Parallel Virtual Machines Placement 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
Independent
Beer Sheva, Israel
maayan.goldstein@gmail.com

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

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

Erez Waisbard
The Open University
Ra’anana, Israel
erezwa@openu.ac.il

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 [15], [24]. 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 [22], selecting a host on which to place the VM may require hundreds of milliseconds in large clouds [3], [6], [16]. 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 [2], [18]–[20], [30], [33], [35], [36] 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 [6], [16]. 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 [3].

Intuitively, one could boost throughput by running multiple schedulers in parallel. However, such an approach may translate to multiple schedulers trying to place requests simultaneously on the same host, leading to race scenarios [26], [28], [32]. In such cases, not all the requests will be successful, and the host may decline some of the requests. This 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 must respond to a flash crowd or a cyber attack [6]. Hence, a provider is typically required to bind the decline ratio, namely, 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) [6], [12], or in the Key Performance Indicators (KPIs) [21].

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
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.

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, place-
ment 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 [7], and the
single-scheduler algorithms proposed in [29]) 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 [3, 6, 16] and the latency
of computing the placement decision itself, which may take
several seconds [32]. 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., scaling out a service chain’s capacity 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 [6]. However, when the cached state becomes
state, 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 [9].
However, our work shows that running multiple independent
schedulers translates to collisions when multiple schedulers
select the same hosts simultaneously. 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 [26, 28]. Further,
the seminal work of [32] 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, rendering 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 [26] (denoted Λ), which works as follows: After ranking the available hosts,
the scheduler randomly assigns the request to one of the top-Λ ranking hosts. In the absence of a rigorous theory studying the
effect of Λ on the system’s performance, its value is commonly
determined using crude estimations and rules of thumb. Our
work helps to configure the parameter Λ properly. Furthermore,
in Section IV we show that the common approach of setting
Λ as a small constant results in poor performance.

The Omega scheduler [32] 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 the 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 [14] uses a
static partition, which occasionally results in non-compulsory
debases due to fragmentation of resources [32]. Namely, a
scheduler may fail to place a request in its partition, even
if hosts in other partitions can accommodate the request.
Mesos [13] suggests using dynamic partitioning, where a central
controller dynamically allocates hosts to schedulers on demand
in order 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 ap-
proach was extensively studied in the context of balanced
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 EXISTING ARCHITECTURES THAT IMPLEMENT THE APPROACH.

| Approach     | #Schedulers | Description   | Throughput | Decline Ratio | Overhead | Examples          |
|--------------|-------------|---------------|------------|---------------|----------|-------------------|
| Global       | Single      | Monolithic    | Low        | Low, guaranteed | High     | Maui [7]          |
| Snapshot     |             |               |            |               |          |                   |
| Fixed        |             | Multiple Snapshots | Low       | Low          | Low      | ASC [2]           |
|               |             | Shared Snapshot | High      | High         | High     | Omega [32]        |
| Partitioning | Fixed       | Static Partition | Mid       | Mid          | Low      | Quincy [14]       |
| Sampling     |             | Dynamic Partition | Mid       | Mid          | Low-Mid  | Mesos [13]        |

These problems essentially assume an (infinite) buffer for pending requests in 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 algorithmic paradigm [1], [24] 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 [28] and Tarcil [5] 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 balance the load evenly [1], [5], [24], [28]. 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], [5], [24], [28], APSR provides guarantees on the decline ratio.

III. SYSTEM MODEL FOR PARALLEL SCHEDULING

We consider a collection $H$ 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 $h \in H$ 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 $R$ of requests, each modeled as a vector of demand for each resource. We assume each request $\vec{r} \in R$ has its vector drawn from some finite set of possible request vectors, or flavors, $C = \{\vec{c}_1, \ldots, \vec{c}_m\}$. A host $h$ 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 $\vec{h} \in H$ resolves concurrent requests 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 $\vec{c} \in C$, we let $k_{\vec{c}}^{(t)}$ denote the number of hosts in $H$ 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 $\delta$ to denote the system’s expected decline ratio (for some set of requests $R$). Since we are handling requests independently, $\delta$ 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 $\varepsilon$, for some $\varepsilon \in [0, 1]$.

