Demand Based Hierarchical QoS Using Storage Resource Pools

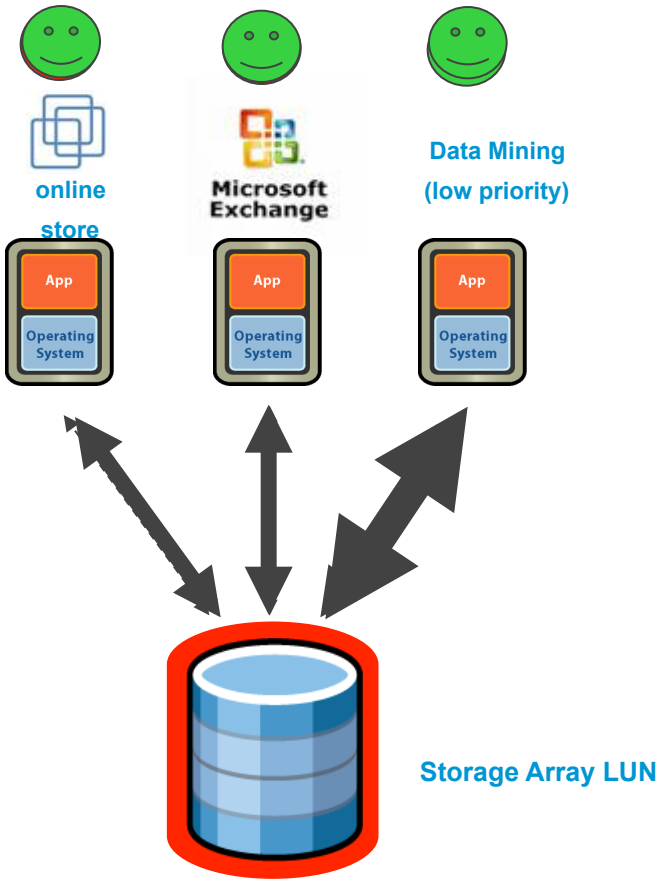
Ajay Gulati, *Ganesha Shanmuganathan*, *Xuechen Zhang**, *Peter Varman*[+]

Distributed Resource Management Team

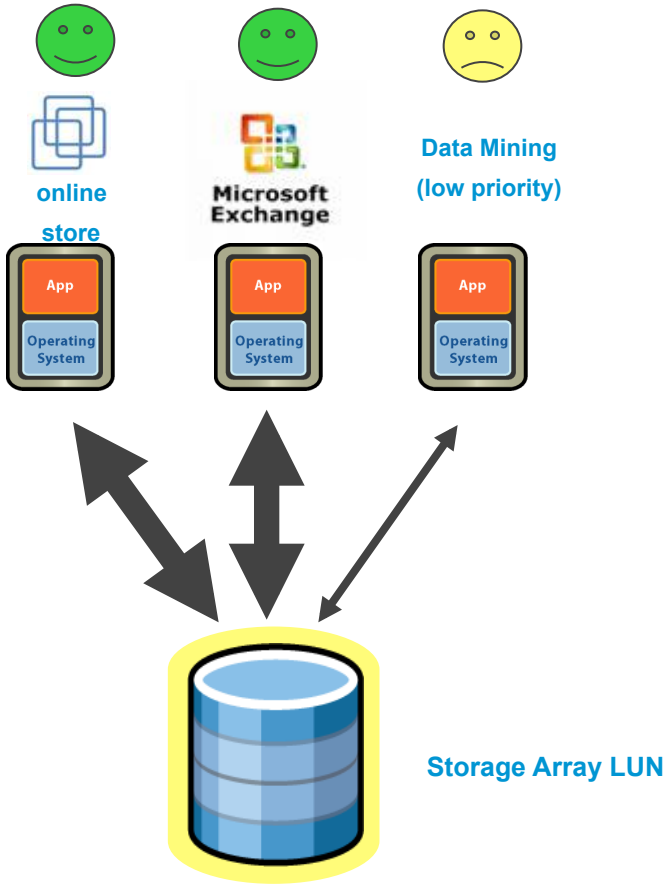
*VMware Inc., *Wayne State University, [+]Rice University*

The Problem

What you see

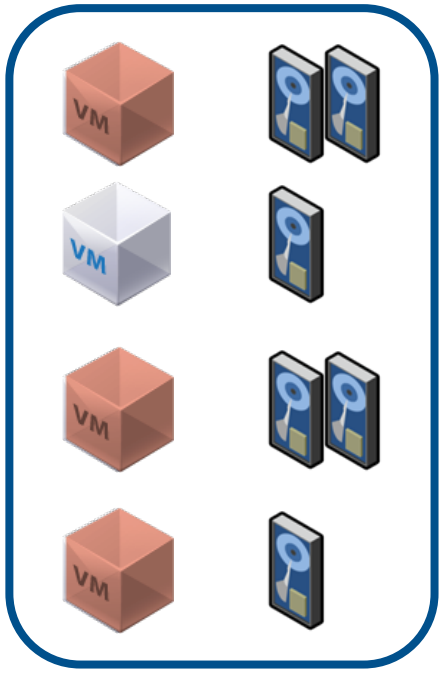


What you want to see

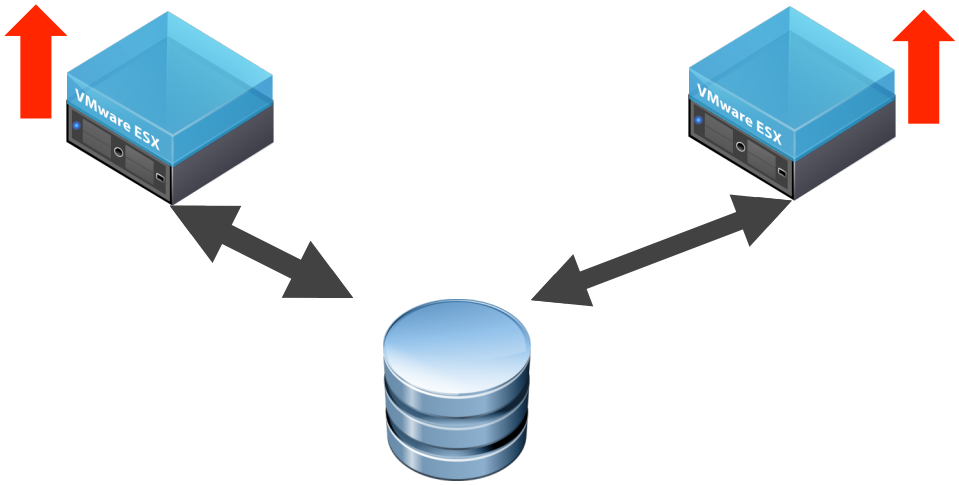


The Problem

- Existing solutions: specify per VM QoS (weights, IOPS or …)
- Sum of per-VM requirement may be < total application requirement



A single virtual App



- **Challenges:**
  - Hard to set per virtual disk value
  - Need support for statistical-multiplexing
  - De-centralized solution needed

