Accelerating VM Placement with Adaptive Caching

Gil Einziger
Nokia Bell Labs
gil.einziger@nokia.com

Maayan Goldstein
Nokia Bell Labs
maayan.goldstein@nokia.com

Yaniv Sa'ar
Nokia Bell Labs
yaniv.saar@nokia.com

***Abstract*—Network Function Virtualization (NFV) allows operators to deploy network functions in virtual machines (VMs) and benefit from on-demand deployment. VMs are placed on one of the hosts in the cloud, and existing resource management algorithms assume full knowledge of the system's state. For large clusters, attaining the system's state creates bottlenecks and therefore it takes a long time to deploy network functionalities. Intuitively, placement can be accelerated if the resource management algorithm operates on a cached system state which is not entirely up to date, but the placement quality may suffer. Our work introduces a new cache refresh method that achieves an up to a 5.3x reduction in placement time with only a slight degradation of quality compared to having the complete and up to date system's state.**

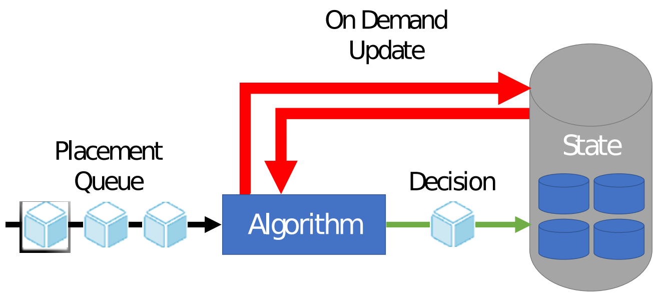
I. Introduction

Network Function Visualization (NFV) is a raising paradigm that promises flexible and scalable network services on cloud infrastructure. Specifically, NFV is about running network functionalities such as firewall, deep packet inspection, load balancing and monitoring in a virtual manner rather than with dedicated middle-boxes [10], [25].

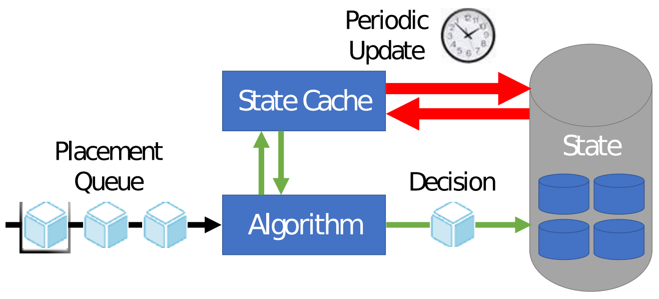
Ideally, the use of NFV allows the operator to deploy functionalities as needed, scale up their capacity to match the current traffic load and even to migrate services from one data center to another. In practice, there are multiple bottlenecks that need to be addressed. For example, [13] optimizes the virtual switching overheads of service chains. Similarly, ClickOS optimizes the time and memory required to create a VM achieving 30 microseconds [15]. Using a container instead of a VM further reduces these overheads, with Kubernetes as the most famous example.

However, to deploy a network service we also need to select a physical host to accommodate the VM. VM placement is often studied as an optimization problem where heuristics are used to optimize various performance aspects such as power consumption [27], [4], virtual switching overheads [13], [19], fault tolerance [14] and resource utilization [17], [21], [31].

Placement algorithms are designed with the common assumption of knowing the complete system's state. In a gist, such algorithms first receive the current state and then perform some calculation on that state according to their unique heuristics and goals. In practice, attaining the state of all available resources is a time-intensive operation which means performance bottlenecks regardless of the placement algorithm used. Indeed, attaining the system's state was reported as a performance bottleneck in both OpenStack [5] and in Kubernetes [11]. For large clouds, OpenStack is reported to take



(a) A standard placement algorithm. Upon each request, the algorithm gathers the current state of all resources (which takes time) and then performs a calculation resulting in a placement decision.



(b) When using a state cache, a snapshot of the state is maintained close to the algorithm. Speedup is achieved as no external resources are needed to reach a placement decision. However, it is unclear how such approach affects the decision's quality.

Fig. 1: An overview of the scenario considered by this paper.

between several hundred microseconds to a few seconds to reach a single placement decision [5], while the vast majority of that time is spent on attaining a fresh snapshot of the system's state. In NFV context, adaptive mechanisms may be too slow to respond to emerging traffic conditions such as a flash crowd or an attack. This may have a noticeable impact on the perceived service quality.

The OpenStack community recognized this problem and suggested state caching as a method to speed up placement decisions. This technique is illustrated in Figure 1. As can be observed, the lengthy periodic update is done in the background and the placement algorithm utilizes the last known state. This technique was reported to yield faster decisions [5], but its impact on the decision's quality was never studied.

A. Our Contributions

We evaluate the impact of caching the system's state on existing algorithms under real workloads. Our evaluation shows that properly configuring the cache refresh interval is crucial for maintaining the placement quality. We identify cases that require low and high cache refresh intervals. This motivates dynamically adapting the refresh interval.

We then introduce *Adaptive Scheduler Cache (ASC)* that uses state cache but dynamically adjusts the refresh interval to the current workload characteristics. Specifically, ASC offers the performance benefit of using a cache along with the placement quality of having a fresh state.

We evaluate ASC along with multiple well-known placement algorithms. We show that the capability to utilize a state cache differs from one algorithm to the next. Yet, in many cases, the placement algorithm achieves similar results when deployed with ASC and with a fresh system state.

Finally, we implement ASC in OpenStack and evaluate it in these settings and show that it achieves up to x5.3 speedup compared to OpenStack's default scheduler. In addition, we show that ASC attains a similar placement quality as the default scheduler despite the latter using a fresh system state.

