Poster Abstract: Parallel VM Placement with Provable Guarantees

Itamar Cohen*, Gil Einziger*, Maayan Goldstein†, Yaniv Sa'ar‡, Gabriel Scalosub*, and Erez Waisbard†
* Ben-Gurion University of the Negev, Beer Sheva, Israel ({itamarq@post.,gilein@,sgabriel@}bgu.ac.il)
† Nokia Bell Labs ({maayan.goldstein,erez.waisbard}@nokia.com)
‡ Independent (yaniv.saar.mail@gmail.com)

***Abstract*—Efficient on-demand deployment of VMs is at the core of cloud infrastructure but the existing resource management approaches are too slow to fulfill this promise. Parallel resource management is a promising direction for boosting performance, but when applied naïvely, it significantly increases the communication overhead and the decline ratio of deployment attempts. We propose a new dynamic and randomized algorithm, APSR, for parallel assignment of VMs to hosts in a cloud environment. APSR is guaranteed to satisfy an SLA containing decline ratio constraints, and communication overheads constraints. Furthermore, via extensive simulations, we show that APSR obtains a higher throughput than other commonly employed policies (including those used in OpenStack) while achieving a reduction of up to 13x in 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. NFV services are composed of *Service Chains* of multiple *Virtual Network Functionalities*, thus when we need to scale up a service we may need to allocate an entire service chain in a timely manner. Current resource management architectures either provide guaranteed performance (using a centralized view of the system state), or provide high throughput at the cost of increased decline ratio (e.g., via partitioning or parallelism) [2]–[4].

We consider a set $\mathbf{H}$ of n *hosts* where each $\vec{h} \in \mathbf{H}$ is a *state* vector describing the multi-dimensional capacity of the resources available at the host (in terms of, e.g., memory, CPU, or disk space). We further consider a set $\mathbf{R}$ of *requests*, each modeled as a vector of demand for each resource. We assume time is slotted, and in each slot requests may arrive at the system, and are queued for assignment to hosts. We denote by s the number of *parallel schedulers* that may perform scheduling decisions of pending requests to hosts simultaneously in any single time slot. In each time slot t, each of the first s pending requests are matched to distinct schedulers that assign them to one of the hosts. Each such scheduler may query some subset of hosts for their resource state, and assigns the request to one of the queried hosts. The assignment is successful if the host has enough resources to satisfy the request, and is a failure otherwise. The host "loses" resources on successful assignments, and regains these resources when the requests terminate. We note that when $s > 1$, multiple schedulers may concurrently assign pending requests to the same host; these are resolved in an arbitrary order. For any specific time slot, we let k denote the estimated number of hosts that may accommodate a request that may arrive at that time.

The *decline ratio* is the ratio between the number of failed requests and the total number of requests handled by the system. We assume the system is subject to a *Service Level Agreement (SLA)* which limits the decline ratio to be at most ε, for some $\varepsilon \in [0, 1]$. We further assume we are given some budget B such that the number of queried hosts in every time slot is at most B. In every time slot, we denote by d the number of hosts queried by any scheduler with a pending request. A *valid configuration* of schedulers determines s and d, such that $s \cdot d \leq B$, and the probability of a failed request is at most ε.

Our goal is to find a valid configuration that maximizes the number of parallel schedulers (s). We refer to this problem as the *Constrained Maximum Parallelism (CMP)* Problem.

II. Common Algorithms Employed

Current systems make use of a variety of algorithms for making assignment decisions. In our evaluations we focused on those most commonly used, including: (i) *WorstFit (WF)*, which serves as OpenStack's default placement algorithm [2], (ii) *FirstFit (FF)*, (iii) the *Adaptive* algorithm, which switches from WF to FF once the load passes a threshold T (in our evaluations, $T = 0.6$), (iv) *WorstFit-Rand (WFR)* and *FirstFit-Rand (FFR)*, which pick a random host from the top ℓ-ranking hosts in WF, and FF, accordingly (in the spirit of the option available in OpenStack. In our evaluation of WFR and FFR we set $\ell = 5$), and (v) *Random*, which selects a host uniformly at random among the available hosts.

Our initial evaluation indicates that random approaches do much better than classic deterministic ones, and the decline ratio significantly increases as we use more schedulers. This motivates our approach of combining *dynamic* and *random* approaches to parallel scheduling.

III. The APSR Algorithm

We suggest the *Adaptive Partial State Random (APSR)* algorithm, which implements an efficient random policy that dynamically adjusts the number of schedulers (s) according to the system's perceived utilization as captured by estimating the number of available hosts (k). Whenever APSR uses parallel schedulers ($s > 1$) it is guaranteed to satisfy the SLA and budget constraints.

s	APSR	Rand	FF	FFR	WF	WFR	Adapt
1		0.3	0.0	0.0	0.3	0.3	0.3
5	0.4	0.4	11.1	2.5	4.0	1.0	2.2
10	$\bar{s}=14$	0.5	23.3	5.2	8.2	2.1	3.1
20		0.7	35.7	10.0	12.1	3.3	11.6
50		0.8	39.0	10.8	16.7	3.9	16.0

TABLE I

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 AVERAGE NUMBER OF SCHEDULERS ($\bar{s}$) IS LISTED BELOW THE DECLINE RATIO. THE SLA DECLINE RATIO CONSTRAINT IS $\varepsilon = 5\%$.

