Demand Based Hierarchical QoS Using Storage Resource Pools

Ajay Gulati, Ganesha Shanmuganathan
VMware Inc
{*agulati, sganesh*}*@vmware.com*

Xuechen Zhang
Wayne State University
xczhang@wayne.edu

Peter Varman
Rice University
pjv@rice.edu

Abstract

The high degree of storage consolidation in modern virtualized datacenters requires flexible and efficient ways to allocate IO resources among virtual machines (VMs). Existing IO resource management techniques have two main deficiencies: (1) they are restricted in their ability to allocate resources across multiple hosts sharing a storage device, and (2) they do not permit the administrator to set allocations for a group of VMs that are providing a single service or belong to the same application.

In this paper we present the design and implementation of a novel software system called *Storage Resource Pools* (SRP). SRP supports the logical grouping of related VMs into hierarchical pools. SRP allows reservations, limits and proportional shares, at both the VM and pool levels. Spare resources are allocated to VMs in the same pool in preference to other VMs. The VMs may be distributed across multiple physical hosts without consideration of their logical groupings. We have implemented a prototype of storage resource pools in the VMware ESX hypervisor. Our results demonstrate that SRP provides hierarchical performance isolation and sharing among groups of VMs running across multiple hosts, while maintaining high utilization of the storage device.

1 Introduction

Shared storage access and data consolidation is on the rise in virtualized environments due to its many benefits: universal access to data, ease of management, and support for live migrations of virtual machines (VMs). Multi-tiered SSD-based storage devices, with high IO rates, are driving systems towards ever-higher consolidation ratios. A typical virtualized cluster consists of tens of servers, hosting hundreds of VMs running diverse applications, and accessing shared SAN or NAS based storage devices.

To maintain control over workload performance, storage administrators usually deploy separate storage devices (also called as LUNs or datastores) for applications requiring strong performance guarantees. This approach has several drawbacks: growing LUN sprawl, higher management costs, and over-provisioning due to reduced benefits from multiplexing. Encouraging LUN sharing among diverse clients requires systems to provide better controls to isolate the workloads and enable QoS differentiation. Recently, PARDA [6] and mClock [8] have been proposed to provide storage QoS support. However, these and other existing approaches like SFQ(D) [12], Triage [13] , Façade [15], Zygaria [24], pClock [7] etc. either provide only proportional allocation or require a centralized scheduler (see Section 2.2).

In this paper, we present a new software system called **storage resource pools** (SRPs) with the following desirable properties:
Rich Controls: QoS can be specified using throughput *reservations* (lower bounds), *limits* (upper bounds) and *shares* (proportional sharing). These may be set for individual VMs or collectively for a group of related VMs known as a resource pool. Reservations are absolute guarantees, that specify the minimum amount of service that a VM (or group) must receive. Limits specify the maximum allocation that should be made to the VM or the group. These are useful for enforcing strict isolation and restricting tenants to contractually-set IOPS based on their SLOs. Shares provide a measure of relative importance between VMs or groups, and are used for proportional allocation when capacity is constrained.
Hierarchical Pooling: Storage administrators can define *storage resource pools* (SRPs) to logically partition IO resources in a hierarchical manner. SRPs allow related VMs to be treated as a single unit for resource allocation. These units can be aggregated into larger SRPs to create a resource pool hierarchy. Resource pooling has several advantages; it (1) spares the user from having to set per-VM controls that are hard to determine; (2) allocates resources to divisions or departments based on organizational structure; and (3) allocates resources to a

group of VMs that are working together to provide a single service. The latter scenario is becoming increasingly common with dynamic websites like e-Commerce and social-networking, where a webpage may be constructed by involving several virtual machines.

Dynamic Allocation based on Demand: SRPs can *dynamically* reallocate LUN capacity (IOPS) among VMs based on the current workload demands, while respecting user-set constraints (see Section 2).

Distributed and Scalable Operation: VMs comprising a resource pool may be distributed across multiple servers (hosts), and a single server may run VMs belonging to many different resource pools. Such distributed architectures are very common in virtualized datacenters.

Providing these controls is quite challenging for several reasons: (1) VMs in the same pool may be distributed across multiple hosts; (2) there is no central location to implement an IO scheduler that sees the requests from all the hosts; and (3) workload demands and device IOPS are highly variable and need to be tracked periodically for an effective implementation of resource pooling.

We have implemented a prototype of storage resource pools on the VMware ESX Server hypervisor [19]. In our prototype, an administrator can create one resource pool per storage device. Our extensive evaluation with multiple devices and workloads shows that SRPs are able to provide the desired isolation and aggregation of IO resources across various VM groups, and dynamically adapt allocation to the current VM demands.

The rest of the paper is organized as follows. Section 2 presents an example to motivate the need for storage resource pools and discusses related work in this area. Section 3 presents the SRP design in detail. In Section 4 we discuss implementation details and storage-specific issues. Section 5 presents the results of extensive performance evaluation that demonstrates the power and effectiveness of storage resource pools. Finally we conclude with some directions for future work in Section 6.

2 Motivation and Related Work

In this section we first motivate the need for storage resource pools using a simple example and discuss the challenges in implementing them in a distributed cluster. We then review the literature on IO resource management and the limitations of existing QoS techniques.

2.1 Need For Storage Resource Pools

Consider an enterprise that has virtualized its infrastructure and consolidated its IO workloads on a small set of storage devices. VMs from several different divisions, (say sales and finance for example), may be deployed on the same device (also called as datastore). The administrator sets up a pool for the divisions with settings reflecting the importance of their workloads. The VMs of the sales division (handling sales in different continents) may need an overall reservation of 1000 IOPS. This total reservation is flexibly shared by these VMs based on the peaks and troughs of demand in different time zones. The finance division is running background data analytics in their VMs and the administrator wants to restrict their combined throughput to 500 IOPS, to reduce their impact on critical sales VMs.

In addition to reservation and limit controls, the administrator may want the VMs from the sales division to get more of the spare capacity (*i.e.*capacity left after satisfying reservations) than VMs from the finance division. This is only relevant during contention periods when demand is higher than the current capacity. For this, one can set shares at the resource pool level and the allocation is done in proportion to the share values. Shares can also be used to do prioritized allocation among the VMs within the same pool.

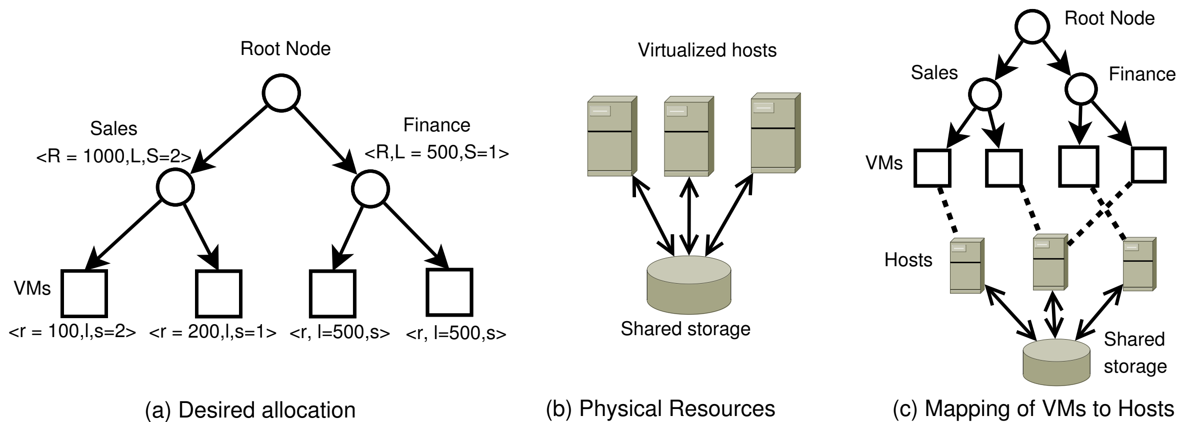
Many of these requirements can be met by using hard VM-level settings. However, that will not allow the IOPS to be dynamically shared among VMs of a group based on demand, as needed by the sales division. Similarly, one will have to individually limit each VM, which is more restrictive than setting the limit on a group of them. Moreover if a VM gets idle, the resource should flow first to the VMs of the same group rather than to a different group.

