Parallel VM Deployment with Provable Guarantees

Itamar Cohen
Politecnico di Torino
Torino, Italy
itamar.cohen@polito.it

Gil Einziger
Ben-Gurion University of the Negev
Beer Sheva, Israel
gilein@bgu.ac.il

Maayan Goldstein
Nokia Bell Labs
Kfar Saba, Israel
maayan.goldstein@nokia.com

Yaniv Sa'ar
Independent
Israel
yaniv.saar.mail@gmail.com

Gabriel Scalosub
Ben-Gurion University of the Negev
Beer Sheva, Israel
sgabriel@bgu.ac.il

Erez Waisbard
Nokia Bell Labs
Kfar Saba, Israel
erez.waisbard@nokia.com

***Abstract*—Network Function Virtualization (NFV) carries the potential for on-demand deployment of network algorithms in virtual machines (VMs). In large clouds, however, VM resource allocation incurs delays that hinder the dynamic scaling of such NFV deployment. Parallel resource management is a promising direction for boosting performance, but it may significantly increase the communication overhead and the decline ratio of deployment attempts. Our work analyzes the performance of various placement algorithms and provides empirical evidence that state of the art parallel resource management dramatically increases the decline ratio of deterministic algorithms, but hardly affects randomized algorithms. We therefore introduce APSR – an efficient parallel random resource management algorithm that requires information only from a small number of hosts and dynamically adjusts the degree of parallelism to provide provable decline ratio guarantees. We formally analyze APSR, evaluate it on real workloads, and integrate it into the popular OpenStack cloud management platform. Our evaluation shows that APSR matches the throughput provided by other parallel schedulers, while achieving up to 13x lower decline ratio and a reduction of over 85% in communication overheads.**

I. Introduction

The *Network Function Virtualization (NFV)* paradigm enables network infrastructure to be virtually deployed on standard cloud infrastructure. Specifically, NFV allows running firewalls, deep packet inspection, load balancing, and monitoring without relying on physical middleboxes [12], [29]. NFV is composed out of (often long) service chains that each packet needs to traverse. One of the main advantages of NFV is the ability to scale the service chain on demand without any physical change to the network. Unfortunately, current cloud placement is not optimized for high-throughput placement, making large service chains slow to deploy.

In principle, once the user issues a request to allocate a new *Virtual Machine (VM)*, a *scheduler* selects a host to accommodate the VM. While the deployment time of optimized VMs or containers (e.g., using Kubernetes) can be tens of milliseconds [18], selecting a host on which to place the VM may require hundreds of milliseconds in large clouds [2], [4], [13]. It follows that the potential performance boost of using NFV remains largely unfulfilled in large clouds due to bottlenecks in scheduling deployment requests.

The main reason that the host selection process takes so long is that most current resource management algorithms [15], [16], [25], [28] require complete information about the availability of resources on the system's hosts. In a large cloud, gathering the current state from hundreds and sometimes thousands of hosts translates to high communication overheads, resulting in a performance bottleneck [4], [13]. In particular, some experiments show that when the number of hosts is above 400, more than 90% of the scheduling time is wasted in collecting the fresh system's state information [2].

Intuitively, one could boost throughput by running multiple schedulers in parallel. However, such an approach may translate to having multiple schedulers try and place requests simultaneously on the same host, leading to race scenarios [22], [24], [27]. In such cases, not all the requests will be successful, and the host may *decline* some of the requests. Such a decline translates to having the scheduler retry to serve the same request, resulting in excessive latency. This added latency may be unacceptable in an NFV environment, which may have to respond to bursts of requests, e.g., when the system has to respond to a flash crowd or a cyber attack [4]. Hence, a provider is typically required to bind the *decline ratio*, the ratio between the number of declined requests and the total number of requests. The maximum allowed decline ratio is typically defined in the *Service Level Agreement (SLA)* [4], [9], or in the *Key Performance Indicators (KPIs)* [17].

An efficient VM placement algorithm should, therefore, strive to (i) increase parallelism, while (ii) maintaining a low communication overhead, and (iii) ensuring a bounded decline ratio. However, to the best of our knowledge, no previous work has studied the interplay between these conflicting aspects.

Our contributions. Our work starts by studying the impact of parallelism on the decline ratio of various popular placement algorithms. We show that parallelism may drastically increase the decline ratio, where we attribute this increase to the determinism of most algorithms. Interestingly, we find that

This work was done while the first author was with Ben-Gurion University.

ISBN 978-3-903176-39-3 © 2021 IFIP

randomly placing VMs in suitable hosts allows for a large degree of parallelism without a significant impact on the decline ratio. That is, random placement is very efficient in parallel settings. Our study further shows that in the random policy, the decline ratio depends on the number of parallel schedulers and the number of hosts that can accommodate each VM. In general, low-utilization environments allow for more schedulers than high-utilization ones.

Equipped with these observations, we introduce our proposed algorithm, APSR, that dynamically adjusts the number of parallel schedulers according to the system's utilization and incorporates randomness into its decision making. APSR guarantees that the expected decline ratio is always within a predefined requirement. Furthermore, APSR is inherently optimized to query only a small number of hosts, thus reducing the communication overheads.

We formally analyze the performance of APSR where we provide guarantees as to its communication overhead and its expected decline ratio. We also evaluate the performance of APSR for three real-life datasets and show that it enables a high degree of parallelism (e.g., effectively running 20-100 schedulers) in various realistic scenarios. We further show that APSR reduces the communication overhead by over 85% compared to state-of-the-art algorithms. Finally, we integrate and implement APSR within the OpenStack framework and show that it matches the throughput of the fastest OpenStack configuration while significantly reducing the decline ratio and the communication overhead. Due to space constraints, for some of our technical claims, we provide only a proof sketch.

II. Related Work

This section provides a short survey of commonly used VM placement paradigms. For each such approach, we discuss the various algorithms that apply it in their design. We further provide insight into the main differences between our suggested solution and these algorithms, summarized in Table I.

The global snapshot-based approach. Traditionally, placement algorithms take a snapshot of the entire system's state before handling each request. This precise state information allows for a single *monolithic* scheduler (e.g., Maui [5]) to select a host to accommodate the request while prioritizing the hosts in some manner. The monolithic approach guarantees a low decline ratio, as the scheduler operates alone on an up-to-date view of the available resources. However, the per-request overhead of this approach is substantial due to both the communication overhead of querying all hosts [2], [4], [13] and the latency of computing the placement decision itself, which may take several seconds [27]. Such a long latency might be reasonable when scheduling large batch jobs (e.g., in HPC environments) but is prohibitively costly when a prompt reaction is critical, e.g., when scaling out the capacity of a service chain due to an increase in demand.

