Multi-Dimensional Scheduling in Cloud Storage Systems

Zhihao Yao
Computer and Information Technology
Purdue University
West Lafayette, Indiana 47906
Email: yao86@purdue.edu

Ioannis Papapanagiotou
Computer and Information Technology
Purdue University
West Lafayette, Indiana 47906
Email: ipapapan@purdue.edu

Robert D. Callaway
Cloud Solutions Group
NetApp, Inc.
RTP, North Carolina 27709
Email: bob.callaway@netapp.com

***Abstract*—The increasing demand for elastic and scalable cloud block storage requires flexible and efficient ways to provision volumes. The scheduling of volume requests in physical storage nodes or virtualized storage pools is usually based on a single criterion, such as the available capacity or the number of volumes per backend. Those properties are exposed to the cloud block storage scheduler through drivers, and may vary based on the workload. Hence, most cloud storage providers refrain from describing Service Level Objectives (SLOs). In this paper, we present the design and implementation of a new scheduling algorithm for block storage systems that has the following advantages over the currently implemented scheduler in OpenStack. It provides guaranteed SLOs even in a dynamic workload, it increases the I/O throughput of the volumes that have been already provisioned in the backend systems, it can be scalable to a higher arrival rate for the volume requests, and finally it can minimize the number of active hosts (or else the energy consumption). The volume placement process is based on an APX-hard multi-dimensional Vector Bin Packing (VBP_d) algorithm. In order to reduce the complexity we propose a heuristic named Modified Vector Best Fit Decreasing (MVBFD). Our scheduler design for block storage systems is based on the principles of the OpenStack's Cinder scheduler; hence it can be deployed with only minor modifications to an OpenStack block storage deployment.**

Keywords*—*Cloud Storage, Infrastructure as a Service, Resource Scheduling, Vector Bin Packing

I. Introduction

Cloud computing has gained immerse popularity as a paradigm for the dynamic provisioning of computing services deployed on virtual machines (VM). The cloud infrastructure, otherwise called infrastructure as a service (IaaS), allows high flexibility, availability, reduction of hardware cost, and operational expenses. At the same time, the cloud can store and process vast amount of data generated by the tenants, hence being useful for a number of industries e.g. healthcare [1]. In cloud storage there are three main categories on how the data can be stored: (a) *ephemeral storage*: data is stored on the provisioned VMs and is lost after the reboot or the release of that VM; (b) *block storage*: data is stored in a persistent block-level storage space (i.e. volume) which is able to be attached or detached to a VM with a lifetime that is separate from the VM itself. Typical utilized capacity is in the low TBs; (c) *object storage*: objects and files are written to multiple disk drives spread throughout servers in the data center and can be accessed through service API from all cloud Services. A typical usage includes 10s of TBs of data storage.

As applications and databases deployed on the cloud grow over time, there is an eventual need to increase the storage capacity. Adding physical storage or upgrading to a storage system with higher capacity requires high capital investment and cannot be scalable. Cloud block storage offers a reliable, high-performance, on-demand storage for applications hosted on the cloud. In addition, it can support resource-hungry workloads seamlessly, and can scale very fast by just asking for more space, i.e. a volume extend request. However, the problem of scheduling storage requests in a manner that maximizes the use of resources, minimizes the energy consumption and maintains the Service Level Agreements (SLA) between the tenant and the cloud service provider has not been properly studied. This problem becomes even more relevant, as new open source IaaS solutions, like OpenStack [2], are being deployed in commercial practice by more than 200 companies.