Figure 1(a) shows a storage resource pool with two children Sales and Finance. These child nodes have reservation (R) and limit (L) settings as per the company policy. Both of them also have two child VMs with settings as shown. The reservation for the Sales node (1000) is higher than the sum of the reservations (300) of its child VMs; the excess amount will be dynamically allocated to the children to increase their statically-set reservations (r), based on their current demand and other settings. Similarly, in the case of the Finance node, the parent limit (500) is less than the sum of the limits (1000) of the individual child VMs; hence, we need to dynamically set the limit on the two child VMs to sum to the parent's value. Once again the allocation is made dynamically based on the current distribution of demand among these VMs, and the other resource control settings.

Also note that we have assigned twice the number of shares to the Sales pool, which means that the capacity at the root will be allocated among the Sales and Finance pools in the ratio 2:1, unless that would lead to violating the reservation or limit settings at the node. We have also allocated different shares to VMs in the Sales pool to allow them to get differentiated service during periods of contention.

Technique	*Distributed*	*Reservation*	*Limit*	*Share*	*Hierarchical*	Storage Allocation
Proportional sharing techniques	No	No	No	Yes	No	Yes
Distributed mechanisms (PARDA [6])	Yes	No	No	Yes	No	Yes
Centralized IO schedulers	No	Yes	Yes	Yes	Some	Yes
ESX CPU scheduler	No	Yes	Yes	Yes	Yes	No
ESX Memory scheduler	No	Yes	Yes	Yes	Yes	No
Storage Resource Pools	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Table 1: Comparison of storage resource pools with existing resource allocation schemes



(a) Desired allocation (b) Physical Resources (c) Mapping of VMs to Hosts

Figure 1: Storage resource pools description and mapping to physical resources

This task of enforcing the desired controls in SRP is challenging because the storage device is accessed by multiple hosts in a distributed manner using a clustered file system like VMFS [1] (in our case) or NFS, with no centralized control on the IO path as shown in Figure 1(b). Finally, based on the requirements for other resources such as CPU and memory, the VMs may get dynamically placed or moved among hosts using live migration. Thus, the system should adapt to the dynamic placement of VMs and cannot rely on static settings. Figure 1(c) shows an example mapping of the VMs to hosts.

Resource Pool Semantics: In summary, the resource pool semantics dictate the following allocation at each level of the resource pool (RP) tree: (1) Distribute the parent's reservation among its children in proportion to their shares while ensuring that each child gets at least its own reservation and no more than its demand or static limit; (2) Distribute the parent's limit among its children in proportion to their shares while making sure that no child gets more than its own static limit or demand; and (3) Distribute the parent's share to its children in proportion to their shares.

2.2 Previous Work

We classified existing work on QoS controls for storage into three categories, as discussed below. Table 1 provides a summary of existing approaches and their comparison with Storage Resource Pools.

Proportional Sharing: Many approaches such as Stonehenge [10], SFQ(D) [12] have been proposed for proportional or weighted allocation of IO resources. These techniques are based on fair-queuing algorithms proposed for network bandwidth allocation (WFQ [3], SFQ [5]) but they deploy storage-specific optimizations to maintain higher efficiency of the underlying storage system. DSFQ [23] proposed proportional allocation for distributed storage, but it needs specific cooperation between the underlying storage device and storage clients.

Different from throughput allocation, Argon [21] and Fahrrad [16] proposed time-sliced allocation of disk accesses to reduce interference across multiple streams accessing the device. Façade [15] presented a combination of EDF based scheduling and queue depth manipulation to provide SLOs to each workload in terms of IOPS and latency. The reduction in queue depth to meet latency bounds can have severe impact on the overall efficiency of the underlying device. SARC+Avatar [25] improved upon that concern by providing better bounds on the queue depth and a trade-off between throughput and latency.

Unlike these centralized schedulers, PARDA [6] provided a distributed proportional-share algorithm to allocate LUN capacity to VMs running on different hosts.

PARDA also runs across a cluster of ESX hosts, but it doesn't support reservation and limit controls. A limitation of pure share-based allocation is that it cannot guarantee a lower bound on absolute VM throughput. Consequently, VMs with strict QoS requirements suffer when the aggregate IO rate of the LUN drops or if new VMs are added on the same LUN. In addition, PARDA does not support resource pooling so VMs running on different hosts are completely independent.

Triage [13] uses a centralized control mechanism that creates an adaptive model of the storage system and sets per-client bandwidth caps to allocate a specific share of the available capacity. Doing per-client throttling using bandwidth caps can underutilize array resources if the workloads become idle. Triage also doesn't support resource pooling unlike SRP.

Algorithms with Reservation Support: The problem of resource reservations for CPU, memory and storage management are well studied. Several approaches [4, 17] have been proposed to support CPU reservations for real-time applications while maintaining proportional resource sharing. Since CPU capacity is fixed and not significantly affected by workloads, it is relatively straightforward to provide CPU reservation in MHz. The VMware ESX server [20, 22] has been providing reservation, limit and shares based allocation to VMs for both CPU and memory since 2003.

For the allocation of storage resources, mClock [8] proposed a per-host local scheduler that provides all three controls (reservation, limit and shares) for VMs running on a *single host*. mClock does this by using three separate tags per client, one for each of the controls. The tags are assigned using real-time instead of virtual time and the scheduler dynamically switches between the tags for scheduling. However, in a clustered environment, a host-level algorithm alone cannot control the LUN capacity available to a specific host due to workload variations on other hosts in the cluster. Hence, *any solution local to a single host* is unable to provide guarantees across a cluster and is not sufficient for our use case.

Hierarchical Resource Management: CPU and memory resource pools [20] implemented by the VMware ESX server since 2003, were proposed for hierarchical resource management. However, the existing solutions were not designed for storage devices which are stateful and have fluctuating capacity. More importantly, both CPU and memory are local to a host and a centralized algorithm suffices to do resource allocation. Zygaria [24] proposed a hierarchical token-bucket based centralized IO scheduler to provide hierarchical resource allocation while supporting reservations, limits and a statistical notion of shares.

Storage resource pools, by contrast, need to work across a cluster of hosts that are accessing a storage device in a distributed manner. This makes it harder to use any centralized IO scheduler. Furthermore, earlier approaches use a fixed queue depth for the underlying device, which is hard to determine in practice; SRP varies the queue depth in order to ensure high device utilization. Finally, SRP adjusts VM-level controls adaptively based on the demand, so a user does not have to specify all of the per-VM settings.