Paper outline: The rest of this paper is organized as follows: Section II explains the concept and modeling of state caching in placement algorithms and provides background on OpenStack. Section III studies the impact of state caching on placement algorithms, and shows that the desired refresh interval differs from one workload to the next as well as for different stages throughout the workload. Motivated by this understanding, Section IV presents ASC, a cache refresh algorithm that dynamically sets the cache refresh interval. Section V evaluates ASC in OpenStack environment and shows that it provides the benefits of a cached scheduler with only a minor degradation of quality. Section VII surveys related work and we conclude with a discussion in Section VIII.

II. Preliminaries

This section begins with formally modeling resource management and state caches. Then, we briefly survey OpenStack and explain its scheduling process and how state caching effects this process.

A. Modeling Cache Schedulers

We first define the model discussed in this work. We consider the set of resources $\mathbf{R}$. Each resource is a compute node which has multidimensional parameters such as memory, CPU, or disk space. Each resource $r_i \in \mathbf{R}$ is, therefore, a vector and each coordinate correspond to its current amount of resources in each dimension.

A placement request v_k is a vector of demand for each resource. If a placement request v_k is placed on resource r_i then the placement is successful if for every coordinate $r_i > v_k$ and fails otherwise. Upon a successful placement, we update $r_i = r_i - v_k$ to mark the now occupied resources. We model the arrival rate of placement requests as a Poisson process where at each step a request appears with probability λ_a.

Additionally, each successful placement request has a certain time to live. That is, after a certain period the resources claimed by the request are released and we update $r_i = r_i + v_k$. We model this as a Poisson process with a probability of λ_d to drop a request at each step. That is, the average lifetime of a request is: $L = \lambda_d^{-1}$. We use the Poisson process to model situations where the need for the service is temporary. For example, a scale-up performed to address peak demand may no longer be required at a later time.

The cache contains a snapshot of $\mathbf{R}$ and is updated once per τ seconds. Specifically, when $\tau = 1$, then the cache is updated once per second. Alternatively, when $\tau = 10$ it is only updated once per 10 seconds. Note that, the actual number of requests within these periods varies according to the workload.

We assume that the user has some *Service Level Agreement (SLA)* with the cloud provider. That is, the provider assures that the portion of rejected requests is at most $th_{decline}$. We assume that the system satisfies the SLA when operating with the minimal refresh interval ($\tau = 1$).

Table I summarizes notations used in this work.

TABLE I: Notations

Set of total available resources	$\mathbf{R}$
Resources of server i	r_i
Resources required by request k	v_k
Probability to create request (per time unit)	λ_a
Probability to terminate request (per time unit)	λ_d
Average life time of request	$L = \lambda_d^{-1}$
Cache refresh interval [Seconds]	τ
Maximum allowed portion of declined requests	$th_{decline}$

B. OpenStack Background

OpenStack is an open source cloud management platform that manages compute, storage and networking resources. Since its inception in 2010, the project has been growing popularity [28] and is a widely used open source cloud management platform.

In OpenStack, Compute resources are managed by a project called Nova. Nova is also responsible for the entire process between a placement request and a deployed VM. This includes creating the VM, attaining the system's state, making a placement decision, and deploying the VM on the selected host.

Placement decisions are at the heart of Nova, and its scheduler is composed of three stages: a *State* stage, a *Filter* stage and a *Weight* stage as illustrated in Figure 2. In the State stage, the scheduler obtains a snapshot of the resource available in all hosts. This state includes free CPUs, as well as free memory and disk space. In the Filter stage, the scheduler goes over all of the available hosts and outputs only the hosts that have enough resources to accommodate the requested VM. In the Weighting stage, it grades all filtered hosts and then selects the highest graded one to accommodate the new VM. This enables users to customize the scheduler to their needs.

Nova offers multiple scheduling algorithms that differ from each other in their State stage. Specifically, *Filter* scheduler computes a fresh snapshot for each request, while *Caching* scheduler, maintains a periodically updated cached snapshot.

When there are many hosts, the Filter scheduler's throughput plummets as each placement request requires fetching the updated state from every host in the cloud. Attaining a fresh state was shown to be the bottleneck [5] and our work reinforces this claim. Moreover, a similar problem was also observed in Kubernates [11].

This bottleneck motivated the creation of the Caching scheduler, that operates considerably faster but uses a cached system state. However, this work is the first to evaluate the impact of this approach on placement quality.

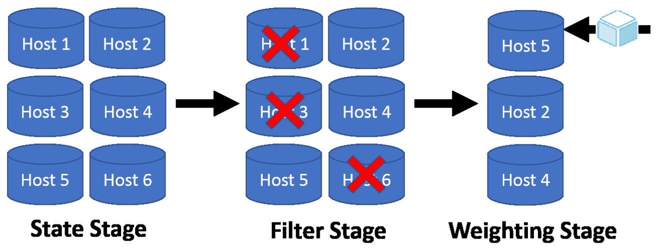


Fig. 2: An illustration of Nova scheduler. The scheduling happens in three stages. In the State stage, the system state is acquired. Next, in the filter stage, hosts that lack the resources to accommodate the request are removed. Finally, in the Weighting stage, a user-defined weighting scheme is used to prioritize the hosts. In this example, Host 5 is selected.

III. The Effect of Cache on Placement Algorithms

This section evaluates the impact of a state cache on various placement algorithms that were originally designed with a complete system state in mind [17], [8]. The evaluated algorithms are heuristics for the following problem: given placement requests, with demand in each dimension (CPU, memory, bandwidth, disk, etc.), and given physical hosts capacities, place as many VMs as possible. We evaluate multiple (fixed) refresh intervals in real placement traces.

A. Evaluated Algorithms

We now list the evaluated algorithms and provide some brief description of their operation. The algorithm *DistFromDiag* [17] is based on the geometry of host and placement request resources. The goal of DistFromDiag is to balance the resource consumption in the host according to its proportions. For example, if a host has 100GB disk and 10GB RAM, it attempts to keep the utilized ratio of disk and RAM at 10:1.

