On Non-Preemptive VM Scheduling in the Cloud

Konstantinos Psychas, and Javad Ghaderi
Columbia University

ABSTRACT

We study the problem of scheduling VMs (Virtual Machines) in a distributed server platform, motivated by cloud computing applications. The VMs arrive dynamically over time to the system, and require a certain amount of resources (e.g. memory, CPU, etc) for the duration of their service. To avoid costly preemptions, we consider non-preemptive scheduling: Each VM has to be assigned to a server which has enough residual capacity to accommodate it, and once a VM is assigned to a server, its service *cannot* be disrupted (preempted). Prior approaches to this problem either have high complexity, require synchronization among the servers, or yield queue sizes/delays which are excessively large. We propose a non-preemptive scheduling algorithm that resolves these issues. In general, given an approximation algorithm to Knapsack with approximation ratio r, our scheduling algorithm can provide $r\beta$ fraction of the throughput region for $\beta < r$. In the special case of a greedy approximation algorithm to Knapsack, we further show that this condition can be relaxed to $\beta < 1$. The parameters β and r can be tuned to provide a tradeoff between achievable throughput, delay, and computational complexity of the scheduling algorithm. Finally extensive simulation results using both synthetic and real traffic traces are presented to verify the performance of our algorithm.

KEYWORDS

Scheduling Algorithms, Stability, Queues, Knapsack Problem, Cloud

ACM Reference Format:
Konstantinos Psychas, and Javad Ghaderi. 2018. On Non-Preemptive VM Scheduling in the Cloud. In *SIGMETRICS'18 Abstracts: ACM SIGMETRICS International Conference on Measurement & Modeling of Computer Systems Abstracts, June 18–22, 2018, Irvine, CA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3219617.3219644

1 INTRODUCTION

There has been an enormous momentum recently in moving storage, computing, and various services to the cloud. By using cloud, clients no longer require to install and maintain their own infrastructure and can instead use massive cloud computing resources on demand. Clients can procure Virtual Machines (VMs) with specific configurations of CPU, memory, disk, and networking in the cloud. In a more complex scenario, clients can put together an entire service by procuring and composing VMs with specific capabilities.

This work was supported by NSF Grant CNS-1652115.

SIGMETRICS'18 Abstracts, June 18–22, 2018, Irvine, CA, USA

ACM ISBN 978-1-4503-5846-0/18/06.
https://doi.org/10.1145/3219617.3219644

In this paper, we consider a distributed server platform, consisting of possibly a large number of servers. The servers could be inhomogeneous in terms of their capacity (e.g. CPU, memory, storage). As an abstraction in our model, VM is simply a multi-dimensional object (vector of resource requirements) that *cannot* be fragmented among the servers.

We are interested in scalable non-preemptive scheduling algorithms that can provide high throughput and low delay. To maintain scalability, we would like the scheduling decisions to be made by the servers individually in a distributed manner. In this work, we propose an algorithm to meet these objectives and we characterize its theoretical performance.

2 SYSTEM MODEL

Cloud Cluster and VM-based Job Model. We consider a collection of L servers denoted by the set $\mathcal{L}$. Each server $\ell \in \mathcal{L}$ has a limited capacity for various resource types (e.g., memory, CPU, storage). We assume there are R different types of resources. There is also a collection of J VM types denoted by the set $\mathcal{J}$. Each VM type $j \in \mathcal{J}$ requires fixed amounts of the various resources. So each VM type is a R-dimensional vector of resource requirements.

Job Arrivals and Service Times. We assume VMs of type j arrive according to a Poisson process with rate λ_j. Each VM must be placed in a server that has enough remaining resources to accommodate it. Once a VM of type j is placed in server, it departs after an exponentially distributed amount of time (service time) with mean $1/\mu_j$, independently of the other existing VMs in the server.