3 Storage Resource Pool Design

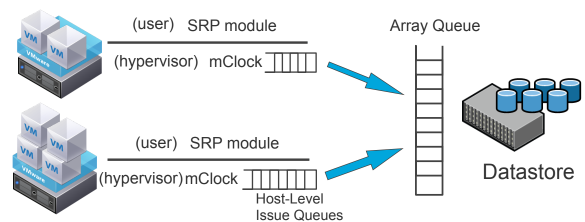


Figure 2: SRP system architecture

In this section, we discuss the key concepts and overall design of storage resource pools. Figure 2 shows the overall system architecture with multiple virtualized hosts accessing a shared storage array. Each host has two main components: an **SRP module** and a local IO scheduler **mClock** [8], that synergistically control the allocations to the VMs. The SRP module is responsible for determining how much of the array capacity should be provided to each VM and each host. mClock is responsible for scheduling the VMs on each host in accordance with the allocations.

The SRP module is a user-level process that runs directly on the ESX hypervisor. Each SRP module periodically decides how much of the array capacity to allocate to this host and the VMs running on it for the next interval. The amount is based on several factors, both static and dynamic: the structure of the RP tree, the control settings (static R,L,S) on the nodes, the dynamic demand of the VMs, the dynamic array capacity for the current workload mix, and the mapping of the VMs to hosts.

SRP computes two quantities periodically: (1) the dynamic VM settings (R,L,S) in accordance with the global resource pool constraints and VM demands, and (2) the *issue queue depth* per host. The maximum number of requests that a host can keep outstanding at the array is bounded by the issue queue depth (also called **host queue depth**), as shown in Figure 2. The size of the issue queue reflects the array capacity currently allocated to the host. Next we discuss the functionality of SRP Module in more detail.

3.1 SRP Module

Algorithm 1 provides a high-level description of the SRP module. It performs three major tasks: 1) updates demand on the RP-tree nodes; 2) computes new values for the reservations, limits, and shares for the VMs for use by the mClock scheduler – these are also called dynamic R,L,S values respectively; and 3) estimates the new array capacity and divides it among the hosts. The inputs to the module are the statistics collected during the previous monitoring interval, specifically the VM demands and measured latency.

Algorithm 1: SRP Module

/* Run periodically to reallocate IO resources */

1. **Update demand in RP Tree**
   (*a*) Update demand of VMs in RP tree
   (*b*) Update demands at internal RP-tree nodes by aggregating demands of its children
2. **Compute dynamic R,L,S**
   (*a*) Update VM Reservation (R Divvy)
   (*b*) Update VM Limits (L Divvy)
   (*c*) Update VM Shares (S Divvy)
3. **Update Array and Host Queue Depths**
   (*a*) Estimate new *array queue depth*
   (*b*) Compute VM entitlement
   (*c*) Set *host queue depth*

3.1.1 Update and Aggregate Demand

The ESX hypervisor maintains stats on the aggregated latency and total number of IOs performed by each VM. Using these stats, the SRP module determines the average latency (*avgLatency*) and average IOPS (*avgIops*), and computes the **VM demand** in terms of the average number of outstanding IOs (*demandOIO*) using Little's law [14] (see equation 1). Each SRP module owns a block in a shared file on the underlying datastore and updates the VM-level stats in that file. By reading this file, every host can get the VM demand values in terms of outstanding IOs (also called as OIOs). The SRP module on each host then converts the *demandOIO* (see equation 2) to a normalized demand IOPS value (*demandIops*), based on the storage device congestion threshold latency ($\mathscr{L}_c$). This helps to avoid overestimating a VM's demand based on local latency variations.

$$demandOIO = avgLatency \times avgIops \quad (1)$$

$$demandIops = demandOIO / \mathscr{L}_c \quad (2)$$

$$demandIops = min(max(demandIops, R), L) \quad (3)$$

The congestion threshold is the maximum latency at which the storage device is operated. This ensures a high utilization of the underlying device. Based on our experiments we have found the range between 20 to 30 *ms* to be good enough for disk-based storage devices. For SSD-backed LUNs, $\mathscr{L}_c$ can be set to a lower value (*e.g.* 5 ms). Our implementation is not sensitive to this control but the utilization of the underlying device may get impacted. The SRP module controls the array queue depth to keep the latency close to $\mathscr{L}_c$, so that we utilize the device in an efficient manner.

The *demandIops* value is then adjusted to make sure that it lies within the lower and upper bounds represented by the reservation and limit settings for each VM (see equation 3). Finally the demand is aggregated level-by-level at each node of the tree by summing the *demandIops* of its children and then applying the bound checks (equation 3) at the parent.

3.1.2 Computing Dynamic R,L,S for VMs

This step computes dynamic reservation, limit and share values for VMs based on the structure of the RP tree, the static (user-specified) reservation, limit and shares settings on the nodes, as well as the demand of VMs and internal nodes computed in Step 1. These operations are called R-*divvy*, L-*divvy* and S-*divvy* respectively. The exact divvy algorithm is explained in Section 3.2 followed by an example divvy computation in Section 3.3.

The R-divvy distributes the total reserved capacity at the root node among the currently active VMs. The allocation proceeds in a top-down hierarchical manner. First the root reservation is divided among its children based on their control settings. For the divvy, the limit of a child node is temporarily capped at its demand. This allows resources to preferentially flow to the nodes with higher current demand. Since the reservation at a node usually exceeds the sum of the reservations of its currently active children, the R-divvy will assign a higher reservation value per VM than its static setting.

The L-divvy is similarly used to provide higher dynamic limits to VMs with higher shares and demands. For instance, the user may set each VM limit to max (unlimited), but place an aggregate limit on the RP node. At run time, the aggregate limit needs to be allocated to individual VMs. The S-divvy similarly divides up the shares at a node among its children. Unlike the R and L divvies, the S-divvy does not use the demands but only the static share settings in doing the computation.

3.1.3 Update Array and Host Queue Depths

In this step, the SRP module computes the new array capacity, and the portion to be allocated to each host.

Since there is no centralized place to do scheduling across hosts, it is not possible to directly allocate IOPS to the hosts. Instead, we use the host queue depth (Q_h) as a control to do across-host allocation. We describe the three steps briefly below.

Update Array Queue Depth: To determine the new array queue depth we use a control strategy inspired by PARDA [6]. The queue depth is adjusted to keep the measured latency within the congestion threshold, using equation 4 below.

$$Q(t+1) = (1-\gamma)Q(t) \; + \; \gamma\left(\frac{\mathscr{L}_c}{Lat(t)}Q(t)\right) \quad (4)$$

Here $Q(t)$ denotes the array queue depth at time t, $Lat(t)$ is the current measured average latency, $\gamma \in [0,1]$ is a smoothing parameter, and $\mathscr{L}_c$ is the device congestion threshold.

Compute VM OIO Entitlement: We first convert the array queue depth value computed above to an equivalent array IOPS capacity using Little's law:

$$arrayIOPS = Q(t+1)/\mathscr{L}_c. \quad (5)$$

We then use the divvy algorithm (Algorithm 2) described in Section 3.2, to divide this capacity among all the VMs based on their settings. This results in the VM IOPS entitlement denoted by E_i. The conversion from queue depth to IOPS is done because the resource pool settings used for the divvy are in terms of user-friendly IOPS, rather than the less transparent OIO values.