Resource Allocation Models

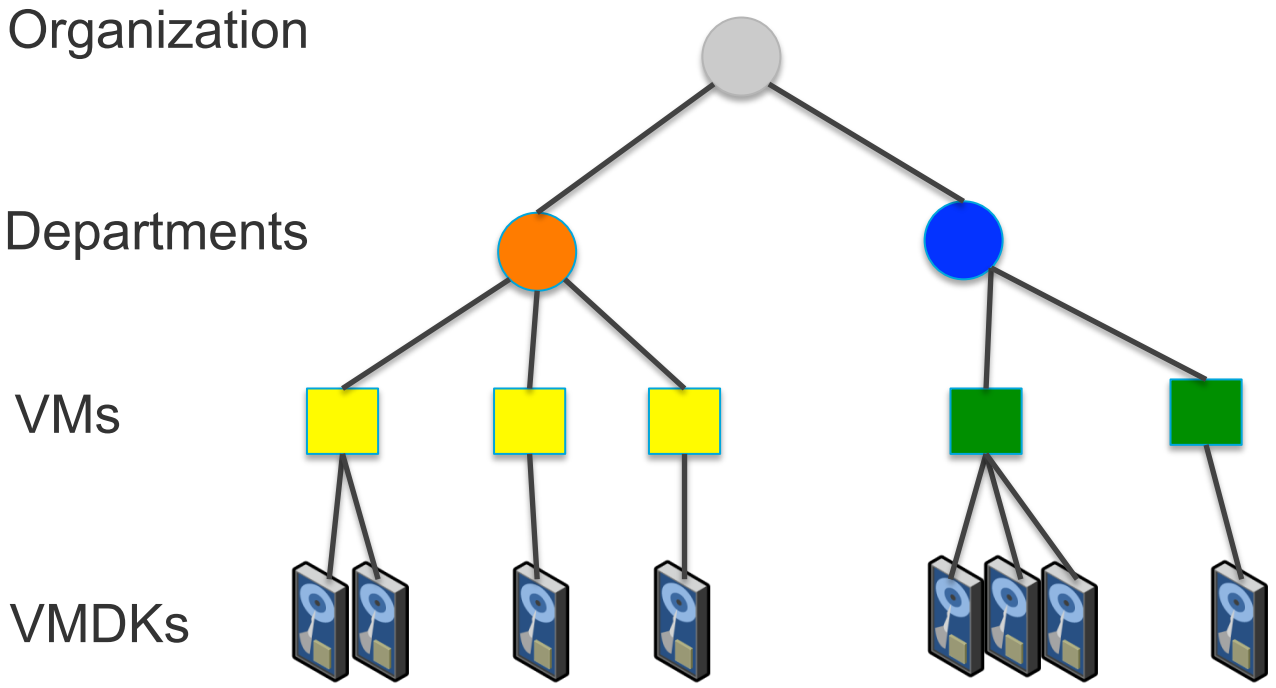
- Various models proposed in literature
- Assuming three clients: 1, 2 and 3

Weights	Latency	Weights, Latency, Burst	Weights, Reservation, Limit
W_1	Lat_1	W_1, Lat_1, B_1	W_1, R_1, L_1
W_2	Lat_2	W_2, Lat_2, B_2	W_2, R_2, L_2
W_3	Lat_3	W_3, Lat_3, B_3	W_3, R_3, L_3

- Weights: *SFQ(D), Stonehedge, Argon, Zygaria, PARDA, …*
- Latency: *Façade, SARC+Avatar*
- Weights, Latency, Burst: *pClock*
- Weights, R, L: *mClock (single host)*

What is a Storage Resource Pool

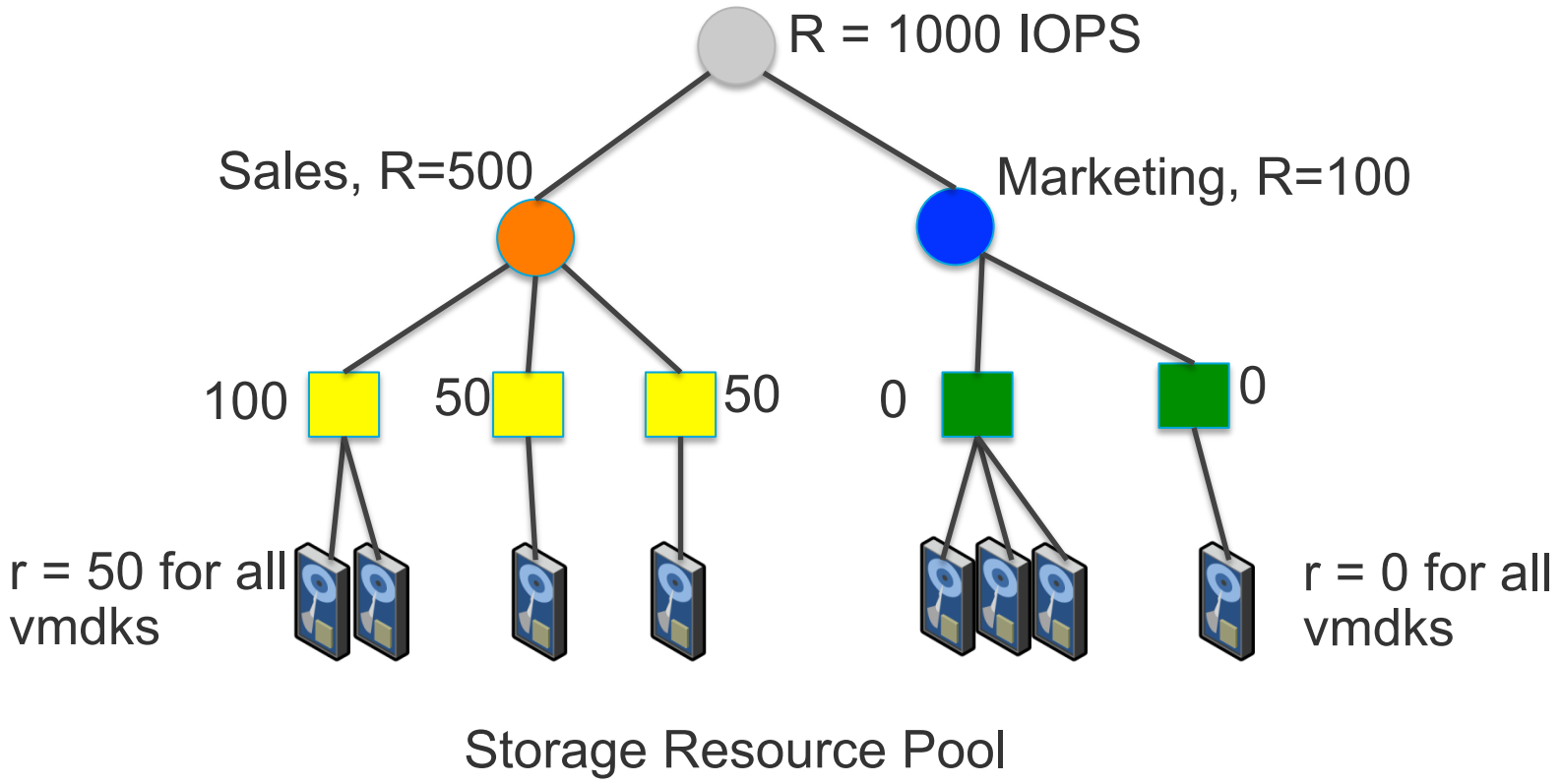
- A powerful abstraction for aggregation and isolation of IO resources
- Customer creates a resource pool and sets business requirements
- Sharing within a pool, isolation across pools



