Accelerating Serverless Computing by Harvesting Idle Resources

Hanfei Yu
hyu25@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 automates fine-grained resource scaling and simplifies the development and deployment of online services with stateless functions. However, it is still non-trivial for users to allocate appropriate resources due to various function types, dependencies, and input sizes. Misconfiguration of resource allocations leaves functions either under-provisioned or over-provisioned and leads to continuous low resource utilization. This paper presents *Freyr*, a new resource manager (RM) for serverless platforms that maximizes resource efficiency by dynamically harvesting idle resources from over-provisioned functions to under-provisioned functions. *Freyr* monitors each function's resource utilization in real-time, detects over-provisioning and under-provisioning, and learns to harvest idle resources safely and accelerates functions efficiently by applying deep reinforcement learning algorithms along with a safeguard mechanism. We have implemented and deployed a *Freyr* prototype in a 13-node Apache OpenWhisk cluster. Experimental results show that 38.8% of function invocations have idle resources harvested by *Freyr*, and 39.2% of invocations are accelerated by the harvested resources. *Freyr* reduces the 99th-percentile function response latency by 32.1% compared to the baseline RMs.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → *Planning and scheduling*.

KEYWORDS

Serverless computing, resource harvesting, reinforcement learning

1 INTRODUCTION

The emergence of serverless computing has extensively simplified the way that developers access cloud resources. Existing serverless computing platforms, such as AWS Lambda, Google Cloud Functions, and Azure Functions, have enabled a wide spectrum of cloud applications, including web services [34], video processing [3, 11], data analytics [19, 27], and machine learning [7, 35] with automated resource provisioning and management. By decoupling traditional monolithic cloud applications into inter-linked microservices executed by stateless *functions*, serverless computing frees developers from infrastructure management and administration with fine-grained resource provisioning, auto-scaling, and pay-as-you-go billing [20].

Existing serverless computing platforms enforce *static* resource provisioning for functions. For example, AWS Lambda allocates function CPU cores in a fixed proportion to the memory size configured by users [6], leading to either CPU over-provisioned or under-provisioned for the function execution. Therefore, serverless service providers are enduring poor resource utilization due to users' inappropriate function configuration—some functions are assigned with more resources than they need [14]. The high concurrency and fine-grained resource isolation of serverless computing further amplify such inefficient resource provisioning.

A few recent studies attempted to address the above issues. Some researchers proposed to maximize resource utilization and reduce the number of cold-starts by predicting the keep-alive windows of individual serverless functions [12, 31]. Fifer [14] incorporated the awareness of function dependencies into the design of a new resource manager to improve resource utilization. COSE [1] attempts to use Bayesian Optimization to seek for the optimal configuration for functions. Furthermore, several works [21, 22, 32] aimed to accelerate functions and improve resource efficiency by adjusting CPU core allocations for serverless functions in reaction to their performance degradation during function executions.

However, none of the existing studies has *directly* tackled the low resource efficiency issue raised by the inappropriate function configurations. There are three critical challenges from the perspective of serverless service providers to address this issue. *First*, a user function is secured as a black box that shares no information about its internal code and workloads, making it hardly possible for the serverless system to estimate the precise resource demands of user functions. *Second*, decoupling monolithic cloud applications to serverless computing architectures generates a variety of functions

with diverse resource demands and dynamic input workloads. *Third*, the resource provisioning for serverless functions is fine-grained spatially (*i.e.*, small resource volumes) and temporally (*i.e.*, short available time).

In this paper, we address the aforementioned challenges by presenting *Freyr*, a new serverless resource manager (RM) that dynamically harvests idle resources to accelerate functions and maximize resource utilization. *Freyr* estimates the CPU and memory saturation points respectively of each function and identifies whether a function is over-provisioned or under-provisioned. For those over-provisioned functions, *Freyr* harvests the wasted resources according to their saturation points; for those under-provisioned functions, *Freyr* tries to accelerate them by offering additional, and just-in-need allocations to approach saturation points. We apply an experience-driven algorithm to identify functions over-supplied and under-supplied by monitoring a series of performance metrics and resource footprints, including CPU utilization, memory utilization, and function response latency to estimate the actual resource demands of running functions. To deal with the highly volatile environment of serverless computing and large numbers of concurrent functions, we propose to apply the Proximal Policy Optimization (PPO) algorithm [30] to learn from the realistic serverless system and make per-invocation resource adjustments. Besides, we design a safeguard mechanism for safely harvesting idle resources without introducing any performance degradation to function executions that have resource harvested.

We implement *Freyr* based on Apache OpenWhisk [4], a popular open-source serverless computing platform. We develop a Deep Reinforcement Learning (DRL) model and training algorithm using PyTorch and enable multi-process support for concurrent function invocations. We evaluate *Freyr* with the other three baselines on an OpenWhisk cluster using realistic serverless workloads. Compared to the default resource manager in OpenWhisk, *Freyr* reduces the 99th-percentile function response latency of invocations[1] by 32.1%. Particularly, *Freyr* harvests idle resources from 38.8% of function invocations while accelerating 39.2% on the OpenWhisk cluster. Notably, *Freyr* only degrades a negligible percentage of function performance under the system performance variations of the OpenWhisk cluster.

2 BACKGROUND AND MOTIVATION

This section first introduces the status quo of resource provisioning and allocation in serverless computing. Then, we use real-world experiments to demonstrate that serverless functions can easily become under-provisioned or over-provisioned, and motivate the necessity to accelerate under-provisioned functions and optimize resource utilization by harvesting idle resources at runtime.

2.1 Resource Provisioning and Allocation in Serverless Computing

Existing serverless computing platforms (*e.g.*, AWS Lambda, Google Cloud Functions, and Apache OpenWhisk) request users to define memory up limits for their functions and allocate CPU cores according to a fixed proportion of the memory limits [4, 5, 13, 36].

---

[1]In this paper, a function denotes an executable code package deployed on serverless platforms, and a function invocation is a running instance of the code package.
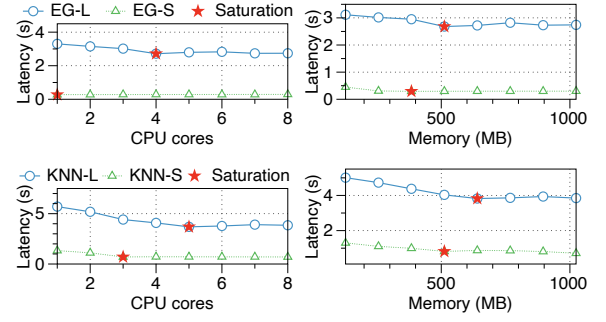

Figure 1: Saturation points of EG and KNN with small (S) and large (L) workload sizes. EG-S (L) generates 1K (10K) emails, and KNN-S (L) inputs 2K (20K) data samples.

Obviously, the fixed proportion between CPU and memory allocations leaves serverless functions either under-provisioned or over-provisioned because functions' CPU and memory demands differ significantly.