A great amount of work has been devoted in optimizing management of a data center. Most studies have focused on the optimal VM placement in data centers [3], or energy aware VM allocation [4]. Some more recent studies have focused on adding SLA during the VM placement process [5], or optimizing the job completion time in big data clusters [6]. In our recent work [7], we proposed an SLA-aware resource scheduling for cloud storage, assuming one request per unit time. Hence, the problem could be solved with relatively simple scheduling operations. The main objective of this work is to present our vision, and discuss the research challenges in the volume allocation in cloud storage systems that are comprised of multiple storage entities and multiple requests on a unit time. We develop a dynamic scheduling algorithm that can provide efficient and rapid allocation of volume requests, and minimize the number of backend systems in the cloud storage infrastructure. More specifically, we design a scheduler based on multi-dimensional Vector Bin Packing (VBP_d), where d is the number of resources that the scheduler can use, such as capacity, I/O through etc. The problem is equivalent to given balls and bins with sizes $\mathbb{R}^d$, multi-dimensional bin packing assigns the balls to the fewest number of bins. This is done to maximize the volume allocation, and minimize the number of used backend systems. While the problem is APX-hard [8], which means that there is no asymptotic polynomial-time approximation scheme (PTAS) for the problem, unless $P = NP$. Several heuristics can be adopted to solve this problem [9]. However, these heuristics consider balls of a fixed

size and therefore the problem of volume request allocation with multiple sizes is totally different. At the same time, to the author's knowledge, such ideas have not been applied in cloud storage systems. In summary, our work aims to:

- Develop a scheduling framework that is modular enough and can be integrated in cloud infrastructure platform for block storage;
- Schedule volume request to backend system, or a pool of scheduling resources, considering multi-dimension property including available space and I/O throughput;
- Manage the performance of block storage infrastructure in terms of utilization, energy and SLO maintenance.

The paper is organized as follows: In Section II, we present the background information and the OpenStack scheduling process for block storage. In Section III, we present new scheduling approaches, and in Section IV, we perform a simulation analysis. Finally, in Section V, we conclude our work, and present our future directions.

II. Block Storage in Cloud

1) Background on block storage SLAs: Block storage aims to provide reliable, high-performance, on-demand and persistent storage capacity for various cloud applications. In general, the storage capacity can be accessed in the form of virtualized logical volumes. The block storage system manages the creation, attaching and detaching of these volumes to VMs which are usually provided by the computing service. To deploy block storage service in cloud infrastructure requires building a dedicated storage server cluster. A storage node or backend system often consists of a number of hard drives in the form of RAID to provide redundancy and massive storage capacity. In some cases, Solid State Drive (SSD) or customized SSD [10] are used to meet the critical requirement of I/O performance. The storage service can be accessed via a predefined service API so that the cloud tenant is able to send request to create, attach, detach or migrate the volume on demand.

The block storage service on the cloud also needs to meet the needs of customers. This is guaranteed from a legal document, i.e. the SLA. The SLA is the entire agreement that specifies what service is to be provided, how it is supported, times, locations, costs, performance, and responsibilities of the parties involved. The level of each service in the SLA is described as an SLO. SLOs are specific measurable characteristics of the SLA such as availability, throughput, frequency, response time, or latency. In the occasions that the block storage provider, fails to achieve the SLA, a penalty is applied on the income. In the cloud storage domain, there are generally two kinds of SLOs: availability SLO and performance SLO. The availability SLO specifies the level that the cloud service must be accessed successfully during the contract period or the period defined by the provider. For instance, in Amazon's Elastic Block Store (EBS) [11], there is a guaranteed SLA of a monthly up-time performance at least 99.9%. Similarly, other cloud storage providers have set really high standards on the provided availability SLO. The performance SLO limits the lowest bound of data transition speed on storage platform.