One of the ways suggested for decreasing the overhead of the monolithic scheduler is to periodically *cache* a snapshot of the system's state [4]. However, when the cached state becomes stale, the scheduler may be unaware of resources that have recently become available, resulting in an increased number of needlessly declined requests. Furthermore, this approach achieves low throughput as it only employs a single scheduler.

Running multiple schedulers in parallel is a straightforward technique to increase throughput. Indeed, OpenStack allows for multiple parallel schedulers to increase the throughput [7]. However, our work shows that running multiple independent schedulers translates to collisions when multiple schedulers simultaneously select the same hosts. Such collisions result in excessive decline ratios, degrading performance. Interestingly, the OpenStack community acknowledges this problem and mitigates its impact by allowing the user to add a certain degree of randomness to the schedulers [22], [24]. Further, the seminal work of [27] shows that when system utilization is high, Google's schedulers require more than two attempts to place each request. Our work shows (in Section IV) that parallel scheduling yields high decline ratios for a variety of placement algorithms and that random placement is more robust than deterministic placement. Intuitively, deterministic algorithms select the same "best" host, which renders them inferior to random algorithms.

To insert some degree of randomness into the scheduling process, the OpenStack community introduced the parameter `scheduler_host_subset_size` [22] (denoted ℓ), which works as follows: After ranking the available hosts, the scheduler randomly assigns the request to one of the top-ℓ ranking hosts. In the absence of a rigorous theory studying the effect of ℓ on the system's performance, its value is commonly determined using crude estimations and rules of thumb. Our work is useful to configure the parameter ℓ properly. Furthermore, in Section IV, we show that the common approach of setting ℓ as a small constant results in poor performance.

The Omega scheduler [27] suggests a new approach that aims at optimizing the usage of a global snapshot by multiple schedulers. Omega decreases the communication overheads by allowing multiple schedulers to share state information. However, Omega does not provide guarantees on the decline ratio. Thus, our approach is also useful in Omega's framework.

The partitioning-based approach. *Partitioning* the hosts between different schedulers is a simple approach that removes conflicts between schedulers and decreases the pre-placement communication overheads as each scheduler only acquires state information about some of the hosts. Quincy [11] uses a *static* partition, that occasionally results in non-compulsory declines due to fragmentation of resources [27]. Namely, a scheduler may fail to place a request in its partition, even if hosts in other partitions can accommodate the request. Mesos [10] suggests using *dynamic* partitioning, where a central controller dynamically allocates hosts to schedulers on demand to minimize fragmentation at the expense of complexity. Note that Quincy and Mesos provide no guarantees on the impact of fragmentation on the decline ratio.

The sampling-based approach. The sampling-based approach was extensively studied in the context of balanced allocation problems [1], [3], [20], [24]. These problems essentially assume an (infinite) buffer for pending requests in

TABLE I

COMPARISON OF APPROACHES FOR SCHEDULING REQUESTS IN A MULTI-HOST SYSTEM. THE DIFFERENT APPROACHES ARE COMPARED IN TERMS OF THEIR (I) THROUGHPUT (RATE OF ASSIGNMENT ATTEMPTS), (II) DECLINE RATIO (EXPECTED RATIO OF ATTEMPTS THAT FAIL), AND (III) OVERHEAD (AMOUNT OF COMMUNICATION/SYNCHRONIZATION REQUIRED TO GATHER THE STATE INFORMATION FOR MAKING AN ASSIGNMENT DECISION). FOR EACH OF THE APPROACHES, WE PROVIDE SOME CONCRETE EXAMPLES OF ACTUAL ARCHITECTURES THAT IMPLEMENT THE APPROACH.

Approach	#Schedulers	Description	Throughput	Decline Ratio	Overhead	Examples
Global Snapshot	Single	Monolithic	Low	Low, guaranteed	High	Maui [5]
		Cached Snapshot	Low	Low	Low	ASC [4]
	Fixed	Multiple Snapshots	High	High	High	OpenStack [7]
		Shared Snapshot	High	High	Low-Mid	Omega [27]
Partitioning	Fixed	Static Partition	Mid	Mid	Low	Quincy [11]
		Dynamic Partition	Mid	Mid	Low-Mid	Mesos [10]
Sampling	Adaptive	Adaptive sample size	Mid-High [1]	Low, guaranteed	Low, guaranteed	Our APSR algorithm

each host, and the goal is to allocate requests to hosts in a way that minimizes the maximum load on all hosts. The celebrated power-of-two-choices [1], [20] algorithmic paradigm shows that sampling only a few (e.g., two) hosts and selecting the least-loaded sampled host provides strong guarantees on the expected maximal load. Sparrow [24] and Tarcil [3] implement variants of this concept in concrete cloud environments.

However, balanced allocation problems are inherently different from the ones addressed in our work as they consider infinite capacity hosts that never decline requests, and instead their algorithms make an effort to evenly balance the load [1], [3], [20], [24]. In contrast, we consider finite-capacity hosts that decline requests that exceed their capacity limitations, making load balancing based algorithms incomparable with our work. That said, our APSR is part of the sampling-based approach as it queries a small number of hosts, and while load-balancing based algorithms provide guarantees on the maximum load [1], [3], [20], [24], APSR provides guarantees on the decline ratio.

III. SYSTEM MODEL FOR PARALLEL SCHEDULING

We consider a collection $\mathbf{H}$ of n hosts where each host has some multi-dimensional capacity corresponding to several types of resources, e.g., memory, CPU, or disk space. Formally, we model each $\vec{h} \in \mathbf{H}$ as a vector whose coordinates correspond to the currently available resources of each type. We refer to this vector as the *state* of the host. We further consider a collection $\mathbf{R}$ of *requests*, each modeled as a vector of demand for each resource. We assume each request $\vec{r} \in \mathbf{R}$ has its vector drawn from some finite set of possible request vectors, or *flavors*, $\mathbf{C} = \{\vec{c}_1, \ldots, \vec{c}_m\}$. A host $\vec{h}$ is considered *available* for request $\vec{r}$ if it has enough resources of each type, i.e., if $\vec{r} \leq \vec{h}$, coordinate-wise.

