## Abstract

We examine the placement of virtual machines in an OpenStack deployment, and explore possible algorithms to optimize the utilization of the infrastructure.

## OpenStack Instance Placement

In OpenStack, a *compute host* is a server on which virtual machines are scheduled. In response to a user-request, the Nova service invokes a scheduler that looks across a fleet of compute hosts and determines where to place the requested virtual machine. *Affinity* and *anti-affinity* policies further allow for the specification of placement of virtual machines with respect to other virtual machines. Scheduling is based on available resources on the compute host. OpenStack tracks and schedules VMs based on vCPU, memory, and local disk requirements. Over-subscription is a mechanism whereby a single instance of a resource on a compute host could be allocated to multiple virtual machines. Over-subscription of a resource (vCPU, memory, and disk) are defined on for each compute host.

Requests are not known ahead of time. The order in which requests are received is not deterministic. This is because requests are received when an application is brought onto the system. The OpenStack cluster is a multi-tenant system with a number of applications coexisting on a set of shared compute hosts. In the most general case of the problem, an OpenStack cluster has many compute hosts, and many applications which come on to the platform over a period of time.

## The Perfect Scheduler

We now define a useful construct called the “perfect scheduler”.

Consider the set of applications which are all going to be (over time) brought onto an OpenStack cluster. Assume that it known in advance what virtual machines each of them will be requesting, what resources are required for each of them, what affinity and anti-affinity rules will be specified, and what oversubscription is being used on each compute host, for each resource. Assume further that there are a set of possible virtual machine placements across the compute hosts that will allow all requests to be satisfied with the minimum number of compute hosts used. Let us call this set {***S***}.

The “perfect scheduler” is an algorithm that will receive and process requests for placement of virtual machines with no knowledge of what requests will be received in the future. The Perfect Scheduler places virtual machines on the compute hosts in such a way that when all requests have been received and processed, the state of the cluster ***s*** is in ***S*** (i.e. ***s*** ε ***S***).

## Similarity to known problems

This problem is tantalizingly similar to the traditional “bin packing” problem which attempts to pack a number of fixed sized objects into a collection of bins of fixed (and identical) size so as to occupy the least volume in total. The solution to this problem is known to be NP-Hard[[1]](#footnote-1).

Another similar class of problems is the “0/1 knapsack problem” which attempts to maximize the weight of objects of different weights placed into a knapsack, subject to some upper bound. This problem is also known to be NP-Hard[[2]](#footnote-2).

This problem appears to resemble the issue of stacking blocks in the game of Tetris. The algorithms for packing in Tetris are well understood and the greedy (lamebrain) algorithm is known to work quite well.

However, the problem at hand is not like any of these problems in some important respects.

First, each placement request must be satisfied with incomplete information – what requests will follow, and the order in which they will follow is not known. In this regard, the problem at hand is like the Tetris problem.

Second, in the traditional bin-packing problem in 3-space, every unit of space in the bin may be used. In the VM packing problem this is not the case. Consider a vCPU that is defined to be subscribed at a ratio of N:1. That means that at most N virtual machines may be allocated to each vCPU. Without loss of generality, we can reduce any N:1 oversubscription into a 1:1 subscription model by merely assuming that the compute host has N vCPUs for each vCPU actually present in the compute host. In other words, a compute host with 32 virtual cores and subscribed as 3:1 can be modeled as a compute host with 96 vCPUs where each vCPU can only be assigned to a single virtual machine.

Third, in Tetris, when the lowest row is solid, it gets consumed and the whole board moves down. There is no equivalent in the virtual machine packing problem.

## Simplified graphical representation

As described in the preceding section, the virtual machine placement problem is not the traditional bin-packing problem, but a similar graphical representation can be created.

|  |  |
| --- | --- |
|  |  |

Figure 1. The traditional bin-packing problem (at left) and the Virtual Machine packing problem (at right).

On the left, the traditional bin-packing problem in 2-space. We show a total of 8 objects packed into a grid of 10x8. On the right, the VM packing problem showing four VMs occupying the same 10x8 grid. In this representation, we have reduced all resources to their 1:1 subscribed equivalent. Without loss of generality this representation can be extended into N-space.

## Problem Statement [#1]

A system consists of a number of boxes. At the outset, all boxes are empty.

Over a period of time, two kinds of requests are received by the system. The first kind of request is a placement. The second kind of request is a removal request.