Further it is non-trivial for users to accurately allocate appropriate amounts of resource for their functions [1, 32] due to various function types, dependencies, and input sizes. Users are prone to oversize their resource allocation to accommodate potential peak workloads and failures [18, 32]. Finally, users' inappropriate resource allocations and providers' fixed CPU and memory provisioning proportion jointly degrade the resource utilization in serverless computing as resources allocated to functions remain idle (more discussion in Supplementary Materials E).

2.2 Resource Saturation Points

We further demonstrate how easily a serverless function becomes under-provisioned or over-provisioned by introducing a new notion of *saturation points*. Given a function and an input size, there exists a resource allocation *saturation point*—allocating resource beyond this point can no longer improve the function's performance, but allocating resource below this point severely degrades the performance.

We profile the saturation points of two applications: email generation (EG) and K-nearest neighbors (KNN), representing two popular serverless application categories: web applications and machine learning, respectively. We identify the allocation saturation points of CPUs and memory separately by measuring the response latency of functions allocated with different number of CPU cores and different sizes of memory. When adjusting a function's CPU (memory) allocation, we fix its memory (CPU) allocation to 1,024 MB (8 cores).

Figure 1 shows that saturation points vary from functions and input sizes. It is non-trivial for users to identify the saturation points for every function with specific input sizes in their applications. Particularly, serverless functions are typically driven by events with varying input sizes. Without dynamic and careful resource allocations, functions tend to become either over-provisioned or under-provisioned.

2.3 The Need for Harvesting Idle Resources

Resource harvesting is a common methodology in virtual environments that increases resource utilization by reallocating idle

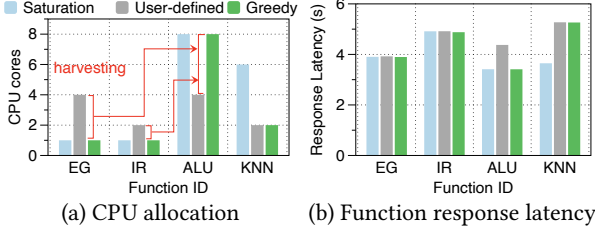(a) CPU allocation (b) Function response latency

Figure 2: The CPU allocation and response latency of four real-world functions: EG, image recognition (IR), arithmetic logic units (ALU), and KNN, where the EG generates 100K emails, the IR classifies ten images, the ALU calculates 20M loops, and the KNN inputs 20K data samples.

resources to under-provisioned services without degrading the performance of services being harvested [2, 38, 40].

To motivate the need for dynamic resource harvesting in serverless computing, we compare the function response latency achieved by the default resource manager (Fixed RM) and greedy resource manager (Greedy RM) when executing four real-world serverless functions. The Fixed RM simply accepts and applies a fixed resource allocation pre-defined by users, such as the RM in OpenWhisk and AWS Lambda. The Greedy RM dynamically harvests CPU cores from functions over-provisioned and assigns the harvested CPU cores to functions under-provisioned in a first-come-first-serve manner based on the estimated function saturation points learned from functions' recent resource utilization (details in Section 5). In this experiment, we collect historical resource utilizations of four functions and profile their saturation points.

Figure 2(a) shows the Greedy RM accelerates the ALU by harvesting three CPU cores from the EG (*i.e.*, the EG function invocation) and one CPU core from the IR. Though the KNN is also under-provisioned, the Greedy RM assigns all harvested CPU cores to the ALU since the ALU is invoked before the KNN. As a comparison, Figure 2 also plots the saturation points of each function invocation and their response latency when allocated with saturated resources. Figure 2(b) shows the Greedy RM can increase resource utilization and accelerate under-provisioned functions without sacrificing over-provisioned functions' performance in the motivation scenario.

2.4 Deep Reinforcement Learning

Due to the volatility and burstiness of serverless computing, it is non-trivial to accurately estimate the saturation points based on functions' recent resource utilization, and the greedy resource harvesting and re-assignment can hardly minimize the overall function response latency. Thus, we propose to utilize reinforcement learning (RL) algorithms to learn the optimal resource harvesting and re-assignment strategies.

At every timestep t, the agent is in a specific state s_t, and evolves to state s_{t+1} according to a Markov process with the transition probability $\mathbb{P}(s_t, a_t, s_{t+1})$ when action a_t is taken [33]. The immediate reward for the agent to take action a_t in state s_t is denoted as r_t. The goal of the agent is to find a policy π that makes decisions regarding what action to take at each timestep, $a_t \sim \pi(\cdot|s_t)$, so as to maximize the expected cumulative rewards, $\mathbb{E}_\pi[\sum_{t=1}^\infty \gamma^{t-1} r_t]$, where $\gamma \in (0, 1]$ is a discount factor.

To capture the patterns of real-world systems and address the curse-of-dimensionality, deep reinforcement learning (DRL) has been introduced to solve scheduling and resource provisioning problems in distributed systems [23–26], where deep neural networks serve as the *function approximators* that describe the relationship between decisions, observations, and rewards.

3 OVERVIEW

3.1 Design Challenges

Unlike long-running VMs with substantial historical traces for demand prediction and flexible time windows for resource harvesting, function executions in serverless computing are highly concurrent, event-driven, and short-lived with bursty input workloads [9], making it hardly practical to reuse the existing VM resource harvesting methods. To enable efficient and safe resource harvesting and performance acceleration in serverless computing, *Freyr*'s design tackles three key challenges:

Volatile and bursty serverless environments. The heterogeneity of serverless functions, the high concurrency of invocation events, and the burstiness of input workloads jointly make it nontrivial to accurately determine whether a function execution has idle resources to be harvested. Besides, serverless functions are sensitive to the latency introduced by resource harvesting and re-assignment due to their short lifetime and event-driven nature.

Huge space of harvesting and re-assignment decisions. Unlike the default resource managers that enforce a fixed proportion between the CPU and memory allocations, we decouple the resource provisioning for CPU and memory for more accurate resource harvesting and re-assignment, leading to a two-dimensional resource pool for *Freyr* to seek for the optimal resource allocation. This is an immense action space for the DRL agent. For example, AWS Lambda allows any memory sizes between 128 MB and 10,240 MB and up to 6 CPU cores—60,672 choices in total. Such a huge action space complicates the DRL algorithm design and extensively increases the computation complexity to train the DRL agent.

Potential performance degradation. While *Freyr* harvests resources from functions deemed as over-provisioned and improves the entire workload, one necessary requirement is to prevent the performance of those functions from degrading. It is vital to guarantee Service Level Objectives (SLOs) of each individual function, *i.e.*, harvested functions have no significant performance degradation.

3.2 *Freyr*'s Architecture

Freyr is a resource manager in serverless platforms that dynamically harvests idle resources from over-provisioned function invocations and reassign the harvested resources to accelerate under-provisioned function invocations. It is located with the controller of a serverless computing framework and interacts with the container system (*e.g.*, Docker [10]) that executes function invocations.

