

[44] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *Proc. of IEEE INFOCOM*.

[45] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proc. of ACM EuroSys*.

[46] Zhaojie Wen, Yishuo Wang, and Fangming Liu. 2022. StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows. In *Proc. of*

the IEEE International Conference on Computer Communications (INFOCOM).

[47] David L Wheeler, Tanya Barrett, Dennis A Benson, Stephen H Bryant, Kathi Canese, Vyacheslav Chetvernin, Deanna M Church, Michael DiCuccio, Ron Edgar, Scott Federhen, et al. 2007. Database Resources of the National Center for Biotechnology Information. *Nucleic Acids Research* (2007).

[48] Hanfei Yu, Athirai A Irissappane, Hao Wang, and Wes J Lloyd. 2021. FaaS-Rank: Learning to Schedule Functions in Serverless Platforms. In *Proc. of IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS).*

[49] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2022. Accelerating Serverless Computing by Harvesting Idle Resources. In *Proc. of the ACM Web Conference (WWW).*

[50] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proc. of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP).*