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

2Current algorithms are oblivious to such constraints, and might violate this requirement. Our APSR algorithm takes such constraints into account, and produces solutions that provably satisfy them.
valid configuration of schedulers determines \( s \) and \( d \), such that \( s \cdot d \leq B \), and the probability of a failed assignment attempt is at most \( \varepsilon \). 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., [24].

OpenStack’s default placement algorithm is the WorstFit (WF) algorithm [9]. 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) [8] 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 [30] 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 [30] 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 \( \Lambda \) 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 \( \Lambda = 5 \).

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 with sufficiently many hosts to accommodate all the requests (see Section VI for details on choosing the number of hosts).

NFV Dataset was collected from a proprietary large NFV management and orchestration (MANO) system [6]. 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 \( \langle \text{memory, storage} \rangle \). 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.

Google Dataset, recorded in a Google’s cluster [51], holds data from 12,477 virtual machines characterized by tuples of \( \langle \text{CPU, memory} \rangle \). The normalized CPU values vary between 0.25, 0.5, and 1, whereas the memory values can be grouped around five levels: 0.125, 0.25, 0.5, 0.75, and 1 [17]. The hosts capacities are either \( \langle 1, 2 \rangle \) or \( \langle 2, 1 \rangle \) in equal proportions [30]. Table IV provides the breakdown of flavors for this dataset.

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

NFV Dataset was collected from a proprietary large NFV management and orchestration (MANO) system [6]. 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 \( \langle \text{memory, storage} \rangle \). 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.

Google Dataset, recorded in a Google’s cluster [51], holds data from 12,477 virtual machines characterized by tuples of \( \langle \text{CPU, memory} \rangle \). The normalized CPU values vary between 0.25, 0.5, and 1, whereas the memory values can be grouped around five levels: 0.125, 0.25, 0.5, 0.75, and 1 [17]. The hosts capacities are either \( \langle 1, 2 \rangle \) or \( \langle 2, 1 \rangle \) in equal proportions [30]. Table IV provides the breakdown of flavors for this dataset.

| Symbol | Meaning |
|--------|---------|
| \( H \) | Set of hosts |
| \( n \) | Number of hosts (bins) |
| \( h \) | Host in \( H \) (resources availability vector) |
| \( R \) | Set of requests |
| \( r' \) | Request in \( R \) (resources demand vector) |
| \( C \) | Set of requests flavors |
| \( c' \) | Flavor in \( C \) of a request |
| \( s \) | Number of schedulers (agents) |
| \( \delta \) | Actual decline ratio |
| \( \varepsilon \) | Maximum allowed decline ratio by the SLA |
| \( B \) | Budget for overall number of queries |
| \( d \) | Number of hosts queried by each scheduler |
| \( n_{c'} \) | Number of hosts queried for requests of flavor \( c' \) |
| \( k_{c'} \) | Number of available hosts for flavor \( c' \) |
| \( k_e \) | Number of available hosts for any request |
| \( F_r \) | Number of potentially happy agents |
| \( H_s \) | Number of happy agents |
| \( \sigma \) | See [3] |
| \( Bin(a, b, c) \) | See [5] |
| \( \lambda_a \) | Poisson arrival rate |
| \( \lambda_d \) | Poisson departure rate |

| Symbol | Meaning |
|--------|---------|
| \( H \) | Set of hosts |
| \( n \) | Number of hosts (bins) |
| \( h \) | Host in \( H \) (resources availability vector) |
| \( R \) | Set of requests |
| \( r' \) | Request in \( R \) (resources demand vector) |
| \( C \) | Set of requests flavors |
| \( c' \) | Flavor in \( C \) of a request |
| \( s \) | Number of schedulers (agents) |
| \( \delta \) | Actual decline ratio |
| \( \varepsilon \) | Maximum allowed decline ratio by the SLA |
| \( B \) | Budget for overall number of queries |
| \( d \) | Number of hosts queried by each scheduler |
| \( n_{c'} \) | Number of hosts queried for requests of flavor \( c' \) |
| \( k_{c'} \) | Number of available hosts for flavor \( c' \) |
| \( k_e \) | Number of available hosts for any request |
| \( F_r \) | Number of potentially happy agents |
| \( H_s \) | Number of happy agents |
| \( \sigma \) | See [3] |
| \( Bin(a, b, c) \) | See [5] |
| \( \lambda_a \) | Poisson arrival rate |
| \( \lambda_d \) | Poisson departure rate |