The second heuristic is *Balanced Utilization Placement (BUP)* [8]. BUP is a multidimensional version of Worst Fit [6] strategy and takes into consideration the exact amount of available resources in the hosts. It tries to maximize the remaining amount of resources by placing a new request in the least loaded host. BUP is the default algorithm in OpenStack.

The third algorithm is FirstFit [6] that accommodates a request in the most loaded host that has sufficient resource. The reason for this approach is to minimize the number of required hosts. OpenStack uses this heuristic as an optional method to facilitate energy saving.

The fourth algorithm, *Adaptive* [17], combines FirstFit [6] and BUP strategies as follows: it fills in the hosts using BUP as long as the hosts are below a certain threshold. Once the threshold is crossed it switches to FirstFit. That is, it starts by using all hosts in an equal manner, but when it gets close to the limits it tries to fit requests to the leftover space. Finally, we evaluate the *Random* heuristic, that randomly chooses a host that has sufficient resources to accommodate the request.

Our evaluation compares different refresh intervals to two extremes, *Always* where the cache is refreshed prior to accommodating each placement request, and *Never* where the cache is never refreshed. Note that the common complete state knowledge assumption is modeled as 'Always'.

B. Datasets

We use three datasets collected from various real systems. For each dataset, we approximate the minimal number of hosts to accommodate all requests. This problem is NP-hard and therefore we use the approximation suggested by [17]. Briefly, we run each placement algorithm and let it "allocate" new hosts when it fails to accommodate a request. We repeat this process multiple times, with a randomly selected order of requests. The approximation is the minimal number of hosts attained by an algorithm for any request order.

The first dataset (NFV) is taken from an NFV management and orchestration (MANO) system, which contains placement requests taken from a large proprietary NFV system. The placement requests are for VMs of different sizes (flavors) and all hosts are of identical size. In this dataset, hosts and placement requests are characterized by a tuple of $< memory, storage >$. The host size is normalized to $< 1, 1 >$, and VM flavors are normalized to that size. Table II, shows a breakdown of the number of VMs by memory and storage. We replicated the requests in Table II 10 times (total of 4370 requests) and randomly ordered them. In this case, 279 hosts are required to accommodate all the 4370 requests.

TABLE II: Normalized breakdown of requests for VM image by memory and storage, obtained from proprietary NFV dataset.

Mem. \Storage	0.01	0.04	0.1	0.3	0.54	Total
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 second dataset is taken from Google cluster management data [18]. It represents data collected for $12,477$ machines, each characterized by a tuple of $< CPU, memory >$. For the purposes of our evaluation, we assumed that these are characteristics of VMs that should be placed on hosts. As summarized by [12], 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. Half the hosts are of size $< 1, 2 >$ and the other half are of size $< 2, 1 >$ [17]. Table III gives the breakdown of the number of VMs of different sizes. In this case, 5995 hosts are needed to accommodate all the requests.

TABLE III: Breakdown of the number of placement request by CPU and memory, obtained from the Google dataset.

Mem. \CPU	0.25	0.5	1.0	Total
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

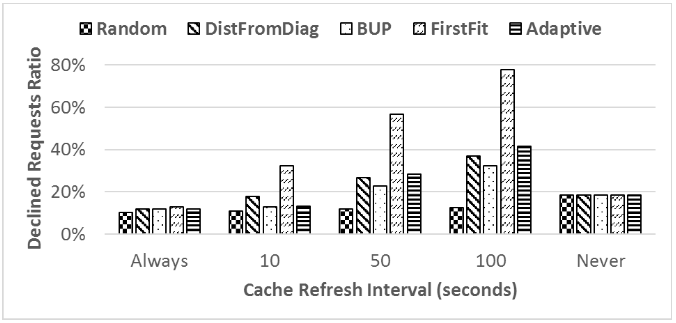
Finally, we considered a randomly generated dataset based on data from Amazon EC2 hosts and VM flavors [16], [17]. The normalized values of all the possible $< CPU, memory >$ requests are taken from [17] and are shown in Table IV. Each column in the table represents one possible tuple. The requests were generated randomly, such that there were 1000 small placement requests and 100 large ones (total of 1100 requests). We considered a request as 'small' if the requested CPU value was lower than 0.4. In this dataset, 126 hosts are sufficient to accommodate all the requests.

C. Experiments

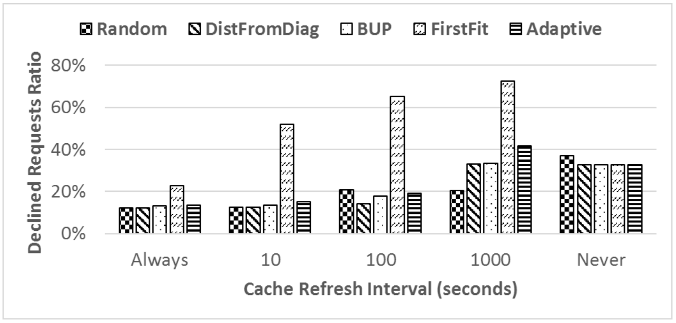
Our first experiment considers the classic placement scenario where we attempt to accommodate as many requests as possible. The requests have an infinite lifetime, i.e., once a request is accommodated its resources remain occupied for the entire experiment. For each dataset, we used our approximation for the minimal number of hosts required to accommodate all requests as mentioned in the description of each dataset. We chose to start with this evaluation scenario as it is commonly considered by previous works [17], [8], [13]. For simplicity, we play one request per second.

Our results are illustrated in Figure 3. As expected, the best performance is with fresh data ('Always') for all datasets. Yet, even with fresh data, the algorithms cannot place all the requests. This is due to the greedy nature of these algorithms which is sensitive to the order of the requests. Therefore, state caching is natural, as placement algorithms already trade quality for speed by using greedy heuristics rather than solving an NP-hard problem. Thus, we can justify some minor quality degradation to gain further speedup through caching.