Server Configuration and System Configuration. We denote by k_j^ℓ the number of type-j VMs that are accommodated by server ℓ. For each server ℓ, a vector $\mathbf{k}^\ell = (k_1^\ell, \cdots, k_J^\ell) \in \mathbb{N}_0^J$ is said to be a feasible configuration if the server can simultaneously accommodate k_1^ℓ type-1 VMs, k_2^ℓ type-2 VMs, ..., k_J^ℓ type-J VMs, without violating its capacity. We use $\mathcal{K}_\ell$ to denote the set of all feasible configurations for server ℓ excluding the 0-configuration $\mathbf{0}_J$, and $\bar{\mathcal{K}}_\ell$ to denote $\mathcal{K}_\ell \cup \{\mathbf{0}_J\}$.

Queueing Dynamics and Stability. When VMs arrive, they are queued and later served by the servers. We use $Q_j(t)$ to denote the number of type-j VMs waiting in the queue to get service at time t. The respective vector of all queue sizes is denoted by $\mathbf{Q}(t)$. The system is said to be stable if the expected size of queues is bounded as time goes to infinity. A vector of arriving rates $\boldsymbol{\lambda}$ and a vector of mean service times $1/\boldsymbol{\mu}$ are said to be supportable if there exists a scheduling algorithm under which the system is stable. Let $\rho_j = \lambda_j/\mu_j$ be the workload of type-j VMs. We will define the throughout region of the cluster as

$$C = \{\mathbf{x} \in \mathbb{R}_+^J : \mathbf{x} = \sum_{\ell \in \mathcal{L}} \mathbf{x}^\ell,\ \mathbf{x}^\ell \in Conv(\bar{\mathcal{K}}^\ell), \ell \in \mathcal{L}\}, \qquad (1)$$

where $Conv(\cdot)$ is the convex hull operator. It has been shown that the set of supportable workloads is the interior of C.

3 ALGORITHM

Before describing the algorithm, we make two definitions.

Definition 1 (weight of a configuration). The weight of configuration $\mathbf{k}^\ell$ for server ℓ, given a queue size vector $\mathbf{Q}$, is defined as

$$f(\mathbf{k}^\ell, \mathbf{Q}) := \sum_{j \in \mathcal{J}} Q_j k_j^\ell. \tag{2}$$

Definition 2 (r-max weight configuration). Given a constant $r \in (0, 1]$, and a queue size vector $\mathbf{Q}$, an r-max weight configuration for server ℓ is a feasible configuration $\mathbf{k}^{(r)\ell} \in \mathcal{K}_\ell$ such that

$$f(\mathbf{k}^{(r)\ell}, \mathbf{Q}) \geq r f(\mathbf{k}^\ell, \mathbf{Q}),\ \forall \mathbf{k}^\ell \in \mathcal{K}_\ell. \tag{3}$$

Under our scheduling algorithm, each server at any time is either in an *active* period or in a *stalled* period. We will also refer to the state of a server as active or stalled depending on the period in which it is at a certain time.

Active period: In this period, the server schedules VMs from the queues according to a *fixed* configuration. Formally, let the configuration of server ℓ in an active period be $\tilde{\mathbf{k}}^\ell = (\tilde{k}_j^\ell : j \in \mathcal{J})$. The server can contain at most $\tilde{k}_j^\ell$ VMs of type j, $j \in \mathcal{J}$, at any time. If there are not enough type-j VMs in the system, the server reserves the remaining *empty slots* for future type-j arrivals.

Stalled period: In this period, the server does not schedule any more VMs, even if there are VMs waiting for service that can fit in the server. It only processes VMs which already exist in the server. The stalled period ends when all the existing VMs in the server finish their service and leave, at which point the server will enter a new active period.

Note that by the above definitions, an arriving VM of type j will not be queued (i.e., it enters the queue but immediately gets service) if there is an *empty slot* available for it in any of the active servers as it will be scheduled in one of the empty slots immediately. Also the change of configuration in a server can only happen when the server is empty and stalled and that change results in a transition from a stalled period to an active period.

Our scheduling algorithm described in Algorithm 1 determines the following:

(1) **Transition from active to stalled**. Suppose server ℓ is in an active period with configuration $\tilde{\mathbf{k}}^\ell$. The server makes a transition to a stalled period if upon departure of a VM from the server at time t,

$$f(\tilde{\mathbf{k}}^\ell, \mathbf{Q}(t)) < \beta f(\mathbf{k}^{(r)\ell}(t), \mathbf{Q}(t)), \tag{4}$$

where $\mathbf{k}^{(r)\ell}(t)$ is an r-max configuration, given the queue size vector $\mathbf{Q}(t)$, and $0 < \beta < 1$ is a constant parameter of the algorithm.