In a placement request, numbered objects of different sizes are received, and these objects are placed into boxes. All placements are final, and an object once placed in a box shall remain in that box[[3]](#footnote-3).

A maximum size is defined for each box. The sum of the size of all objects in a box shall not exceed this maximum size.

Some boxes are grouped into an *aggregate*. A *default aggregate* is also defined. On arrival, some objects have an intended aggregate. If no aggregate is specified, then the default aggregate is used.

Some placement requests have an associated placement policy. A placement policy is used to determine which box the object must be placed in. Three policies are defined, a *NULL* policy, an *affinity* policy, and an *anti-affinity* policy. In the affinity policy, an object must be placed in the same box as some previous object. In the anti-affinity policy an object must be placed in a box such that it is not in the same box as some other previous objects which had the same policy. In the NULL policy, there are no restrictions (other than total size of the box) on which box the object may be placed.

The second kind of request is the removal request. A removal request indicates the number of the object that must be removed from the system. In response to a removal request, the system will locate the numbered object and remove it from the box.

The receipt and execution of any request is treated as a single atomic operation. When a first request is received before a second request, the first request shall be completely executed before the second request is considered. The processing of the first request may not depend on any information related to the second request. When two requests are received at exactly the same time, the system is free to consider them for execution in any order.

Each object placement request needs to be considered based on the state of the system at the time, without any knowledge of future requests.

## Algorithm [#1]

Assume that box there are N boxes, Bi where each[[4]](#footnote-4) has a maximum allowed size of Mi. The total size of all objects currently in Bi is Ti. A request for placement of object Oj is received by the system, and the size of object Oj is Wj. We place Oj in the fullest box into which it can be placed.

|  |
| --- |
| LET L = 0, LB = 0  FOR i = 1 TO N:  // can we place Oj in Bi?  IF Wj + Ti ≤ Mi:  // is this the first feasible box  // (L == 0), or is this box fuller  // than a previous feasible box  // (LB < Ti)? If yes, choose this  // box.  IF L == 0 OR LB < Ti:  // choose this box  L = i  // update the fullest size  LB = Ti  END-IF  END-FOR |

Algorithm . Identifying placement based on fullest box

At the end of this L indicates the box into which the object Oj is to be placed. If L is 0 then no suitable box has been found, and the placement request cannot be satisfied. If L is non-zero, then we place Oj in box L.

This simplified algorithm presented above does not consider either the aggregate, or the placement policy and without loss of generality, we can extend this algorithm to consider those two things.

## Problem Statement [#2]

We now extend the earlier problem statement to reflect the fact that a compute host contains multiple resources, all of which are limited.

Therefore, each object (Oj) has a number of resource requirements <W(j,1), W(j,2), … W(j,R)>. Currently the resources consumed in each box Bi are <T(i,1), T(i,2), … T(i,R)>, and their associated limits are <M(i,1), M(i,2), … M(i,R)>.

## Algorithm [#2]

We update the earlier algorithm as shown below. We use a slightly modified mechanism to determine the ‘fullness’ of a compute host.

// Compute the fullness of a box as the

// linear combination of total sizes with

// an array of relative weights ()

DEF FULLNESS(Bi):

RETURN

// check whether sufficient amount of

// each resource is available in the box

DEF FITS (Oj, Bi):

FOR r = 1 to R:

IF W*(j,r)* + T*(i,r)* > M*(i,r)*:

RETURN FALSE

END-IF

END-FOR

RETURN TRUE

// ----------------------------

// the updated algorithm

// ----------------------------

LET L = 0, LB = 0

FOR i = 1 TO N:

// can we place Oj in Bi?

IF FITS (Oj, Bi):

IF L == 0 OR LB < FULLNESS(Bi):

// choose this box

L = i

// update the fullest size

LB = FULLNESS(Bi)

END-IF

END-FOR

Algorithm . Identifying placement based on fullest box and weighted sizes

1. Determining whether or not a solution exists is known to be NP-complete. [↑](#footnote-ref-1)
2. Determining whether or not a solution exists is known to be NP-complete. [↑](#footnote-ref-2)
3. Till such time that a removal request is received by the system [↑](#footnote-ref-3)
4. Assume that the indices for boxes are 1 based, 1 ≤ i ≤ N. We use 0 to indicate an invalid index [↑](#footnote-ref-4)