We assume time is slotted, such that in every time slot, some requests arrive at the system, and are queued, pending assignment to hosts. We denote by s the number of *parallel schedulers* that may perform scheduling decisions simultaneously in any single time slot. In each time slot t, given a queue consisting of some q requests pending at t, each scheduler dequeues a request. Schedulers may query (sample) the state of some subset of hosts, and assign the request to an available host (if they queried such a host). We note that when $s > 1$, multiple schedulers may concurrently assign their pending requests to the same host.

Any host $\vec{h} \in \mathbf{H}$ resolves concurrent requests being assigned to $\vec{h}$ at the same time slot in some arbitrary order. The resolution of request $\vec{r}$ being assigned by some scheduler to host $\vec{h}$ *fails* if the host is no longer available when it resolves $\vec{r}$, and is *successful* otherwise. The host updates its available capacity upon a successful resolution by setting $\vec{h} = \vec{h} - \vec{r}$. Requests live for some time, and the host regains the resources used by completed requests. If request $\vec{r}$ placed on host $\vec{h}$ is completed we update the resource state of the host by setting $\vec{h} = \vec{h} + \vec{r}$. The above model implies that a request fails if either (i) the scheduler does not find an available host, or (ii) the chosen host is no longer available once it resolves the request.

In every time slot t, and for every request flavor $\vec{c} \in \mathbf{C}$, we let $k_{\vec{c}}^{(t)}$ denote the number of hosts in $\mathbf{H}$ that are available for a request of flavor $\vec{c}$ at time t. We further let $k^{(t)}$ denote an *estimate* of the number of hosts that may accommodate *any* request that may arrive at time t. We note that $k^{(t)}$ may be a pessimistic estimate (e.g., by setting $k^{(t)} = \min_{\vec{c}} k_{\vec{c}}^{(t)}$), or it may incorporate some information about the workload distribution, or otherwise the system state. We will usually be omitting the superscript of (t), and refer to $k_{\vec{c}}$, and k, when the time slot in question is clear from the context.

The *decline ratio* is the ratio between the number of failed assignment attempts and the total number of assignment attempts performed by the system. We use δ to denote the system's expected decline ratio (for some set of requests $\mathbf{R}$). Since we are handling requests independently, δ is the a posteriori probability of having a declined assignment attempt.

We assume the system is subject to a *Service Level Agreement (SLA)* which requires that the decline ratio is at most ε, for some $\varepsilon \in [0, 1]$.[2] To control the overheads, we limit the maximal number of hosts queried (by all schedulers) in each time slot to B. In every time slot t, we denote by d the number of hosts queried by any scheduler with a pending request at t. A *valid configuration* of schedulers determines s and d, such that

[1] Throughput is inversely proportional to system utilization, and adapts to the amount of resources available in the system.

[2] Current algorithms are oblivious to such constraints, and might violate this requirement. Our APSR algorithm takes such constraints into account, and produces solutions which provably satisfy them.

TABLE II
LIST OF SYMBOLS. THE TOP SECTION CORRESPONDS TO OUR SYSTEM MODEL (SECTION III), THE MIDDLE SECTION CORRESPONDS TO OUR PERFORMANCE GUARANTEES (SECTION VI), AND THE BOTTOM SECTION CORRESPONDS TO OUR EVALUATION (SECTION VIII)

Symbol	Meaning
$\mathbf{H}$	Set of hosts
n	Number of hosts (bins)
$\vec{h}$	Host in $\mathbf{H}$ (resources availability vector)
$\mathbf{R}$	Set of requests
$\vec{r}$	Request in $\mathbf{R}$ (resources demand vector)
$\mathbf{C}$	Set of requests flavors
$\vec{c}$	Flavor in $\mathbf{C}$ of a request
s	Number of schedulers (agents)
δ	Actual decline ratio
ε	Maximum allowed decline ratio by the SLA
B	Budget for overall number of queries
d	Number of hosts queried by each scheduler
$n_{\vec{c}}$	Number of hosts queried for requests of flavor $\vec{c}$
$k_{\vec{c}}$	Number of available hosts for flavor $\vec{c}$
k	Number of available hosts for any request
F_s	Number of potentially happy agents
H_s	Number of happy agents
σ	See (2)
$Bin(a,b,c)$	See (4)
λ_a	Poisson arrival rate
λ_d	Poisson departure rate

$s \cdot d \leq B$, and the probability of a failed assignment attempt is at most ε. We seek the valid configuration maximizing the number of parallel schedulers (s). Table II summarizes the notation used in our model, as well as further notation defined in later sections.

IV. PARALLELISM AND PLACEMENT ALGORITHMS

We begin by evaluating the effect of parallel schedulers on the decline ratio of existing placement algorithms.

A. Evaluated Algorithms

We briefly introduce some common placement algorithms. For further details see, e.g., [19].

OpenStack's default placement algorithm is the *WorstFit (WF)* algorithm [7]. WF places requests on one of the least loaded hosts to maximize the hosts' remaining resources. For the multi-dimensional settings, we implement a pessimistic variant of WF, where we consider a host load to be the maximum load over all the possible resources.

The *FirstFit (FF)* [6] algorithm assigns a request to the first available host, assuming some arbitrarily fixed ordering of the hosts. This approach aims at minimizing the number of utilized hosts, thus reducing energy consumption.

The *Adaptive* algorithm [25] combines WF and FF as follows: It begins like WF; once the load passes a threshold, the algorithm switches to an FF regime. Throughout our evaluation, we used 0.6 as the threshold for the Adaptive algorithm.

The algorithm *DistFromDiag* [25] attempts to balance the host's resource consumption according to its proportions. For example, if a host has 100GB disk and 10GB RAM, it aspires for a 10:1 ratio between available disk and RAM.

We also consider two algorithms that incorporate randomization into WF and FF. These variants, referred to as *WorstFit-Rand (WFR)* and *FirstFit-Rand (FFR)*, respectively, weigh the hosts based on the WF and FF strategies but randomly select a host from the ℓ top-ranking available hosts (in the spirit of the option available in OpenStack, as described in Section II). In our evaluation of WFR and FFR, we set $\ell = 5$.

Finally, we evaluate the *Random* algorithm, which selects a host uniformly at random among the available hosts.

B. Datasets

We use three datasets that capture requests made in real systems. We evaluate each workload in a cloud environment that has sufficiently many hosts to accommodate all the requests (see Section IV-C for details on choosing the number of hosts).