	APSR			Random		
	Target Decline Ratio (ε)			Number of Schedulers		
	3%	5%	10%	1	10	100
Number of Queries (K)	1553	811	578	11000		
Throughput [req./slot]	7.2	14	19.6	1	10	19.8
Decline Ratio (δ)	0.4%	0.4%	0.6%	0.3%	0.5%	0.8%

TABLE II

TOTAL NUMBER OF QUERIES, THROUGHPUT AND ACTUAL DECLINE RATIOS OF APSR (WITH AVERAGE NUMBER OF SCHEDULERS) VERSUS RANDOM (WITH VARYING FIXED NUMBER OF SCHEDULERS).

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

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

Evaluation of APSR vs. Other Policies. We consider the SLA decline ratio constraint of $\varepsilon = 5\%$, and let the overall budget of state queries to be $B = n$ (as the budget of a *single* OpenStack scheduler). Table I summarizes the results comparing APSR to the common approaches currently used in cloud environments, for an NFV dataset [1]. We note that APSR manages to run many parallel schedulers in most scenarios, while satisfying the SLA decline ratio and budget constraints.

Table II compares the throughput, 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%. Relaxing (i.e., increasing) APSR's target decline ratio constraint increases its parallelism which in turn increases the throughput. This highlights the tension between decline ratio and parallelism.

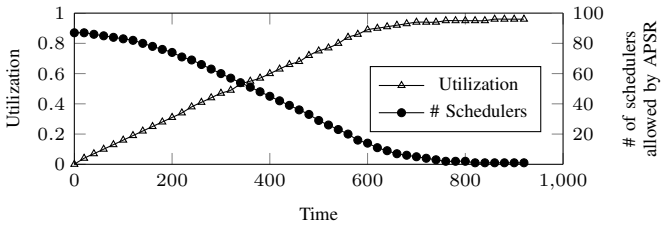
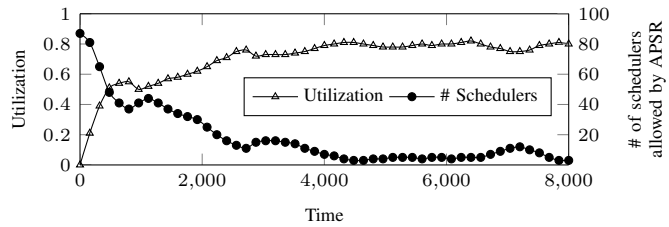
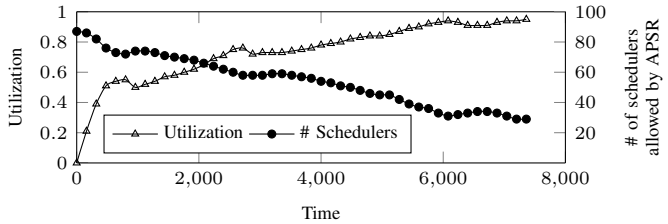


Fig. 1. Cloud resource utilization and the number of schedulers in APSR for the NFV dataset under Poisson arrivals (requests have infinite lifetime).

Under the Hood of APSR. We study the interplay between the system's utilization and the level of parallelism offered by APSR. Fig. 1 depicts the number of schedulers and the system utilization of APSR for infinite duration requests, with Poisson arrivals. As system utilization increases, APSR adapts and decreases the number of schedulers it employs (thus allowing each scheduler to query more hosts while still complying the budget constraint, and also increasing the probability of finding an available host and avoiding collision in assignment attempts).



(a) APSR (actual decline ratio is 0.01%).



(b) APSR_{avg} (actual decline ratio is 1.6%).

Fig. 2. Cloud resource utilization and the number of schedulers allowed by APSR (requests have finite lifetime).

Fig. 2 shows similar results for finite-duration requests, where requests arrival is made bursty by following an MMPP process. Results are presented for a highly conservative estimate of resource availability (APSR) and a more relaxed estimate of resource availability (APSR_{avg}). We note that for the less conservative estimate, the algorithm has a higher level of parallelism, and attains higher throughput (it finishes handling requests earlier). This comes at the cost of a negligible degradation in the decline ratio, well within the SLA constraint.

IV. DISCUSSION, CONCLUSIONS AND FUTURE WORK

We require fast placement of virtual machines to realize the NFV vision of rapidly deploying service chains, and auto-scaling their capacity. However, the existing cloud infrastructure is not fast enough to cope with bursts of parallel placement requests in a timely manner. Our APSR algorithm efficiently implements random placement while minimizing the communication overhead, and dynamically adjusts the degree of parallelism to ensure that decline ratios are kept at their SLA.

REFERENCES

[1] Einziger et al. Faster placement of virtual machines through adaptive caching. In *INFOCOM*, 2019.

[2] OpenStack compute schedulers, 2018. https://docs.openstack.org/.

[3] Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[4] Schwarzkopf et al. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS EuroSys*, 2013.