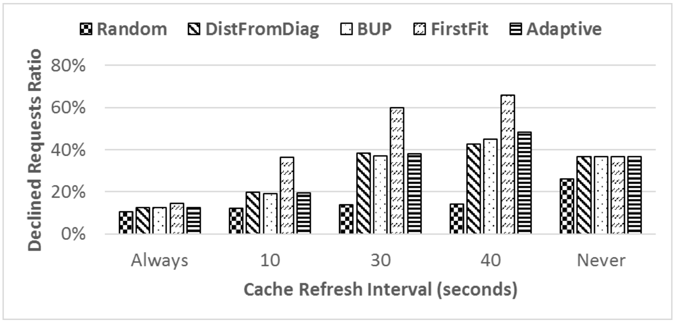
The quality degradation is influenced by multiple factors. For example, it is apparent that FirstFit algorithm is extremely sensitive to the refresh interval. The reason for this is that FirstFit accommodates requests in the most loaded host. When the cache is not refreshed often enough, the algorithm continuously places requests in an already exhausted host.



(a) NFV dataset.



(b) Google dataset.



(c) Amazon dataset.

Fig. 3: Percentage of declined requests for different placement algorithms and cache refresh intervals applied to NFV, Google and Amazon datasets.

On the other hand, the nature of the dataset can also affect the placement quality. Specifically, for all but the FirstFit algorithm, the decline ratio for a 40 seconds interval in the Amazon dataset (Figure 3c), is similar to the decline ratio in the Google dataset (Figure 3b) for a 1000 seconds interval. The different scales for each dataset were chosen to show the gradual degradation of quality. The variation in the shown points, further emphasizes the dependence of the refresh interval on the specific dataset.

Moreover, some algorithms are more suitable for state caching than others. For example, Random degrades in performance less than other algorithms for the same refresh interval. Therefore, when using Random we can justify longer refresh intervals. Observe that for the NFV dataset, when the cache is updated every 100 second, Random discards 12.5% while DistFromDiag discards 38% of the placement requests.

TABLE IV: Breakdown of placement request sizes by CPU and memory, obtained from Amazon EC2 dataset.

CPU	0.354	0.142	0.07	0.035	0.333	0.167	0.083	0.2	0.1	0.4	0.8	0.833	1	0.5	0.5
Mem.	0.062	0.031	0.016	0.008	0.125	0.063	0.031	0.016	0.008	0.031	0.063	0.25	0.25	0.125	0.5

Therefore, we cannot determine the required refresh interval without knowing which placement algorithm is used.

What may look surprising is that for all datasets it is better to never refresh the cache than to refresh it infrequently. That is when the algorithms are given no state at all, they perform better than when given a very stale state. Interestingly, in that case, they see all hosts as empty and thus break the symmetry between them at random. This case is different from the Random heuristic as the Random heuristic only considers hosts that can accommodate the request. On the other hand, a stale state shows less symmetry between the hosts and thus a smaller subset of hosts is repeatedly selected. For example, let's consider BUP that selects the least loaded hosts. At the extreme, it may try and place all the requests until the next cache refresh to the same host because it is slightly less loaded than the rest. In that case, the host quickly runs out of resources before the cache is updated. This results in a large portion of declined requests. Therefore, we deduce that in some cases, stale state information can be worse than having no information at all.

We conclude that the effectiveness of state caching depends on the refresh interval, the dataset, and the specific cache algorithm used. Moreover, state caching should be done carefully as in some cases it degrades performance so much that a blind assignment of requests to hosts becomes a better strategy. Therefore, a single fixed size interval cannot be the solution, and we suggest adapting the interval to the circumstances.

IV. Adaptive Scheduling Cache (ASC)

This section presents *Adaptive Scheduling Cache (ASC)* an algorithm for adapting the cache refresh interval to the workload. Intuitively, ASC attempts to ensure that the decline ratio is always kept below a certain threshold. To do that it monitors the requests issued during the last refresh interval (τ) and calculates their decline ratio. ASC conservatively increases τ when the decline ratio is considerably lower than the threshold and aggressively reduces τ when it is above the threshold. This approach is inspired by TCP that increases the sender's window gradually, but aggressively reduces it upon a packet loss. Initially, we experimented with other heuristics such as an additive increase to the threshold and a simple step function for the refresh interval. We progress with ASC as it showed superior performance in the initial evaluation.

The pseudo-code for ASC is given in Figure 4. The algorithm accepts as the predefined thresholds $th_{decline}$ and th_{low} which are used to dynamically adapt the state refresh interval. Specifically, when the decline ratio over the last τ seconds is higher than $th_{decline}$ the interval is reduced and when it is lower than th_{low} it is increased. It also accepts an initial refresh rate, defined by $startRate$ parameter. We used $\tau = 60$ as the initial value. The parameters $minInterval$ and $maxInterval$ determine the minimum and maximum allowed refresh intervals. Their default values are 1 and ∞ respectively. These intervals are best used according to the system's limitations. E.g., if it takes 5 seconds to perform a cache refresh then the minimal interval should be 5.

Finally, the values $speedUpStep$ and $slowDownStep$ determine how the interval is changed. Specifically, τ is multiplied by these parameters once per interval if the conditions are met. We used $slowDownStep = 1.1$ and $speedUpStep = 0.5$ to make sure that the response for exceeding $th_{decline}$ is aggressive. Alternatively, when the decline ratio is less than th_{low}, we slowly increase the refresh interval to make sure that it remains within the SLA (lower than $th_{decline}$).

V. ASC Evaluation

This section describes our ASC evaluation. We start by showing how it affects the placement algorithms evaluated in Section III. Then, in Section VI we describe our implementation and evaluation of ASC within the OpenStack framework. That evaluation is focused on the existing placement algorithms within OpenStack and their interaction with ASC.