Figure 3 shows an overview of *Freyr*'s architecture. First, concurrent function requests arrive at the frontend to invoke specific functions with user-defined resource allocations. The controller admits the function requests, registers their configurations, and schedules them to the invokers. Before the execution of functions, *Freyr* inputs observations from serverless platform database and
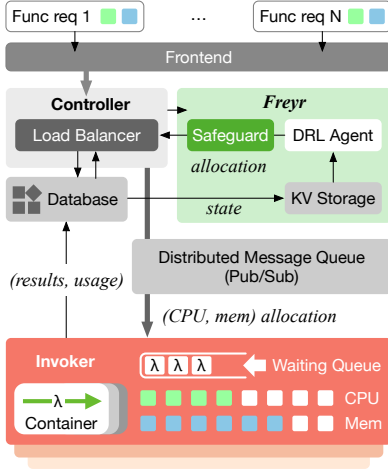
Figure 3: *Freyr*'s architecture.

makes resource harvesting and re-assignment decisions. The controller instructs invokers to enforce the decisions when executing function invocations.

To handle the **volatile and bursty serverless environments**, *Freyr* is designed to be event-driven with multi-process support that the arrival of a function request triggers *Freyr* to make resource harvesting decisions. To shrink the **huge decision space**, *Freyr* trains a score network to justify each allocation option of function invocations, converting the action space of all potential allocation options to a score for individual allocation option. *Freyr* evaluates the score of each allocation option using this score network and enforces the allocation option with the highest score. To avoid **potential performance degradation** of functions with resources harvested, *Freyr* applies a *safeguard mechanism* to prevent those potentially dangerous allocation options and guarantees the SLOs of every function invocation within a workload. The safeguard examines whether the allocation decision made by the DRL agent is below a function's historical resource usage peak. Besides, the safeguard monitors the function's runtime resource utilization and returns all harvested resources by calling a *safeguard invocation* when its resources appear to be fully utilized.

4 DESIGN

4.1 Problem Formulation

We consider a serverless platform that handles a workload W with multiple concurrent function invocations. Let f denote a function invocation in W. We assume the response latency e of f is dominated by CPU and memory. Each function invocation f has a resource allocation $p = (p_c, p_m)$, where p_c and p_m denote a set of CPU and memory resources, respectively. We assume p is non-preemptive and fixed when the platform is executing f, *i.e.*, p is consistently provisioned to f until the execution completes. Thus, we define the relationship between the response latency and the resource allocation as: $e = B(p)$. Section 2.2 demonstrates that a function invocation has a pair of saturation points for CPU and memory denoted by $p^\Xi = (p_c^\Xi, p_m^\Xi)$, respectively.

The platform determines whether it can harvest or accelerate a function invocation f by comparing p with p^Ξ: if $p_c^\Xi < p_c$

$(p_m^\Xi < p_m)$, f has idle CPU (memory), the platform can harvest at most $p_c - p_c^\Xi$ resources without increasing response latency e; if $p_c^\Xi > p_c$ ($p_m^\Xi > p_m$), the allocation of f hasn't saturated, the platform can provide f with at most $p_c^\Xi - p_c$ resources to improve the performance of f, *i.e.*, reduce response latency e. Thus for CPU or memory, function invocations in a workload W can be classified into three groups of invocations: $W = W_h + W_a + W_d$, where W_h denotes the set of invocations that can be harvested, W_a denotes the set of invocations that can be accelerated, and W_d denotes the set of invocations which have descent user configurations ($p^\Xi = p$).

We define a *slowdown* value as the performance metric to avoid prioritizing long invocations while keeping short invocations starving. Recall that W denotes the workload, f denotes a function invocation in W. Function invocations arrive at the platform in a sequential order. At the first invocation of a function, the platform captures the response latency e^b with resources (p_c^b, p_m^b) configured by the user and employs it as a baseline denoted by b. When i-th invocation completes execution, the platform captures the response latency e^i of it. The slowdown of the i-th invocation is calculated as

$$slowdown := \frac{e^i}{e^b}. \tag{1}$$

We normalize the response latency of each invocation with baseline latency of user configuration. Intuitively, the slowdown indicates how a function invocation performs regardless of its duration length. A function invocation may be accelerated while being harvested at the same time (*e.g.*, $p_c^\Xi < p_c$ while $p_m^\Xi > p_m$). In this case, the slowdown is a mixed result. For individual invocations, we only focus on the performance regardless of details of resource allocation, *i.e.*, the invocation is good as long as it yields low slowdown. We use average slowdown to measure how well a workload is handled by the platform with harvesting and acceleration. Hence, the goal is to find a set of resource allocation $p = (p^1, p^2, ..., p^{|W|})$ which minimizes the average slowdown of a workload, defined as

$$avg\_slowdown := \frac{1}{|W|} \sum_{f \in W} \frac{e^i}{e^b} = \frac{1}{|W|} \sum_{f \in W} \frac{B(p^i)}{B(p^b)} \tag{2}$$

However, as introduced in Section 2.2, estimating varying saturation points of sequential function invocations posts a challenging sequential decision problem. The complex mapping from set of p to objective average slowdown can hardly be solved by existing deterministic algorithms. Hence, we opt for DRL and propose *Freyr*, which learns to optimize the problem by replaying experiences through training. *Freyr* observes information from platform level and function level in real time. Figure 4 depicts how *Freyr* estimates CPU/memory saturation points. Given a function invocation, we encode every possible CPU and memory option into a scalar value representing the choice.

4.2 Information Collection and Embedding

When allocating resources for a function invocation, *Freyr* collects information from two levels: platform level and function level, as summarized in Table 1. Specifically, for the platform, *Freyr* captures the number of invocations remaining in the system (*i.e.*, `inflight_request_num`), available CPU cores, and available memory. For the incoming function, *Freyr* queries invocation history
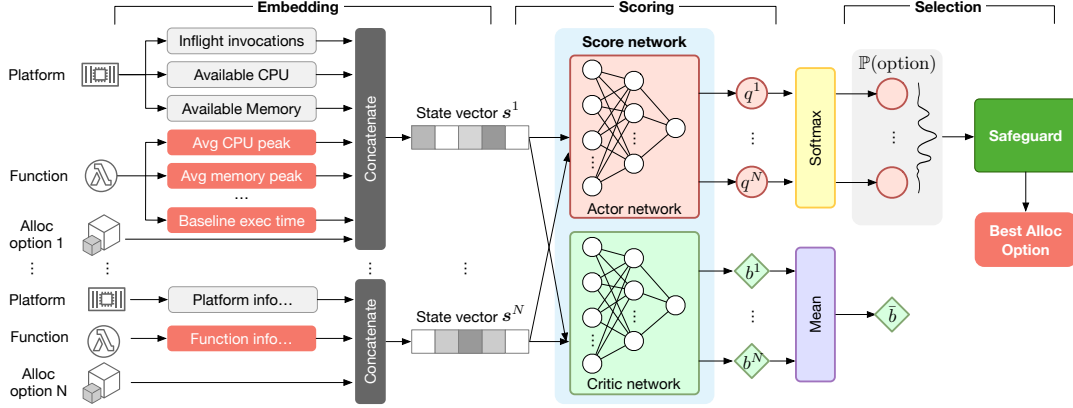
Figure 4: The workflow of *Freyr*.

of the function which records average CPU peak, average memory peak, average inter-arrival time (IAT), average execution time, and baseline execution time (*i.e.*, baseline) with user-requested resources.

