2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)

Deadline-aware Dynamic Resource Management in Serverless Computing Environments

Anupama Mampage, Shanika Karunasekera and Rajkumar Buyya
Cloud Computing and Distributed Systems (CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia
Email: mampage@student.unimelb.edu.au, {karus, rbuyya}@unimelb.edu.au

***Abstract*—Serverless computing enables rapid application development and deployment by composing loosely coupled microservices at a scale. This emerging paradigm greatly unburdens the users of cloud environments, from the need to provision and manage the underlying cloud resources. With this shift in responsibility, the cloud provider faces the challenge of providing acceptable performance to the user without compromising on reliability, while having minimal knowledge of the application requirements. Sub-optimal resource allocations, specifically the CPU resources, could result in the violation of performance requirements of applications. Further, the fine-grained serverless billing model only charges for resource usage in terms of function execution time. At the same time, the provider has to maintain the underlying infrastructure in always-on mode to facilitate asynchronous function calls. Thus, achieving optimum utilization of cloud resources without compromising on application requirements is of high importance to the provider. Most of the current works only focus on minimizing function execution times caused by delays in infrastructure set up and reducing resource costs for the end-user. However, in this paper, we focus on both the provider and user's perspective and propose a function placement policy and a dynamic resource management policy for applications deployed in serverless computing environments. The policies minimize the resource consumption cost for the service provider while meeting the user's application requirement, i.e., deadline. The proposed solutions are sensitive to deadline and efficiently increase the resource utilization for the provider, while dynamically managing resources to improve function response times. We implement and evaluate our approach through simulation using ContainerCloudSim toolkit. The proposed function placement policy when compared with baseline scheduling techniques can reduce resource consumption by up to three times. The dynamic resource allocation policy when evaluated with a fixed resource allocation policy and a proportional CPU-shares policy shows improvements of up to 25% in meeting the required function deadlines.**

***Index Terms*—serverless computing, function placement, dynamic resource management, resource efficiency**

I. INTRODUCTION

The many attractions of the serverless computing paradigm include rapid auto-scaling, strong isolation for applications, a fine-grained billing mechanism and more importantly, the access to a service ecosystem, which automatically handles instance selection, resource management, fault tolerance, monitoring, and security [1].

The core concept behind the serverless execution model is to shift the complexities of application resource management from the developer to the cloud provider. The serverless model requires the provider to autonomously manage resource allocations to functions in real time, in contrast to a service placement scenario under a serverful model (e.g. Infrastructure as a Service), where the user configures the environment with required resources prior to application execution [1]. Serverless platforms have minimal knowledge on the resource requirements of different functions, at the time of initial resource allocation. For instance, AWS Lambda allows the user to specify the amount of memory available to the function during execution and allocates the CPU power linearly in proportion to the configured memory [2]. Google Cloud Functions seems to adopt the same strategy for resource allocations [3]. Studies show that CPU is often a cause for contention in serverless environments, specially when compute intensive applications are involved [4], leading to high application latencies. Thus, any arbitrary resource allocation policy could lead to subsequent resource contentions for the applications during runtime, leading to Service Level Agreement (SLA) violations to the user. Thus arises the need for dynamic resource management techniques.

As per the serverless deployment model, the user is charged only for the resource-time actually consumed by the application during its execution. Regardless of this fact, the cloud provider maintains the underlying Virtual Machine (VM) resources during its entire active life time. The resource-time covered by a VM during its life time comprises of its set up time and its entire active time when one or more functions are using the VM resources either fully or partially. As the serverless model is being increasingly experimented for longer running tasks such as massively parallel task executions as in [5], [6], such partial resource usages is more prevalent. Hence its imperative, that the cloud provider maximizes the utilization of the set of active VMs at any given time, thus reducing the cost of maintaining too many underutilized VMs. On the other hand, when an application has a deadline target on execution as part of the SLA, the placement decision of a function instance on an available VM needs to consider optimizing the resource usage of VMs, without compromising on the stated execution time limitations.

Many existing serverless platforms follow different strategies to manage their underlying infrastructure. Experimenta-

978-1-7281-9586-5/21/$31.00 ©2021 IEEE

DOI 10.1109/CCGrid51090.2021.00058

tion done on AWS Lambda platform indicate that the function placement decision is currently treated as a bin packing problem to maximize VM memory utilization [7]. Azure Functions seems to try not to co-locate concurrent instances of the same function on the same VM, which indicates a spread placement approach [7]. IBM OpenWhisk uses a hash-based first-fit heuristic which aggregates application executions by function type, aimed at improving instance re-use and cache hit rate [8]. Docker Swarm employs a spread placement algorithm which tries to evenly spread tasks across the nodes in a cluster [9].

While some of the above approaches try to maximize resource utilization in function executions, they do not consider application specific details and hence could result in SLA violations, while not achieving optimal resource usages. In the literature, many works exist, which address the problem of reducing function response time to users and optimizing resource cost for the end user [10], [11], [12], [13]. However, the importance of dynamic management of allocated resources to function instances in the runtime, considering user requirements, and also the problem space of efficient resource management on the provider side have not been studied extensively.

Thus, a major challenge for the service provider under this model is to choose a suitable compute node for the function placement and allocate sufficient resources to the containerized function instance, such that the desired user requirements are met, and the cost of resources is maintained at an optimum level.

In this paper, we present a deadline-sensitive heuristic algorithm for selecting a VM for function execution, which tries to manage the VM resources efficiently in order to minimize the provider cost of maintaining the cloud infrastructure. Further, we present an approach to dynamically monitor and manage the allocated CPU resources to function instances in the runtime, targeted at meeting the user deadlines, irrespective of the initial assignment of resources. We implement our proposed policies in ContainerCloudSim [14] simulation environment and conduct experiments using real-world and synthetic traces. The experimental results show that our policies increase the efficiency of VM resources and also perform better in terms of meeting the function deadlines, as compared to baseline techniques.