A. ASC Influence on Placement Algorithms

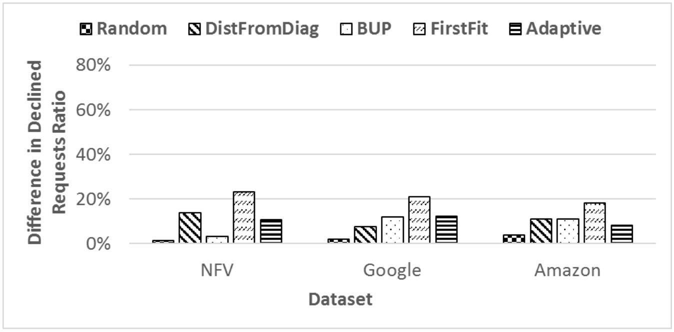
We begin by comparing Random, BUP, FirstFit, DistFromDiag and Adaptive placement algorithms [17], [8] in three different states. First, we evaluate them with complete state information as intended by their authors. Next, we evaluate them with two state caching options: with our own ASC algorithm that dynamically adapts the cache refresh interval and with a fixed interval.

Placement requests arrive in a Poisson process and remain alive for a certain period of time that is determined by another Poisson process. Thus the system's state changes as the cache is unaware of some of the recently placed/dropped VMs. Such a scenario emulates the state in a cloud where VMs are deployed dynamically and may be removed at any time according to customers' needs. The intent of our experiment is to keep the hosts relatively highly utilized as is expected of a successful cloud system.

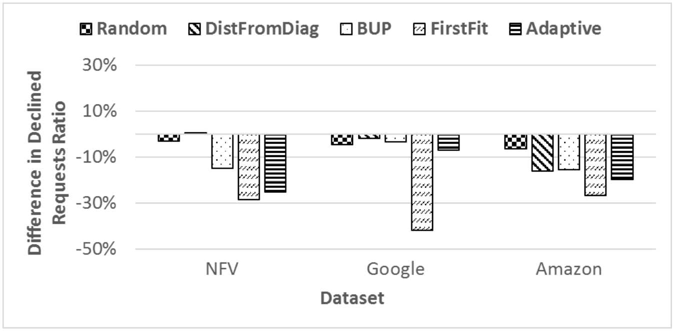
Figure 5 shows the results for this experiment, where we set $th_{decline}$ and th_{low} to their default values (5% and 3% respectively). Figure 5a compares ASC to the complete state situation and it is clearly evident that having a complete state is always better than using a cache. Yet, in some algorithms (e.g. Random) the degradation in quality is low. In contrast, DistFromDiag is the most unpredictable and may suffer a high degradation of quality. The work of [17] showed that all heuristics attain similar quality when they are fed with a complete state, and our evaluation reveals that algorithms are considerably better than when fed with a cached state.

```
function ASC($th_{decline}, th_{low}, startRate, maxInterval, minInterval, speedUpStep, slowDownStep$)
    $\tau \leftarrow startRate$                                  ▷ Initial refresh rate
    $ratio \leftarrow 0$                                         ▷ The decline ratio for the analyzed window of requests
    while True do
        $sleep(\tau)$                                            ▷ Sleep for $\tau$ time units
        $placed \leftarrow$ number of successfully placed requests during $\tau$
        $declined \leftarrow$ number of declined requests during $\tau$
        $ratio \leftarrow declined \div (declined + placed)$
        if $ratio > th_{decline}$ then                           ▷ Percentage of declined requests is over the SLA
            $\tau \leftarrow max(minInterval, \tau \times speedUpStep)$      ▷ Decrease refresh rate up to a predefined minimum
        else if $ratio < th_{low}$ then                          ▷ Percentage of declined requests is very low, sample less frequently
            $\tau \leftarrow min(maxInterval, \tau \times slowDownStep)$     ▷ Increase refresh rate up to a predefined minimum
        end if
    end while
end function
```

Fig. 4: A pseudo-code of ASC.



(a) Complete state compared to ASC.



(b) Fixed 24 second interval compared to ASC with the same **average** interval.

Fig. 5: Evaluating the **placement quality** of ASC compared to no state caching and a fixed refresh interval. Positive means that the alternative is better, negative means that ASC is better.

On these datasets, the average refresh interval of ASC is 24 seconds. In Figure 5b, we compare ASC to a fixed refresh interval of 24 seconds. This evaluation emphasizes the need for dynamically adapting the refresh interval as in all but a single data point ASC achieves a better quality than the fixed interval despite having the same number of cache refreshes on average. This implies that the required refresh interval varies throughout the workload and that we achieve concrete benefit from dynamically adapting it to the current conditions rather than statically configuring it to the workload. In comparison, the default refresh interval for Nova's Caching scheduler is 60 seconds which underperformed in the tested scenario.

VI. OpenStack Implementation Evaluation

We now evaluate ASC in OpenStack environment. For this evaluation we installed OpenStack (Mitaka release) [2] on a high-end HP ProLiant BL460c Gen9 server with two Intel(R) Xeon(R) CPU E5-2680v4, where each processor has 28 cores (56 cores total) running at 2.4 GHz; with total RAM of 256GB.

Our deployment runs the concrete scheduler implementation and the remote hosts are emulated with the OpenStack's Benchmarking for Scheduling project [1]. That is, OpenStack's state is maintained as usual, but the remote hosts only perform bookkeeping and do not actually run the VMs. From the scheduler's perspective, this deployment is completely realistic. That is, the same scheduler code is run in our benchmark and in an actual deployment. We used the NFV dataset that is composed of real VM placement requests. The dataset is summarized in Table II. Moreover, we compared Nova's Filter scheduler and Nova Caching scheduler which with their default parameters as included in the Nova project and compared them to our own ASC scheduler.

A. *OpenStack Throughput Evaluation*

Figure 6 evaluates the throughput of ASC compared to Nova Caching and Filter schedulers when requests are taken from the NFV dataset. In this experiment, we first deploy the system with a varying number of hosts (compute nodes) and then issue 200 placement requests in a batch. These requests are processed by OpenStack partly in parallel until they reach the (single) scheduler. We measure OpenStack's throughput in decisions per second. Note that Nova's Filter scheduler is the default scheduler and that Nova's Caching scheduler is an optional scheduler with a fixed cache refresh interval having a default value of 60 seconds.