Storage Resource Pool

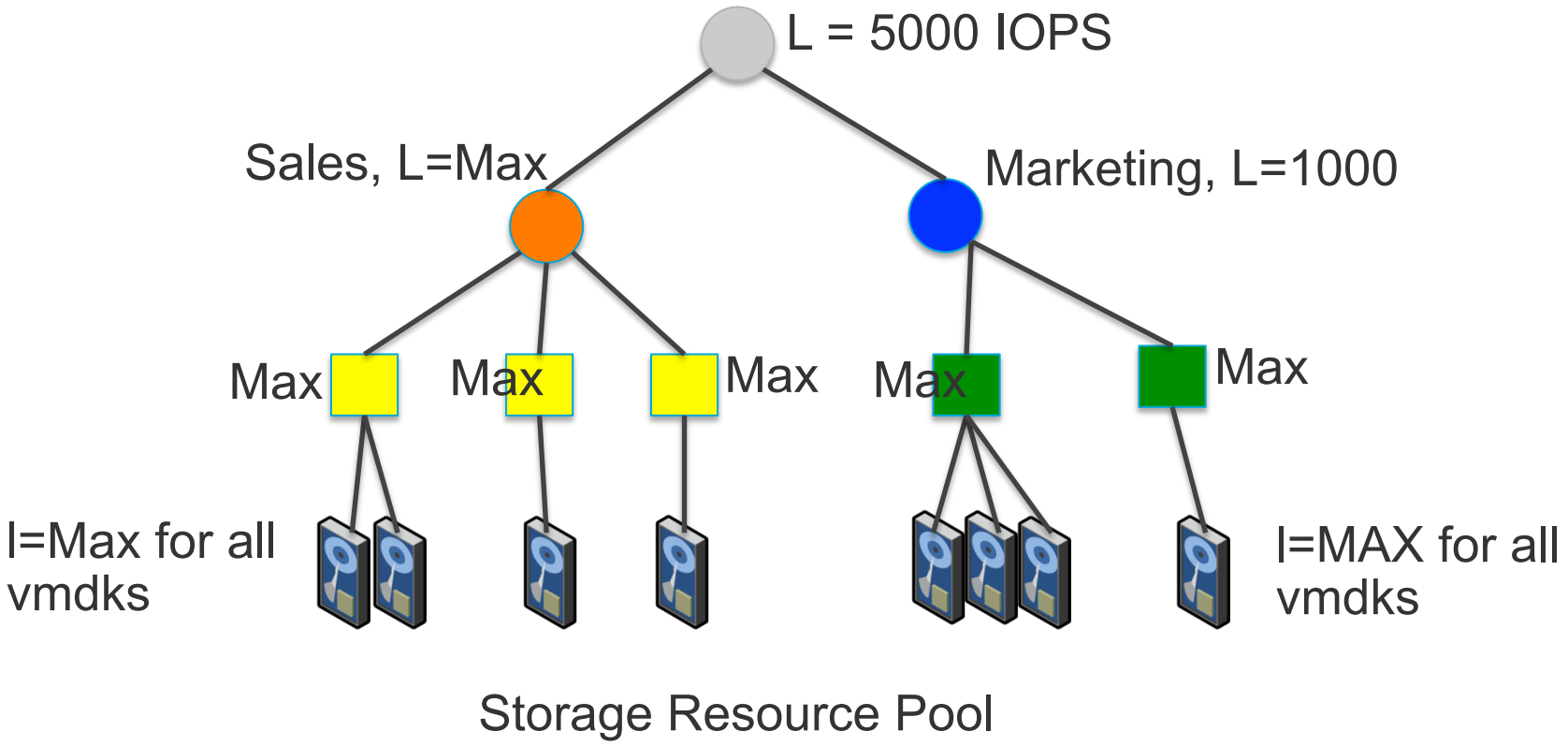
Controls: Reservation (IOPS)

- **Minimum IOPS guaranteed**
- **We need to distribute the R value hierarchically *using current demand***



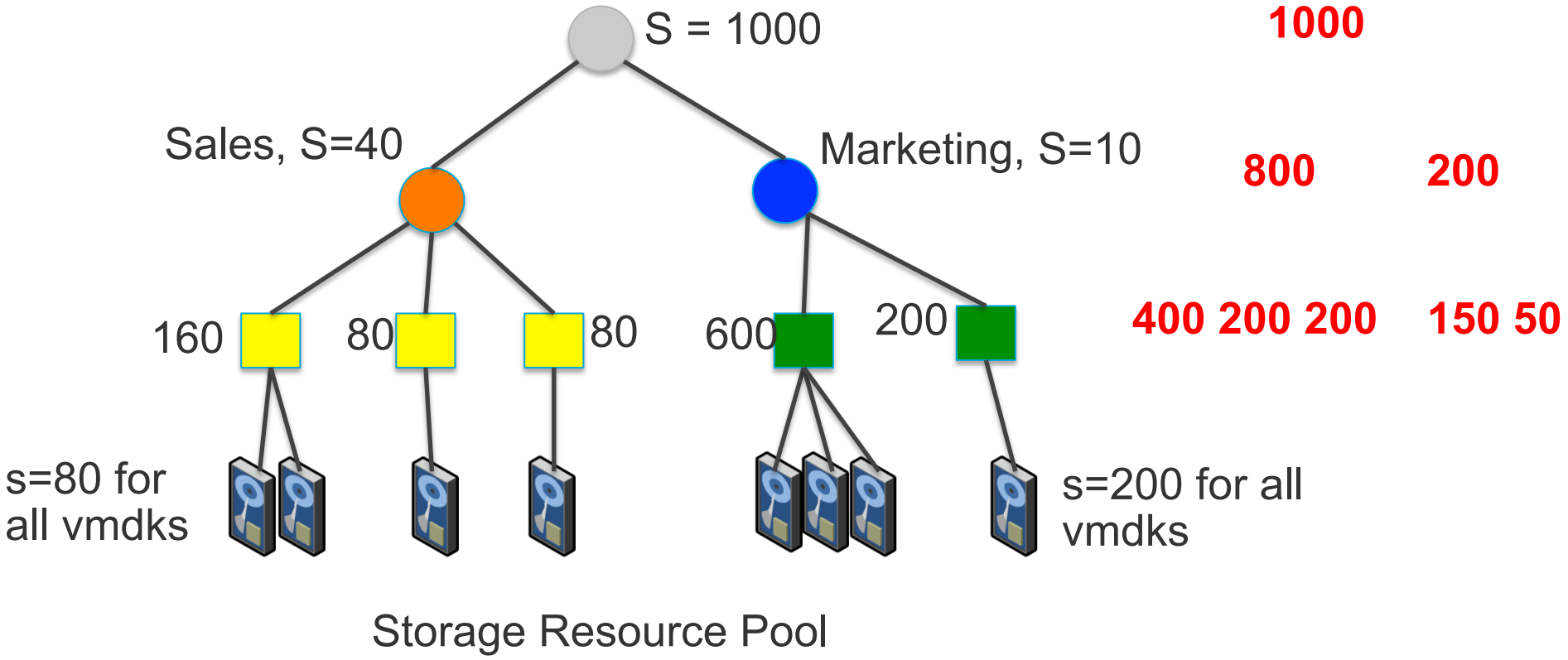
Controls: Limit (IOPS)