The key **contributions** of our work are as follows:

1. An efficient placement algorithm for function requests, which aims to enhance VM resource efficiency.

2. A fine-grained approach to dynamically manage resource allocations to functions.

3. Implementation of our proposed policies in a simulation environment and conducting extensive experiments using both real-world and synthetic workloads.

4. Evaluation of the efficiency of our proposed solution in comparison with baseline load balancing algorithms and two resource allocation policies, namely, a fixed resource allocation policy and a cpu-shares policy in terms of VM resource usage and meeting the function deadlines.

The rest of the paper is organized as follows. Section II, highlights related research. In section III, we show the system model and formulate the scheduling problem. Section IV, presents our proposed approach. In section V, we discuss the experimental set-up and present the performance evaluation of our proposed method. Section VI concludes the paper and highlights future research directions.

II. Related Work

Serverless computing as a cloud application deployment model, is still at an early stage of being widely adopted and explored in different application domains. As such, research work referring to efficient resource management in serverless platforms are still growing and also refer to many diverse aspects. Here we focus on key research work related to application resource management in serverless platforms.

HoseinyFarahabady et al. present a QoS-aware resource allocation controller for serverless platforms [15]. The scheduler aims to dynamically scale resources by predicting the future rate of incoming events using a closed-loop model predictive mechanism. Although the controller tries to maintain a healthy CPU utilization level at each host, specific focus is not given to using application level details to minimize provider cost. Overall this work addresses the challenge of the initial placement of functions, but the handling of sub-optimum resource allocations is not discussed.

A package-aware scheduler is proposed by Abad et al. [11] for serverless functions. The objective is to reduce the cold start latency by assigning functions requiring similar packages to the same node. The efficiency of this model could be affected by functions having multiple, large package dependencies. Further, this model ignores workload characteristics other than the package requirements and does not focus on optimizing cloud resource usage.

Stein et al. [10] present a non-cooperative resource allocation heuristic which tries to predict the number of function instances required to keep the request waiting time below a chosen threshold. This work considers container re-use and pre-warming of containers to reduce response times. Meeting any function specific user requirements has not been focused on, in the proposed approach.

Mahmoudi et al. [16] present a function placement algorithm which uses a machine learning based approach for selecting the VM for a new function invocation, so as to reduce operational cost to the user. They explore a predictive performance model which tries to predict the normalized performance for any workload when deployed to a specific VM. Their approach takes in to account the nature of the workload such as CPU, disk or memory intensiveness in making its decision. The model requires a profiling step each time a new function is deployed in the platform.

A latency-aware function scheduler is presented by Suresh et al. [4]. Their model is focused on dynamically adjusting cpu-shares [17] of containers based on the latency degradation to each application type as a whole. The greedy algorithm presented for VM scaling results in reduced number of VMs

TABLE I
SUMMARY OF LITERATURE STUDY

Work	Application Model		Deadline Awareness	Efficient VM usage	Dynamic Re-sizing	Dynamic Re-scheduling
	Task(Single Function)	DAG(Multiple functions)				
HoseinyFarahabady et al. (2017)	✓		✓	✓		
Abad et al. (2018)	✓					
Stein et al. (2018)	✓					
Mahmoudi et al.(2019)	✓					
Suresh et al. (2019)	✓		✓	✓	✓	
Kaffes et al. (2019)	✓					
Singhvi et al. (2019)		✓	✓			
Das et al. (2020)		✓	✓			
Kim et al. (2020)	✓	✓		✓	✓	
Our proposed work	✓		✓	✓	✓	✓

being used compared to spread placement approaches, but there is no specific focus on reducing partial VM usages. They achieve reduced latency degradation to applications via adjusting cpu-shares of containers. Since cpu-shares is a relative allocation of CPU to each container, the application performance largely depends on the co-located functions in a VM.

A core-granular scheduler for serverless environments is introduced by Kaffes et al. [18]. In this model, the scheduler assigns functions directly to individual cores aiming to eliminate overloading of cores and reducing co-located function interference. Consideration for any workload specific requirements is not observed in the proposed approach.

Singhvi et al. implement a low-latency serverless platform for DAG based applications [12]. The design entails a set of node clusters with semi-global schedulers, which follow deadline-aware function scheduling. Although a deadline is considered for initial function placement, the subsequent management of allocated resources to functions is not discussed.

Das et al. [13] propose a hybrid cloud scheduling framework for multi-function serverless applications. They suggest a greedy algorithm to decide the order and placement of each function in either the private or the public cloud. The objective is to minimize the cost of public cloud use for the consumer and to complete the execution of a batch of jobs within a specified deadline.

Kim et al. [19] propose a technique for CPU resource management for serverless worker processes based on the throttled time and the number of unprocessed functions in the queue of each worker. They try to reduce the function response time and increase the CPU utilization of the worker processes. They do not consider application level details or requirements in their load balancing policy and thus may not be responsive to specific user execution time limitations and achieving optimal resource usage levels.

A summary of the reviewed related works is presented in Table I, comparing them in terms of the focus on efficient VM resource usage, application deadline awareness, dynamic resource re-sizing and re-scheduling, and the application model. Although a few works discuss increasing the VM resource utilization levels in general, they are not specifically focused on reducing provider cost by making use of application level details and requirements.

In our work we present a comprehensive function placement algorithm which aims to manage VM resources efficiently and thereby reduce the underutilization of VMs by considering the function deadlines, in making the placement decision. We also propose an algorithm for the dynamic management of resource allocations to functions instances, in order to meet the application deadlines.

III. SYSTEM MODEL AND PROBLEM FORMULATION

We present the serverless system model used in our work and formulate the problem of scheduling functions in VMs.

A. System Model

We follow a similar approach to Apache OpenWhisk serverless platform [20] while designing the system model for evaluating our proposed approach to address the challenges mentioned above. Figure 1 illustrates the high-level system components involved in our model.