Once collecting such information, *Freyr* encapsulates them with a potential resource allocation option. More precisely, we embed information and the potential configuration option together into a flat state vector as input to *Freyr* agent, with the information embedding process illustrated in Figure 4.

4.3 Score Network

Freyr uses a *score network* to calculate the priority of selecting potential resource allocation options. Figure 4 visualizes the policy network of *Freyr* agent, and illustrates the workflow of how the agent selects the best allocation option based on states. At time t, a function invocation arrives at the platform which has in total N potential resource configuration options. After embedding procedure, *Freyr* collects a batch of state vectors $s_t = (s_t^1, \ldots, s_t^n, \ldots, s_t^N)$, where s_t^n maps the state to the n-th option. *Freyr* inputs s_t to the score network. We implement the score network using two neural networks, an *actor network* and a *critic network*. The actor network computes a score q_t^n, which is a scalar value mapped from the state vector s_t^n representing the priority to select configuration option n. Then *Freyr* applies a Softmax operation to the scores $(q_t^1, \ldots, q_t^n, \ldots, q_t^N)$ to compute the probability of selecting option n based on the priority scores, given by

$$\mathbb{P}_t(option = n) = \frac{\exp(q_t^n)}{\sum_{n=1}^N \exp(q_t^n)},$$

at time t. The critic network outputs a baseline value b_t^n for option n, the average baseline value $\bar{b}_t$ is calculated as

$$\bar{b}_t = \frac{1}{N} \sum_{n=1}^N b_t^n, \tag{3}$$

which is used to reduce variance when training *Freyr*. The whole operation of policy network is end-to-end differentiable.

The score network itself contains no manual feature engineering. *Freyr* agent automatically learns to compute accurate priority score of allocation options through training. More importantly, *Freyr* uses the same score network for all function invocations and all

potential resource allocation options. By embedding options into state vectors, *Freyr* can distinguish between different options and use the score network to select the best option. Reusing the score network reduces the size of networks and limits the action space of *Freyr* agent significantly.

4.4 Safeguard

We design *Freyr* to improve both over-provisioned and under-provisioned functions. However, when harvesting resources from functions deemed as over-provisioned, it is possible that *Freyr* under-predicts their resource demands. The performance of functions degrades when being over-harvested. We devise a safeguard mechanism atop *Freyr* to regulate decisions by avoiding decisions that may harm performance and returning harvested resources immediately when detecting a usage spike. We use this safeguard mechanism to mitigate obvious performance degradation of individual functions.

Algorithm 1 summarizes the safeguard mechanism built atop *Freyr*. We refer safeguard invocation as invoking the function with user-defined resources. When there are no previous invocations, *Freyr* triggers the safeguard to obtain resource usage and calibrate the baseline mentioned in Equation 1 (lines 5–7). For further invocations, *Freyr* queries the history of function and polls the usage peak, allocation of the last invocation, and the highest peak since last baseline calibration (lines 10–12). *Freyr* first checks current status of the function, *i.e.*, over-provisioned or under-provisioned (line 13). We assume functions with resource usage below 80% of user-requested level is over-provisioned. For over-provisioned (harvested) functions, *Freyr* then checks the usage peak of last invocation (line 14). If the usage peak approaches 80% of allocation, we suspect there may be a load spike, which could use more resources than current allocation. This triggers the safeguard invocation and

Table 1: The observation state space of the DRL agent.

Platform State	`avail_cpu`, `avail_mem` `inflight_request_num`
Function State	`avg_cpu_peak`, `avg_mem_peak`, `avg_interval`, `avg_execution_time`, `baseline`

Algorithm 1: Safeguard mechanism atop *Freyr*.

---

1 **while** *request_queue.notEmpty* **do**
2   function_id ← request_queue.dequeue()
3   calibrate_baseline ← False
4   last_request ← QueryRequestHistory(function_id)
5   **if** *last_request == None* **then**
       /* Trigger safeguard */
6   range ← [user_defined]
7   calibrate_baseline ← True
8   **else**
9   threshold ← 0.8
10   last_alloc ← last_request.alloc
11   last_peak ← last_request.peak
12   recent_peak ← GetRecentPeak(function_id)
13   **if** *last_peak < user_defined* **then**
          /* Over-provisioned */
14   **if** *last_peak / last_alloc ≥ threshold* **then**
             /* Trigger safeguard */
15   range ← [user_defined]
16   calibrate_baseline ← True
17   **else**
18   range ← [recent_peak + 1, user_defined]
19   **end**
20   **else**
          /* Under-provisioned */
21   range ← [recent_peak + 1, max_per_function]
22   **end**
23   **end**
24   alloc_option ← Freyr(function_id, range)
25   Invoke(function_id, alloc_option, calibrate_baseline)
26 **end**

---

baseline re-calibration, *Freyr* immediately returns harvested resource to the function at the next invocation (lines 15–16). If there is no usage spike, *Freyr* is allowed to select an allocation option from recent peak plus one unit to a user-requested level (line 18). For under-provisioned functions, *Freyr* is allowed to select from recent peak plus one unit to the maximum available level (line 21). After an allocation option is selected, *Freyr* invokes the function and forwards the invocation to invoker servers for execution.

Supplementary Materials D presents a sensitivity analysis of safeguard thresholds and shows that the safeguard mechanism effectively regulates decisions made by *Freyr* and protects SLOs of functions that have resources harvested.