- **Maximum IOPS allowed**
- **We need to set children limits based on parent's L value, *using current demand***



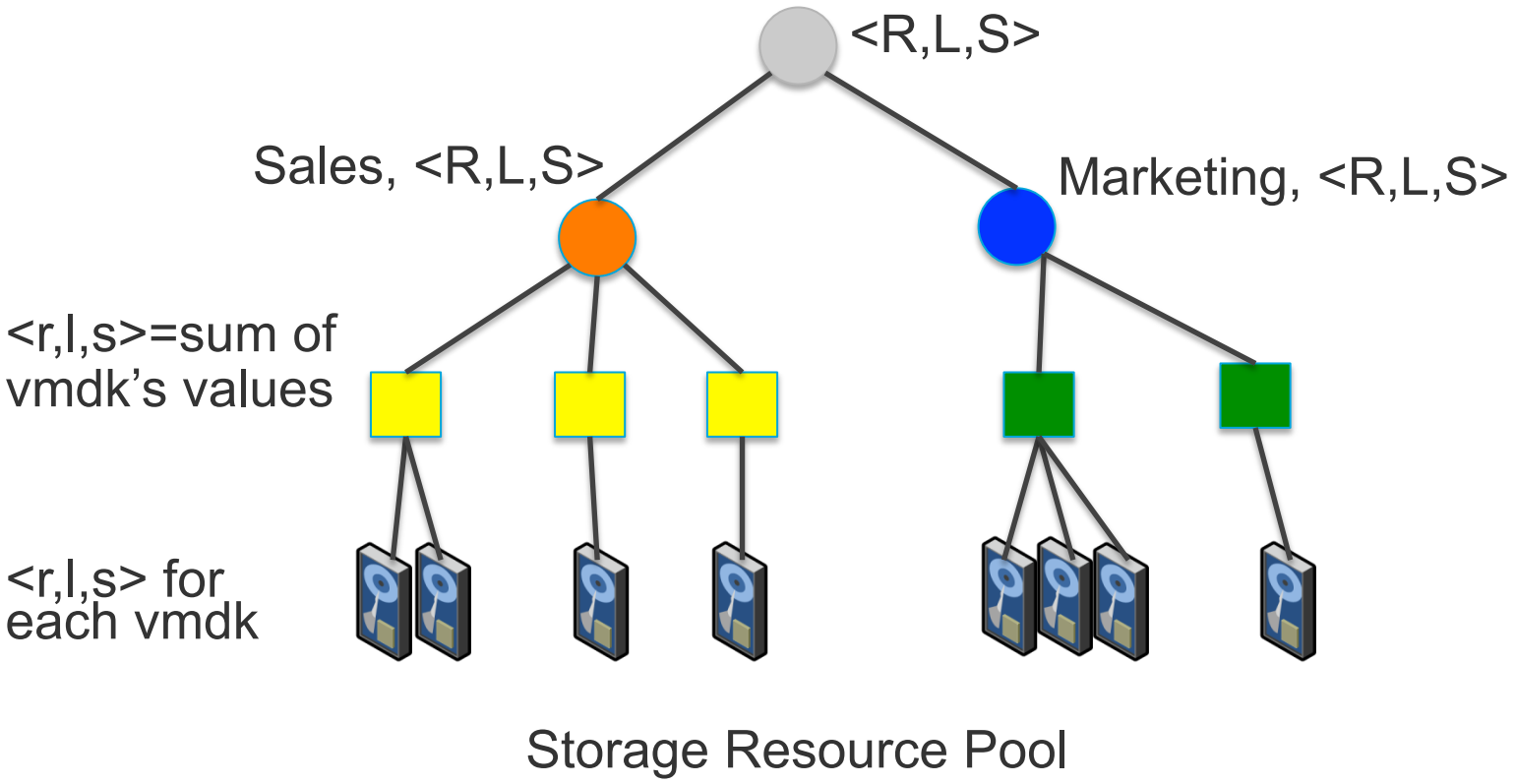
Controls: Shares (aka Weights) (No Units)

- Relative priority between siblings in the tree
- Need to divide parent shares among children
- Straight-forward and not based on demand



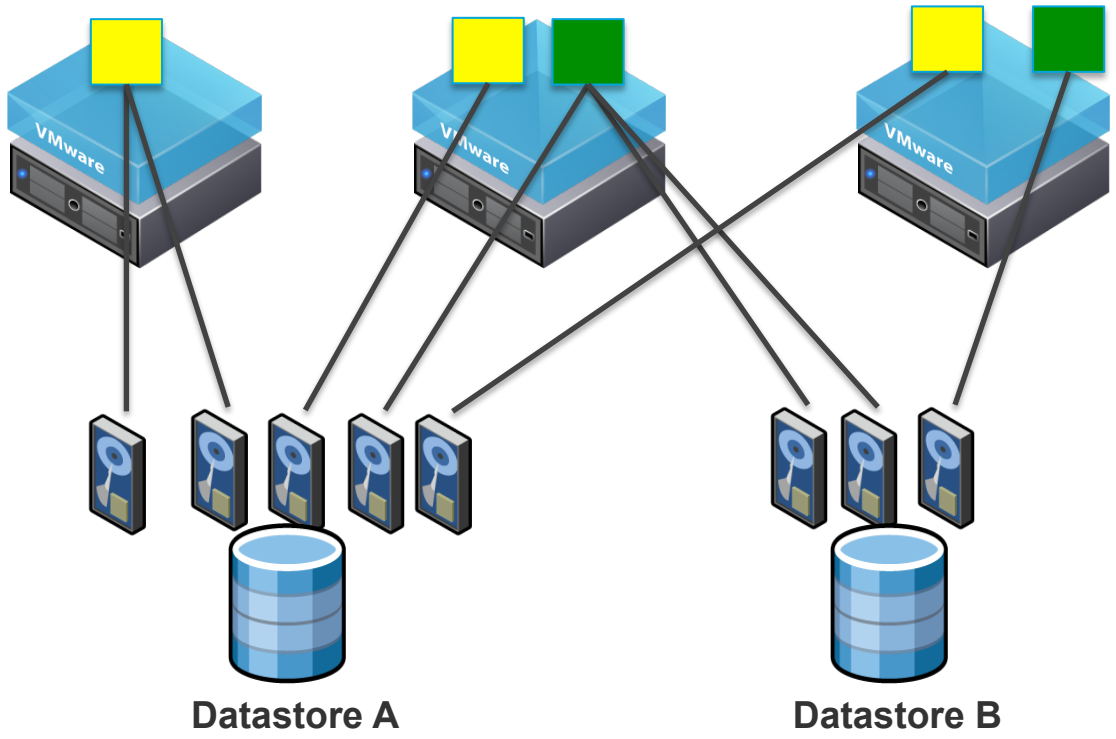
Virtual View of the World

- Storage resource pool as specified by user
- In practice, life isn't so simple!