Function invocation requests created from user initiated events are received at the system controller. The controller contains the function placement logic which handles the load balancing responsibility in choosing a compute node for function execution. The VM monitor module constantly updates and retains meta data on the expected remaining VM runtimes

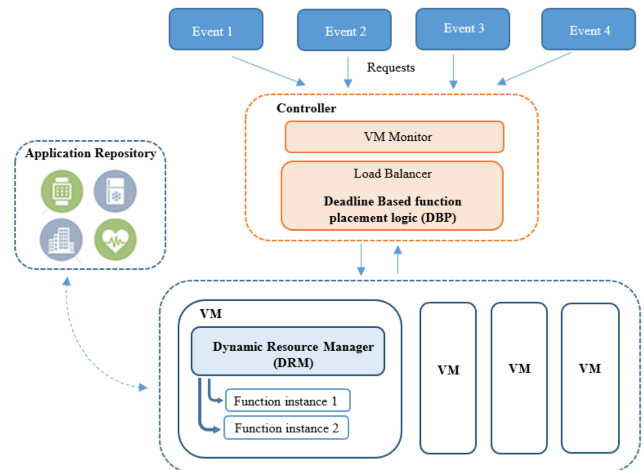

Fig. 1. System Model

and the functions in execution in each VM. This information is used by the load balancer in its decision making. Once a suitable node is selected, the requests are dispatched to the destination nodes.

A compute node represents an active VM available for task execution. An active VM would contain multiple concurrent functions in execution inside containers. A container with the required resource configurations is spawned for a new request execution. The VM loads the required runtime and the associated application code from the application repository, on to the launched container. As per our proposed approach, the Dynamic Resource Manager (DRM) module which is deployed on each VM, is responsible for monitoring the functions in runtime and handling dynamic resource re-provisioning to containers as applications approach their deadlines. It causes a dynamic update to a container's CPU resource limits until the task completes its execution. This module also enables eviction and re-scheduling of low priority task executions to avoid performance degradation in VMs due to resource contentions. Functions which were evicted are re-scheduled on a different node, by sending a new function invocation request to the controller where the function placement logic is called again to find a suitable compute node.

B. Problem Formulation

Based on the system model, we now formulate the function scheduling problem to minimize the provider cost of VM resources usage, and to meet execution deadlines of the functions. For the scope of this work we consider an application to be composed of a single function and hence the terms "function" and "application" are used interchangeably. Table II summarizes the important notations and descriptions presented in this paper.

Given an instance of a serverless platform, let $V = \{v_1, v_2, ..., v_N\}$ be the set of VMs or compute nodes available for function execution, where N is the total number of VMs and v_j, $1 \leq j \leq N$ is the jth VM. Each VM has available resource capacities defined by a two-dimensional vector: CPU and memory, represented as v_j^c and v_j^m respectively. Hence we have, $v_j =< v_j^c, v_j^m >$. The total CPU capacity in a VM is determined as the product of the number of cores and the processing power of each core, denoted in Million Instructions Per Second (MIPS). The available free CPU and memory resources in VM, v_j at time t is denoted by, $v_j^{ac}(t)$ and $v_j^{am}(t)$ respectively.

Let $R = \{r_1, r_2, ..., r_M\}$ be the sequence of function invocation requests received at the scheduler where M is the total number of requests and r_i, $1 \leq i \leq M$ is the ith request. Each request carries five attributes, i.e., $r_i =< r_i^{type}, r_i^{priority}, r_i^{ta}, r_i^d, r_i^m >$ where r_i^{type} represents the application ID of the function to be invoked, $r_i^{priority}$ denotes the user requested priority level for the request i, and r_i^{ta}, r_i^d and r_i^m are the time of arrival, specified deadline and the memory requirement of request i respectively. We assume the initial CPU allocation r_i^c, to the containerized function instance, to be done in proportion to the requested memory, adopting AWS Lambda's initial resource allocation policy [2] (i.e. if r_i^m is the requested memory and v_j^m and v_j^c are the total memory and CPU capacities of the VM, the allocated CPU capacity is $(r_i^m / v_j^m) * v_j^c$). Since we dynamically update CPU allocations to the function instances in the runtime, we use the notation $r_i^{uc}(t)$ to denote the updated CPU allocation to request r_i subsequently in time t.

TABLE II
DEFINITION OF SYMBOLS

Symbol	Definition
v	A VM or compute node available for function execution
N	The total number of available VMs
δ	The index set of all the available VMs, $\delta = \{1, 2, 3, ..., N\}$
M	The total number of function invocation requests
r	Function invocation request
r_i^{type}	Type of the function to be invoked by i^{th}request, r_i
$r_i^{priority}$	Requested priority level for the execution of i^{th}request, r_i
r_i^{ta}	Time of arrival of i^{th}request, r_i
r_i^d	Deadline for i^{th}request, r_i
r_i^m	Memory requirement for i^{th}request, r_i
r_i^c	Initial CPU allocation for i^{th}request, r_i
$r_i^{uc}(t)$	The CPU allocation for i^{th}request, r_i at time t
r_i^w	The waiting time for scheduling i^{th}request, r_i
r_i^p	Total processing time of i^{th}request, r_i
v_j^m	Total memory capacity of jth VM, v_j
v_j^c	Total CPU capacity of jth VM, v_j
$v_j^{am}(t)$	Available free memory in j^{th} VM, v_j at time t
$v_j^{ac}(t)$	Available free CPU capacity in j^{th} VM, v_j at time t

Now the challenge at hand is to decide the mapping of a request execution to an available VM where the application would start its execution inside a container with access to assigned resources, and to manage resource allocations to its containerized instance throughout the life time.

$$Schedule = \{r_i \rightarrow v_j\} \tag{1}$$

Function scheduling and resource allocation would be subject to the following constraints.