4.5 Training the DRL Agent

Freyr training proceeds in *episodes*. In each episode, a series of function invocations arrive at the serverless platform, and each requires a two-dimensional action to configure CPU and memory resources. When the platform completes all function invocations, the current episode ends. Let T denote the total number of invocations in an episode, and t_i denote the arrival time of the i-th invocation. We continuously feed *Freyr* with a reward r after it takes an action to
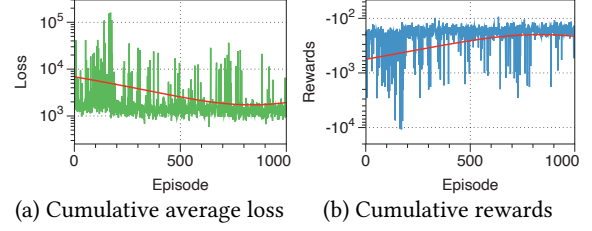

(a) Cumulative average loss   (b) Cumulative rewards

Figure 5: The trends of cumulative average loss (left) and cumulative rewards (right) of *Freyr*'s 1,000-episode training on the OpenWhisk testbed.

handle an invocation. Concretely, we penalize *Freyr* with

$$r_i = - \sum_{f \in S|_{t_{i-1}}^{t_i}} \frac{e^i}{e^b} + R_{(slowdown<1)} - R_{(slowdown>1)},$$

after taking action on the i-th invocation, where W is the set of invocations that finish during the interval $[t_{i-1}, t_i)$, $\frac{e^i}{e^b}$ is the slowdown of an invocation f introduced in Section 4.1, and two constant summaries for awarding good and penalizing bad actions ($R_{(slowdown<1)}$ and $R_{(slowdown>1)}$). The goal of the algorithm is to maximize the expected cumulative rewards given by

$$\mathbb{E}\left[\sum_{i=1}^{T} \gamma^{t-1} \left(- \sum_{f \in S|_{t_{i-1}}^{t_i}} \frac{e^i}{e^b} + R_{(slowdown<1)} - R_{(slowdown>1)} \right) \right]. \quad (4)$$

Similar to [24], we set the discount factor γ in Equation 4 to be 1. Hence, *Freyr* learns to minimize the overall *slowdown* of the given workload.

We use the algorithm 2 to train *Freyr* with 4 epochs per surrogate optimization and a 0.2 clip threshold [30]. We update the policy network parameters using the AdamW optimizer [17] with a learning rate of 0.001. We train *Freyr* with 1,000 episodes. The total training time is about 120 hours. Figure 5 shows the learning curve and cumulative rewards of *Freyr* training on OpenWhisk testbed. In Figure 5(a), the descending loss trendline indicates that *Freyr* gradually learns to make good resource management decisions. In Figure 5(b), the ascending trendline shows that *Freyr* seeks to maximize the cumulative rewards through training. Supplementary Material A and B introduces the details of *Freyr*'s training algorithm and implementation.

5 EVALUATION

We implement *Freyr* with 6K lines of Scala code in Apache Open-Whisk [4] and deploy it to a realistic OpenWhisk cluster. We train and evaluate *Freyr* using realistic workloads from public serverless benchmarks and invocation traces sampled from Azure Functions traces [31] (implementation details in Supplementary Materials B).

5.1 Methodology

Baselines. We compare *Freyr* with three baseline RMs: 1) *Fixed RM*: the default RM of most existing serverless platforms that allocates CPU cores in a fixed proportion to user-defined memory sizes. 2) *Greedy RM* detects a function's saturation points based on its historical resource usage by gradually decreasing (increasing) the
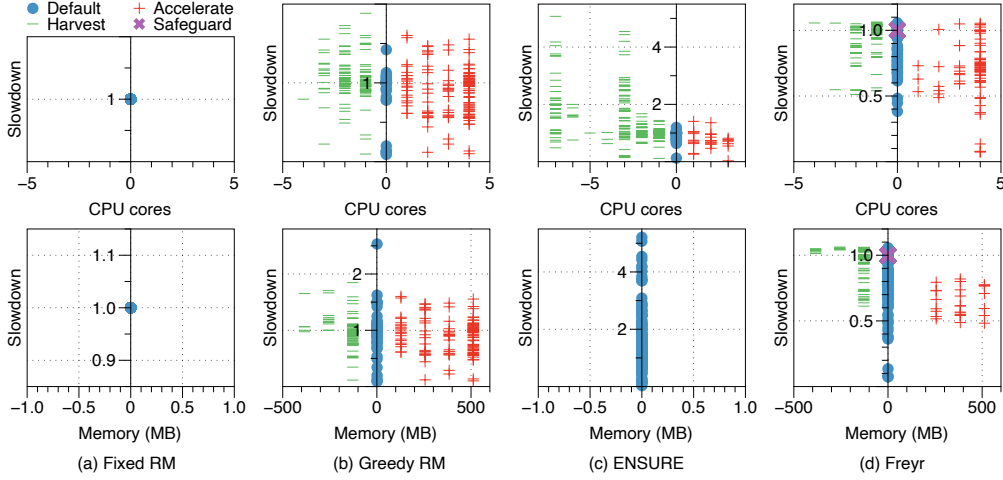
Figure 6: Performance of individual invocations processed by Fixed RM, Greedy RM, ENSURE, and _Freyr_ in OpenWhisk evaluation. Default (●): invocations with user-requested allocation. Accelerate (+): invocations accelerated by supplementary allocation. Harvest (−): invocations with resource harvested. Safeguard (×): invocations protected by the safeguard.

allocation for an over-provisioned (under-provisioned) function in a fine-tuned and fixed step. Our implementation sets the detect step size one core and 64 MBs for CPU and memory, respectively. Besides, Greedy RM allocates resources to functions in a first-come-first-serve manner. 3) _ENSURE_ [32] allocates memory resources as users request and adjusts the CPU cores for each function at runtime when detecting performance degradation.

Evaluation metrics. We use the _slowdown_ value defined in Section 4.1 to measure the performance of a function invocation. Function invocations with lower slowdowns have lower response latency. For resource harvesting, _Freyr_ aims to maximize the amount of harvested resources while having minimal impact on the performance of victim functions. For resource re-assignment, _Freyr_ treats invocations with different lengths of response latency as the same by reducing slowdowns, which improves the overall performance of the workload. We also report the details of SLO violation and 99th-percentile (P99) function response latency of the workload.

Testbed: We deploy and evaluate _Freyr_ on an OpenWhisk cluster with 13 physical servers. Two of the servers host the OpenWhisk components, such as the frontend, the controller, the messaging queue, and database services. One deploys the _Freyr_ agent. The remaining ten servers serve as the invokers for executing functions. The server hosting _Freyr_ agent has 16 Intel Xeon Skylake CPU cores and 64 GB memory, and each of the other 12 servers has eight Intel Xeon Skylake CPU cores and 32 GB memory. Each function can be configured with eight CPU cores and 1,024 MB of RAM at most. Considering the serverless functions' short lifecycle, we monitor their CPU and memory usage per 0.1 second and keep the historical resource usage in the Redis (_i.e._, KV store in Figure 3).

Workloads: We randomly sampled two function invocation sets for OpenWhisk evaluation. Table 2 depicts the two invocation sets (OW-train and OW-test) used in the OpenWhisk evaluation. We use a scaled-down version of the invocation traces, _i.e._, we assume the invocation trace is based on seconds rather than minutes. This rescaling increases the intensity of workloads while speeding up _Freyr_

OpenWhisk training by reducing the total workload duration. We employ ten real-world functions from three serverless benchmark suites: SeBS [8], ServerlessBench [39], and ENSURE-workloads [32] (details of the ten functions in Supplementary Materials C). For DH, EG, IP, KNN, ALU, MS and GD, each is initially configured with four CPU cores and 512 MB memory; for VP, IR and DV, each is initially configured with eight cores and 1,024 MB. We set the initial resource configuration of each function according to the default settings from the suites.