Physical View of the World

- VMs would get placed on different hosts
- VMDKs get placed on different datastores



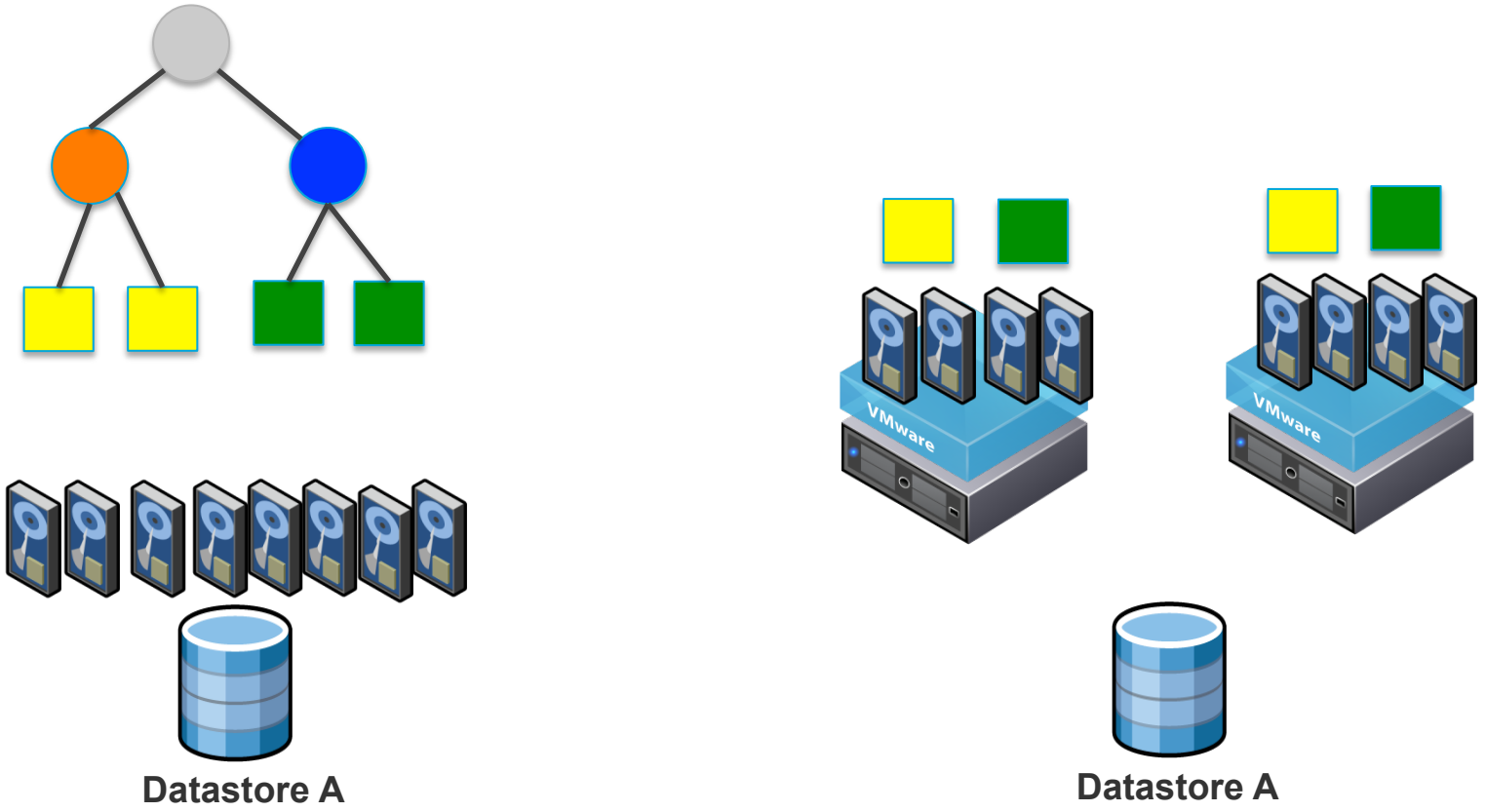
What Do we need to Support Storage Resource Pools

- If (Parent R > Sum of children R)
  We need to distribute spare R among children
- If (Parent L < Sum of children L)
  We need to restrict children to parent limit
- Distribute parent's shares among children
- Use current demand of children for dynamic allocation

- These tasks are done periodically

Per Datastore RP Tree

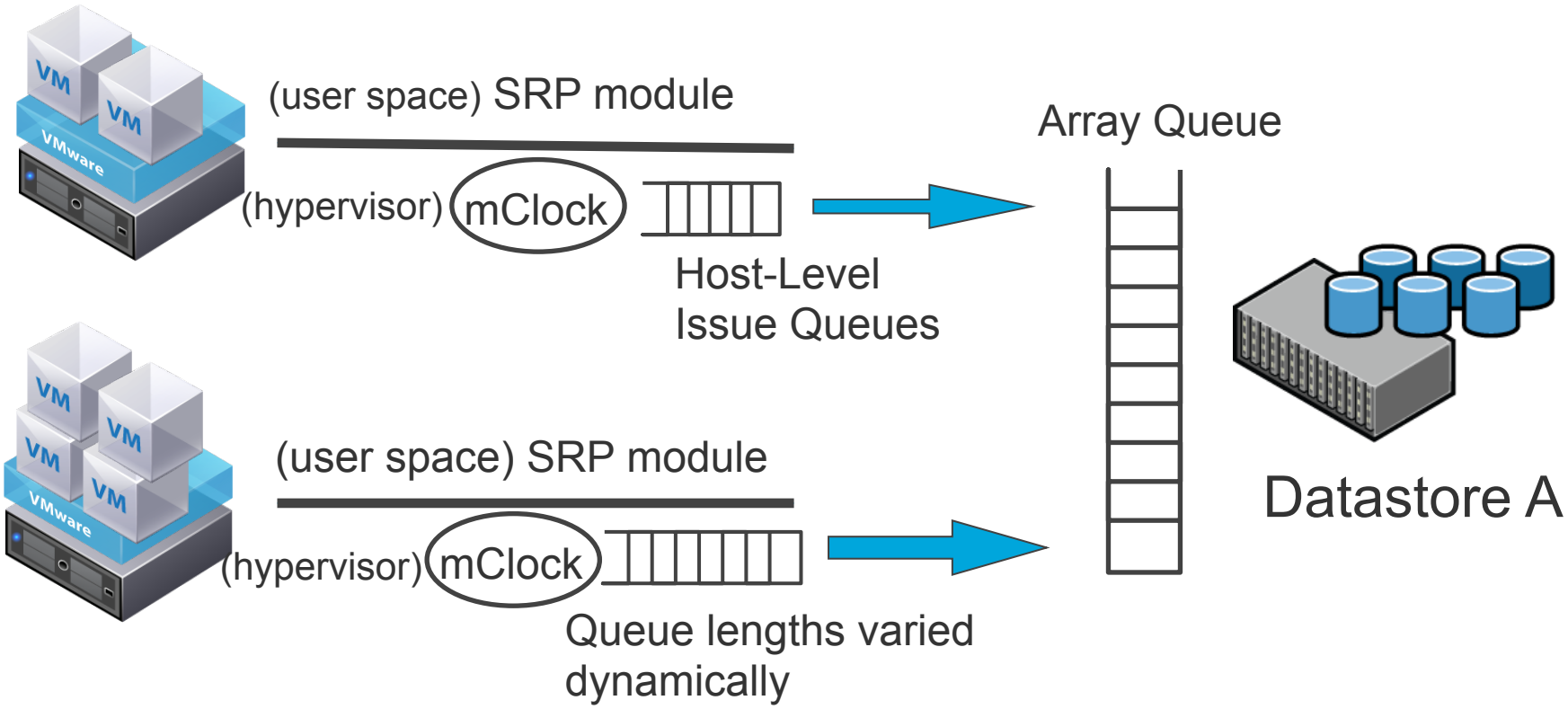
- Focus on per datastore tree
- A tree spanning multiple devices can be broken up across devices
- Need to handle distributed setting with multiple hosts