VM resource capacity constraints: A VM is chosen for function execution only if the requested memory and the initial CPU allocation requirements for a function execution do not exceed the available free memory and CPU capacities of the VM at time t, i.e.,

$$r_i^m \leq v_j^{am}(t) \tag{2}$$

$$r_i^c \leq v_j^{ac}(t) \tag{3}$$

We identify the VM's available memory and CPU resource levels as follows:

$$v_j^{am}(t) = v_j^m - \sum_{i=1}^{M} u_{ij}(t) r_i^m(t) \tag{4}$$

$$v_j^{ac}(t) = v_j^c - \sum_{i=1}^{M} u_{ij}(t) r_i^{uc}(t) \tag{5}$$

where we define a binary variable u_{ij} to indicate whether request i is currently placed in v_j or not, i.e., $\forall j \in \delta$, we have;

$$u_{ij}(t) = \begin{cases} 1, & \text{if request } i \text{ is being executed in } v_j \text{ at time } t \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

The time t in the above expressions: (2), (3), (4), (5) and (6) refers to the request arrival time i.e., r_i^{ta}.

Overall, the primary focus of this study is to minimize the provider expenses of running serverless applications by efficiently utilizing resources in VMs, while also minimizing the violation of user requirements of function execution.

In our work, we assume all the compute nodes to be homogeneous, and thereby the combined uptimes of all the VMs is representative of the provider's opportunity cost of utilizing the same resources for revenue generation from other services. Thus, we formulate the optimization problem for resource-efficient function scheduling as follows:

$$Minimize : T = \sum_{j=1}^{N} t_j \quad s.t. : (2), (3) \tag{7}$$

where t_j is the sum of the active periods of jth VM over the course of the experiment and T is the summation of the active periods of all the VMs used in the experiment. A VM is considered to be active when at least one container is running in it. We assume that VMs are available on-demand without additional start-up delays. Primarily, our proposed function placement logic contributes towards realizing this objective.

We also formulate the objective of minimizing user deadline violations as follows:

$$Minimize : Z = \sum_{i=1}^{M} x_i \quad s.t. : (2), (3) \tag{8}$$

where we define a binary variable x_i to indicate whether request r_i violates its deadline or not, i.e., $\forall i \in M$, we have;

$$x_i = \begin{cases} 1, & \text{if } r_i^w + r_i^p > r_i^d \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

where r_i^w is the waiting time for function scheduling, r_i^p is the processing time, r_i^d is the user specified deadline and Z is the total number of deadline violations. The deadline for a function indicates the expected maximum time to finish execution from the request arrival time. Our approach of dynamic resource management mainly contributes towards meeting the objective of deadline satisfaction.

IV. Proposed Algorithms

We propose a heuristic algorithm for the resource efficient placement of functions, and a dynamic resource alteration algorithm to solve resource contentions in VMs in the runtime and to meet user specified deadline constraints.

A. Function Placement Algorithm

The proposed heuristic function placement algorithm (Algorithm 1) follows a deadline sensitive function aggregation policy. The algorithm aims to align the runtimes of functions executing in a particular VM, such that the underutilization of VM resources is minimized, allowing the instance to be released after experiencing high utilization during its active life time.

We maintain a list of the existing active VMs ($VMList$) in the ascending order of the remaining time each of them are expected to run, depending on the functions already in execution and their stated deadlines. The remaining time to deadline $r_i^{\Delta t}(t)$, for request r_i at time t is expressed as follows:

$$r_i^{\Delta t}(t) = r_i^{ta} + r_i^d - t \tag{10}$$

Algorithm 1 Deadline Based Function Placement (DBP) Algorithm

```
Input: The function invocation request r_i,
       r_i =< r_i^type, r_i^priority, r_i^ta, r_i^d, r_i^m >
Input: The list of active VMs sorted in the ascending order
of the expected remaining runtimes, VMList
Output: VM selected for function execution, v_s
 1: procedure PROCESSVMSELECTION(r_i)
 2:     r_i^Δt(t) ← Time to deadline for r_i
 3:     CPU_max ← 0
 4:     for each VM v_j in VMList do
 5:         if r_i is a request for re-scheduling then
 6:             if v_j = r_i.GetOldVm then
 7:                 continue
 8:         v_j^Δt(t) ← v_j.GetRemainingRunTime
 9:         v_j^util(t) ← v_j.GetCPUUtilization
10:         if Placement of r_i in v_j satisfies resource
                capacity constraint and v_j^util(t) < v^util_T then
11:             v_temp ← v_j
12:             if v_j^Δt(t) ≥ r_i^Δt(t) then
13:                 if r_i^priority = Low then
14:                     v_s ← v_j
15:                     break
16:                 else
17:                     if v_j^ac(t) > CPU_max then
18:                         CPU_max ← v_j^ac
19:                         v_s ← v_j
20:     if v_s = null then
21:         if v_temp! = null then
22:             v_s ← v_temp
23:             return v_s
24:         else
25:             Add a new VM, vmNew to the active pool
26:             v_s ← vmNew
27:             return v_s
28:     else
29:         return v_s
```

where r_i^{ta} is the arrival time and r_i^d is the deadline, of request r_i respectively. We assume, that the longest of the remaining times to deadlines of the current functions running in a VM to be an approximation of its expected remaining runtime. The $VMList$ is updated whenever a new request is allocated to a VM or a function completes its execution.

When a new function invocation request (r_i) arrives at the controller, its time to deadline is calculated (line 2). Next the algorithm starts its iteration over the sorted list of active VMs. While we aim to maintain high VM utilization levels with the provider, for the set of active VMs at all times, we also try to avoid potential performance degradation caused by CPU contentions arising with resource overloading. Therefore, apart from the availability of sufficient free resources, a VM, v_j is considered for function placement only if its current CPU utilization, $v_j^{util}(t)$ is below a defined CPU utilization threshold v^{util_T} (line 10). If these requirements are met, this VM is chosen for execution if the expected remaining time of the VM is greater than the time to deadline of the request and the request bears a low level of priority (line 12-15). In case r_i is a high priority request, we choose the VM with the highest free CPU resources out of the VMs having higher remaining runtime than the time to deadline for r_i (line 17-19). We assume each request to be accompanied with either a high or low priority level and the high priority requests to have a tighter deadline than requests with low priority. Assigning a high priority request to a relatively less congested VM gives a better opportunity to dynamically monitor and increment the allocated CPU resources to the request in the runtime, if needed. It could be that none of the active VMs with sufficient free resources have the required remaining active time. In that case we choose the VM with the highest remaining active time. This ensures that the increase to the expected runtime of the chosen VM by the new function execution will be minimum (line 21-23). In case r_i is a request for re-scheduling, we avoid assigning the request to the same node it was previously being executed in (line 5-7). If none of the active VMs have sufficient free capacity for the function execution, a new free VM is added to the pool (line 25-27).