Set Host Queue Depth: Finally, we set the host queue depth (Q_h) to be the fraction of the array queue depth that the host should get based on its share of the VM entitlements in the whole cluster (using equation 6).

$$Q_h = Q(t+1) \times \frac{\sum_{i \in VM\ on\ host} E_i}{arrayIOPS} \quad (6)$$

At each host, the local mClock scheduler is used to allocate the host's share of the array capacity (represented by the host queue depth Q_h) among its VMs. mClock uses the dynamic VM reservations, limits, and shares settings computed by SRP in step 2 to do the scheduling.

3.2 Divvy Algorithm

The root of the RP tree holds four resource types that need to be divided among the nodes of the tree: (1) RP reservation ($\mathscr{R}$), (2) RP limit ($\mathscr{L}$), (3) RP shares ($\mathscr{S}$), and (4) array IOPS. The first three values are divvied to compute the dynamic R,L,S settings, and the fourth value (array IOPS) is divvied to compute per VM entitlement. We use the same divvy algorithm for all these except for shares. The divvying of shares ($\mathscr{S}$) is much simpler and is based only on the static share values of the child nodes. The shares at a node are divided among the children in the ratio of the children's share settings. Next, we explain the common divvy algorithm for the remaining values. We use the generic term *capacity* to denote the resource being divvied.

Intuitively, the divvy will allocate the parent's capacity to its children in proportion to their shares, subject to their reservations and limit controls. Algorithm 2 presents an efficient algorithm to do the divvying for a given capacity $\mathscr{C}$. The goal is to assign each child to one of three sets: RB, LB, or PS. These represent children whose allocation either equals their reservation (RB), equals their limit (LB), or lies between the two (PS). The children in PS get allocations in proportion to their shares.

We use w_i to denote the fraction of shares assigned to child i relative to the total shares of all the children. We use the terms *normalized reservation* and *normalized limit* to denote the quantities r_i/w_i and l_i/w_i respectively. $\mathscr{V}$ is the ordered set of all normalized reservations and limits, arranged in increasing order. Ties between normalized reservation and limit values of child i are broken to ensure that r_i/w_i appears earlier than l_i/w_i.

Initially we allocate all children their reservations, and place them in set RB. At each step k, we see if there is enough capacity to increase the allocation of the current members of PS to a new target value v_k. This is either the normalized reservation or limit of some child denoted by $index[k]$. In the first case the child is moved from RB to PS, and in the latter case the child is moved from PS to LB. The total weight of the children in PS is adjusted accordingly. This continues till either the capacity is exhausted or all elements in $\mathscr{V}$ have been examined.

The complexity of the algorithm is $O(n \log n)$, bounded by the time to create the sorted sequence $\mathscr{V}$. At the end of the process, children in LB are allocated their limit, those in RB are allocated their reservation, and the rest receive allocation of the remaining capacity in proportion to their shares.

This divvy algorithm is used by the SRP module for R-divvy, L-divvy and entitlement computation. The only difference in these is the parameters with which the divvy algorithm is called. In all cases, the demand of a node is used as its temporary l value during the divvy, while its r and s values are the user set values. If the sum of the demands of the children is smaller than the capacity being divvied at the parent, the user set limits are used instead of the demand. For R-divvy, the reservation set at the root ($\mathscr{R}$) is used as the capacity to divvy, while for L-divvy and entitlement computation they are the root *limit* setting ($\mathscr{L}$) and the *arrayIOPS* respectively.

Algorithm 2: O(n log n) Divvy Algorithm

Data: $\mathscr{C}$: Capacity to divvy
Child c_i, $1 \le i \le n$, parameters: r_i, l_i, s_i.
Result: a_i: allocation computed for child c_i.
Variables: $w_i = s_i / \sum_{j=1}^{n} s_j$
$\mathscr{V}$: Ordered set $\{v_1, v_2, \cdots v_{2n},\ v_i \le v_{i+1}\}$ of elements from set $\{r_i/w_i,\ l_i/w_i,\ 1 \le i \le n\}$.
$index[i]$; equals k if v_i is either r_k or l_k.
$type[i]$: equals L (R) if v_i is a limit (reservation).
Sets: $RB = \{1, \cdots n\}$, $LB = \{\}$, $PS = \{\}$.
$RBcap = \sum_{j=1}^{n} r_j$, $LBcap = 0$, $PSwt = 0$.

```
foreach k = 1,...,2n do
    /*Can allocation in PS be increased to v_k?*/
    if (PSwt * v_k + LBcap + RBcap > C) then
        break
    /* If type[k] is the limit of a child in PS: Transfer the
    child from PS set to LB set */
    if (type[k] = L) then
        LB = LB ∪ {index[k]}
        LBcap = LBcap + l_index[k]
        PS = PS − {index[k]}
        PSwt = PSwt − w_index[k]
    else
        /* type[k] is R: Move child from RB to PS*/
        PS = PS ∪ {index[k]}
        PSwt = PSwt + w_index[k]
        RB = RB − {index[k]}
        RBcap = RBcap − r_index[k]
if i ∈ RB   a_i = r_i; /*allocation equals reservation */
if i ∈ LB   a_i = l_i; /*allocation equals limit */
/* PS members get rest of capacity in shares ratio.*/
if i ∈ PS   a_i = (w_i / Σ_{j∈PS} w_j) × (C − LB_cap − RB_cap);
```

3.3 A Divvy Example

We illustrate the divvy operation using the RP tree in Figure 3. The tuple U denotes static *user* settings, and the tuple D shows the dynamic reservation, limit and share values of each node as computed by the *divvy* algorithm. The demand is also shown for each VM (leaf-nodes).
The first step is to aggregate VM demands (step 1 of Algorithm 1), and use it as a temporary cap on the limit settings of the nodes. Hence the limits on nodes A through D are temporarily set to 600, 400, 400, and 100 respectively. The limit for a non-leaf node is set to the smaller of its static limit and the sum of its children's limits. For nodes E and F the limits are set to 1000 and 500 respectively.

R Divvy: The algorithm then proceeds level-by-level down from the root using Algorithm 2 to divvy the parent reservation among its children. At the root of the tree, $\mathscr{R} = 1200$ is divvied between nodes E and F in the ratio of their shares 3 : 1, resulting in allocations of 900 and 300 respectively. Since these values lie between the reservation and limit values for the nodes, this is the final result of the R-divvy at the root node.

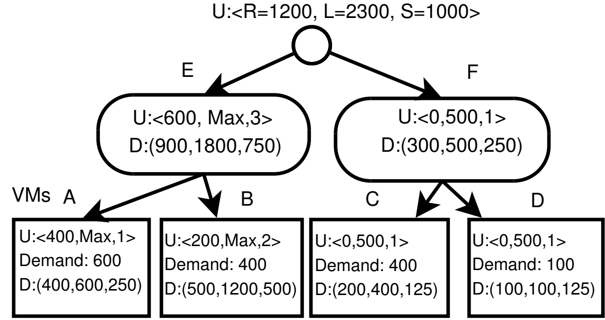


Figure 3: Divvy example for R, L and S

At the next level, the reservation of $R = 900$ at node E is divvied up among VMs A and B. Based on shares (1 : 2), A would be allocated 300, which is below its reservation of 400. Hence, the algorithm would actually give A its reservation amount (400) and B would get the rest (500). For VMs C and D, divvying the parent reservation in the 1 : 1 share ratio would lead to an allocation of 150 each; however, since D's limit has been temporarily capped at its demand, it is given 100 while C gets the remaining amount 200.
L Divvy: The L-divvy is similar and uses Algorithm 2 to divide the parent's limit among its children, level-by-level. The limit of $\mathscr{L}$ = 2300 at the root is divided in the ratio of 3 : 1 among E and F, but is capped at the limit setting of the child; so the resulting allocations to nodes E and F are 1800 and 500 respectively. The dynamic limit settings at the other nodes can be similarly verified.
S-divvy: At each level the shares at the parent are simply divided in the ratio of the user set S values of the children.