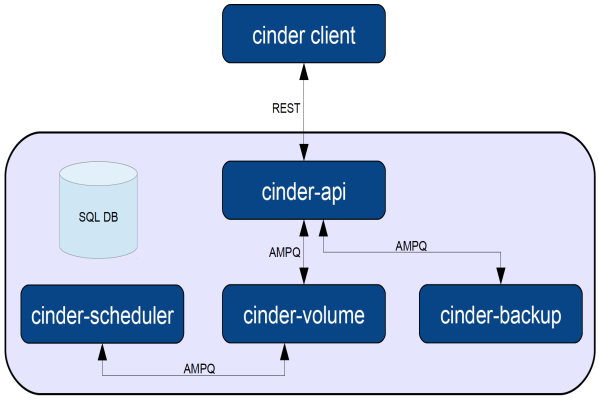

Fig. 1. System Components of OpenStack Cinder's Service

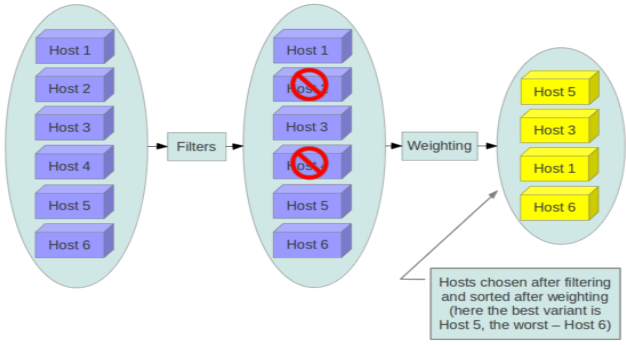

Fig. 2. Filtering and Weighting in Cinder's Scheduling

The most common unit of performance SLO in block storage is Input/Output Operations Per Second (IOPS). Compared to the availability SLO, performance SLO is hardly ever disclosed by any cloud block storage provider. In some occasions, the cloud storage providers may claim that their service is able to deliver a consistent baseline of X IOPS/GB without disclosing a specific SLO in their SLA.

2) OpenStack Cinder: As the most popular open source cloud infrastructure framework, OpenStack is collaborating developers and cloud computing technologists producing the ubiquitous open source framework for public and private clouds. The community and more than 200 companies have joined the development of OpenStack. The project aims to deliver solutions for all types of clouds by being simple to implement, massively scalable, and feature rich. The technology consists of a series of interrelated projects/services delivering various components for a cloud infrastructure solution [2].

In OpenStack, the block storage service is named Cinder. Cinder virtualizes pools of block storage devices and provides end users with a self service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device. An overview of the Cinder architecture is shown in Fig.1. The cinder-scheduler module is responsible for scheduling and routes requests to the appropriate volume service running on storage entity.

Fig.2 shows the scheduling in OpenStack. It works in

two steps: (i) filtering and (ii) weighting. When the Cinder Filter Scheduler receives a request for a logical volume, it first applies filters to determine which hosts are eligible for consideration when dispatching a request. The filter's decision is binary: either a host is accepted by the filter, or it is rejected. It is based on capabilities of the host, namely free space, location and I/O throughput. The default scheduling policy in Cinder is the capacity scheduler. Once the scheduler receives a volume request, it filters out the hosts which do not have sufficient capacity and create a list of eligible hosts. Then the weight of each host is calculated based on the available capacity. The scheduler selects the host with the largest available capacity as the best candidate to serve the request.

A pool is a logical concept that describes a set of storage resources that can be used to serve core Cinder requests, e.g. volumes/snapshots. This notion is almost identical to Cinder volume backend or storage node from the Cinder scheduling perspective, for it has similar attributes (capacity, capability). The main difference is that a pool cannot exist on its own; it must reside in a volume backend. One volume backend can have multiple pools but pools cannot expose sub-pools to Cinder (meaning even they have, sub-Pools do not get exposed to Cinder, yet). The workflow of this recent change is as follows: 1) Volume backends report how many pools and what those pools look like and are capable of to the scheduler; 2) when a request comes in, the scheduler picks the most appropriate pool to serve the request; it passes the request to the backend where the target pool resides; 3) the volume driver gets the message and lets the target pool serve the request as the scheduler instructed. Each pool is also a separate schedulable entity, which means the schedulable object from the Cinder scheduler's perspective is no longer limited to a physical machine. To accommodate this feature, the scheduler has to have great scalability to schedule up to thousands of storage entities.