Once a VM is selected, the request r_i is forwarded to the selected compute node v_s for execution, where a new container with the requested resources is created and the function execution is initiated. Assuming the total number of VMs to be n, the worst case time complexity of Algorithm 1 for selecting a worker node, is $O(n)$.

B. Dynamic Resource Alteration (DRA) Algorithm

The proposed *Dynamic Resource Alteration* (DRA) algorithm (Algorithm 2) aims to alter the resource allocations to function instances approaching the deadline during runtime. The algorithm also efficiently manages resource contentions in the VMs by evicting suitable recently started tasks from constrained nodes to nodes with sufficient free resources.

The algorithm is executed by the VMs each time a task reaches a certain percentage of the task's time to deadline from the arrival time, denoted by d_{check}. We decide this

Algorithm 2 Dynamic Resource Alteration (DRA) Algorithm

Input: The function invocation request r_i
Input: Current VM, v_j and current container, c_{ij} of r_i
Input: The list of requests in execution in v_j sorted by time to deadline in descending order, r_{list}

1: **procedure** PROCESSRESOURCEALTERATION(r_i)
2: $v_j^{util}(t) \leftarrow v_j.GetCPUUtilization$
3: **if** $v_j^{util}(t) \geq v^{util_T}$ or c_{ij} has reached its max CPU allocation **then**
4: **return** $Failure$
5: **else**
6: $c_{ij}.UpdateResources\ (CPU)$
7: $v_j^{util} \leftarrow v_j.GetUpdatedUtilization$
8: **if** $v_j^{util} \geq v_j^{util_T}$ **then**
9: **for** each request r in r_{list} **do**
10: **if** r satisfies re-scheduling criteria (12) **then**
11: $c_r \leftarrow r.container$
12: Re-schedule request r
13: Destroy c_r
14: $v_j^{util} \leftarrow v_j.GetUpdatedUtilization$
15: **if** $v_j^{util} < v^{util_T}$ **then**
16: **break**

checkpoint based on the level of priority requested by each request on arrival (for example, a high priority request would have a better opportunity of meeting the deadline from a lower d_{check} value). At this point, if the function is still in execution, the initially allocated upper limit of CPU processing power to the container is incremented, provided that the underlying VM's CPU utilization level is below v^{util_T} and the container has not reached its maximum CPU allocation (line 3-6). We assume the maximum CPU power allocated to a container to be equal to that of one full vCPU core. Here we use the concept of cpu-quota and cpu-period enabled in Linux Kernel's Completely Fair Scheduler (CFS) [21], in setting and updating the CPU upper limits of the container. The cpu-quota value sets the number of microseconds per cpu-period that the container's access to CPU resources is limited to, before it is throttled [17]. Thus this acts as an effective ceiling and a hard limit for CPU resources allocated to a container. This is in contrast to the concept of cpu-shares used in [4], which adjusts the relative weight of CPU resources accessible to a container when co-located with other containers [17]. Container orchestration technologies allow updating the container resource configurations in the runtime [22]. During the evaluation of this approach, we conduct experiments while varying the d_{check} value and the cpu-quota increment values with the request priority levels.

After each resource update, the VM is checked for resource overloading (line 8). In the presence of resource overloading and performance interference as a result, the algorithm proceeds to efficiently evict some of the most recently scheduled tasks, scheduling them on another suitable node with sufficient free resources (line 9-12). Task eviction is undertaken only if

the re-scheduling cost satisfies criteria (12) below. The re-scheduling cost r_i^{cost} at time t for the ith request is defined by the time spent from arrival of r_i and the waiting time for function re-scheduling r_i^w, i.e.,

$$r_i^{cost}(t) = t - r_i^{ta} + r_i^w \tag{11}$$

$$r_i^{cost}(t) \leq r_i^d \times r^{evict_T} \tag{12}$$

where r_i^d is the task deadline and r^{evict_T} is a defined threshold for function eviction. This eviction policy ensures that the task execution loss is minimized and the evicted task has sufficient re-scheduling time until its deadline. Once a task is chosen for eviction, its running container is destroyed, freeing up resources in the constrained node. Re-scheduling of a task follows the function placement algorithm (Algorithm 1), by sending a function placement request to the controller. A node continuously monitors each function and employs the DRA algorithm, each time the time to deadline approaches the defined checkpoint. The process continues until the function finishes its execution or its container occupies a full vCPU core, which is assumed to be the maximum allocated CPU capacity for a function instance.

If v_j, the current VM of r_i, has r number of requests already in execution, under the worst case scenario, Algorithm 2 has a linear time complexity of $O(r)$.

V. Performance Evaluation

To evaluate the performance of our algorithms, we simulate a serverless computing environment using ContainerCloudSim [14] simulator. It is a simulation toolkit developed for modeling containerized cloud infrastructures. ContainerCloudSim is built on top of CloudSim [23] simulator which is widely used in evaluating resource management and scheduling techniques in cloud environments. We extended the simulator by implementing the Dynamic Resource Manager and VM Monitor modules as described in section III, to include our scheduling and dynamic resource management policies.

A. Baselines

We compare our function placement policy with the following baseline scheduling policies:

Round Robin (RR): This method tries to equally balance the load among the VMs by sending successive function requests to different VMs in a cyclic manner

Random Placement (RP): Function requests are randomly distributed among the VMs

Bin packing First-Fit (BPFF): Each request is directed to the first VM which satisfies the resource requirements of the request out of the active VMs, similar to AWS Lambda's function placement policy [7], packing the requests within a fewer VMs as possible.