System Architecture

- **Two-level scheduling:**

1. **Per host LUN queue depth**
2. **Per VM local IO scheduling using mClock that support R, L, S**



Storage Resource Pools

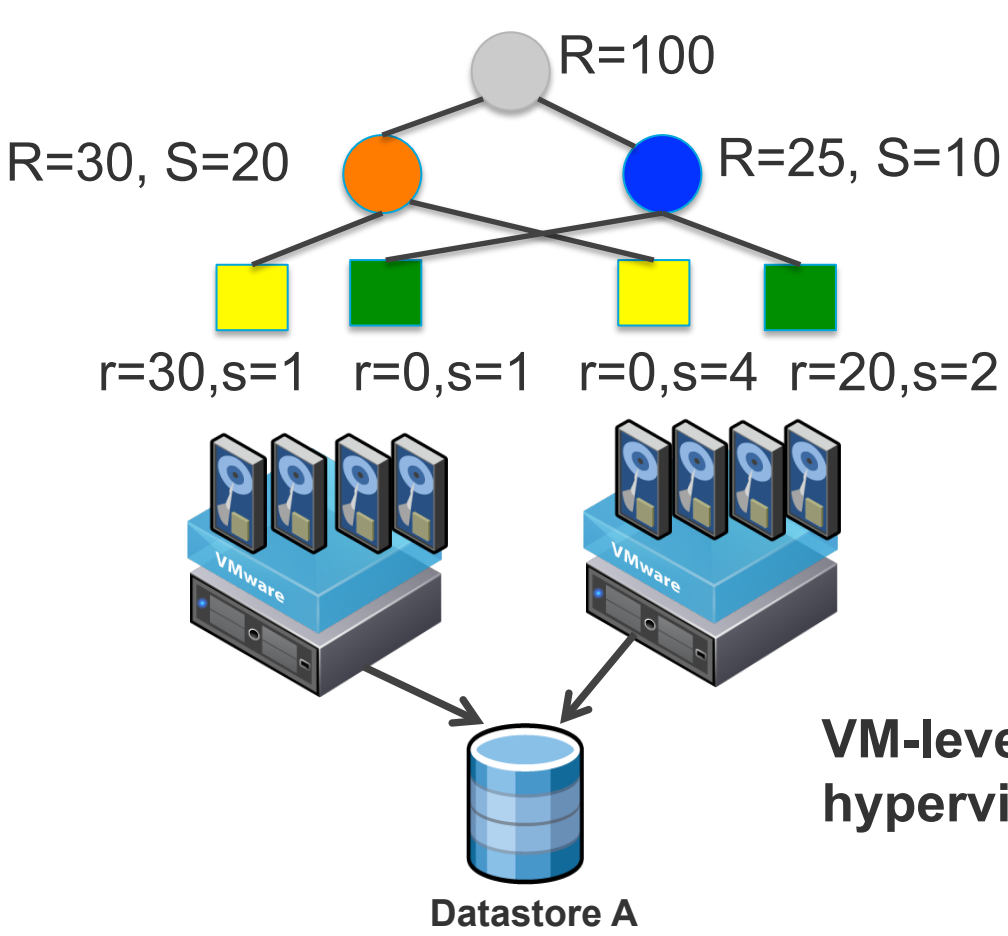
- Two main steps:

1. Compute dynamic R,L,S settings per VM based on demand
2. Adjust per host LUN queue depth for host-level differentiation

- These adjustments are done every few seconds
  (4 sec in our prototype)

Step 1: Per VM Settings

- Compute per VM R, L, S using divvy algorithm on the tree
- Set the controls for local VMs on each host



100

66 33

30 13 36 20

VM-level values are set in the hypervisor for mClock

Step 2: Host-level QueueDepth Setting

- **Compute current array capacity (in terms of total queue depth)**
- **Allocate that to hosts based on VMs running there**
  - Use queue depth per host as the key control
- **Per VM allocation on a host is handled by local scheduler (mClock)**

Step 2: How to compute total QueueDepth?

- **Compute total queue depth**

$$Q(t+1) = (1-\gamma)Q(t) + \gamma\left(\frac{\mathcal{L}}{L(t)}Q(t)\right)$$

Compute queue size ***Q(t)*** using cluster-wide average latency ***L(t)***

$\mathcal{L}$: congestion threshold, operating point for IO latency

γ : smoothing parameter between 0 and 1

Note: Q increases if latency is lower than congestion threshold, decreases otherwise

Step 2: Host-level QueueDepth Setting

- **Compute current array capacity**
- **Allocate that to hosts based on VMs running there**
  - Use queue depth per host as the key control
- **Per VM allocation on a host is handled by local scheduler (mClock)**

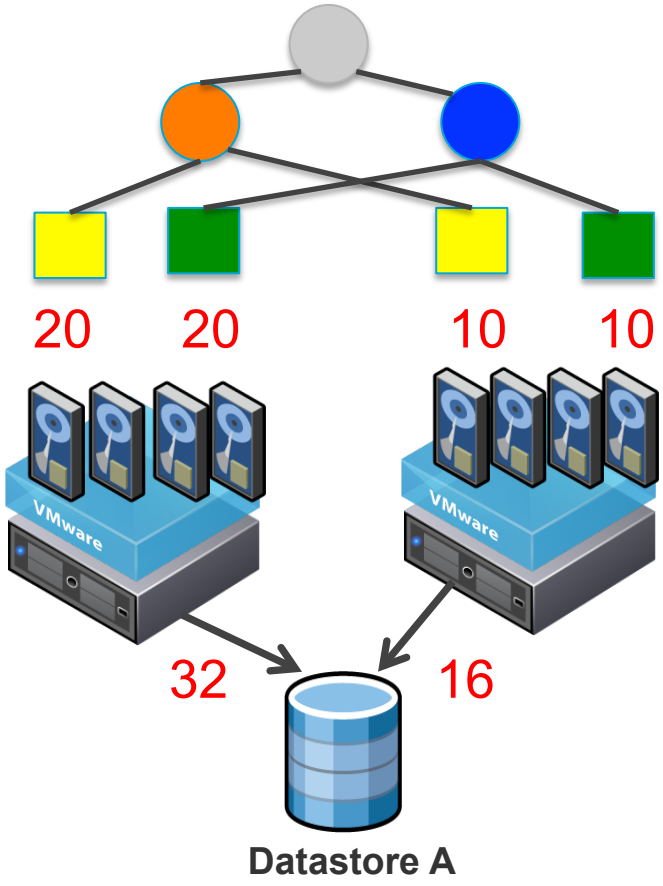
Example:

VM Shares = 20 20 10 10

R= 0 for all

L= Max for all

No idle VMs



Host Queues: 32 16

Lets say total QueueDepth = 48

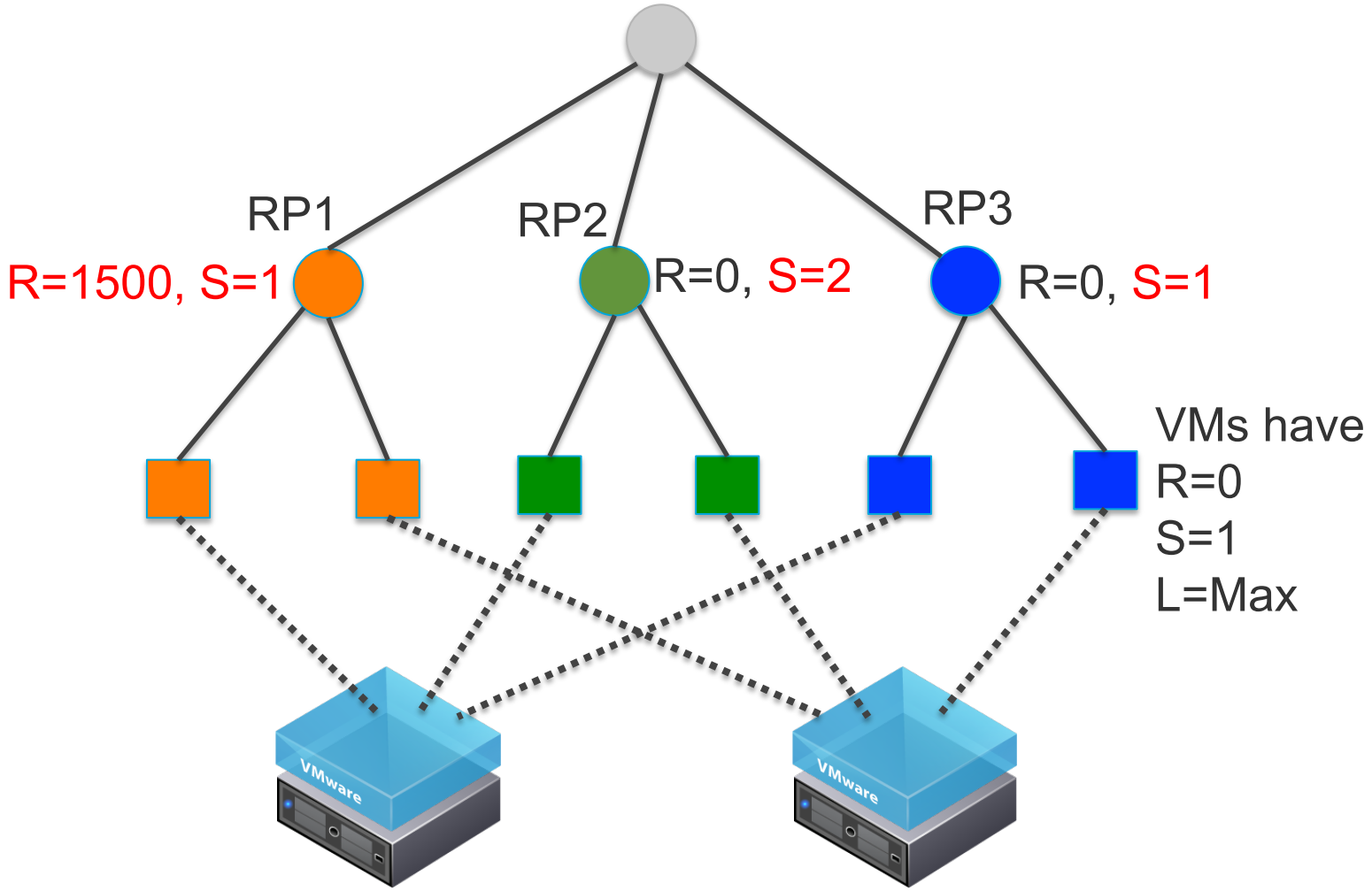
Storage-specific Practical Issues

- **How to set R value at root?**
  - For disk-based devices, we use the random IO performance as upper bound
  - For SSD-based devices, we can use a fraction of total IOPS performance using some read-write ratio
- **Handling of IO sizes**
  - IOPS can be specified for specific IO size range (<= 32KB)
  - Higher IO sizes will lead to lower values
- **Sequential vs. Random workloads**
  - mClock optimizes for sequential workload by using batching
  - Typically virtualization workloads are random due to IO-blending, thin-provisioning, de-duplication etc.

Experimental Setup

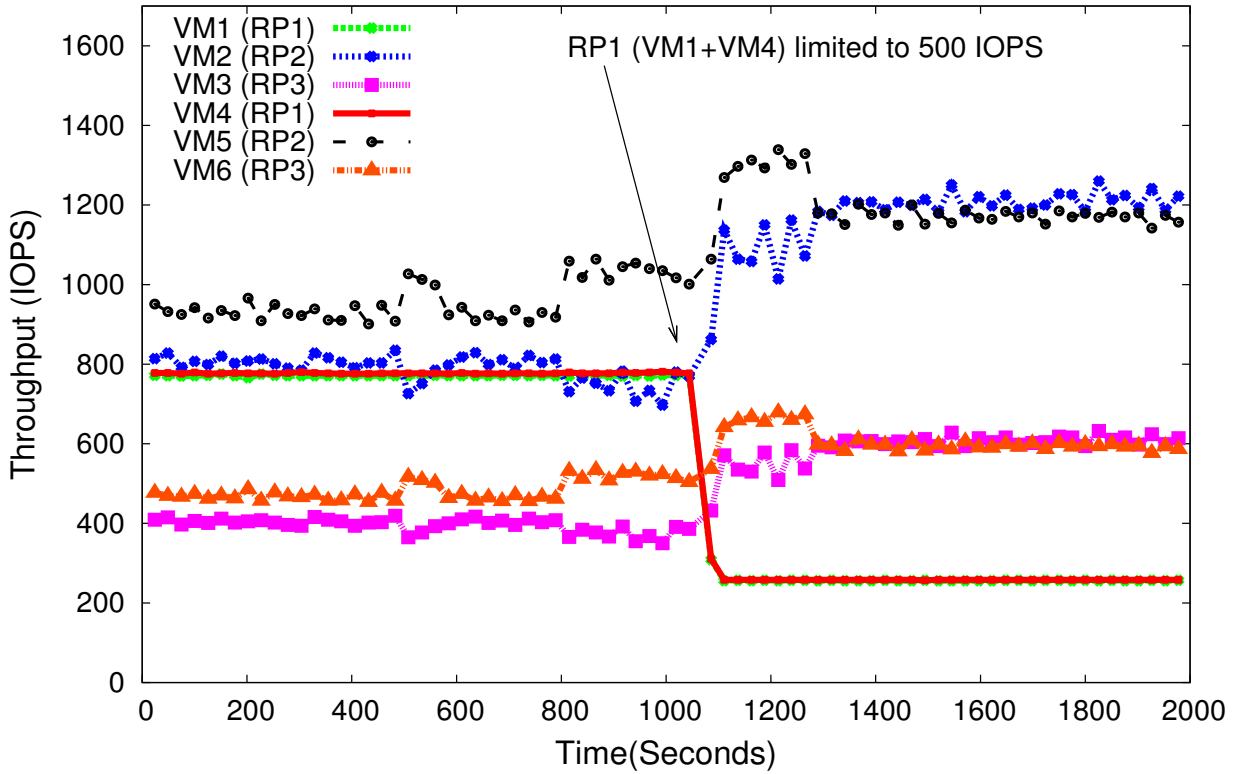
- **Built a prototype on ESX hypervisor**
  - User-space module + kernel changes
- **We use multiple ESX hosts**
- **Experiment 1:**
  - Six Windows Server 2003 VMs running Iometer
  - Each VM has 4 GB OS disk and 8 GB experimental disk
  - Each VM is running a different workload profile
- **Experiment 2:**
  - Eight VMs running different enterprise workloads: webserver, mail, oltp, DvdStore