As can be observed, the Filter scheduler degrades in throughput as the number of nodes increases. At the extreme, for 1000 nodes, it degraded from over 4 requests per second to only 0.75 requests per second. This is a similar observation to the one made by [5]. In contrast, the throughput of ASC and the Caching scheduler is very similar. Both algorithms are less affected by the number of hosts. ASC is up to x5.3 faster than the Filter scheduler. This emphasizes throughput improvement that can be attained from state caching in OpenStack. We conclude that ASC achieves a similar throughput compared to the Caching scheduler.

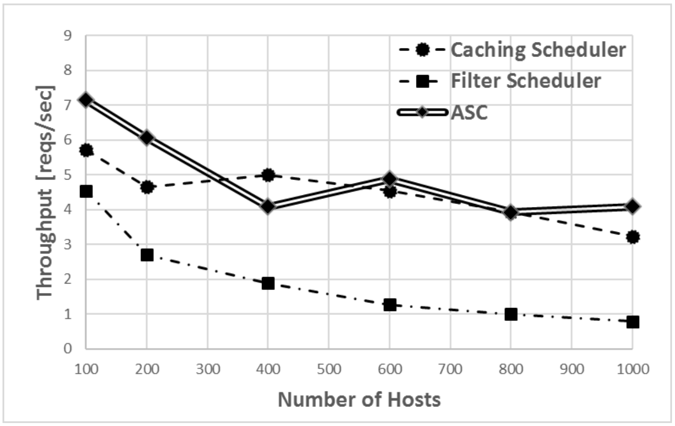


Fig. 6: Effect of the number of hosts on the throughput (in decisions per seconds) for different schedulers.

B. OpenStack Placement Quality Evaluation

Our second experiment exemplifies the operation of ASC over time. We used a deployment of varying number of hosts. We replicated the NFV dataset 10 times and randomly ordered the requests. We then adjusted the average lifetime so that a growing number of hosts would remain nearly full for the entire experiment. We modeled the placement requests as a Poisson arrival process with half a second as the basic time unit and $\lambda_a = 0.5$. That is, a new VM is placed once per second on average. The complete experiment configuration is summarized in Table V.

TABLE V: Parameters used for the results in Table VI

Number of available hosts	h	28, 56, 84
Number of requests	$\#reqs$	4370
Time unit	tu	0.5
Avg life time of request (in secs)	L	437, 874, 1311
Probability to create request	λ_a	0.5
Probability to terminate request	λ_d	$\lambda_d = 1/L * tu$
Declined requests thresholds	th_{low}	0.03
	$th_{decline}$	0.05

Table VI summarizes the decline ratio in the different settings on three deployments 28, 56 and 84 hosts. As can be observed, quality can improve with the number of hosts and as expected the Filter scheduler attains the highest quality and declines only $0.8 - 2.5\%$ of the requests. ASC declines $2.5 - 3\%$ of the requests and thus it is close in quality to the Filter scheduler. Last is the Caching scheduler that declines $7.8 - 11\%$ of the requests. Clearly, its default refresh interval is too long for this experiment. It is also interesting to notice, that although the relative load on hosts remains the same, placement quality improves with the number of hosts. A large cluster is more likely to find a host that can accommodate a VM even if the majority of hosts cannot.

Our third experiment is a faster and larger version of the previous experiment. That is, we increased the rate of placement requests to on average 1.5 per second and evaluated

TABLE VI: Declined requests ratio in the OpenStack environment for ASC and the existing OpenStack schedulers.

Scheduler	Declined Requests Ratio		
	28 hosts	56 hosts	84 hosts
Nova Filter	2.5%	1.7%	0.8%
Nova Caching	11%	7.3%	7.8%
ASC	3%	2.8%	2.5%

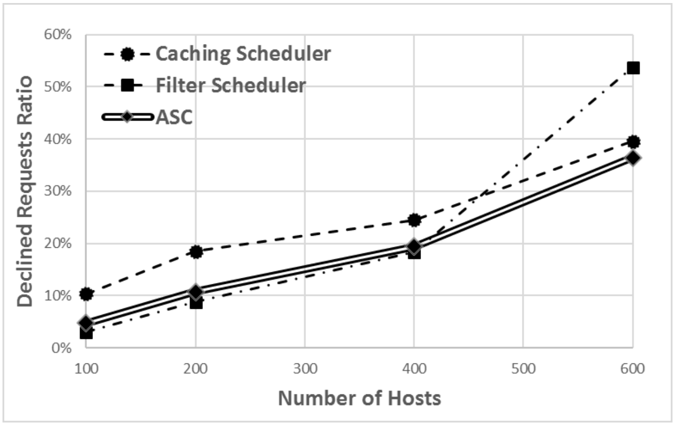


Fig. 7: Percentage of declined requests for different schedulers and a large number of hosts.

a number of hosts ranging from 100 to 600. The experiment uses up to 48k placement requests in each data point. Note that the Filter scheduler is too slow to handle over 600 hosts.

Figure 7 shows the result of this more demanding evaluation. As illustrated, ASC follows the quality of the Filter scheduler closely until 400 nodes, then in 600 the Filter scheduler is too slow to cope with the placement requests and thus ASC actually performs better. It is also interesting to observe that the decline ratio climbs with the number of hosts, the reason for that is the stress placed on the system and the occasional times within the trace where the queue of placement requests overflows and drops some of the requests. Thus, we conclude that ASC achieves a similar quality compared to the Filter scheduler. ASC is also considerably faster than the Filter scheduler and can cope with higher request arrival rates. Therefore, when the request arrival rate is high enough the quality of ASC is better than that of the Filter scheduler.