Further, we compare our Dynamic Resource Alteration (DRA) technique with the following techniques:

Fixed Resource Allocation (FRA): The cpu-quota allocation to each container is done in proportion to the requested container memory and this CPU upper limit is maintained throughout the application lifetime, similar to the policy used in AWS Lambda serverless platform [7], [2].

OpenWhisk Resource Allocation (OW): OpenWhisk [20] sets the cpu-shares for each container proportional to the requested memory for each function, as mentioned in [4]. Cpu-shares indicates the relative weight given to a container in terms of the proportion of CPU time it is given access to when CPU resources are limited [17].

B. Experimental Set-up

We simulate a serverless computing environment with a cluster of VMs, each with four vCPU cores and 3 GB of memory. We follow the CPU configuration of Intel E5-2666 (2.9 GHZ), identified as one of the machine configurations seen in AWS Lambda infrastructure [7]. Since ContainerCloudSim identifies processor capacity in terms of MIPS (Million Instructions Per Second), we refer to CISCO's industry benchmark [24] in converting GHz values to MIPS (2.9 GHz $\rightarrow$ 11600 MIPS). We design experiments using cluster sizes of 12, 25 and 40 VMs each for three load levels of 4x, 8x and 16x requests per second, respectively. We use fixed time durations of 500 ms and 20 ms respectively as the container set-up delay and function scheduling delay in all the simulations. Variations and the impact of container start-up delay is not considered a part of this study. The VM CPU utilization threshold (v^{util_T}) is kept at 85% referring [25] and the threshold for function eviction (r^{evict_T}) is maintained at 20% in our experiments.

Experimental Workloads: We employ a number of synthetic and real-world traces to evaluate our proposed algorithms. The synthetic workloads enable us to observe the behavior of the system, while maintaining a constant request arrival rate at a time, and varying the rates across workloads. Under both the real and synthetic workloads, a request received at the controller consists of the id of the application to be invoked, the level of priority requested for the application execution, the requested container memory size and a user specified deadline parameter associated with each priority level. The deadline is the incremental value derived by increasing the average execution time of each application by a certain percentage. We use two levels of priority as high and low in our experiments, where the high priority requests are associated with a tighter deadline and the low priority requests with a more relaxed deadline.

We create a workload with real-world arrival patterns using trace snippets from Wikipedia [26] and Azure function traces [27]. We extract the set of single function applications from the Azure data set, and refer to the attributes of average container memory size and average execution times (we only consider applications with execution times exceeding 1 second in this study), coupled with the fluctuation of request arrival patterns from Wikipedia traces. We used the arrival patterns from Wikipedia traces since the Azure data set does not contain details of request arrival times. These traces drive the load for 140 application types with a peak load of 16x requests per second, and we run the experiments spanning for a period of one hour, using a cluster of 25 VMs.

Each synthetic workload consists of four synthetic traces which are created with requests arriving for four application types and run in parallel. Average application execution times and requested container memory sizes are generated randomly to be in the range of 1-50 seconds and between 128 - 512 MB in 64 MB increments, respectively. We conduct a series of experiments with multiple synthetic workloads created by varying the request deadline percentages for each priority level, and the arrival rates. In each workload, the inter arrival time of the function invocation requests is modeled using Poisson distribution as in [12], adjusting the Poisson mean to demonstrate different application load levels. A set of experiments are carried out for each workload, adjusting the different system parameters of the task deadline checkpoints (d_{check}) and cpu-quota increment values. At each experiment, we run the workload for a period of approximately 5 minutes in the simulation environment described above.

Performance Metrics: In all the experiments, we observe the performance metrics mentioned below.

1. Total VM uptime during the simulation time. Since we are considering a homogeneous resource environment in our experiments, VM uptime is used as a proxy for the function execution cost efficiency to the provider - we measure the intermittent VM uptimes and add them for all the VMs in the cluster.

2. Percentage of requests meeting the deadline - The number of requests finishing on or before the specified deadline as a percentage of the total number of requests.

C. Results and Analysis

We carry out performance evaluation in two steps for each of the experimental scenarios.

1. We run the experiments with our DBP algorithm (Algorithm 1) for the initial placement of functions and the DRA algorithm (Algorithm 2) for dynamically managing CPU resource allocations. The results are compared with the baseline schedulers: RR, RP and BPFF for load balancing, also coupled with the DRA algorithm for dynamic resource management.

2. The performance of our DBP algorithm for load balancing accompanied with the fine-grained DRA policy is evaluated with a Fixed Resource Allocation (FRA) policy and a cpu-shares policy similar to that adopted by OpenWhisk (OW).

We now discuss the results, primarily in terms of the efficiency in consuming the cloud resources and the level of satisfying the SLA requirements (meeting the function deadlines in this case) of the user.

Evaluation of resource efficiency: The efficient use of cloud resources is evaluated in terms of the total uptime of all the VMs used in each of the scenarios with different load levels of the incoming requests. The total time a cloud provider dedicates its resources for serverless function scheduling could be directly related to the cost incurred by the provider as means of the revenue lost during the same period by rendering other services using those resources. Figure 2 and Figure 3 present results of the resource efficiency study done using the synthetic workloads. Here we compare the VM uptimes using the DBP algorithm with the baseline schedulers under different scenarios, with dynamic provisioning of CPU resources and re-scheduling. Figure 2 depicts results under different load conditions when the incoming function execution requests have tighter deadlines (an increment of 5% and 15% over the average execution time for high and low priority requests respectively), while Figure 3 shows results for the same workload with relatively relaxed deadlines (an increment of 10% and 20% over the average execution time for high and low priority requests respectively). The results show that the DBP method is able to achieve a high resource efficiency similar to the BPFF heuristic, while the RP and RR schedulers show significantly higher resource usage levels and hence, lesser efficiency. We do experiments varying the time point of CPU re-provisioning (d_{check}) for the high and low priority requests, from 65% and 85% of remaining time to deadline (Figure 2(a)) to 55% and 75% (Figure 2(b)). We also incorporate two levels of CPU re-provisioning, changing the incremental cpu-quota/cpu-period value for each of the priority levels from 20% and 40% to 40% and 60% of CPU time of a vCPU core, for the same two scenarios. Results show that as we vary these system parameters to identify resource contentions faster and resort to better resource alterations (earlier checks for CPU re-provisioning and higher CPU quota increments), the overall VM uptimes decrease slightly, yielding better results. It is noticeable that the longer a varying load level prevails, the higher the distinction of resource efficiency between a random or a spread placement method, as compared to an