The Cinder scheduler plays a crucial role in the resource management of block storage services. Nonetheless, there are a number of limitations in the current scheduling approach. First, the default capacity-based scheduling algorithm has not been evaluated in terms of performance. For example, there might be cases in which the storage nodes can be overloaded by many read and write I/O operations, therefore their ability to serve future demands may be limited. The scheduler unaware of this status may schedule a volume request to be served by the entity with more available capacity, but overloaded in terms of the I/O operations. Moreover, the current approach does not consider the case in which competing volume requests arrive in the same time slot. Hence, if the request arrival rate is high, it may lead to suboptimal allocation, SLO violation, higher energy usage and therefore higher operating expenses. Third, the scalability of the current implementation has not been investigated. The scheduler needs to deliver the capability of fast scheduling, cost efficient balancing, and scalability to the cloud storage platform.

III. Optimization Problem And Solution

The goal of the resource allocation is to map volume requests, where each volume represents a tuple containing several dimensions, into a number of storage nodes or pools. In our formulation, we consider each storage entity, as a bin, the dimensions, as its properties, and the goal is to maximize the utilization of a single entity and minimize the number of entities that must be used to serve all the volume requests, respecting the properties and capabilities of the storage entities. This problem is NP-complete [12] and can be solved through Linear Programming (LP) or heuristics.

A. Problem Description

In order to create the LP formulation of the volume request allocation problem, we define two variables: (i) $y_k \in \{0,1\}$, equals to 1 if a scheduling entity (storage node or pool) $k \in K$ is used, and 0 otherwise; (ii) $x_{kv} \in \{0,1\}$, equals to 1 if the volume $v \in V$ is provisioned to a storage entity $k \in K$, 0 otherwise. Hence the objective function can be expressed as follows:

$$\begin{aligned} &\text{min} && \sum_{k \in K} y_k && \\ &\text{subject to:} && \sum_{k \in K} x_{kv} = 1 && \forall v \in V \\ &\text{and} && \sum_{v \in V} u_{kd} x_{kv} \leq c_{kd} y_k && \forall d \in D, \forall k \in K \end{aligned}$$

where u_{kd} is the utilization of dimension d on a node k and c_{kd} is the maximum available resources of dimension d. The objective function aims at minimizing the number of required storage entities. The constraints with the c_{kd} guarantee that each volume request demand does not exceed the appropriate capacity on each dimensions d.

B. Heuristics

The volume placement process can be treated as a Vector Bin Packing Problem (VBP_d), where the d-dimension represents the capabilities of the storage entity across different dimensions, e.g. CPU and IOPS, memory and disk usage, network throughput, etc. These capabilities are measured at the current (or last) time epoch t and exposed to the scheduler. Therefore, the assignment should ensure that the number of nodes is minimized while no capability constraint (i.e. SLO) is violated. Let us assume V volumes $\{I^1, I^2, ..., I^V\}$, where each $I^i \in \mathbb{R}^d$. A valid packing is a partition of the V volumes in K sets (or storage entities) $B_1, ..., B_K$, where for each storage entity j and for each dimension i, $\sum_{l \in B_j} I_i^l \leq 1$. In our problem $d \geq 2$, the vector bin packing problem is known to be APX-hard, which means that there is no asymptotic polynomial-time approximation scheme (PTAS) for the problem, unless $P = NP$ [8].

In reality, the volume requests arrive one at a time in unknown order and the scheduler has to make the decision immediately. The optimization goal is to minimize the number of storage entities used. The simplest approach would be to allocate the volumes to the fullest entity that still has enough resources based on the requested properties. One can naturally observe that this would result to a greedy algorithm that follows the principles of the online bin packing problem. This algorithm is often referred to as *Standard Best Fit* (BF) [13]. BF is guaranteed to find an allocation with at most $\frac{17}{10} \cdot OPT + 1$ entities (where OPT is the number of bins given the optimal solution) [14]. To adopt this heuristic in the