Notice that although VMs C and D have identical static settings, due to the difference in their demands, the dynamic settings are different: $(200, 400, 125)$ and $(100, 100, 125)$ respectively. Similarly, excess reservation was given to VM B over VM A since it has a higher share value; however, to meet A's user-set reservation, B received less than twice A's reservation.

4 Implementation Issues

In this section, we discuss some of the implementation issues and storage-specific challenges that we handled while building our prototype for storage resource pools.

Shared Files. In order to share information across multiple hosts, we use three shared files on the underlying storage device running VMFS [1] clustered file system. The first file contains that structure of the resource pool tree and the static RP node settings. The second file allows hosts to share the current VM demands with each other. Each host is allotted a unique 512-byte block in

this file, that can be read by other hosts when performing the entitlement computation. Heart-beats using generation numbers are used in this file to detect host failures. The mapping of hosts to blocks is kept in a third file, and is the only structure that needs locking when a new host joins or leaves the resource pool.

This information could alternatively be disseminated via a broadcast or multicast network channel between the hosts. We chose to use shared files in our design because it reduces the dependence on other subsystems, so that a failure or congestion in the network does not affect SRP. Our approach only allocates 512 bytes per host and does 2 IOs per host every 4 seconds. We also use a special `MULTI_WRITER` mode for opening the file, which doesn't involve any locking. We have not seen any scalability issues in our testing of up to 32 hosts and don't expect scalability to be a major problem in the near future.

Local Scheduler. As shown in Figure 2, the mClock scheduler [8] is used on each host to allocate the host's resources among its VMs. In some cases, we noticed long convergence times before the scheduling reflected the new policy settings. To fix this, we modified the mClock algorithm to reset the internally used tags whenever control settings are changed by SRP. The update frequency of 4 seconds seems to work fine for mClock.

System Scalability. Scalability is a critical issue for VMware ESX server clusters that may support up to thousands of VMs. Two of the design choices that help us avoid potential performance bottlenecks are: (1) Every host makes its allocation decisions locally after reading the shared VM-demand file using a single IO. Having no central entity makes the system robust to host failures and, together with the efficient file access mentioned earlier, allows it to scale to a large number of hosts. (2) The implementation can also handle slightly stale VM data, and doesn't require a consistent snapshot of the per-VM demand values.

4.1 Storage-specific Challenges

A key question that arises in the SRP implementation is: *How many IOPS can we reserve on a storage device?* This $\mathscr{R}$ value for the root of the RP tree, is an upper bound on the total VM reservations that will be allowed on this LUN by admission control. This is a well-known (and difficult) problem since the throughput is highly dependent on the workload's access pattern.

We suggest and use the following approach in this work: compute the throughput (in IOPS) for the storage device using random read workloads. This can be done either at the time of installation or later by running a micro-benchmark. Some of the light-weight techniques proposed in Pesto [9] can also be used to determine this value. Once this is known, we use that as an upper bound on reservable capacity, providing a conservative bound for admission control and leaving some buffer capacity for use by VMs with no (or low) reservations.

IO sizes pose another problem in estimating the reservable capacity. To handle this we compute the value using a base IO size of 16KB, and treat large IO sizes as multiple IOs. Many functions can be used to determine this relationship as described in PARDA [6] and mClock [8]. We use a simple step function where we charge one extra IO for every 32 KB of IO size. This seems to provide a good approximation to more fine-grained approaches.

We also perform certain optimizations to help with sequential workloads. For instance, mClock schedules a batch of IOs from one VM if the IOs are close to each other (within 4 MB). Similarly, arrays try to detect sequential streams and do prefetching for them. In most virtual environments, however, blending of IOs happens at the array and sequentiality doesn't get preserved well at the backend. Features like thin-provisioning and deduplication also make it difficult to maintain sequentiality of IOs.

An interesting issue that needs to be faced when dealing with IO reservation is the concurrency required of the workload. If the array is being operated at a latency equal to the congestion threshold $\mathscr{L}_c$ and the reservation is R IOPS, steady-state operation requires the number of outstanding IOs to be $\mathscr{L}_c$ x R. For example, if the congestion threshold is $20ms$, a single threaded VM application (doing synchronous IOs) can get a maximum throughput of 50 IOPS. In order to get a reservation of 200 IOPS, the application or VM should have 4 outstanding IOs most of the time. This issue is similar to meeting CPU reservations in a multi-vcpu VM. A VM with 8 1GHz virtual CPUs and 8 GHz reservation requires at least 8 active threads to get its full reservation.

Impact of SSD Storage. We expect SRP to work even better for SSD-based LUNs. First, the overall IOPS capacity is much higher and more predictable for SSDs, making it is easier to figure out the IOPS that can be reserved. Second, the issue of random vs. sequential IOs is also less pronounced in case of SSDs. Given the small response times, we can even run the algorithm more frequently than every 4 seconds to react faster to workload changes. This is something that we plan to explore as part of future work.

5 Experimental Evaluation

In this section, we present results from a detailed evaluation of our prototype implementation of storage resource pools in the VMware ESX server hypervisor [19]. Our experiments examine the the following four questions: (1) How well can SRP enforce resource controls (reservations, limits and shares) for VMs and resource pools

spanning multiple hosts? (2) How effective is SRP in flowing resources between VMs in the same pool? (3) How does SRP compare with PARDA and mClock running together? (4) How well do we handle enterprise workloads with dynamic behavior in terms of IO type, locality and IO sizes?

5.1 System Setup

We implemented storage resource pools as a user level process running on ESX, and implemented the necessary APIs to set mClock controls in the hypervisor. We use mClock as the underlying local IO scheduler in ESX. For the experiments, we used a cluster consisting of five ESX hosts accessing a shared storage array. Each host is a HP DL380 G5 with a dual-core Intel Xeon 3.0GHz processor, 16GB of RAM and two Qlogic HBAs connected to multiple arrays. We used two arrays for our experiments: (1) EMC CLARiiON CX over a FC SAN, and (2) Dell Equallogic array with iSCSI connection. The storage volumes are hosted on a 10-disk RAID-5 disk group on the EMC array, and a 15-disk (7 SSD, 8 SAS) pool on the Dell Equallogic array.

We used multiple micro and macro benchmarks running in separate VMs for our experiments. These include Iometer, DVDStore and Filebench based oltp, varmail and webserver workloads. The Iometer VMs have 1 virtual CPU, 1 GB RAM, 1 OS virtual disk of size 4GB and a 8 GB data disk. The DVDStore VM is a Windows Server 2008 machine with 2 virtual CPUs, 4 GB RAM, and three virtual disks of sizes 50 (OS disk), 25 (database disk) and 10 (log disk) GB respectively. The Filebench VMs have 4 virtual CPUs, 4 GB RAM and an OS disk of size 10 GB. For mail and webserver workloads we use a separate 16 GB virtual disk and for the oltp workload we use two separate disks of sizes 20 GB (data disk) and 1 GB (log disk) respectively.

5.2 Micro Benchmark Evaluations

In this section we present several experiments based on micro-benchmarks that show the effectiveness of SRP in doing allocations within and across resource pools.