| Symbol | Meaning |
|--------|---------|
| \( H \) | Set of hosts |
| \( n \) | Number of hosts (bins) |
| \( h \) | Host in \( H \) (resources availability vector) |
| \( R \) | Set of requests |
| \( r' \) | Request in \( R \) (resources demand vector) |
| \( C \) | Set of requests flavors |
| \( c' \) | Flavor in \( C \) of a request |
| \( s \) | Number of schedulers (agents) |
| \( \delta \) | Actual decline ratio |
| \( \varepsilon \) | Maximum allowed decline ratio by the SLA |
| \( B \) | Budget for overall number of queries |
| \( d \) | Number of hosts queried by each scheduler |
| \( n_{c'} \) | Number of hosts queried for requests of flavor \( c' \) |
| \( k_{c'} \) | Number of available hosts for flavor \( c' \) |
| \( k_e \) | Number of available hosts for any request |
| \( F_r \) | Number of potentially happy agents |
| \( H_s \) | Number of happy agents |
| \( \sigma \) | See [3] |
| \( Bin(a, b, c) \) | See [5] |
| \( \lambda_a \) | Poisson arrival rate |
| \( \lambda_d \) | Poisson departure rate |
Amazon Dataset is based on data from Amazon EC2 hosts and VM flavors [23], [30]. Table V depicts the flavors of the normalized \((CPU, memory)\) in this dataset, where each 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).

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. Our experiments 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 complete. 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 slight randomization into traditional algorithms (as captured by FFR and WFR). However, statically setting \(\Lambda = 5\) 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 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

### Table V

|               | Small | Large |
|---------------|-------|-------|
| CPU           |       |       |
| memory        |       |       |
| 0.035         | 0.008 | 0.083 |
| 0.07          | 0.016 | 0.083 |
| 0.083         | 0.008 | 0.031 |
| 0.1           | 0.008 | 0.031 |
| 0.142         | 0.063 | 0.016 |
| 0.167         | 0.125 | 0.062 |
| 0.2           | 0.333 | 0.354 |
| 0.008         | 0.031 | 0.012 |
| 0.016         | 0.125 | 0.056 |
| 0.031         | 0.063 | 0.016 |
| 0.083         | 0.25  | 0.25  |

Fig. 1. Decline ratios for different placement algorithms and a 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 [11]. Thus, we approximate this number as suggested in [30]. 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.
which may cause multiple agents to select the same available
balls-and-bins
Algorithm 1
APSR Controller (n, ε, B, T)

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

scheduler queries per request. The controller determines the
above parameters to ensure the validity of the configuration.

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 estimates 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 various 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:

$$\frac{E[H_s]}{s} \geq 1 - \varepsilon. \quad (1)$$

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 · d ≤ B.

Given n, k, ε, and B, our goal is to find the largest number
of agents s, and the 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.,

$$E[H_s] = \sum_{f=1}^{n-k} \Pr(F_s = f) \cdot E[H_s|F_s = f]. \quad (2)$$

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:

$$\sigma = 1 - \left(\frac{n-k}{n}\right)^d. \quad (3)$$

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

$$\Pr(F_s = f) = Bin(f, s, \sigma), \quad (4)$$

where

$$Bin(f, s, \sigma) \equiv \binom{s}{f} \sigma^f (1-\sigma)^{s-f}. \quad (5)$$

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, . . . , f,
and each available bin with a sequence number 1, . . . , 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. Let i be an agent – not necessarily a potentially happy
agent. Denote by Q_i the set of bins which agent i queries
and finds available. Let q_i denote the random variable for
the number of bins which agent i finds available, namely, |Q_i| = q_i.
Then Pr(q_i = x) captures the probability that agent i finds x
distinct available bins in his overall d samples.

For each available bin ℓ, we let B_ℓ denote a binary random
variable, indicating whether agent i samples bin ℓ. Namely,
which succeeds iff at least one agent places its ball in bin with (8), the probability that agent \( i \) of success of successes in

If this succeeds, bin \( j \) is available. By (6) it follows that for every \( \ell = 1, \ldots, k \),

By Proposition 1, the probability that potentially-happy agent \( i \) selects available bin \( j \) is \( \frac{1}{k} \sum_{x=1}^{k} x \cdot \Pr(q_i = x) \).

If agent \( i \) samples available bin \( j \), then she selects \( j \) w.p. \( \frac{1}{k} \).