volume allocation scenario, we transform the problem to an offline version of BF, *Best Fit Decreasing* (BFD), and propose a Modified Vector BFD (MVBFD) algorithm that take into account the aforementioned higher dimensions. The dilemma that online bin packing algorithm usually have is it is difficult to pack large items properly, especially if they arrive late in the sequence. This situation can be circumvented by pre-sorting the input sequence or sub-sequence and packing the large item first, which is the offline BFD algorithm. BFD is able to achieve at most $\frac{11}{9} \cdot OPT + 4$ nodes [15]. In our implementation, the MVBFD is applied by changing the way of triggering scheduling algorithm. The MVBFD algorithm no longer runs based on first come first serve model. It is triggered periodically. The volume requests that arrive in a short period are stored in a message queue, i.e. Advanced Message Queuing Protocol. The MVBFD is able to sort the requests in decreasing order before allocating them.

Algorithm 1 Modified Vector Best Fit Decreasing (MVBFD)

Input: hostList, volumeList
Output: : Volume Request Allocation
  for all v in volumeList **do**
    calculate w(I)
  end for
  for all b in hostList **do**
    calculate w(B)
  end for
  volumeList.sortDecreasing(w(I))
  hostList.sortDecreasing(w(B))
  for all v in volumeList **do**
    for all b in hostList **do**
      $candidateHost \leftarrow NULL$
      for all d in dimensions **do**
        if $b.d.getUtil() + v.d < b.d.getReource()$ **then**
          $candidateHost \leftarrow b$
        else
          $candidateHost \leftarrow NULL$
          break
        end if
      end for
    end for
    if $candidateHost \neq NULL$ **then**
      allocate v to $candidateHost$
      calculate new w(B) of $candidateHost$
      update hostList
    end if
  end for

In order for the proposed heuristic to be modular, we propose to make the weighter decision based on a scalar that is produced as a weighted sum across the dimensions i.e. $w(I) = \sum_{i \leq d} \alpha_i I_i$. The vector $\vec{\alpha} = \alpha_1, ..., \alpha_d$ is a scaling vector that defines the coefficients based on the importance of each dimensions. For example the cloud storage provider may wish to have a higher weight on the I/O throughput compared to thw network throughput to decrease the energy footprint or satisfy an SLO. For simplicity we set the $\vec{\alpha}$ elements to 1 [16] and assume each volume request is independent of each other.

In Algorithm 1, we show the pseudo-code of the implemented algorithm. The worst case complexity of the algorithm is $\mathcal{O}(n \cdot j \cdot d)$, where n is the number of volumes in the message list, j is the number of storage hosts and d is the number of resources that are taken into account. The volume requests and storage hosts are sorted based on their weight. Then we check the hosts that can support the volume request. Given the multidimensional nature of the volume request, all dimensions must be satisfied. Once a volume request is allocated on a host, the algorithm recalculates the weight of that host and update the order of host list.

IV. Performance Analysis

In this section, we discuss the performance analysis of the cloud storage heuristic presented in the previous section compared to the default scheduler implementation in OpenStack Cinder. As the system under evaluation is an IaaS it is essential to achieve repeatability of experiments under different circumstances. For this reason, we have chosen to use simulations to evaluate the performance of the proposed heuristics. We developed a simulator in Java to simulate a storage cluster. The source code will be open-sourced for public validation in the future. We implemented both the proposed scheduling algorithm, as well as the default Cinder scheduling algorithm in our simulator.

A. Performance Metrics

We use three metrics to evaluate the performance of proposed scheduling algorithm and default scheduler in OpenStack.

1) *SLO violation*: The SLO violation occurs when the cloud service can not achieve the service objective as promised. We gather data for the availability SLO violation rate and the I/O throughput performance SLO violation rate respectively. The availability SLO violation rate is calculated by the number of failed to be scheduled requests divided by the total number of arrival requests. The I/O performance violation happens when a volume is running at a speed that lower than I/O performance SLO.
2) *Volume speed performance*: This metric includes two aspects: I/O speed that an active volume is running at and available I/O speed that a storage entity can provide to the next allocated volume. Cloud customers always expect their volumes could run faster. On the other hand, it is also important for cloud provider to offer fast enough performance to the next customer.
3) *Number of Active Nodes*: We count how many storage host are actually used to serve volume requests. The unused hosts are turned sleep or suspend mode to save energy and maintenance cost.