5.2.1 Enforcement of Resource Pools Controls

First we show that the resource controls set at the VM and resource pool level are respected. For this experiment, we ran six VMs distributed across two hosts as shown in Figure 4. Host 1 runs VMs 1, 2 and 3, and the other VMs run in Host 2. All the VMs are accessing the EMC CLARiiON array.There are three resource pools RP1, RP2 and RP3 each with two VMs; all the resource pools have one VM on each of the two hosts. VMs 1 and 4 are in RP1,

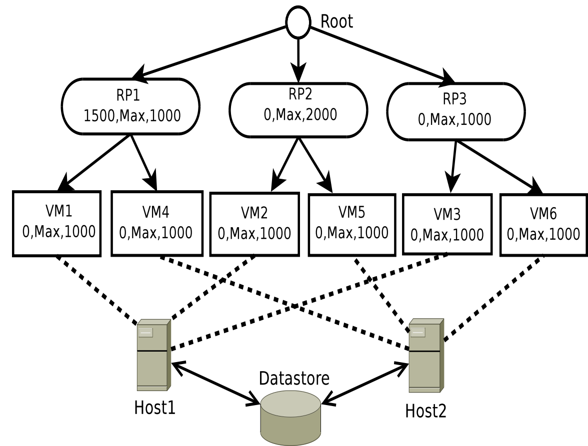


Figure 4: SRP tree configuration for micro benchmark based experiments

VMs 2 and 5 are in RP2 and VMs 3 and 6 are in RP3. The initial resource pool settings are shown in Table 2.

Resource Pool	Reservation	Shares	Limit	VMs
RP1	1500	1000	Max	1, 4
RP2	0	2000	Max	2, 5
RP3	0	1000	Max	3, 6

Table 2: Initial resource pool settings

VM	Size, Read%,Random%	Host	Resource Pool
VM1	4K, 75%, 100%	1	RP1
VM2	8K, 90%, 80%	1	RP2
VM3	16K, 75%, 20%	1	RP3
VM4	8K, 50%, 60%	2	RP1
VM5	16K, 100%, 100%	2	RP2
VM6	8K, 100%, 0%	2	RP3

Table 3: VM workload configurations

To demonstrate the practical effectiveness of SRP, we experimented with workloads having very different IO characteristics. We used six workloads, generated using Iometer on Windows VMs. All VMs are continuously backlogged with a fixed number of outstanding IOs. The workload configurations are shown in Table 3. The VMs do not have any reservation or limits set (which default to 0 and *max* respectively), and they all have equal shares of 1000.

At the start of the experiment, RP1 has a reservation of 1500 IOPS and 1000 shares. SRP should give RP1 its reservation of 1500 IOPS, and allocate additional capacity between RP2 and RP3 in the ratio of their shares (2 : 1), until their allocations catch up with RP1. The allocation to RP1 should be divided equally between VMs 1 and 4, which should receive allocation of 750 IOPS each.

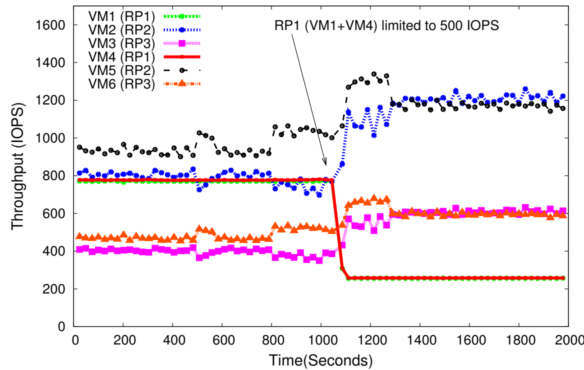


Figure 5: RP1's reservation is changed to 0 and its limit is set to 500 at t = 1000 sec. SRP always satisfies reservations and limits while doing allocation in proportion to shares.

Figure 5 shows the experimentally measured throughputs of all the VMs. The throughputs of VMs 1 and 4 are close to 750 IOPS as expected. The total throughput of RP2 (VMs 2 and 5) is a little less than twice that of RP3 (VMs 3 and 6). The reason is because VMs 3 and 6 have highly sequential workloads (80% and 100%), and get some preferential treatment from the array, resulting in a little higher throughput than their entitled allocations. After about 1000 seconds, the reservation of RP1 is set to 0 and its limit is reduced to 500 IOPS. Now VMs 1 and 4 only get 250 IOPS each, equally splitting the parent's limit of 500 IOPS. The rest of the capacity is divided between RP2 and RP3 as before in a rough 2 : 1 ratio.

We also experimented with setting the controls directly on the VMs instead of the RP nodes. We set reservations of 750 each for the VMs in RP1, and shares of 2000 (1000) to each of the VMs in RP2 (respectively RP3). The observed VM througphputs were similar to the initial portion of Figure 5. The ability to set controls at the RP nodes instead of individual VMs provides a very convenient way to share resources using very few explicit settings.

5.2.2 VM Isolation in Resource Pools

In this experiment we show how RPs allow for stronger sharing and better multiplexing among VMs in a pool so that resources stay within it. This has advantages when VMs are not continuously backlogged; the capacity freed up during idle periods is preferentially allocated to other (sibling) VMs within the pool rather than spread across all VMs.

The setup is identical to the previous experiment. At the start, the throughputs of the VMs shown in Figure 6 match the initial portion of Figure 5 as expected. Starting at time 250 seconds, VM 1 goes idle. We see that the entire slack is picked up by VM4, its sibling in RP1, whose throughput rises from 750 to 1500. The other VMs do not get to use this freed-up reservation since VM4 has first priority for it and it has enough demand to use it completely.

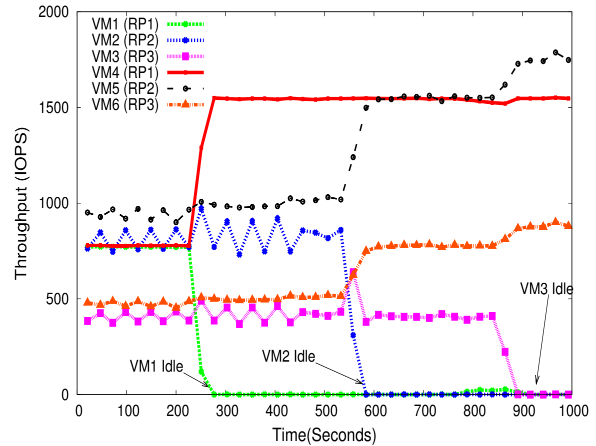


Figure 6: VMs 1, 2 and 3 get idle at t = 250, 500 and 750 sec respectively. Spare IOPS are allocated to the sibling VMs first

At time 550 seconds, VM2 goes idle, and its sibling VM5 on the other host sees the benefit within just a few seconds. VM6 which runs on the same host as VM5 also gets a slight boost from the increase in queue depth allocated to this host. The array also becomes more efficient and this benefit is given to all the active VMs in proportion to their shares. After VM2 becomes idle, RP2 gets higher IOPS than RP3 due to its higher shares.

Finally VM3 goes idle and VM6 gets the benefit. There is not much benefit to the other workloads when the sequential workload becomes idle. But still the reservations are always met and the workloads under RP2 and RP3 are roughly in the ratio of 2 : 1. This experiment shows the flow of resources within a resource pool and the isolation between pools.