The NFV Dataset was collected from a proprietary large NFV management and orchestration (MANO) system [4]. In this scenario, hosts are identical, and the placement requests are for VMs of preset sizes (flavors). Hosts and placement requests are two-dimensional tuples of the form $\langle memory, storage \rangle$. The sizes are normalized such that hosts' capacity is $\langle 1, 1 \rangle$, and each VM requires a certain fraction of this capacity. Table III shows the distribution of flavors for this dataset.

TABLE III
NORMALIZED BREAKDOWN OF REQUESTS FOR VM IMAGES BY MEMORY AND STORAGE, OBTAINED FROM THE NFV DATASET.

		$storage$					
		0.01	0.04	0.1	0.3	0.54	Total
$memory$	0.001	14	22	14	3	13	66
	0.016	7	93	0	2	0	102
	0.032	83	165	0	14	0	262
	0.064	1	1	1	0	0	3
	0.19	0	2	0	0	2	4
	Total	105	283	15	19	15	437

The Google Dataset, recorded in a Google's cluster [26], holds data from 12,477 virtual machines characterized by tuples of $\langle CPU, memory \rangle$. The normalized CPU values vary between 0.25, 0.5, and 1, while the memory values can be grouped around five levels: 0.125, 0.25, 0.5, 0.75, and 1 [14]. The hosts capacities are either $\langle 1, 2 \rangle$ or $\langle 2, 1 \rangle$ in equal proportions [25]. Table IV provides the breakdown of flavors for this dataset.

TABLE IV
BREAKDOWN OF THE NUMBER OF PLACEMENT REQUEST SIZES BY CPU AND MEMORY, OBTAINED FROM THE GOOGLE DATASET.

		CPU			
		0.25	0.5	1.0	Total
$memory$	0.125	0	60	0	60
	0.25	123	3,835	0	3,958
	0.5	0	6,672	3	6,675
	0.75	0	992	0	992
	1.0	0	4	788	792
	Total	123	11,563	791	12,477

The Amazon Dataset is based on data from Amazon EC2 hosts and VM flavors [19], [25]. Table V depicts the flavors of the normalized $\langle CPU, memory \rangle$ in this dataset, where each

TABLE V
BREAKDOWN OF PLACEMENT REQUEST FLAVORS OF $\langle CPU, memory \rangle$ OBTAINED FROM THE AMAZON EC2 DATASET. FLAVORS ARE SORTED BY CPU.

	Small									Large					
CPU	0.035	0.07	0.083	0.1	0.142	0.167	0.2	0.333	0.354	0.4	0.5	0.5	0.8	0.833	1
$memory$	0.008	0.016	0.031	0.008	0.031	0.063	0.016	0.125	0.062	0.031	0.125	0.5	0.063	0.25	0.25

column represents one possible flavor of requests. We partition requests' flavors into two types: *small* flavors, which have a CPU requirement below 0.4, and *large* flavors, which consist of all remaining flavors. We generate a sequence of 1000 small requests and 100 large ones (i.e., a total of 1100 requests) and select a flavor for each request uniformly at random from the corresponding flavor types. In this scenario we consider hosts with capacities of either $\langle 1, 2 \rangle$, or $\langle 2, 1 \rangle$ in equal proportions (similarly to the host setup used in the Google dataset).

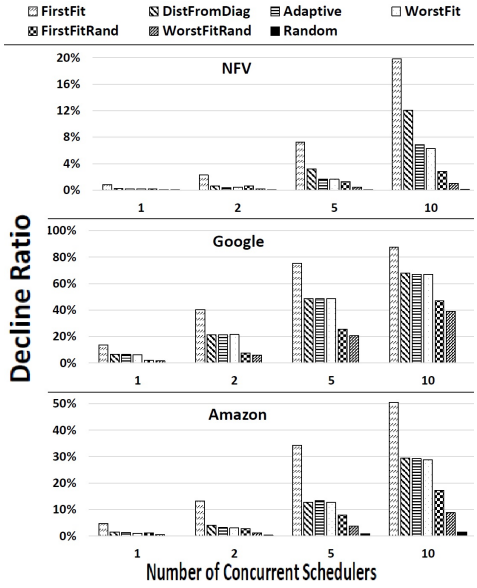

Fig. 1. Decline ratios for different placement algorithms and varying number of parallel schedulers on the NFV, Google and Amazon datasets. Note that the decline ratio (y-axis) ranges corresponding to the various datasets are distinct.

C. Experiments

We now turn to study the effect of running multiple parallel schedulers with existing algorithms. We select a number of hosts that enable placing all requests at once (by some algorithm). Evaluating the required number of hosts to accommodate all the requests in a given trace is equivalent to the multi-dimensional bin packing problem, which is NP-hard [8] and thus we approximate this number as suggested in [25]: We run the trace for each algorithm multiple times, each time with a randomly-generated order of requests. Whenever the placement algorithm fails to accommodate a request with the currently available resources, we open a new host. The approximated value is the minimal number of open hosts in all runs.

To simulate large clouds, we replicated the NFV dataset to have 4730 requests with 279 hosts. The Amazon dataset is evaluated with 126 hosts, and the Google dataset with 5989 hosts. In our experiments, we make just one attempt to place any request (i.e., we do not retry placing declined requests).

Our results are illustrated in Fig. 1. When using a single scheduler, there are very few failures in all policies. Yet, the decline ratio in Random remains low also for higher levels of parallelism. This result is intuitive as randomly allocating requests to hosts minimizes the probability of having many schedulers select the same host concurrently. In contrast, FirstFit is the worst, as all the schedulers select the same host even if it is close to being full. In other algorithms like WorstFit, once a host is nearly full, it is less attractive, and thus the schedulers distribute their placement decisions upon a larger number of hosts.

The decline ratio of the deterministic algorithms becomes very high, even when running only 10 schedulers. This problem is somewhat mitigated by OpenStack's solution of introducing small randomization into traditional algorithms (as captured by FFR and WFR). However, statically setting $\ell = 5$ is insufficient when having ten schedulers. These results show that the OpenStack community correctly identified the problems with parallelism and introduced a valid workaround. However, the interplay between parallelism and decline ratio has not been studied. Our work builds upon the insights drawn from the above results and claims that one should use randomness to maximize the parallelism in resource management. In particular, our goal is to study the scaling laws of parallelism when combined with random VM placement.