B. Experimental Setup

The simulator's deployment scenario is based on commercial off-the-shelf (COTS) hardware proposed by Rackspace [17]. We assume a fairly standard load of 70% reads and 30% for all volume requests, and 8K of IO block size. A single hard drive is able to achieve 175 IOPS. We use the RAID calculator [18] and find that the maximum I/O throughput per storage host is 8800 IOPS. The storage hardware characteristics are described in Table I. In our evaluation, the scheduler takes into account two dimensions $d = 2$: the capacity of a

TABLE I. STORAGE HOST HARDWARE CONFIGURATION

CPU/RAM	1 core at 2.0GHz/16GB
Network Interface	10G Ethenet
Hard Drive	600GB SAS 15k rpm
Number of Hard Drives	60
RAID	0+1
Storage Capacity per node	18TB
Cluster Capacity	18 - 36PB

TABLE II. SIMULATION SETUP

Volume size	100GB 500GB 1000GB
Volume Duration	2hours 4hours 6hours
Default Arrival rate	3 requests/second (Poisson Distribution)
I/O Performance SLO	300 IOPS
Cluster Scale	1000 to 2000 nodes
Simulation Length	3 days
Data Sampling Period	12h to 60h

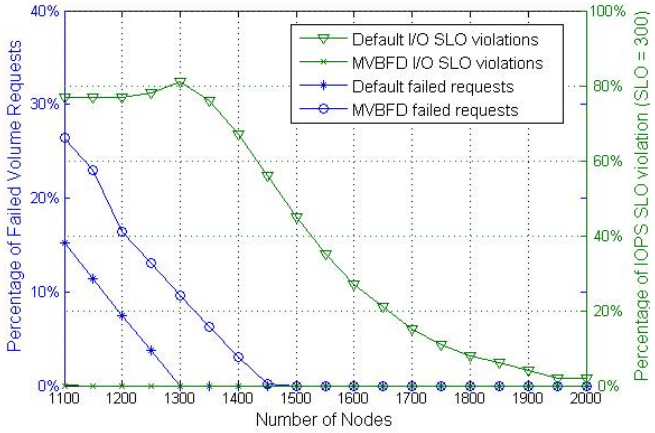

Fig. 3. SLO Violation Rate

host machine and the I/O throughput. While, we could have used more dimensions, these constraints are two of the most important factors during the allocation decision making. The MVBFD algorithm is enabled every 2 seconds. Each volume request contains three fields: volume capacity, performance SLO and duration. The volume capacity is randomly generated based on the following values: 100GB, 500GB, 1TB. The volume requests arrival time is distributed based on a Poisson distribution with $\lambda = 3$ requests/seconds. Each volume will be running for 2 hours, 4 hours and 6 hours with a probability of 25%, 50% and 25%, respectively. The data are sampled from 12nd to 60th hour to achieve steady state performance. Each experiment has been run 10 times and the mean values of the metrics are calculated. The full list of simulation parameters are shown in Table II. We scale the storage cluster from 1000 to 2000 scheduling entities, as a single node can export a large number of pools to the Cinder scheduler. We do not present lower values, because the system would have reached saturation give the above workload.