5.2 Results

We summarize the slowdown and resource allocation of function invocations of the testing workload in Figure 6. In each subgraph, each point (_i.e._, ●, +, −, and ×) indicates a function invocation. The y-axis indicates the slowdown values of function invocations, and the x-axis shows the CPU and memory allocation of function invocations relative to their user configurations. The negative CPU and memory values indicates that RMs harvest corresponding resources from those invocations, and the positive means that those invocations are provided with additional resources.

Overall performance. _Freyr_ outperforms other baseline RMs with the best overall performance. For processing the same testing workload, _Freyr_ achieves a lowest average slowdown of 0.82, whereas Fixed RM, Greedy RM, ENSURE are 1.0, 1.12, and 1.78, respectively. Recall in Section 4.1, a lower slowdown indicates a faster function

Table 2: Characterization of training and testing workload sets in the OpenWhisk evaluation. Metrics include: total number of unique traces, total number of invocations (Calls), average inter-arrival time (IAT), and requests per second.

Set	Traces	Calls	Avg IAT (s)	Reqs/sec
OW-train	1,000	26,705	2.21	0.44
OW-test	10	268	2.20	0.45

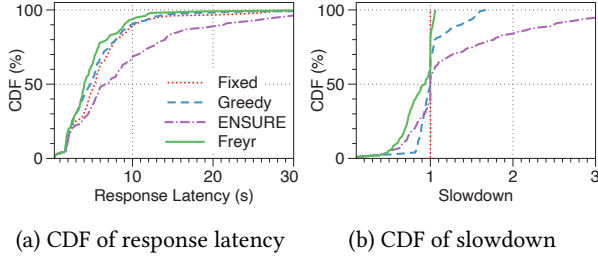(a) CDF of response latency   (b) CDF of slowdown

Figure 7: The CDF of function response latency (left) and slowdown (right) in OpenWhisk experiment respectively.

response. Compared to the default RM in OpenWhisk, *Freyr* provides an average of 18% faster function executions and 32.1% lower P99 response latency for the testing workload. *Freyr* harvests idle resources from 38.8% of function invocations and accelerates 39.2% of invocations.

Harvesting and acceleration. Figure 6 shows the performance of 268 individual invocations processed by four RMs. Fixed RM has no resource adjustment during its workload processing. Greedy harvests an average of 1.7 cores and 168 MB from victim invocations and accelerates under-provisioned functions with an average of 3 cores and 392 MB. ENSURE's policy also harvests and accelerates invocations with CPU cores but makes no changes to memory resources. ENSURE harvests an average of 3.4 cores from victims and accelerates under-provisioned functions with an average of 1.9 cores. *Freyr* harvests an average of 1.5 cores and 380 MB from victims and accelerates under-provisioned functions with an average of 3.6 cores and 164 MB. *Freyr* re-assign harvested resources to accelerate under-provisioned invocations, which speeds up for under-provisioned function invocations up to 92%.

SLO violation. Figure 6 shows that both Greedy RM and ENSURE severely violate function SLOs since there are some function invocations with slowdown values much larger than 1. Fixed RM has no violation as it performs no harvesting or acceleration. Greedy RM degrades the performance of some victim invocations over 60%. ENSURE violates SLOs of some victim invocations over 500% when harvesting CPU cores. Compared to Greedy RM and ENSURE, *Freyr* rationally harvests idle resources from under-provisioned invocations, as the performance degradation of victim invocations is limited within 6%. When harvesting idle resources, *Freyr* calls safeguard for 21.8% of invocations to avoid potential performance degradation due to usage spike.

P99 latency. Figure 7(a) shows the CDF of function response latency of the testing workload. *Freyr* has a P99 function response latency in less than 19 seconds, whereas Fixed RM, Greedy RM and ENSURE are 28, 25, and 38 seconds, respectively. Figure 7(b) shows the CDF of the slowdown of the testing workload. *Freyr* maintains P99 slowdowns below 1.06 for all invocations, whereas Greedy RM and ENSURE are 1.58 and 4.5, respectively. As Fixed RM adjusts no resources, the slowdown stays 1.0 for all percentile.

6 RELATED WORK

Resource harvesting. Research has been conducted on VM resource management in traditional clouds for years. SmartHarvest [38]

proposes a VM resource harvesting algorithm using online learning. Unlike *Freyr*, which uses harvested resources to accelerate function executions, SmartHarvest offers a new low-priority VM service using harvested resources. Directly replacing *Freyr* with SmartHarvest is not feasible as SmartHarvest is not designed for serverless computing. Zhang *et al.* [40] proposed to harvest VMs for serverless computing, while *Freyr* harvests idle resources of serverless functions directly.

Resource provisioning. Spock [15] proposes a serverless-based VM scaling system to improve SLOs and reduce costs. For resource management in serverless, [22] and [32] both aim to automatically adjust CPU resource when detecting performance degradation during function executions, which help mitigate the issue of resource over-provisioning. Unlike [22] and [32] that only focus on CPU, *Freyr* manages CPU and memory resources independently. Kaffes *et al.* [21] propose a centralized scheduler for serverless platforms that assigns each CPU core of worker servers to CPU cores of scheduler servers for fine-grained core-to-core management. *Freyr* focuses on resource allocation rather than scheduling or scaling. Fifer [14] tackles the resource under-utilization in serverless computing by packing requests to fewer containers for function chains. Instead of improving packing efficiency, *Freyr* directly harvests idle resources from under-utilized functions.

Reinforcement learning. SIREN [35] adopts DRL techniques to dynamically invoke functions for distributed machine learning with a serverless architecture. Our work *Freyr* leverages DRL to improve the platform itself rather than serverless applications. Decima [25] leverages DRL to schedule DAG jobs for data processing clusters. Metis [37] proposes a scheduler to schedule long-running applications in large container clusters. TVW-RL [26] proposes a DRL-based scheduler for time-varying workloads. George [23] uses DRL to place long-running containers in large computing clusters. Differ from the above works, *Freyr* learns resource management in serverless computing using DRL.

7 CONCLUSION

This paper proposed a new resource manager, *Freyr*, which harvests idle resources from over-provisioned functions and accelerates under-provisioned functions with supplementary resources. Given realistic serverless workloads, *Freyr* improved most function invocations while safely harvesting idle resources using reinforcement learning and a safeguard mechanism. Experimental results on the OpenWhisk cluster demonstrate that *Freyr* outperforms other baseline RMs. *Freyr* harvests idle resources from 38.8% of function invocations and accelerates 39.2% of invocations. Compared to the default RM in OpenWhisk, *Freyr* reduces the 99th-percentile function response latency by 32.1% for the same testing workload.

8 ACKNOWLEDGEMENTS

REFERENCES

[1] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring Serverless Functions using Statistical Learning. In *Proc. of IEEE INFOCOM*.