V. ADAPTIVE PARTIAL STATE RANDOM (APSR)

This section presents our algorithm *Adaptive Partial State Random (APSR)*. Motivated by our observations from Section IV, APSR implements an efficient random policy that dynamically adjusts the number of schedulers (s) according to the system's perceived utilization. Whenever APSR uses parallel schedulers ($s > 1$), it is guaranteed to satisfy the SLA and budget constraints.

Upon receiving a placement request, each APSR scheduler does the following: (i) queries d hosts (for some value d), (ii) filters out hosts that cannot accommodate the request, (iii) randomly selects an available host out of the remaining set of hosts, and (iv) sends the request to the chosen host.

APSR relies on a centralized controller called the *APSR controller* to do the following periodically: (i) estimate the system's utilization, captured by the estimate k of the number of available hosts, (ii) determine the number s of parallel schedulers, and (iii) determine the number d of hosts each scheduler queries per request. The controller determines the above parameters to ensure the validity of the configuration.

Algorithm 1 APSR Controller (n, ε, B, T)

```
s ← 1, k ← n
GenerateSchedulers(1, B)
for every time slot t = T, 2T, 3T, ... do
    k ← EstimateK(...)
    (s, d) ← MaximizeParallelism(n, ε, B, k)
    GenerateSchedulers(s, d)
end for
```

Algorithm 1 illustrates the APSR controller algorithm. The procedure GenerateSchedulers(s, d) adjusts the number of schedulers to s and the number of hosts queried by each scheduler to d. The method EstimateK provides an estimate of the number of hosts k that can accommodate a request. We do not specify the arguments for this method since it can be implemented in a variety of ways (see details in Section VII). The procedure MaximizeParallelism considers the system state and the SLA constraints and outputs the number of schedulers s and the number of hosts each scheduler queries (d).

VI. Analysis

We now establish the correctness of our approach. We start with a simplified *balls-and-bins* model where hosts are unit-size *bins*, and requests are unit-size *balls*, implying that each bin can store at most one ball. Each scheduler is an *agent* assigning balls to bins. We show sufficient conditions for satisfying the SLA requirement in this simplified model. Our conditions provide a lower bound on the number of parallel agents for a given failure probability. We further show that the decline ratio serves as an upper bound on the original model's decline ratio. These results imply that when APSR utilizes parallelism, the decline ratio is at most ε, and the total number of queries performed by all agents is at most B.

A. Balls-and-bins Model

Assume s identical agents acting in parallel, trying to place balls in available bins. Each agent queries d random bins and possibly finds some of them available. If the agent does not find any available bins, the ball assignment fails. Otherwise, the agent selects an available bin uniformly at random and tries to place its ball in that bin.

Agents are unaware of the decisions made by other agents, which may cause multiple agents to select the same available bin. In such a case, one of the agents succeeds, and the rest of them fail. We use the term *potentially happy agent* to refer to an agent that finds an available bin. Similarly, the term *happy agent* refers to an agent that successfully places a ball in an available bin. Finally, we use the term *unhappy agent* to refer to an agent that fails to place its ball (either due to collision or due to not finding an available bin).

We let the random variables, F_s and H_s, denote the number of potentially happy agents and happy agents. We denote by k a lower bound on the number of available bins in some time slot where agents contend for assigning balls into bins.

We view the SLA requirement of having a decline ratio of at most ε as a lower bound on the probability that an arbitrary agent attempting to assign a ball to some bin is happy. Formally, this requirement translates to ensuring that: $\frac{E[H_s]}{s} \geq 1 - \varepsilon$.

We also require that the total number of bins queried by our agents is no more than a prescribed budget (B), which translates to requiring that: $s \cdot d \leq B$.

Given n, k, ε and B, our goal is to find the largest number of agents s, and number of bin queries per agent d, that satisfy the above conditions.

We calculate the expected number of happy agents $E[H_s]$ in order to estimate the failure probability. Observe that $E[H_s]$ can be expressed by conditioning the number of happy agents H_s on the number of potentially happy agents F_s. I.e.,

$$E[H_s] = \sum_{f=1}^{s} \Big[\Pr(F_s = f) \cdot E[H_s | F_s = f] \Big]. \tag{1}$$

We now turn to evaluate the probability distribution of F_s, and then calculate the conditional expectation $E[H_s|F_s = f]$.

To evaluate the distribution of the number of potentially happy agents F_s, observe that an agent fails to find an available bin with probability $\left(\frac{n-k}{n}\right)^d$. Therefore the probability that an agent is potentially happy is:

$$\sigma = 1 - \left(\frac{n-k}{n}\right)^d. \tag{2}$$

One can interpret F_s as the result of s independent Bernoulli trials with success probability σ. Therefore:

$$Pr(F_s = f) = Bin(f, s, \sigma), \tag{3}$$

where

$$Bin(f, s, \sigma) \equiv \binom{s}{f} \sigma^f (1-\sigma)^{s-f}. \tag{4}$$

For calculating $E[H_s|F_s = f]$, we examine the process of the potentially happy agents placing their balls from the point of view of the k free bins. For ease of presentation, we associate each potentially happy agent with a sequence number $1, \ldots, f$, and each available bin with a sequence number $1, \ldots, k$.

The following proposition shows that the probability that an arbitrary potentially happy agent selects an arbitrary available bin is uniform over all available bins.

Proposition 1. *If agent i is potentially happy, then it places its ball on available bin j with a probability of $\frac{1}{k}$.*

Proof sketch. The proof follows from considering the conditional probability of an agent finding a specific available bin j, conditioned on having found x available bins in its set of sampled bins. This probability is $\frac{x}{k}$ due to the uniformity of the agent's choice. Selecting any specific available bin j has the same probability, which implies that the probability for finding a specific bin is $\frac{1}{k}$, as required. □

By Proposition 1, the probability that agent i does *not* place its ball in bin j is $1 - \frac{1}{k} = \frac{k-1}{k}$. As the agents are mutually independent, the probability that none of the f potentially happy agents places its ball in bin j is $\left(\frac{k-1}{k}\right)^f$. The probability that at least one of the f potentially happy agents tries to place its ball in bin j is $1 - \left(\frac{k-1}{k}\right)^f$. From the point of view of bin j, this process is equivalent to a Bernoulli trial, which succeeds iff at least one agent places its ball in bin j. If this succeeds, bin j is exclusively associated with a single happy agent.