It follows that

Observe that agent \( i \) is potentially happy iff she samples at least one available bin, that is, if \( q_i > 0 \). The probability for this event is \( \sum_{x=1}^{k} \Pr(q_i = x) \). Combining this observation with (5), the probability that agent \( i \) samples available bin \( j \)
given that \( i \) is potentially happy is \( \frac{1}{k} \).

By Proposition 1 the probability that potentially-happy agent \( i \) does not place its ball in bin \( j \) is \( 1 - \frac{k-1}{k} = \frac{k-1}{k} \). 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.

Applying the analysis above for each of the \( k \) free bins, we obtain that \( \mathbb{E}[H_s|F_s = f] \) 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,

Combining (2) with (4) and (9), we obtain

The following corollary is a direct consequence of (1) and (10).

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

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.

\textbf{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.

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

\textbf{Proof.} Let \( H_{\varepsilon} \) denote the set of hosts with enough resources for accommodating a request of flavor \( \varepsilon \). Using our notation, it follows that \( |H_{\varepsilon}| = k_{\varepsilon} \). Let \( \varepsilon^* = \arg \min_{\varepsilon} \{k_{\varepsilon}\} \).

Consider the following compacting process:

1) Consider all the hosts in \( H_{\varepsilon^*} \) as available for all flavors.
2) Consider the other hosts as unavailable for any request.
3) Determine that once a scheduler allocates a request in a host, it becomes unavailable.

We claim that compacting the system can only increase its decline ratio for the following reasons: First, as for each \( \varepsilon \in \mathbb{C} \) we have \( k_{\varepsilon^*} \leq k_{\varepsilon} \), steps 1 and 2 can only decrease the number of hosts available for each flavor. This reduces the expected number of available hosts found by each scheduler. Second, steps 1 and 2 define the available hosts of any flavor to be exactly \( H_{\varepsilon^*} \). This compacting may only increase the probability that multiple schedulers will end up assigning their requests to the same host. Finally, a host may accommodate multiple parallel requests providing it has enough resources while step 3 disallows it, which implies a potential increase in the decline ratio. Thus, any algorithm satisfying the SLA in the compacted system also satisfies it in the original system.

We now note that the compacted system is equivalent to our balls-and-bins model. To see this, observe that once the sets of available hosts for every request become identical (due to steps 1 and 2), the requests themselves are also virtually...
identical and thus become equivalent to the identical balls in our balls-and-bins model. Furthermore, as every host can accommodate only a single request (due to step 3), the hosts can be modeled as identical bins, where each available bin can accommodate merely a single ball.

By Corollary 2, MaximizeParallelism satisfies the SLA requirement in the balls-and-bins model, which is equivalent to guaranteeing SLA also in the compacted system. As the decline ratio in the compacted system serves as an upper bound on the decline ratio ($\varepsilon$), 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.

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

Proof. We have to show that increasing the number of hosts available for every request flavor, while keeping the number schedulers $s$ and the sample size $d$ unchanged can only decrease the decline ratio. We do so by checking the effect of increasing the number of available bins $k$ on our balls-and-bins analysis. As we now vary $k$, we add to the notation of our random variables a superscript indicating its value. That is, $F^k_s$ and $H^k_s$ denote the random variable for the number of potentially happy and happy agents, respectively, when there are $k$ available bins. Recalling the SLA requirement in (1), it suffices to show that $E[H^k_{s+1}] \geq E[H^k_s]$.

Using our modified notation, we rewrite (2) as

$$E[H^k_{s}] = \sum_{f=1}^{s} \Pr(F^k_s = f) \cdot E[H^k_s | F^k_s = f]$$  \hspace{1cm} (11)

We now handle each of the components in the product appearing on the right-hand side of (11) separately, namely (i) the probability distribution of the number potentially happy agents and (ii) the expected number of happy agents, given that there are $f$ potentially happy agents.