C. Under the hood of ASC

Next, we provide some insight into how ASC operates. To do so, we monitor its window size and the decline ratio across a full experiment. Specifically, Figure 8, demonstrates its internal behavior in the 28 hosts experiment. The solid line illustrates the requests decline ratio while the variation in cache refresh interval is shown via a dashed line. As can be observed, ASC starts with a large interval of 60 seconds but it quickly observes that the decline ratio is higher than $th_{decline}$. Therefore, it drastically reduces the refresh interval. Initially, the hosts are relatively empty and ASC can attain low decline ratios with long refresh intervals. As the hosts become crowded it shortens the interval to keep the decline ratio

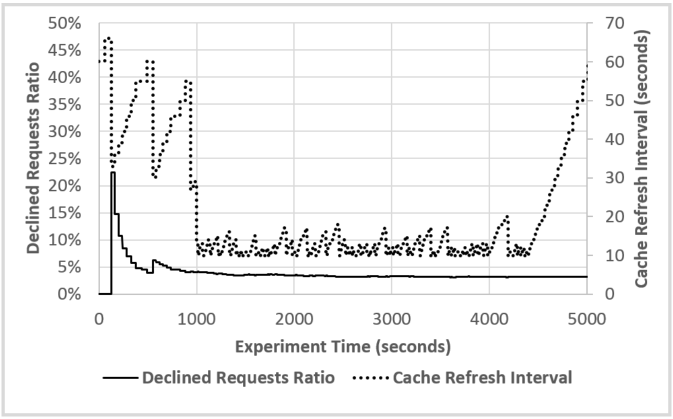


Fig. 8: Percentage of declined requests (solid line) and ASC's cache refresh interval (dashed line) on the NFV dataset.

within the SLA. Finally, once all requests are placed, there are no declined requests in each interval, and ASC gradually increases the refresh interval. This saves resources when the system is idle. Note that the decline ratio shown by the solid line is the total ratio measured over all of the requests so far, while ASC examines requests issued within the last τ seconds.

VII. Related Work

VM placement within clouds is an extensively studied research field [14], [27], [4], [30], [17], [9], [31], [21]. Various algorithms differ from each other in their objectives. For example [4] attempts to place VMs in as few servers as possible. This potentially enables the cloud provider to save power. Additionally, OpenStack also offers a placement policy with the same goal. Other optimization goals include migration overheads [4], fault tolerance [14], NFV switching [13], [19] and the resource utilization [17]. Finally, the impact of network bandwidth on VM placement is studied in [9], [31], [21].

The works of [17], [8] evaluates placement algorithms designed to optimize the effective capacity of hosts. That is, to maximize the number of placement requests that can be accommodated by the hosts. An assumption which is almost universal among placement algorithms is the need to know the system's state prior to the placement decision. The algorithms in [17], [8] are an example of this assumption. Therefore, our work investigates the effects state caching has on the performance of such algorithms.

Generally, cloud systems [2], [23] offer a wide array of services such as network, storage, monitoring, visualization, orchestration and VM placement. Specifically for VM placement, OpenStack [2] suggests a scheduler called Filter scheduler that can be customized by the user. Given a placement request, the Filter scheduler fetches the up to date system's state and then filters out the hosts that lack the resources to satisfy the request. Finally, user-defined filter weights are applied to break the symmetry between capable hosts.

An extensive evaluation of open source cloud management systems is found in [26]. Their work concludes that OpenStack is competitive compared to the alternatives. This motivated our work to choose OpenStack as our evaluation target. Additionally, Our work is orthogonal to works that suggest new placement algorithms to be deployed within OpenStack [7].

Omega [20] is Google's scheduler for large clusters. Omega suggests various kinds of schedulers and Nova's current schedulers are based on Omega. Google's concern is with the throughput of the scheduler and in Omega they suggest parallelism with complex synchronization between schedulers as a mean to extend throughput. Our work accelerates the single scheduler which is complimentary to Omega.

Some machine learning techniques for adaptive placement have been studied in the past by [3], [24], [29]. Such methods are also orthogonal to our approach. Specifically, ASC can also be used when the placement decision is done using these techniques. An extensive survey of autonomous resource management is found in [22]. The survey leads us to the conclusion that our work is the first to study the impact of state caching on the quality of placement algorithms.

VIII. Conclusions and Future Work

Much of the NFV promise lies in the capability to rapidly deploy network functions and exploit existing cloud infrastructure. However, in the current technology, this remains unachieved for large clouds. Specifically, in order to run a virtual network function, one needs to create the functionality in a VM or container and also accommodate it on one of the hosts. While the creation of process can be done rapidly, existing placement algorithms assume knowledge of the exact state and attaining a fresh snapshot of the system's state is a timely operation in a large cloud. Therefore, regardless of the specific placement algorithm, the time to deploy NFV services is dominated by the decision process.

Placement speed in large clusters is a known concern for the industry [20] as well as for the open source community for the last few years. Specifically, the work of [5] showed that OpenStack's default scheduler does not scale to large clusters and that the reason for this slowness is obtaining a fresh snapshot for each VM placement. This process is common among management systems and indeed similar problems were also observed in Kubernetes [11]. Caching the system's state and running placement algorithms on a cached state was suggested as an effective method to accelerate performance.

However, the impact of such an approach on quality remained unclear. Our work began with studying the effect of using a state cache on the placement quality of a variety of previously suggested algorithms. Our study showed that quality degrades when the cache refresh interval is too long. However, there is a range where the impact on quality is minimal. Yet, the desired range is algorithm and workload dependent which motivates dynamic refresh intervals.

We presented ASC, an adaptive method that adjusts the cache refresh interval based on the circumstances. To do so, ASC receives an SLA from the user defining the maximum allowed portion of declined placement requests. It then shortens the refresh interval when the portion of declined requests exceeds the SLA. Similarly, if the portion of declined requests is very small, ASC increases the refresh interval. We evaluated ASC when deployed with five different placement algorithms and used three real request datasets.