Algorithm 2 MaximizeParallelism (n, ε, B, k)

1: $s \leftarrow 1$ ▷ initialization
2: **while** SatisfySLA $\left(n, \varepsilon, k, s+1, \left\lfloor \frac{B}{s+1} \right\rfloor\right)$ **do**
3: $\quad s \leftarrow s+1$
4: **end while**
5: return $s, \left\lfloor \frac{B}{s} \right\rfloor$

Applying the analysis above for each of the k free bins, we obtain that $E[H_s|F_s = f]$ is equivalent to the expected number of successes in k independent Bernoulli trials, with probability of success $1 - \left(\frac{k-1}{k}\right)^f$ each. Hence,

$$E[H_s|F_s = f] = k\left[1 - \left(\frac{k-1}{k}\right)^f\right]. \quad (5)$$

Combining (1) with (3) and (5), we obtain

$$E[H_s] = k\sum_{f=1}^{s}\left[1 - \left(\frac{k-1}{k}\right)^f\right] \cdot Bin(f, s, \sigma). \quad (6)$$

The following corollary is a direct consequence of (6)

Corollary 2. *If* $k\sum_{f=1}^{s}\left[1 - \left(\frac{k-1}{k}\right)^f\right] \cdot Bin(f, s, \sigma) \geq s(1-\varepsilon)$ *then the expected decline ratio with* s *agents, where each agent queries* d *bins, is at most* ε*.*

Based on Corollary 2, we now describe the details of the MaximizeParallelism method, which maximizes the parallelism while satisfying the SLA and budget constraints. The method is detailed in Algorithm 2. After initially setting $s = 1$, the algorithm repeatedly increases the value of s, while maintaining feasibility by having SatisfySLA validate that the condition of Corollary 2 is satisfied for the given configuration.

B. SLA Guarantees with Availability Lower Bounds

We first show that if k is the precise number of available hosts for any request, then MaximizeParallelism indeed generates a valid configuration.

Theorem 3. *Assume* k *is the number of available hosts that may accommodate any request flavor. If MaximizeParallelism*$(n, \varepsilon, B, k) = (s, d)$ *and* $s > 1$ *then employing* s *schedulers, each querying* d *hosts, guarantees an expected decline ratio of at most* ε*.*

Proof sketch. Our proof will consider the following *compacting* process: (i) Consider all the hosts in $\mathbf{H}_{\vec{c}^*}$ as available for all flavors, (ii) consider the other hosts as unavailable for any request, and (iii) determine that once a scheduler allocates a request in a host it becomes unavailable. The compacted system is equivalent to our balls-and-bins model. Hence, by Corollary 2, the compacted system satisfies the SLA constraint. One can then show that the decline ratio in the compacted system serves as an upper bound on the decline ratio (ε) in the original system. The result follows. □

The proof of Theorem 3 implicitly suggests that all the requests are handled in a time slot belonging to the flavor with the minimum number of available hosts. Furthermore, it suggests that two requests can never be placed in parallel on the same host. Thus, we expect better decline ratios in practice.

The following corollary shows that for providing performance guarantees, it is sufficient to know only a *lower bound* on the number of hosts available for every request flavor.

Algorithm 3 EstimateK(k)

1: **for all** $\vec{c} \in \mathbf{C}$ **do** ▷ for each flavor
2: $\quad n_{\vec{c}}^{(tot)} \leftarrow \sum_{i=1}^{s} n_{\vec{c}}^{(i)}$,$k_{\vec{c}}^{(tot)} \leftarrow \sum_{i=1}^{s} k_{\vec{c}}^{(i)}$
3: **end for**
4: $\tilde{k} \leftarrow n \cdot \min_{\vec{c} \in \mathbf{C}}\left[\frac{k_{\vec{c}}^{(tot)}}{n_{\vec{c}}^{(tot)}}\right]$
5: return $\alpha \cdot \tilde{k} + (1-\alpha) \cdot k$

Corollary 4. *Theorem 3 holds whenever* k *is a lower bound on the number of available hosts for every request flavor.*

Proof sketch. The proof is based on showing that increasing the number of hosts available for every request flavor while keeping the number schedulers s and the sample size d unchanged does not increase the decline ratio. We introduce a k superscript to our various notations to indicate the values for a specific value of k. We then rearrange (1), capturing the expected number of happy agents, and show that it can be cast as $E\left[H_s^k\right] = \sum_{f=0}^{s-1}\left[\Pr(F_s^k > f) \cdot D(k, f)\right]$, where we let $D(k, f) = E\left[H_s^k | F_s^k = f+1\right] - E\left[H_s^k | F_s^k = f\right]$. It can be shown that both $\Pr(F_s^k > f)$ and $D(k, f)$ are monotone non decreasing in k, thus completing the proof. □

VII. Practical Implementation of APSR

We now discuss practical aspects of implementing APSR. The main caveat in implementing APSR is to estimate the number of available hosts for any request flavor (k).

A straightforward option is to compute k explicitly by running a centralized periodic task that gathers the state from all hosts. We note that the APSR controller may execute such a task (in line 4). When the task is performed every time step (i.e., by setting $T = 1$ in APSR), then the guarantees of Theorem 3 hold. However, this approach incurs the communication overhead of querying all the hosts.

Alternatively, we propose to estimate k by relying on the statistics which the schedulers gather during their regular operation. Algorithm 3 describes our proposed algorithm EstimateK(k) for estimating k.

Our algorithm assumes that each scheduler i maintains counters $n_{\vec{c}}^{(i)}$ and $k_{\vec{c}}^{(i)}$, which keep track of the overall number of hosts queried, and the total number of available hosts of flavor $\vec{c}$, respectively. These counters are reset before each call to algorithm EstimateK. The algorithm uses these counters to estimate the *overall* number of hosts queried and the overall number of available hosts for each flavor. These values can be used to estimate the *percentage* of hosts available for each request flavor. The normalized minimum of all flavors is chosen as the pessimistic estimate of k. We then use exponential averaging to produce an updated estimate of k.

We emphasize that our approach does not require any additional querying of hosts. We note that Algorithm 3 does not ensure that our estimate is a lower bound on the available

resources in the system, as is required by Corollary 4. However, due to the conservative approach in making the estimate (mainly, line 4 in Algorithm 3) our estimation method is effective when incorporated within our APSR Algorithm.

VIII. APSR Evaluation

This section positions APSR with respect to known placement algorithms and evaluates the interplay between parallelism, utilization, decline ratio, and throughput.