(2) **Transition from stalled to active.** Suppose a server is in a stalled period. When the server becomes empty the server makes a transition to an active period.

(3) **Server configuration during an active period**. Suppose server ℓ enters an active period at time $t_{(a)}$. The configuration of server ℓ for the entire duration of its active period, $\tilde{\mathbf{k}}^\ell$, is *fixed* and set to $\mathbf{k}^{(r)\ell}(t_{(a)})$, an r-max weight configuration based on the queues at time $t_{(a)}$.

Algorithm 1 Basic Non-preemptive Scheduling

When a VM of type j arrives at time t:

Add the VM to the queue j
if exists *empty slots* for type-j VMs **then**
  Schedule the VM in the first empty slot.
end if

When a VM of type j in server ℓ is completed at time t:

if ℓ is *active* with configuration $\tilde{\mathbf{k}}^\ell$ **then**
  if condition (4) holds **then**
    Switch ℓ to *stalled*.
  else
    Schedule a type-j VM in server ℓ from queue j. If queue j is empty, register an *empty slot* of type j in server ℓ.
  end if
end if
if ℓ is empty and *stalled* **then**
  Switch ℓ to *active*.
  Find an r-max weight configuration $\mathbf{k}^{(r)\ell}$.
  Set the configuration of server ℓ during its active period to be fixed and equal to $\mathbf{k}^{(r)\ell}$.
  for $j \in \mathcal{J}$ **do**
    Schedule $k_j{}^{(r)\ell}$ VMs of type j in server ℓ. If there are not enough VMs in queue j, register an *empty slot* for each unused slot.
  end for
end if

The following theorem states the main result about the performance of the algorithm.

Theorem 3.1. *Consider Algorithm 1 with parameter $r \in (0, 1]$ and $0 < \beta < r$. Then the algorithm can support any workload vector $\boldsymbol{\rho}$ in the interior of $\mathcal{C}_{r\beta}$ ($r\beta$-fraction of the capacity region C).*

Proofs and extensions to the theorem as well as algorithms that find an r-max weight configuration for different values of r are in full version of the paper [3].

4 SIMULATIONS

In this section, we verify our theoretical results and also compare the performance of our algorithm with two other algorithms, the randomized sampling algorithm [1] and the MaxWeight at local refresh times [2], which will refer to them as G16 and M14 respectively.

4.1 Inefficiency of other algorithms

Example 1 (Instability of M14: MaxWeight based on local refresh times). Consider one server with capacity 6 units and two VM types, type-1 VMs require 4 units and type-2 VMs require 1 units. Service rates are the same for both VMs and arrival rate of the small VM type is 8 times higher than the large one. The workload vector is chosen to be $0.89 \times (0.5, 4)$, which is supportable. When the server starts scheduling according to configuration $(1, 2)$, the arrival

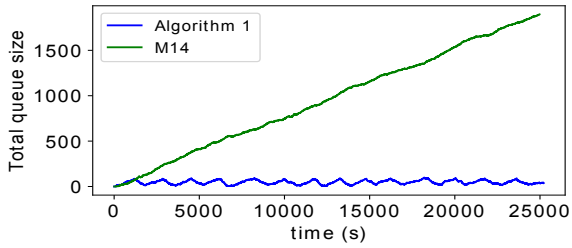

Figure 1: M14 fails in Example 1 while Algorithm 1 still stabilizes the queues.

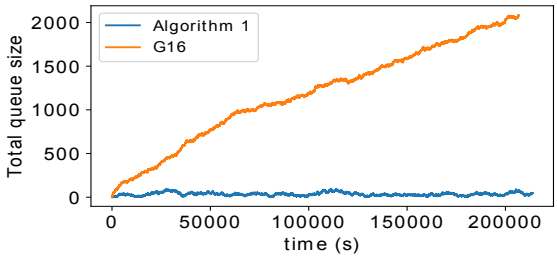

Figure 2: G16 performs poorly in Example 2 although it theoretically converges. Algorithm 1 performs much better.

rate of small VMs will be higher than their service rate. That will result in the queue of small VMs to grow and configuration never resets with a non-zero probability. Figure 1 depicts the total queue size (sum of the queue sizes) under our algorithm and M14. The sawtooth behavior under our algorithm indicates the configuration reset times.