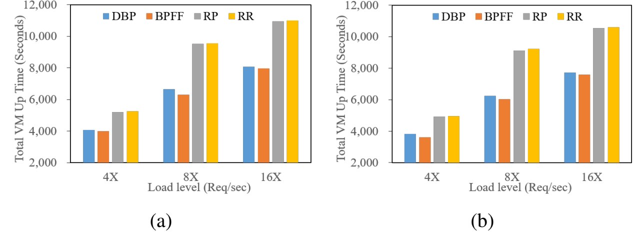

Fig. 2. VM uptime comparison for the different load balancing algorithms when requests have tighter deadlines at both priority levels (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%

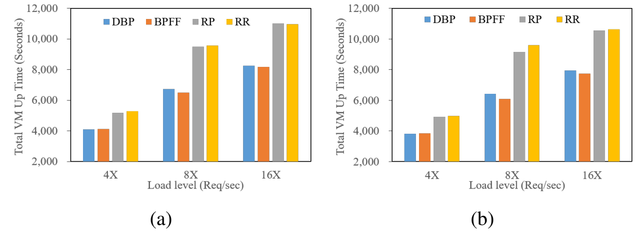

Fig. 3. VM uptime comparison for the different load balancing algorithms when requests have relaxed deadlines at both priority levels (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%

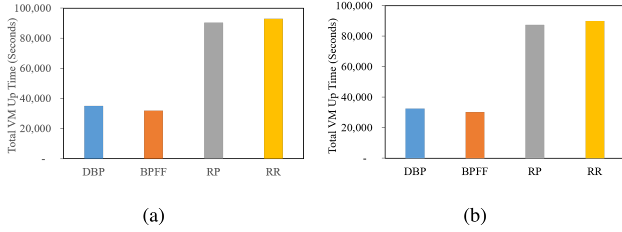

Fig. 4. VM uptime comparison for the different load balancing algorithms using real-world traces (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%

efficient bin-packing method. This is further emphasized by the experimental results from real-world traces, shown in Figure 4, where the VM uptimes recorded when using RR and RP algorithms are approximately 3 times that from DBP and BPFF algorithms.

Evaluation of deadline requirements: Evaluation of application/user SLA requirements is done by taking the percentage of functions meeting the set deadline. The results discussed here are from the same set of experiments described in the above section, for both the synthetic and real-world traces. As shown in Figure 5 and Figure 6, in all the scenarios with dynamic resource provisioning and re-scheduling, it is evident that the RP and RR load balancing algorithms are able to show higher levels of meeting deadlines. This is because they have a better opportunity of dynamically provisioning CPU resources to functions approaching their deadlines as required, since the initial function placement tends to happen in VMs with more free resources. Despite having relatively lower deadline met percentages, the DBP method is able to maintain a low level of deadline violations while also reducing the provider cost by the efficient use of cloud resources as discussed in the previous section. This is because the DBP algorithm considers the deadline priority level of the request in choosing a VM with either higher or lower free CPU quota levels. In general, the BPFF heuristic shows poor performance with higher deadline violations since it always tries to pack the function executions to a minimum number of VMs and hence show lesser flexibility in the ability to face resource contentions in the runtime. When compared with BPFF, DBP

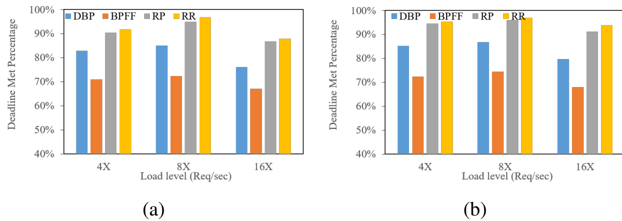

Fig. 5. Comparison of the percentage of requests meeting the deadline for the different load balancing algorithms when requests have tighter deadlines at both priority levels (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%

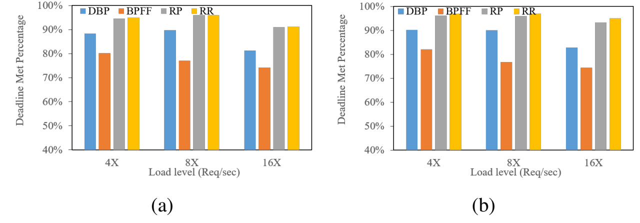

Fig. 6. Comparison of the percentage of requests meeting the deadline for the different load balancing algorithms when requests have relaxed deadlines at both priority levels (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%

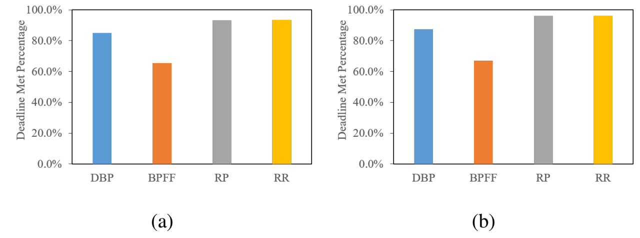

Fig. 7. Comparison of the percentage of requests meeting the deadline for the different load balancing algorithms using real-world traces (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%

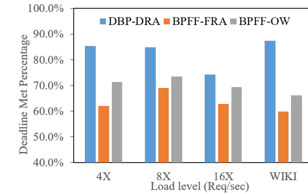

Fig. 8. Comparison of the percentage of requests meeting the deadline under different resource management methods

performs better, when function deadlines are tighter as well. As the load level increases to 16 requests/second, a slight increase in deadline violations is seen in all the scenarios. The results also show that dynamic resource provisioning and re-scheduling is better able to improve SLA violations when the VM CPU contentions are addressed early and with higher CPU quota increments (Figure 5(a) Vs. 5(b) and Figure 6(a) Vs. 6(b)). Figure 7 shows results from the workload created from real-world traces. The ability of DBP algorithm to maintain a higher level of deadline satisfaction compared to the BPFF algorithm, when the load level varies, is clearly observed here.