Intuitively, the probability of having more than $f$ potentially happy agents is non-decreasing in the number of free bins $k$. Indeed, combining (3), (4) and (5) shows that

$$\Pr(F^{k+1}_s > f) \geq \Pr(F^k_s > f).$$  \hspace{1cm} (12)

To quantify the impact of the number of potentially happy agents $f$ on the expected number of happy agents we let $D(k,f)$ denote the difference function

$$D(k,f) = E[H^k_s | F^k_s = f + 1] - E[H^k_s | F^k_s = f].$$  \hspace{1cm} (13)

$D(k,f)$ captures the contribution of adding one potentially happy agent to the expected number of happy agents. As $E[H^k_s | F^k_s = 0] = 0$, we have $D(k,0) = E[H^k_s | F^k_s = 1]$. We can therefore rewrite (11) as follows:

$$E[H^k_s] = \sum_{f=1}^{s} \Pr(F^k_s = f) \cdot E[H^k_s | F^k_s = f] = \Pr(F^k_s = 1) \cdot D(k,0) + \Pr(F^k_s = 2) \cdot (D(k,0) + D(k,1)) + \cdots + \Pr(F^k_s = s) \cdot [D(k,0) + D(k,1) + \cdots + D(k,s-1)] = \sum_{f=0}^{s-1} \Pr(F^k_s > f) \cdot D(k,f).$$  \hspace{1cm} (14)

By combining (12) and (14), it suffices to show that $D(k+1,f) > D(k,f)$ for every $f$.

For proving that (15) is satisfied, we assign $\alpha$ to the definition of $D(s)$ in (13), and obtain:

$$D(k,f) = k \left(\left(\frac{k-1}{k}\right)^f - \left(\frac{k-1}{k}\right)^{f+1}\right) = \left(\frac{k-1}{k}\right)^f - \left(\frac{k-1}{k}\right)^{f+1}$$  \hspace{1cm} (16)

Hence,

$$D(k+1,f) - D(k,f) = \left(\frac{k}{k+1}\right)^f - \left(\frac{k-1}{k}\right)^f \geq 0$$  \hspace{1cm} (17)

where the last inequality is satisfied for every $k > 0$.

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 estimating $k$ by relying on the statistics 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 $n_{c}^{(i)}$ and $k_{c}^{(i)}$, which keep track of the overall number of hosts queried, and the total number of available hosts of flavor $c$, 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 (namely, 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.

A. Simulation Settings

We model the arrival of requests using a Poisson process with parameter $\lambda_a$. Unless stated otherwise, we set $\lambda_a$ to 20, and $\varepsilon$ (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. 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.

B. Comparing APSR to other algorithms

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 Dist-FromDiag 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 $\lambda_n = 20$, 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 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.

C. Under the hood of APSR

Our next set of experiments studies the interplay between the system’s utilization and the level of parallelism offered by APSR. For these experiments, we use solely the NFV dataset.

Fig. 2a depicts the number of schedulers and the system utilization of APSR. Initially, APSR allows many schedulers as there are many available hosts for any flavor. As the utilization increases and the number of available hosts decreases, APSR gradually reduces the number of schedulers. Intuitively, reducing the number of schedulers serves two goals: First, it allows each scheduler to query more hosts while still complying with the budget constraint. This increases the probability of finding an available host. Second, having fewer schedulers reduces the collision probability.

Recall that APSR uses a conservative approach in estimating the number of available hosts ($k$). This conservative approach indeed yields a very low decline ratio (0.4%) – but at the cost of throttling parallelism when utilization ramps up. We, therefore, consider a variant of APSR, which we dub APSR$_{avg}$. As its name suggests, this variant differs from Algorithm 3 in Line 4, where it updates $k$ according to the average number of available hosts taken over all flavors.

Fig. 2b shows that APSR$_{avg}$ allows a significantly higher number of schedulers than APSR, for any given level of utilization. As a result, APSR$_{avg}$ finishes handling all requests much faster than APSR, implying a higher throughput. Indeed, switching from APSR to APSR$_{avg}$ doubles the actual decline
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 |
|---------|-----|------|------|-----|-----|-----|-----|------|-------|
|         | 1   | 0.3  | 0.0  | 0.0 | 0.3 | 0.3 | 0.7 | 0.3  |       |
|         | 5   | 0.4  | 11.1 | 2.5 | 4.0 | 1.0 | 5.3 | 2.2  |       |
|         | 10  | 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  | 56.2 | 15.5| 42.0| 16.4| 42.7| 42.0 |       |
|         | 10  | 3.4  | 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   | 0.6  | 18.2 | 4.2 | 6.5 | 1.5 | 7.2 | 6.3  |       |
|         | 10  | 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 (\( \varepsilon \)) | Number of Schedulers |
|------|--------|-----------------------------------------|---------------------|
|      |        | 3% | 5% | 10% | 1 | 10 | 20 |
| NFV  |        | 1553K | 811K | 578K | 11000K |
|      |        | 7.2 | 14 | 19.6 | 1 | 10 | 19.8 |
|      |        | 0.4% | 0.4% | 0.6% | 0.3% | 0.5% | 0.8% |
| Google |    | 3920K | 3860K | 3823K | 74724K |
|      |        | 19.8 | 19.9 | 19.9 | 1 | 10 | 19.9 |
|      |        | 3.0% | 3.1% | 2.9% | 2.3% | 2.4% | 2.4% |
| Amazon |   | 469K | 370K | 354K | 6745K |
|      |        | 15.3 | 19.3 | 19.9 | 1 | 10 | 19.9 |
|      |        | 0.7% | 0.8% | 1.0% | 0.5% | 1.0% | 1.4% |