C. Simulation Results

We first present the results for the SLO violation rate. Fig. 3 shows that the Cinder scheduler achieves 0 failed request i.e. 100% service availability with 1300 nodes, whereas the

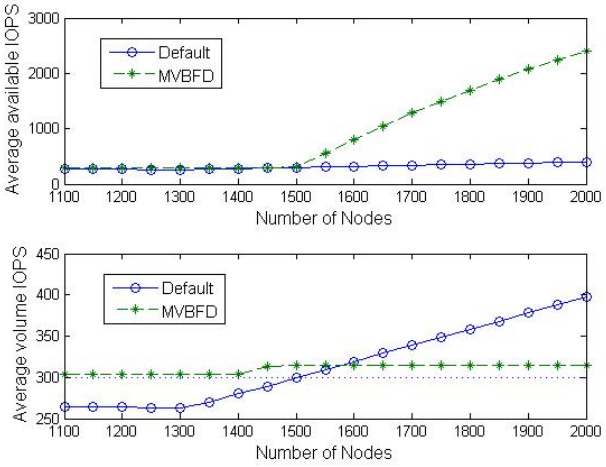

Fig. 4. Available IOPS and Volume Speed Performance

proposed MVBFD algorithm has to utilize 1500 nodes to reach the same goal. However, MVBFD algorithm is able to accomplish 0% in the I/O performance SLO violation rate, or else 100% SLO performance is guaranteed independently on the number of nodes used. On the other hand, the default scheduler has 80% violation rate with 1300 nodes, and requires more than 2000 block storage scheduling entities to be able to achieve an acceptable violation rate. In fact, the default scheduling policy cannot reach 0% in the violation rate of the performance SLO even above 2000 block storage scheduling entities. Hence, the combination of these results shows that the MVBFD algorithm is able to obtain 0 availability and IOPS SLO violation with 1500 storage scheduling entities (nodes or pools).

In Fig.4, we performed an I/O volume speed analysis. The sub-figure in the top indicates that the MVBFD algorithm can deliver higher volume I/O throughput to the next volume request than the default scheduling algorithm. The MVBFD curve increases linearly after 0 SLO violations when the number of nodes scale up. This is because the available resources of the storage pools can be fully utilized by future requests. In the bottom figure, we present the I/O throughput of the already active volumes. In this case, the default scheduler has a small advantage in terms of the I/O access speed compared to the MVBFD after 1500 pools. However, for lower than 1500 pools, the average speed is lower than the SLO, which is the reference line in the figure. This means the cloud storage provider has to pay an SLO violation penalty. On the other hand, the MVBFD achieves steady performance and the I/O throughout is always above the the SLO irrespective of the number of storage nodes. Hence, the benefit of the cloud storage provider from the MVBFD implementation is that the available IOPS for future volumes requests can scale better and as the provisioned resources scale up and down, there is no negative performance effect on the already active storage volumes.

Finally, we look into the maximum number of active nodes used by each scheduling algorithm by varying the λ of the Poisson distribution from 0.5 to 4. Note that the number of available nodes in the storage cluster is 2000. Fig.5 shows that the default scheduler uniformly utilizes all the available

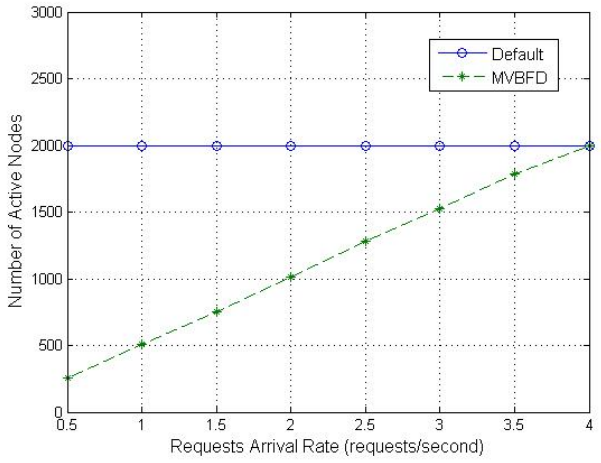

Fig. 5. Number of Active Nodes

nodes at all times, as it is based on allocating the volumes to the pools/nodes that have the most available capacity. On the other hand, the MVBFD will allocate the volumes to the tightest spots, hence it will start with 250 nodes for an arrival rate of 0.5 requests/second and it will reach saturation around 4 requests/second. In other words, MVBDF utilizes only the necessary nodes to serve the provided workload. At the same time, all active nodes fully satisfy the SLO constrains. The benefit for the cloud storage provider from the implementation of MVBFD is that fewer nodes are active, hence less power needs to be consumed as the inactive nodes will be put in sleep/suspend mode.