Figure 8 shows the performance of our load balancing approach of DBP with dynamic resource alteration (DRA), compared with a fixed CPU allocation policy (FRA) and a proportional cpu-shares policy (OW), both using BPFF as the load balancing method. It is seen that sub-optimal initial resource allocations and CPU performance variations in the runtime result in higher deadline violations under the fixed CPU allocation method. Under the cpu-shares policy,

the cpu-shares determine the relative weight of CPU power available to each function instance in the presence of CPU contentions [17]. Hence the functions co-located in a VM would largely affect each other's performance and the presence of a function instance with a larger cpu-share could cause a function instance with a relatively smaller share to perform poorly. In comparison, our policy of SLA-aware dynamic handling of the CPU time available to each function, is able to result in better performance under all the load levels.

VI. Conclusions and Future Work

Cloud service providers have an increased responsibility in serverless computing environments, in scheduling and provisioning resources for function executions. An increased level of abstraction in specifying resource requirements by the developers encourages a serverless platform to infer resource requirements on its own. Setting fine-grained limits on container CPU usage, dynamic monitoring and updating of resource allocations to containerized function instances is a promising approach in this regard, with many benefits. As evidenced by our experiments, such a mechanism allows for finer-grained control of CPU usage, reduces CPU contentions among containers fighting for the same set of resources in a VM and thereby enables higher opportunities for meeting SLA requirements of the cloud user. Further, the increased granularity in billing under the serverless scenario raises the requirement for the provider to be more considerate on their resource costs and our proposed algorithm for packing requests on VMs is able to maximize resource efficiency, while satisfying application specific SLA requirements.

In future work, we plan to explore following research directions; extend our dynamic resource management policy to include more precise resource usage predictions by profiling workloads, explore provider resource efficiency under heterogeneous cloud environments, enable greater flexibility and Quality of Service (QoS) offerings to users in terms of latency, address data dependency challenges in DAG based serverless application scheduling, and demonstrate the practical applicability of our proposed resource scheduling and provisioning strategy on Apache Openwhisk, an open source serverless platform.

References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[2] "Aws lambda - developer guide," https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf, (Accessed on 08/31/2020).

[3] "Quotas — cloud functions documentation — google cloud," https://cloud.google.com/functions/quotas, (Accessed on 08/31/2020).

[4] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," in *Proceedings of the 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 1–10.

[5] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless linear algebra," *arXiv preprint arXiv:1810.09679*, 2018.

[6] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 445–451.

[7] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the Annual Technical Conference*, 2018.

[8] M. Stein, "Adaptive event dispatching in serverless computing infrastructures," Ph.D. dissertation, Brunel University London, 2018.

[9] "Deploy services to a swarm — docker documentation," https://docs.docker.com/engine/swarm/services/, (Accessed on 08/31/2020).

[10] M. Stein, "The serverless scheduling problem and noah," *arXiv*, pp. arXiv–1809, 2018.

[11] G. Aumala, E. Boza, L. Ortiz-Avilés, G. Totoy, and C. Abad, "Beyond load balancing: Package-aware scheduling for serverless platforms," in *Proceedings of the 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019, pp. 282–291.

[12] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman, and A. Akella, "Archipelago: A scalable low-latency serverless platform," *arXiv preprint arXiv:1911.09849*, 2019.

[13] A. Das, A. Leaf, C. A. Varela, and S. Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," in *Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 609–618.

[14] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "Containercloudsim: An environment for modeling and simulation of containers in cloud data centers," *Software: Practice and Experience*, vol. 47, no. 4, pp. 505–521, 2017.

[15] M. HoseinyFarahabady, Y. C. Lee, A. Y. Zomaya, and Z. Tari, "A qos-aware resource allocation controller for function as a service (faas) platform," in *Proceedings of the International Conference on Service-Oriented Computing*. Springer, 2017, pp. 241–255.

[16] N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu, "Optimizing serverless computing: introducing an adaptive function placement algorithm," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, 2019, pp. 203–213.

[17] "Runtime options with memory, cpus, and gpus — docker documentation," https://docs.docker.com/config/containers/resource_constraints/, (Accessed on 10/16/2020).

[18] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 158–164.

[19] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya, "Automated fine-grained cpu cap control in serverless computing platform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2289–2301, 2020.

[20] "Apache openwhisk is a serverless, open source cloud platform," https://openwhisk.apache.org/, (Accessed on 11/23/2020).

[21] P. Turner, B. B. Rao, and N. Rao, "Cpu bandwidth control for cfs," in *Proceedings of the Linux Symposium*. Citeseer, 2010, p. 245.

[22] "docker update — docker documentation," https://docs.docker.com/engine/reference/commandline/update/, (Accessed on 09/03/2020).

[23] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[24] "vbootcamp_performance_benchmark.pdf," https://www.cisco.com/c/dam/global/da_dk/assets/docs/presentations/vBootcamp_Performance_Benchmark.pdf, (Accessed on 10/01/2020).

[25] Z. Zhong, J. He, M. A. Rodriguez, S. Erfani, R. Kotagiri, and R. Buyya, "Heterogeneous task co-location in containerized cloud computing environments," in *Proceedings of the 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2020, pp. 79–88.

[26] "Wikipedia access traces — wikibench," http://www.wikibench.eu/?page_id=60, (Accessed on 12/02/2020).

[27] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (ATC 2020)*, 2020, pp. 205–218.