Example 2 (Large queue size under G16: Randomized sampling). In the second example we show that although G16 guarantees stability it can yield large queue sizes. Consider a setting of 4 different servers with 1, 2, 4, 8 resource units and 4 types of VMs with resource requirements 1, 2, 4, 8. Arrival and service rates are the same for all VMs and traffic load is 0.89. Figure 2 depicts the total queue size under G16 and our algorithm.

4.2 Experiment with Google trace dataset

In this experiment, we use a real traffic trace from a large Google cluster. From the original dataset, we extracted the arrival times of tasks and their service times by taking the difference of the deployment time and the completion time.

Resource requirements involve two resources (CPU and memory) and are collected once a VM is submitted. The resources are not treated as discrete; their range in the original dataset is normalized to have a maximum of 1. To map the VMs to a tractable number of types, we took the maximum out of the two resources and rounded it up to the closest integer power of $1/2$. All tasks that are mapped to the same power of two are considered to belong to the same type. The highest power of $1/2$ considered was 7, since lower valued VMs account for less than 1% of requests.

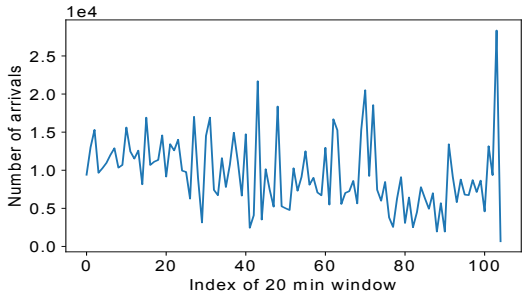

Figure 3: Number of arrivals over time in the Google trace, computed over 20-minute time windows.

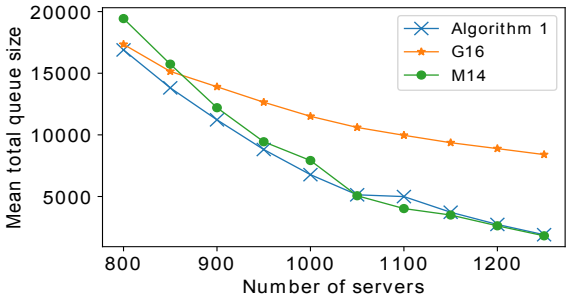

Figure 4: The performance of different algorithms under the Google trace, for different number of servers.

In actual trace the number of servers changes dynamically with servers being added, removed or modified. To keep things simpler we assumed that the sizes of all servers are all 1 which is the maximum possible and their number is fixed throughout a run.

In simulations we work with a window of 1 million arrivals. The traffic intensity for that part of trace is depicted in Figure 3. We evaluate the performance of all the algorithms using the above trace and for different number of servers that ranges from 800 to 1250. The change in the number of servers implicitly controls the traffic intensity. All runs were repeated 3 times and the reported results which appear in Figure 4 are the average of these runs. As we can see, our algorithm has the best overall performance in the whole range of the number of servers. The performance of G16 deteriorates as the number of servers scales up, while the performance of M14 deteriorates as the number of servers scales down.

REFERENCES

[1] Javad Ghaderi. 2016. Randomized algorithms for scheduling VMs in the cloud. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9. https://doi.org/10.1109/INFOCOM.2016.7524536

[2] Siva Theja Maguluri and R Srikant. 2014. Scheduling jobs with unknown duration in clouds. *IEEE/ACM Transactions on Networking* 22, 6 (2014), 1938–1951.

[3] Konstantinos Psychas and Javad Ghaderi. 2017. On Non-Preemptive VM Scheduling in the Cloud. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 2, Article 35 (Dec. 2017), 29 pages. https://doi.org/10.1145/3154493