[2] 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 *Proc. of USENIX OSDI*.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proc. of ACM SoCC*.

[4] Apache. 2018. Apache OpenWhisk Official Website. https://openwhisk.apache.org. [Online; accessed 1-May-2018].

[5] AWS. 2018. AWS Lambda: Serverless Compute. https://aws.amazon.com/lambda/. [Online; accessed 1-May-2018].

[6] AWS. 2021. AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html. [Online; accessed 1-May-2021].

[7] 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 ACM SoCC*.

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

[9] DataDog. 2020. The State of Serverless. https://www.datadoghq.com/state-of-serverless-2020/. [Online; accessed 1-July-2021].

[10] Docker. 2021. Docker: Empowering App Development for Developers. https://www.docker.com. [Online; accessed 1-May-2021].

[11] 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*.

[12] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proc. of ACM ASPLOS*.

[13] Google Cloud. 2018. Google Cloud Function:Event-Driven Serverless Compute Platform. https://cloud.google.com/functions. [Online; accessed 1-May-2018].

[14] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T Kandemir, and Chita R Das. 2020. Fifer: Tackling Underutilization in the Serverless Era. In *Proc. of ACM Middleware*.

[15] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. 2019. Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud. In *Proc. of IEEE CLOUD*.

[16] IBM. 2021. IBM Cloud Functions. https://www.ibm.com/cloud/functions.

[17] Loshchilov Ilya and Hutter Frank. 2019. Decoupled Weight Decay Regularization. In *Proc. of ICLR*.

[18] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *Proc. of USENIX ATC*.

[19] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proc. of ACM SoCC*.

[20] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs.OS]

[21] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-Granular Scheduling for Serverless Functions. In *Proc. of ACM SoCC*.

[22] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya. 2020. Automated Fine-Grained CPU Cap Control in Serverless Computing Platform. *IEEE Transactions on Parallel and Distributed Systems* (2020).

[23] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. 2021. George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints. In *Proc. of ACM SoCC*.

[24] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proc. of ACM HotNets*.

[25] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proc. of ACM SIGCOMM*.

[26] Shanka Subhra Mondal, Nikhil Sheoran, and Subrata Mitra. 2021. Scheduling of Time-Varying Workloads Using Reinforcement Learning. In *Proc. of AAAI*.

[27] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proc. of ACM SIGMOD*.

[28] PyTorch. 2018. PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration. https://pytorch.org. [Online; accessed 1-May-2018].

[29] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. 2017. Trust Region Policy Optimization. arXiv:1502.05477 [cs.LG]

[30] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG]

[31] 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*.

[32] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. 2020. ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments. In *Proc. of ACSOS*.

[33] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. MIT press Cambridge.

[34] Mayur Tanna and Harmeet Singh. 2018. *Serverless Web Applications with React and Firebase: Develop real-time applications for web and mobile platforms*. Packt Publishing Ltd.

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

[36] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *Proc. of USENIX ATC*.

[37] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. 2020. Metis: Learning to Schedule Long-Running Applications in Shared Container Clusters at Scale. In *Proc. of ACM SC*.

[38] 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*.

[39] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with ServerlessBench. In *Proc. of ACM SoCC*.

[40] 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 ACM SOSP*.

A THE TRAINING ALGORITHM

Freyr uses a policy gradient algorithm for training. Policy gradient methods are a class of RL algorithms that learn policies by performing gradient ascent directly on the parameters of neural networks using the rewards received during training. When updating policies, large step sizes may collapse the performance, while small step sizes may decrease the sampling efficiency. We use the Proximal Policy Optimization (PPO) algorithms [30] to ensure that *Freyr* takes appropriate step sizes during policy updates. More specifically, given a policy π_θ parameterized by θ, the PPO algorithm updates policies at the k-th episode via

$$\theta_{k+1} = \arg\max_\theta \mathbb{E}_{s,a\sim\pi_{\theta_k}} \left[\mathbb{L}(s, a, \theta_k, \theta) \right],$$

where $\mathbb{L}$ is the *surrogate advantage* [29], a measure of how policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy. We use the PPO-clip version of a PPO algorithm, where $\mathbb{L}$ is given by

$$\mathbb{L}(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \ g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right),$$

and $g(\epsilon, A)$ is a clip operation defined as

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A, & \text{if } A \geq 0, \\ (1-\epsilon)A, & \text{otherwise,} \end{cases}$$

where A is the advantage calculated as rewards r subtracted by baseline values b; ϵ is a hyperparameter that restricts how far the new policy is allowed to deviate from the old. Intuitively, the PPO algorithm sets a range for step sizes of policy updates, which prevents the new policy from deviating too much from the old (either positive or negative).

Algorithm 2 presents the training process of *Freyr*. For each episode, we record the whole set of trajectories including the states, actions, rewards, baseline values predicted by the critic network, and the logarithm probability of the actions for all invocations. After each training episode finishes, we use the collected trajectories to update the actor and critic networks.

B IMPLEMENTATION DETAILS

Apache OpenWhisk is an open-source, distributed serverless platform that powers IBM Cloud Functions [16]. Figure 3 illustrates the architecture of *Freyr* based on OpenWhisk. OpenWhisk exposes an NGINX-based REST interface for users to interact with the platform. Users can create new functions, invoke functions, and query results of invocations via the frontend. The Frontend forwards function invocations to the Controller, which selects an Invoker (typically hosted using VMs) to execute invocations. The Load Balancer inside the Controller implements the scheduling logic by considering Invoker's health, available capacity, and infrastructure state. Once choosing an Invoker, the Controller sends the function invocation request to the selected Invoker via a Kafka-based distributed messaging component. The Invoker receives the request and executes the function using a Docker container. After finishing the function execution, the Invoker submits the result to a CouchDB-based Database and informs the Controller. Then the Controller returns the result of function executions to users synchronously or asynchronously. Here we focus on resource management for containers.

Algorithm 2: *Freyr* Training Algorithm.

1 Initial policy (actor network) parameters θ_0 and value function (critic network) parameters ϕ_0

2 for *episode $k \leftarrow$ 0, 1, 2, …* **do**

3   Run policy $\pi_k = \pi(\theta_k)$ in the environment until T-th invocation completes

4   Collect set of trajectories $\mathbb{D}_k = \{\tau_i\}$, where $\tau_i = (s_i, a_i), i \in [0, T]$

5   Compute reward $\hat{r}_t$ via Equation 4

6   Compute baseline value $\bar{b}_t$ via Equation 3

7   Compute advantage $\hat{\mathbb{A}}_t = \hat{r}_t - \bar{b}_t$

8   Update actor network by maximizing objective using stochastic gradient ascent:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathbb{D}_k|T} \sum_{\tau \in \mathbb{D}_k} \sum_{t=0}^{T} \mathbb{L}(s_t, a_t, \theta_k, \theta)$$

9   Update critic network by regression on mean-squared error using stochastic gradient descent:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathbb{D}_k|T} \sum_{\tau \in \mathbb{D}_k} \sum_{t=0}^{T} (\bar{b}_t - \hat{r}_t)^2$$

10 end

We modify the following modules of OpenWhisk to implement our resource manager:

Frontend: Initially, 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 limit and enable the Frontend to take CPU and memory inputs from users. Users are allowed to specify CPU cores and memory of their functions, and the Frontend forwards both CPU and memory limits to the Controller.