5.2.3 Comparison with Parda+mClock

We compared Storage Resource Pool with a state-of-the-art system that supports reservation, limit and shares. We ran both PARDA [6] and mClock [8] together on ESX hosts. PARDA does proportional allocation for VMs based on the share settings. PARDA works across a cluster of hosts by leveraging a control algorithm that is very similar to FAST TCP [11] for flow control in networks. mClock is used as the local scheduler which supports reservations, limits and shares for VMs on the same host.

Since PARDA and mClock do not support resource pools, we created a single pool with VMs as its children and set the controls only at the VM-level. We found that

VM	R_i	L_i	S_i
VM1(Host1)	750	Max	1000
VM2(Host1)	750	Max	1000
VM3(Host1)	0	Max	1000
VM4(Host2)	0	Max	1000
VM5(Host2)	0	Max	1000
VM6(Host2)	0	Max	1000

Table 4: VM settings used for comparison of SRP versus PARDA+mClock

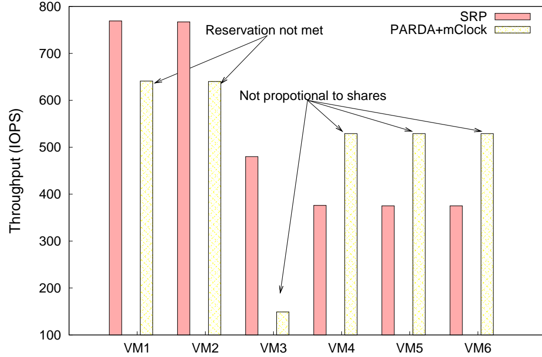


Figure 7: Comparison of throughput by SRP versus PARDA+mClock

even without the benefit of setting controls at the resource pool level, SRP is better than PARDA+mClock in two aspects. First PARDA+mClock could not satisfy VM reservations because PARDA adjusted the window sizes based solely on shares, while mClock tried to satisfy the reservation based on whatever window size was allocated by PARDA. This also showed that the local reservation penalized some VMs more than the others.

The second benefit of SRP is that when the settings are changed or when the workload changes in the VMs, SRP converges much faster than PARDA+mClock. We discuss each of these in more detail below.

Reservation Enforcement. We used six VMs running on two different ESX hosts. Per VM settings and VM-to-host mappings are shown in Table 4. We picked a simple case where a reservation of 750 IOPS was set for VMs 1 and 2. All the VMs have equal shares of 1000. VMs 1, 2 and 3 ran on host 1 and VMs 4, 5 and 6 ran on host 2. Both hosts ran PARDA and mClock initially. PARDA sees equal amount of shares on both hosts and allocates a host queue depth of 45 to both hosts. However, host 1 has two VMs with reservations of 750, and mClock tries to satisfy this by penalizing the third VM.

As seen in Figure 7, the third VM only gets 149 IOPS; the first two VMs are also not able to meet their reservations due to interference from host 2. When SRP is enabled in lieu of PARDA, it increases host 1's queue depth to 55 and reduces host 2's queue depth to 32. This enables VM1 and VM2 to meet their reservations and VM3 also gets IOPS that are much closer to VMs 4, 5 and 6. Thus with SRP the reservations are met and the other VMs get IOPS roughly in proportional to their shares.

Convergence Time. The response time to react to dynamic changes and converge to new settings is one of the critical performance factors in a distributed system. Quick convergence is usually desired to react to changes in a timely manner. In this section, we compare the convergence times of SRP to PARDA+mClock.

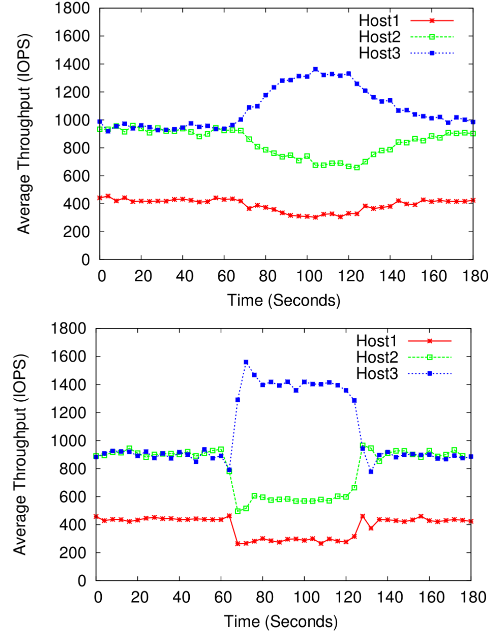


Figure 8: Average throughput of hosts while using (a) PARDA + mClock (top figure) and (b) SRP (bottom figure). SRP converges much faster as compared to PARDA+mClock.

To do the comparison, we ran VM1, VM2 and VM3 on separate hosts, with an initial shares ratio of 1 : 2 : 2. The VMs did not have reservations or limits set. Later the shares of VM3 were doubled at t = 60 second and reduced by half at t = 120 second. Each VM was running Windows IOmeter with the same configuration of 4KB, 100% random read, and 32 outstanding IOs at all times. Figure 8 shows the average throughput for the hosts using PARDA+mClock (top) and SRP (bottom) respectively. In general, both of the algorithms achieve resource sharing in proportion to their shares.

PARDA takes much longer (30 seconds) to converge to the new queue depth settings as compared to SRP (8 seconds). This is because PARDA runs a local control equation to react to global changes, where as SRP does a quick divvy of overall queue depth.

5.3 Enterprise Workloads

Next we tested storage resource pools for more realistic enterprise workloads with bursty arrivals, variable locality of reference, variable IO sizes and variable demand. We used four different workloads for this experiment: Filebench-Mail, Filebench-Webserver, Filebench-Oltp and DVDStore.

The first three workloads are based on different personalities of the well-known Filebench workload emulator [18], and the fourth workload is based on DVDStore [2] database test suite, which represents a complete online e-commerce application running on SQL database. For each VM, we used one OS disk and one or more data disks. These eight VMs, with a total of nineteen virtual disks, were spread across three ESX hosts accessing the same underlying device using VMFS.

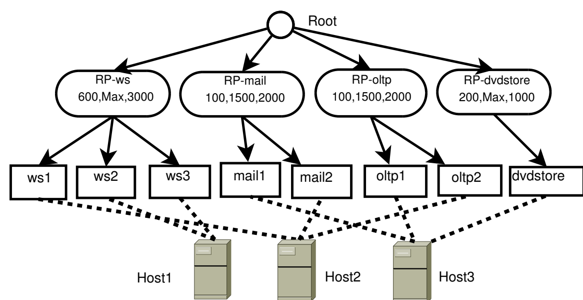


Figure 9: SRP configuration for Enterprise workloads

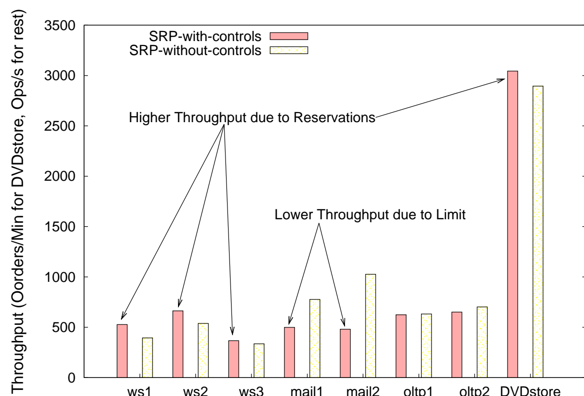


Figure 10: Comparison of application level throughput with and without SRP

For this experiment, we partitioned these four workloads into four different resource pools. These pools are called RP-mail, RP-oltp, RP-ws and RP-dvdstore respectively. RP-mail contains two VMs running the mail server workload, RP-ws contains three VMs running the webserver workload, RP-oltp contains two VMs running the oltp workload and RP-dvdstore contains one VM running the DVDStore workload.