Trace-based Simulation: We model the arrival of requests using a Poisson process with parameter λ_a. Unless stated otherwise, we set λ_a to 20, and ε (APSR's target decline ratio) to 5%. We set APSR's query budget to be $B = n$. That is, the *overall* number of samples made by all of our parallel schedulers is the same as the number of samples done by *a single* OpenStack scheduler. We set APSR's time interval for estimating the state of the cloud to be $T = 10$ and set $\alpha = 0.1$ for the EstimateK method.

We consider requests of unbounded duration as it is a common (though somewhat unrealistic) benchmark for placement algorithms [4]. These settings provide a clear demonstration of the relationship between utilization and parallelism. Due to space constraints, we omit our simulation results for finite duration requests but note that these results have similar qualitative characteristics for such settings.

We use the workloads described in Section IV-B, and simulate large clouds with 30 replicas of the NFV dataset, seven replicas of the Amazon dataset, and one replica of the Google dataset, attaining a total of 13110, 7700, and 12477 requests, respectively. We determine the number of hosts as the number of hosts needed for successfully placing all the requests at once (by some offline algorithm), as described in Section IV-C; we use 837 hosts for NFV, 876 hosts for Amazon, and 5989 hosts for Google. As discussed in Section IV-C, for every algorithm, we make a single attempt to place each request and compute the decline ratio accordingly.

We study the interplay between parallelism and the decline ratio of APSR and other common placement algorithms. We let APSR adapt the number of schedulers according to its estimate of the system utilization and report the throughput of APSR, captured by the average number of active schedulers that handle requests. For the competing algorithms, we consider various values for the (fixed) number of schedulers.

Table VI summarizes the results. The algorithms DistFromDiag and Adaptive are abbreviated by Diag and Adapt, respectively. The average number of active schedulers used by APSR is indicated below its decline ratio. Note that APSR's decline ratio is always within the SLA requirement ($\varepsilon = 5\%$), and it uses between 14 and 20 active schedulers on average. Since the average number of arriving requests per cycle is $\lambda_a = 20$, it follows that it might be beneficial to occasionally have more than 20 schedulers to handle bursts of arrivals, but only 20 requests arrive per time unit on average. Random and APSR yield the lowest decline ratio, both within the SLA constraint but the communication overhead of APSR is much lower than that of Random: the total number of queries made by *all* the schedulers which APSR uses is the same as that of a *single* scheduler of Random. We note that this less accurate view of the system state causes APSR's decline ratio sometimes to be slightly higher than that of Random (although always within the SLA).

TABLE VI
Decline ratios (in %, lower is better) of APSR and other placement algorithms when varying the (fixed) number of schedulers (s, higher is better). APSR's throughput, captured by the average number of active schedulers ($\bar{s}$), is listed below its decline ratio.

Dataset	s	APSR	Rand	FF	FFR	WF	WFR	Diag	Adapt
NFV	1		0.3	0.0	0.0	0.3	0.3	0.7	0.3
	5	0.4	0.4	11.1	2.5	4.0	1.0	5.3	2.2
	10	$\bar{s}$ = 14	0.5	23.3	5.2	8.2	2.1	7.8	3.1
	20		0.7	35.7	10.0	12.1	3.3	11.7	11.6
	50		0.8	39.0	10.8	16.7	3.9	16.4	16.0
Google	1		2.3	0.4	1.3	8.7	8.7	2.2	8.7
	5	3.1	2.4	56.2	15.5	42.0	16.4	42.7	42.0
	10	$\bar{s}$ = 20	2.4	77.8	29.9	64.1	26.1	62.7	64.5
	20		2.4	87.8	48.1	79.8	36.4	73.8	79.3
	50		2.4	88.9	51.4	81.2	40.2	76.7	81.2
Amazon	1		0.5	0.0	0.0	0.4	0.4	1.3	0.2
	5	0.8	0.6	18.2	4.2	6.5	1.5	7.2	6.3
	10	$\bar{s}$ = 19	1.0	33.6	9.6	20.7	3.4	15.8	20.3
	20		1.2	49.1	16.0	61.4	6.2	31.9	60.5
	50		1.4	52.8	17.7	64.9	7.7	37.8	65.0

TABLE VII
Total number of queries, throughput and actual decline ratios of APSR versus Random.

	APSR			Random		
	Target Decline Ratio (ε)			Number of Schedulers		
	3%	5%	10%	1	10	20
	NFV					
Number of Queries	1553K	811K	578K	11000K		
Throughput [req./slot]	7.2	14	19.6	1	10	19.8
Decline Ratio (δ)	0.4%	0.4%	0.6%	0.3%	0.5%	0.8%
	Google					
Number of Queries	3920K	3860K	3823K	74724K		
Throughput [req./slot]	19.8	19.9	19.9	1	10	19.9
Decline Ratio (δ)	3.0%	3.1%	2.9%	2.3%	2.4%	2.4%
	Amazon					
Number of Queries	469K	370K	354K	6745K		
Throughput [req./slot]	15.3	19.3	19.9	1	10	19.9
Decline Ratio (δ)	0.7%	0.8%	1.0%	0.5%	1.0%	1.4%

Table VII compares the throughput, the decline ratios, and the total number of queries of APSR and Random. Note that APSR reduces the total number of queries by at least 85%. Increasing APSR's target decline ratio increases its parallelism, which in turn increases the throughput. This tradeoff highlights the tension between the decline ratio and the degree of parallelism. The best achievable throughput is 20, as it is the average arrival rate. Indeed, APSR and Random with fixed 20 schedulers are very close to the maximal throughput. Also, recall that, unlike Random, APSR may fail due to not finding an available host in the queried hosts; thus, its decline ratio is sometimes higher.

OpenStack Evaluation: We now evaluate APSR in an OpenStack environment (Mitaka release) [23] on an HP ProLiant BL460c Gen9 server with two Intel(R) Xeon(R) E5-

TABLE VIII
OPENSTACK: NUMBER OF QUERIES, AVERAGE NUMBER OF SCHEDULERS, THROUGHPUT, AND ACTUAL DECLINE RATIOS OF APSR AND FILTER SCHEDULER FOR THE NFV DATASET.