Our evaluation showed that ASC generally achieves a similar quality to the vanilla algorithms and that some algorithms are more suitable to state caching than others. We believe that cache friendliness is a new trait of placement algorithms that should guide the design of future heuristics.

Finally, we implemented ASC in OpenStack and evaluated it compared to the Filter and Caching schedulers. We used a real NFV workload and showed that the Caching scheduler and ASC achieve very similar throughput and that the default Filter scheduler is considerably slower. When compared to the Filter scheduler, ASC achieves a speedup of up to $5.3x$ with only a minor degradation in placement quality. When compared to the Caching scheduler, ASC achieves better placement quality and operates at a similar speed. ASC combines both speed and quality and exposes an attractive point in the scheduler design space. It constitutes an important step toward realizing the NFV vision in clouds and we believe that it is a strong candidate for inclusion in future OpenStack releases. Moreover, ASC is also relevant to the general cloud community and can be implemented in other systems as well.

We observe that even after our improvements, the time required to deploy a network functionality is typically dominated by the placement algorithm. The reason for that is that placement algorithms perform a calculation which digests the entire system state. Thus, in the future, we plan to develop algorithms that only requires a partial system state.

Acknowledgments This work is partially funded by the European Union under the H2020 and 5G-PPP Phase2 programs, under Grant Agreement No. 761 557 (project NGPaaS). We thank Shachar Beiser for exposing us to this problem.

References

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

[2] OpenStack. https://www.openstack.org/, 2017.

[3] I. Ari, A. Amer, E. Miller, S. Brandt, and D. Long. Who is more adaptive? ACME: adaptive caching using multiple experts (ext. abstract). In *Workshop on Distributed Data and Structures (WDAS)*, pages 143–158, 2002.

[4] U. Bellur, C. S. Rao, and S. D. M. Kumar. Optimal placement algorithms for virtual machines. *CoRR*, abs/1011.5064, 2010.

[5] Y. Cheng. Dive into nova scheduler performance. Open Stack Summit 2016. https://www.openstack.org/assets/presentation-media/7129-Dive-into-nova-scheduler-performance-summit.pdf.

[6] L. Epstein and L. M. Favrholdt. *On-Line Maximizing the Number of Items Packed in Variable-Sized Bins*, pages 467–475. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[7] J. Han and J. Kim. Analytics-leveraged box visibility for resource-aware orchestration of smartx multi-site cloud. In *2016 International Conference on Information Networking (ICOIN)*, pages 321–323, Jan 2016.

[8] F. Hao, M. Kodialam, T. V. Lakshman, and S. Mukherjee. Online allocation of virtual machines in a distributed cloud. *IEEE/ACM Transactions on Networking*, 25(1):238–249, Feb 2017.

[9] T. Huang, C. Rong, Y. Tang, C. Hu, J. Li, and P. Zhang. Virtualrack: Bandwidth-aware virtual network allocation for multi-tenant datacenters. In *IEEE ICC 2014*.

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

[11] Kubernetes. Scheduling performance issues. https://github.com/kubernetes/kubernetes/issues/20540 https://github.com/kubernetes/kubernetes/issues/32361 https://github.com/kubernetes/kubernetes/issues/18266.

[12] Z. Liu and S. Cho. Characterizing machines and workloads on a google cluster. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 397–403, Sept 2012.

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

[14] F. Machida, M. Kawato, and Y. Maeno. Redundant virtual machine placement for fault-tolerant consolidated server clusters. In *IEEE Network Operations and Management Symposium*, 2010.

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

[16] K. Mills, J. Filliben, and C. Dabrowski. Comparing vm-placement algorithms for on-demand clouds. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 91–98, Nov 2011.

[17] D. Raz, I. Segall, and M. Goldstein. Multidimensional resource allocation in practice. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, pages 1:1–1:10, New York, NY, USA, 2017. ACM.

[18] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, Nov. 2011. Revised 2012.03.20. Posted at URL http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[19] Y. Sa'ar, D. Raz, and M. C. Luizelli. Optimizing NFV chain deployment through minimizing the cost of virtual switching. In *Conference on Computer Communications, IEEE INFOCOM*, pages 1–9, 2018.

[20] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[21] 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. *, an International Journal on Engineering Science and Technology*, 20(2):616 – 628, 2017.

[22] S. Singh and I. Chana. Qos-aware autonomic resource management in cloud computing: A systematic review. *ACM Comput. Surv.*, 48(3):42:1–42:46, Dec. 2015.

[23] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, Sept. 2009.

[24] S. Sulaiman, S. M. Shamsuddin, A. Abraham, and S. Sulaiman. Intelligent web caching using machine learning methods.

[25] 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*, 2015.

[26] V. N. Van, L. M. Chi, N. Q. Long, G. N. Nguyen, e. C. S. Le, Dac-Nhuong", S. K. Raju, K. J. Mandal, and V. Bhateja. *A Performance Analysis of OpenStack Open-Source Solution for IaaS Cloud Computing*, pages 141–150. Springer India, New Delhi, 2016.

[27] A. Verma, P. Ahuja, and A. Neogi. pmapper: Power and migration cost aware application placement in virtualized systems, 12 2008.

[28] X. Wen, G. Gu, Q. Li, Y. Gao, and X. Zhang. Comparison of open-source cloud management platforms: Openstack and opennebula. In *9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 2457–2461, May 2012.

[29] Z. Xiao, W. Song, and Q. Chen. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1107–1117, June 2013.

[30] J. Xu and J. A. B. Fortes. Multi-objective virtual machine placement in virtualized data center environments. In *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 179–188, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Y. Yao, J. Cao, and M. Li. A network-aware virtual machine allocation in cloud datacenter. In *Proceedings of the 10th IFIP International Conference on Network and Parallel Computing - Volume 8147*, IFIP NPC 2013, pages 71–82.