V. Conclusion

In conclusion, we formulated the volume allocation problem as a multi-dimensional VBP and optimized based on the number of active storage entities. In order to solve the problem we proposed a Modified Vector Best Fit Decreasing algorithm (MVBFD). The proposed scheduler takes into account multiple resources, pre-sorts the volume requests that are located in the messaging queue, apply SLO constraints on each dimension, and chooses the proper storage node. Our proposal was designed based on the principles of OpenStack so that it can be easily integrated to most cloud platforms. However, the same properties could be used in other cloud storage schedulers. Our simulation based experiment showed that the proposed MVBFD algorithm regards its performance on SLO maintenance, volume I/O throughput, scalability and the number of active entities. The results showed a better performance of our approach compared to the default scheduling algorithm in OpenStack Cinder. In addition, the MVBFD algorithm reveals further potential on energy and operation cost reduction.

In summary, the MVBFD scheduling algorithm has demonstrated excellent standing in terms of resource utilization, SLO maintenance, scalability, energy and cost efficient. As a future work, we plan to contribute the scheduling and optimization policies to the OpenStack Cinder project. At the same time, we are currently investigating novel optimizations in the backend systems, and we are planning to deploy the aforementioned policies in a bare metal hardware infrastructure.

Acknowledgment

The authors would like to thank Ben Swartzlander, architect at NetApp, for his useful feedback.

References

[1] R. Ghosh, I. Papapanagiotou, and K. Boloor, "A survey on research initiatives for healthcare clouds," *Cloud Computing Applications for Quality Health Care Delivery*, pp. 1–18, 2014.

[2] "Openstack." [Online]. Available: www.openstack.org

[3] J. Xu and J. A. Fortes, "Multi-objective virtual machine placement in virtualized data center environments," in *IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*. IEEE, 2010, pp. 179–188.

[4] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755–768, 2012.

[5] K. Lu, R. Yahyapour, P. Wieder, C. Kotsokalis, E. Yaqub, and A. Jehangiri, "Qos-aware vm placement in multi-domain service level agreements scenarios," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, June 2013, pp. 661–668.

[6] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 351–364.

[7] Z. Yao, I. Papapanagiotou, and R. D. Callaway, "SLA-aware Resource Scheduling for Cloud Storage ," in *IEEE International Conference on Cloud Networking (CloudNet)*. IEEE, 2014.

[8] G. J. Woeginger, "There is no asymptotic PTAS for two-dimensional vector packing," *Information Processing Letters*, vol. 64, no. 6, pp. 293–297, 1997.

[9] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," *Microsoft Research, Tech. Rep.*, 2011.

[10] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "Sdf: Software-defined flash for web-scale internet storage systems," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 471–484.

[11] "Amazon elastic block store." [Online]. Available: http://aws.amazon.com/ebs/

[12] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, 1979.

[13] L. T. Kou and G. Markowsky, "Multidimensional bin packing algorithms," *IBM Journal of Research and development*, vol. 21, no. 5, pp. 443–448, 1977.

[14] M. Yue, "A simple proof of the inequality FFD (L) 11/9 OPT (L)+ 1, L for the FFD bin-packing algorithm," *Acta mathematicae applicatae sinica*, vol. 7, no. 4, pp. 321–331, 1991.

[15] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 3, no. 4, pp. 299–325, 1974.

[16] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation algorithms for virtualized service hosting platforms," *Journal of Parallel and Distributed Computing*, vol. 70, no. 9, pp. 962–974, 2010.

[17] K. Levenstein. (2013) Configuring openstack block storage. [Online]. Available: https://www.rackspace.com/blog/laying-cinder-block-volumes-in-openstack-part-2-solutions-design/

[18] Raid performance calculator. [Online]. Available: http://wintelguy.com/raidperf.pl