	APSR			Filter Scheduler		
	Target decline ratio (ε)			Number of schedulers		
	2%	3%	5%	1	8	16
Number of queries	110K	102K	108K	2240K		
Avg # of schedulers allowed	13.5	16	16	1	8	16
Throughput [req./sec.]	2.6	2.8	2.6	1	2.6	2.7
Actual decline ratio (δ)	1.0%	0.7%	0.7%	0%	3.8%	13.6%

2680v4 processors with 28 cores (56 cores total) running at 2.4 GHz, and a total RAM of 256GB. We run a functional scheduler implementation and use OpenStack's Benchmarking to emulate the remote hosts [21]. We periodically send 200 request batches from the NFV dataset and wait for the scheduler to place all of them. In total, we send 2800 requests to place on 400 hosts, attaining a resource utilization of ≈90% at a steady state. We set APSR's parameters to $T = 10sec$, $B = 100$ and $\varepsilon \in \{2\%, 3\%, 5\%\}$.

Table VIII compares the throughput, decline ratio, and the total number of queries of APSR and the default Filter scheduler. The table shows that APSR's decline ratio is always within the target bound. Furthermore, APSR attains a similar throughput to running 8 Filter schedulers in parallel while keeping a much lower decline ratio than that presented by 8 Filter schedulers. Finally, APSR reduces the number of host queries by $\approx 90\%$.

IX. DISCUSSION, CONCLUSIONS AND FUTURE WORK

Our work seeks high-throughput placement of virtual machines to better cope with long service chains. Parallelism improves throughput, but many placement algorithms behave poorly in parallel settings. Our APSR algorithm implements random placement while minimizing the communication overhead and dynamically adjusting the degree of parallelism to ensure that decline ratios satisfy their SLA requirements. We formally prove the correctness of APSR and provide insights into the possibilities and limitations of parallel resource management.

We evaluate APSR on three real workloads and demonstrate its capability to provide high degrees of parallelism with small decline ratios and low communication overheads. We then integrate APSR into the OpenStack cloud management platform. We show that APSR matches the best throughput of OpenStack's default Filter Scheduler while reducing the decline ratio from up to 13.6% to $\approx 1\%$, and the communication overheads by $\approx$ 20x. That is, APSR also implies less clutter and drain on the system.

Looking into the future, we observe that OpenStack only gains up to 3x speedup from parallelism, whereas APSR easily supports many parallel schedulers. Thus, we plan to carefully benchmark OpenStack, identify its current bottlenecks, and unleash its full potential for parallel resource management.

REFERENCES

[1] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. E. Rasmussen. Parallel randomized load balancing. *Rand. Struct. Alg.*, 13(2):159–188, 1998.

[2] Y. Cheng. Dive into nova scheduler performance. In *OpenStack Summit*, 2016. https://www.openstack.org/assets/presentation-media/7129-Dive-into-nova-scheduler-performance-summit.pdf.

[3] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *ACM SoCC*, pages 97–110, 2015.

[4] G. Einziger, M. Goldstein, and Y. Sa'ar. Faster placement of virtual machines through adaptive caching. In *INFOCOM*, pages 2458–2466, 2019.

[5] A. C. Enterprises. Inc. maui scheduler administrator's guide, version 3.2, january 2014. http://docs.adaptivecomputing.com/maui/.

[6] L. Epstein and L. M. Favrholdt. On-line maximizing the number of items packed in variable-sized bins. *Acta Cybern.*, 16(1):57–66, 2003.

[7] OpenStack compute schedulers, 2018. https://docs.openstack.org/newton/config-reference/compute/schedulers.html.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[9] J. G. Herrera and J. F. Botero. Resource allocation in NFV: A comprehensive survey. *IEEE Trans. Net. Serv. Manag.*, 13(3):518–532, 2016.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[11] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *ACM SIGOPS*, pages 261–276, 2009.

[12] Y. Kanizo, O. Rottenstreich, I. Segall, and J. Yallouz. Optimizing virtual backup allocation for middleboxes. *IEEE/ACM Trans. Netw.*, 25(5):2759–2772, 2017.

[13] Kubernetes. Scheduling performance issues, 2016. https://github.com/kubernetes/kubernetes/issues/32361 https://github.com/kubernetes/kubernetes/issues/18266.

[14] Z. Liu and S. Cho. Characterizing machines and workloads on a google cluster. In *ICPPW*, pages 397–403, Sept 2012.

[15] M. C. Luizelli, D. Raz, and Y. Sa'ar. Optimizing NFV chain deployment through minimizing the cost of virtual switching. In *INFOCOM*, pages 2150–2158, 2018.

[16] M. C. Luizelli, D. Raz, Y. Sa'ar, and J. Yallouz. The actual cost of software switching for NFV chaining. In *IM*, pages 335–343, 2017.

[17] J. Martín-Pérez, F. Malandrino, C.-F. Chiasserini, and C. J. Bernardos. Okpi: All-kpi network slicing through efficient resource allocation. In *INFOCOM*, pages 804–813, 2020.

[18] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *USENIX NSDI*, pages 459–473, 2014.

[19] K. Mills, J. Filliben, and C. Dabrowski. Comparing VM-placement algorithms for on-demand clouds. In *CloudCom*, pages 91–98, 2011.

[20] M. Mitzenmacher. The power of two choices in randomized load balancing. *Trans. on Paral. Dist. Sys.*, 12(10):1094–1104, 2001.

[21] OpenStack benchmarking for scheduling. https://github.com/cyx1231st/nova-scheduler-bench, 2016.

[22] OpenStack configuration options, 2019. https://docs.openstack.org/nova/queens/configuration/config.html.

[23] OpenStack. https://www.openstack.org/.

[24] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *SOSP*, pages 69–84, 2013.

[25] D. Raz, I. Segall, and M. Goldstein. Multidimensional resource allocation in practice. In *ACM SYSTOR*, pages 1:1–1:10, 2017.

[26] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format+schema. Technical report, Google Inc., 2011.

[27] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS EuroSys*, pages 351–364, 2013.

[28] T. Shabeera, S. M. Kumar, S. M. Salam, and K. M. Krishnan. Optimizing vm allocation and data placement for data-intensive applications in cloud using aco metaheuristic algorithm. *Int. J. Eng. Sci. Tech.*, 20(2):616 – 628, 2017.

[29] T. Taleb, M. Corici, C. Parada, A. Jamakovic, S. Ruffino, G. Karagiannis, and T. Magedanz. EASE: EPC as a service to ease mobile core network deployment over cloud. *IEEE Network*, 29(2):78–88, 2015.