Controller: The Load Balancer makes scheduling decisions for the Controller. When selecting an Invoker, the Load Balancer considers available memory of Invokers. We modify the Load Balancer also to check available CPU cores of Invokers—the Load Balancer selects Invokers with enough available CPU cores and memory to execute function invocations.

Invoker: The Invoker uses a semaphore-based mechanism to control containers' access to available memory. We apply the same mechanism to control access to available CPU cores independently.

Container: By default, OpenWhisk uses `cpu-shares` parameter to regulate CPU power of containers. When plenty of CPU cycles are available, all containers with `cpu-shares` use as much CPU as they need. While `cpu-shares` improves CPU utilization of Invokers, it can lead to performance variation of function executions. We change the CPU parameter to `cpus` which restricts how many CPU cores a container can use. This is aligned with the CPU allocation policy of AWS Lambda [6]. For each function invocation, we monitor the CPU cores and memory usage of its container using

Table 3: Characterizations of serverless applications used in OpenWhisk evaluation. (DH: Dynamic HTML, EG: Email Generation, IP: Image Processing, VP: Video Processing, IR: Image Recognition, KNN: K Nearest Neighbors, GD: Gradient Descent, ALU: Arithmetic Logic Units, MS: Merge Sorting, and DV: DNA Visualization.)

Function	Type	Dependency
DH	Web App	Jinja2, CouchDB
EG	Web App	CouchDB
IP	Multimedia	Pillow, CouchDB
VP	Multimedia	FFmpeg, CouchDB
IR	Machine Learning	Pillow, torch, CouchDB
KNN	Machine Learning	Scikit-learn, CouchDB
GD	Machine Learning	NumPy, CouchDB
ALU	Scientific	CouchDB
MS	Scientific	CouchDB
DV	Scientific	Squiggle, CouchDB

cgroups. We record the usage peak during function execution and keep it as history for *Freyr* to query.

DRL agent: We implement the *Freyr*'s agent using two neural networks, each with two fully connected hidden layers. The first hidden layer has 32 neurons, and the second layer has 16 neurons. Each neuron uses Tanh as its activation function. The agent is implemented in 2K lines of Python code using PyTorch [28]. *Freyr* is lightweight because the policy network consists of only 1858 parameters (12 KB in total). Mapping a state to an action takes less than 10 ms.

C WORKLOAD CHARACTERIZATIONS

Table 3 describes the type and dependency of 10 serverless applications from benchmark suites. DH downloads HTML template, populates the templates based on input, and uploads them to CouchDB. EG generates emails based on the input and returns them to the CouchDB. IP downloads images, resizes them, and uploads them to CouchDB. VP downloads videos, trims and tags them with a watermark, and uploads to CouchDB. IR downloads a batch of images, classifies them using ResNet-50, and uploads them to CouchDB. KNN downloads the dataset, performs the KNN algorithm on it, and uploads the result to CouchDB. GD performs three kinds of gradient descent based on input and uploads the result to CouchDB. ALU computes the arithmetic logic based on input and uploads the result to CouchDB. MS performs merge sorting based on input and uploads the result to CouchDB. DV downloads a DNA sequence file, visualizes the sequence, and uploads the result to CouchDB. We profile the ten applications configured with eight CPU cores and 1,024 MB memory, which is the maximum allocation in our experimental environment.

D SAFEGUARD SENSITIVITY ANALYSIS

Safeguard threshold. We set the default threshold value in the safeguard algorithm to be 0.8, which allows *Freyr* to trigger the safeguard just before detecting a full utilization. The threshold is
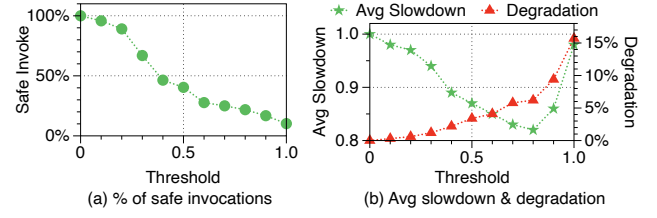

Figure 8: Sensitivity analysis of safeguard thresholds.

tunable—a high threshold may allow *Freyr* to presumptuously harvest idle resource and deteriorate performance, while a low threshold may too conservatively restrict the harvesting and under-utilize resources. We conduct a threshold analysis on our OpenWhisk testbed using the workload OW-test from Table 2 to evaluate the sensitivity of safeguard threshold in *Freyr*. We increase *Freyr*'s safeguard threshold from 0 to 1 with a step of 0.1 and run the same workload using *Freyr*. Figure 8(a) shows the percentage of safe invocations (invocations allocated with user-defined CPU/memory) under each threshold. Figure 8(b) shows the average slowdown and percentage of degraded invocations under each threshold. When increasing the threshold, the rate of safe invocation drops down as *Freyr* gradually harvests idle resources wildly. The percentage of degraded invocations gradually rises because *Freyr*'s harvesting policy becomes more and more unrestricted. For average slowdown of the workload, *Freyr* achieves better and better overall performance until its threshold reaching 0.8. Due to severe performance degradation, *Freyr* yields a worse performance for thresholds 0.9 and 1.0.

To deploy *Freyr* in a production environment, service providers can tune the safeguard threshold based on their own criteria, *i.e.*, tightening the threshold to conservatively harvest or loosing the threshold to actively harvest idle resources.

Safeguard effectiveness. To examine safeguard effectiveness in *Freyr*, we also evaluate a variant of *Freyr* with safeguard turned off. We run the workload OW-test from Table 2 on our OpenWhisk testbed using safeguard-off *Freyr* and obtain the average slowdown and performance degradation. *Freyr* without safeguard processes the testing workload with an average slowdown of 1.28 while degrading at most 15.7% to function response latency, which is 36% slower and has 9.5% more degradation than the original version. The result shows that *Freyr*'s safeguard effectively regulates the decision-making process, thus guaranteeing the performance of individual functions.

E DEPLOYING *FREYR*

In industrial serverless computing environments, such as OpenWhisk, AWS Lambda, and Google Cloud Functions, integrating *Freyr* lead to merits for both service providers and users. For service providers, *Freyr* carefully harvests idle resources and reuses them to accelerate function invocations, which improves the overall serverless platform's resource utilization. For users who mistakenly configured insufficient resource allocation for their functions, *Freyr* transparently brings potential performance protection (*i.e.*, faster function executions) using harvested idle resources without violating other users' SLOs.