ratio to 0.8% – but this value is still well below the target decline ratio of \( \varepsilon = 5\% \).

Our next experiment explores how both APSR and APSR\(_{avg}\) dynamically adjust the number of schedulers when the utilization fluctuates. To generate fluctuations in the utilization, we modeled the request arrival process as a variant of a Markov Modulated Poisson Process (MMPP) [10]. Specifically, the number of requests arriving per slot is a Poisson process with mean \( \lambda_a \) throughout the experiment. However, for the first 20\% of the requests we use \( \lambda_a = 20 \), to fill up the system; while for the rest of the requests we fix \( \lambda_a = 5 \). Furthermore, in this experiment, requests have a finite lifetime. That is, the number of allocated requests leaving per time slot follows a Poisson process with mean \( \lambda_d = 4 \). Finally, we use here 100 replicas of the NFV dataset (instead of 30 used in the rest of this section) so that even when requests leave their hosts, the hosts become utilized again with more arriving requests. These settings are intended to let utilization first build-up, and then stay at some (high) level, with mild fluctuations. The results of this experiment are shown in Fig. 3. Both APSR (Fig. 3a) and APSR\(_{avg}\) (Fig. 3b) dynamically adapt the number of allowed schedulers to the utilization. However, APSR\(_{avg}\) allows more schedulers than APSR. It obtains shorter total run-time but experiences a higher decline ratio (1.6\% for APSR\(_{avg}\) versus 0.01\% for APSR). Note that both algorithms are below the maximum allowed decline ratio (5\%).

We now investigate the effect of the query budget \( B \) on the number of schedulers. We use the same settings of long-lived VMs as in the experiment used for the results presented in Table VI. In this experiment, we vary the budget \( B \) on the overall number of accesses made by all the schedulers from 20\% to 100\% of the number of hosts. We report the level of parallelism, captured by the average number of active schedulers which APSR employs along the run.
D. OpenStack Evaluation

We now evaluate APSR in an OpenStack environment (Mitaka release) [27] on an HP ProLiant BL460c Gen9 server with two Intel(R) Xeon(R) E5-2680v4 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 [25]. We periodically send 200 request batches from the NFV dataset and wait for the scheduler to place all of them. We set APSR’s parameters to \( T = 10\ sec, B = 100 \) and \( \epsilon \in \{2\%, 3\%, 5\%\}\).

Table IX 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 \( \approx 90\% \).

Our experiments suggest that in current OS implementation, solving the bottleneck of parallelism addressed in our work, raises new challenges and bottlenecks. Therefore, to fully exploit the benefits of our work, a further investigation of the bottlenecks in OS is required.

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 \( \approx 20\times \). That is, APSR also implies less clutter and drain on the system.

Looking into the future, we observe that OpenStack only gains up to \( 3\times \) 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.

TABLE VIII

| Budget | Throughput [req./slot] |
|--------|------------------------|
| 20%    | 6.5                    |
| 40%    | 9.8                    |
| 60%    | 11.7                   |
| 80%    | 12.8                   |
| 100%   | 14                     |
| Number of queries | 110K | 102K | 108K | 2240K |
|--------------------|------|------|------|-------|
| Throughput [req./sec.] | 2.6 | 2.8 | 2.6 | 1.26 |
| Actual decline ratio (δ) | 1.0% | 0.7% | 0.7% | 0% | 3.8% | 13.6% |