Figure 9 shows the settings for these resource pools and the individual VMs. First we ran these workloads with no reservation, infinite limit and equal shares. Then we set reservations and limit at the pool level to favor some workloads over others. The VM-level settings were unchanged. We set a reservation of 600 IOPS for webserver pool (RP-ws) and a reservation of 200 IOPS for RP-dvdstore. This reflects the user's concern that these workloads have a smaller latency. We also set a limit of 1500 IOPS for both RP-mail and RP-oltp pools. This is done to contain the impact of these very bursty VMs on others. We ran all these workloads together on the same underlying Equallogic datastore for 30 minutes, once for each setting.

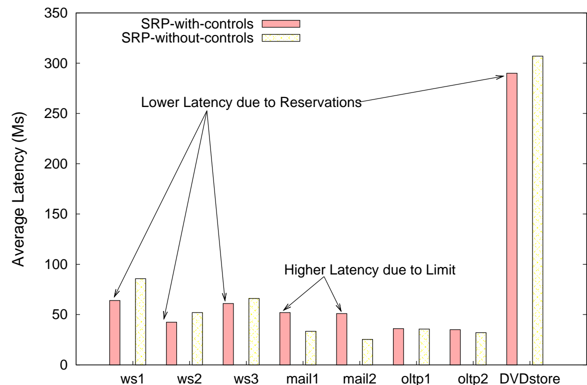


Figure 11: Comparison of application level average latency with and without SRP

Figures 10 and 11 show the overall application-level throughput in terms of Ops/sec (orders/min in case of DVDStore) and application-level average latency (in ms) for all VMs. Since we had set a reservation on the webserver and dvdstore pools, those VMs got lower latency and higher IOPS compared to other VMs. On the other hand the mail server VMs got higher latency because their aggregate demand is higher than the limit of 1500 IOPS. Interestingly, the effect on oltp VMs was much smaller because their overall demand is close to 1500 IOPS, so the limit didn't have as much of an impact.

This shows that by setting controls at the resource pool level, one can effectively isolate the workloads from one another. An advantage of setting controls at the resource pool level is that one doesn't have to worry about per VM controls, and VMs within a pool can take advantage of each other's idleness.

6 Conclusions

In this paper we studied the problem of doing hierarchical IO resource allocation in a distributed environment, where VMs running across multiple hosts are accessing the same underlying storage. We propose a novel and powerful abstraction called *storage resource pools* (SRP),

which allows setting of rich resource controls such as IO reservations, limits and proportional shares at VM or pool level. SRP does a two-level allocation by controlling per host queue depth and computing dynamic resource controls for VMs based on their workload demand using a resource divvying algorithm.

We implemented storage resource pools in VMware ESX hypervisor. Our evaluation with a diverse set of workloads shows that storage resource pools can guarantee high utilization of resources, while providing strong performance isolation for VMs in different resource pools. As future work, we plan to automate the resource control settings in order to provide application level SLOs and test our approach on multi-tiered storage devices.

Acknowledgments

We would like to thank Jyothir Ramanan, Irfan Ahmad, Murali Vilyannur, Chethan Kumar, and members of the DRS team, for discussions and help in setting up our test environment. We are also very grateful to our shepherd Rohit Chandra and anonymous reviewers for insightful comments and suggestions. We would also like to thank Bala Kaushik, Naveen Nagaraj, and Thuan Pham for discusions and motivating us to work on these problems.

References

[1] CLEMENTS, A. T., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized Deduplication in SAN Cluster File Systems. In *Usenix ATC '09*.

[2] DELL. DVD Store. `http://www.delltechcenter.com/page/DVD+store`.

[3] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience 1*, 1 (September 1990), 3–26.

[4] GOYAL, P., GUO, X., AND VIN, H. M. A hierarchical cpu scheduler for multimedia operating systems. In *Usenix OSDI* (January 1996).

[5] GOYAL, P., VIN, H. M., AND CHENG, H. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. Tech. Rep. CS-TR-96-02, UT Austin, January 1996.

[6] GULATI, A., AHMAD, I., AND WALDSPURGER, C. PARDA: Proportionate Allocation of Resources for Distributed Storage Access. In *Usenix FAST '09* (Feb. 2009).

[7] GULATI, A., MERCHANT, A., AND VARMAN, P. *p*Clock: An arrival curve based approach for QoS in shared storage systems. In *Proc. of ACM SIGMETRICS* (June 2007), pp. 13–24.

[8] GULATI, A., MERCHANT, A., AND VARMAN, P. J. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Usenix OSDI'10* (Oct. 2010).

[9] GULATI, A., SHANMUGANATHAN, G., AHMAD, I., WALDSPURGER, C., AND UYSAL, M. PESTO: online Storage Performance Management in Virtualized Datacenters. In *Symposium on Cloud Computing (SOCC '11)* (Oct. 2011).

[10] HUANG, L., PENG, G., AND CKER CHIUEH, T. Multi-dimensional storage virtualization. In *ACM SIGMETRICS* (June 2004).

[11] JIN, C., WEI, D., AND LOW, S. FAST TCP: Motivation, Architecture, Algorithms, Performance. *Proceedings of IEEE INFOCOM* (March 2004).

[12] JIN, W., CHASE, J. S., AND KAUR, J. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS* (June 2004), pp. 37–48.

[13] KARLSSON, M., KARAMANOLIS, C., AND ZHU, X. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage 1*, 4 (2005), 457–480.

[14] LITTLE, J. D. C. A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research 9*, 3 (1961).

[15] LUMB, C., MERCHANT, A., AND ALVAREZ, G. Façade: Virtual storage devices with performance guarantees. *Usenix FAST* (March 2003).

[16] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T., AND MALTZAHN, C. Guaranteed Disk Request Scheduling with Fahrrad. In *EUROSYS* (March 2008).

[17] STOICA, I., ABDEL-WAHAB, H., AND JEFFAY, K. On the duality between resource reservation and proportional share resource allocation. *Multimedia Computing and Networking 3020* (1997), 207–214.

[18] SUN MICROSYSTEMS. Filebench Benchamarking Tool. `http://solarisinternals.com/si/tools/filebench/index.php`.

[19] VMWARE, INC. *Introduction to VMware Infrastructure*. 2007. http://www.vmware.com/support/pubs/.

[20] VMWARE, INC. *vSphere Resource Management Guide: ESX 4.1, ESXi 4.1, vCenter Server 4.1*. 2010.

[21] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: performance insulation for shared storage servers. In *FAST'07* (2007), pp. 5–5.

[22] WALDSPURGER, C. A. Memory Resource Management in VMware ESX Server. In *Usenix OSDI* (Dec. 2002).

[23] WANG, Y., AND MERCHANT, A. Proportional-share scheduling for distributed storage systems. In *Usenix FAST'07*.

[24] WONG, T. M., GOLDING, R. A., LIN, C., AND BECKER-SZENDY, R. A. Zygaria: Storage performance as managed resource. In *Proc. of RTAS* (April 2006), pp. 125–34.

[25] ZHANG, J., SIVASUBRAMANIAM, A., WANG, Q., RISKA, A., AND RIEDEL, E. Storage performance virtualization via throughput and latency control. In *Proc. of MASCOTS* (Sep 2005).