Experiment 1 Setup



Case 1: High Reservation At Pool Level

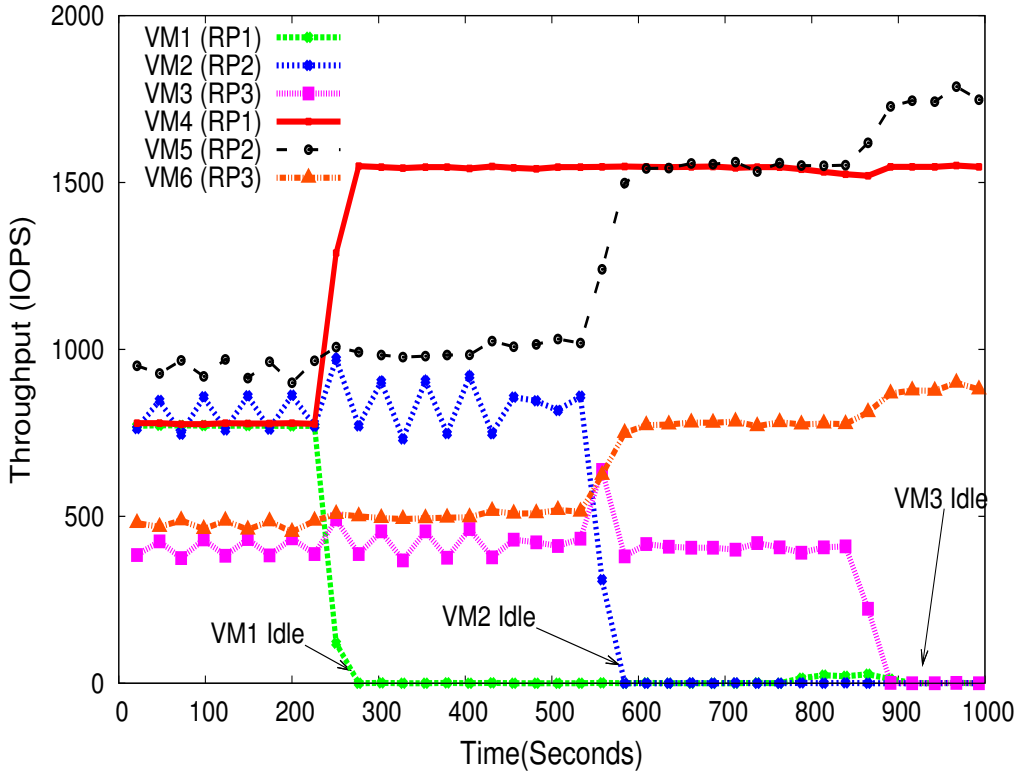
- At t=1000 sec, RP1 settings are changed to **R=0 and L = 500**



Reservations & Limit controls are enforced at pool level
Remaining capacity allocated in proportion of shares (2:1)

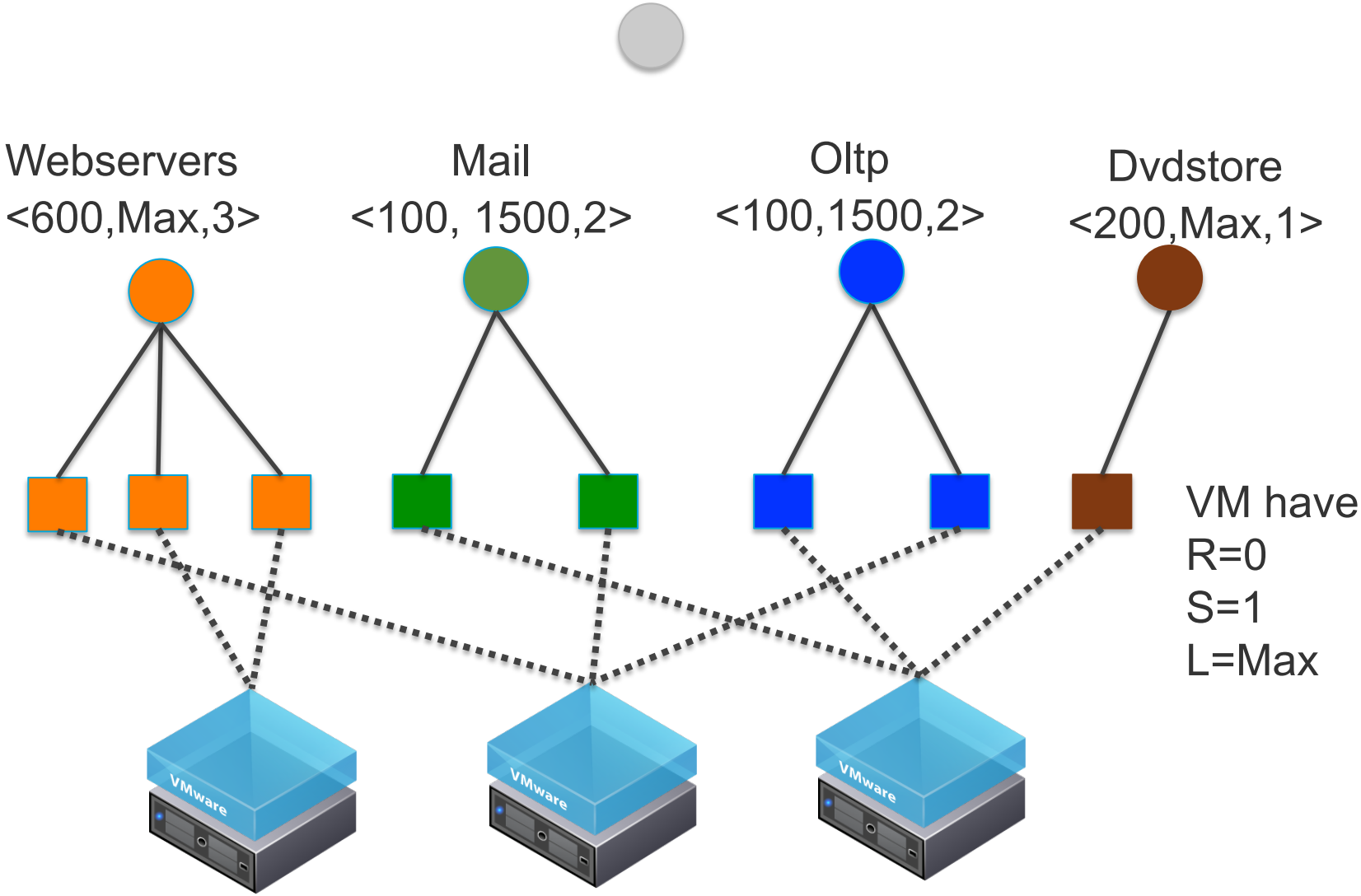
Case 2: Allocation Within a Pool

- VM1, 2 and 3 get idle at t=250, 500 and 750 seconds respectively



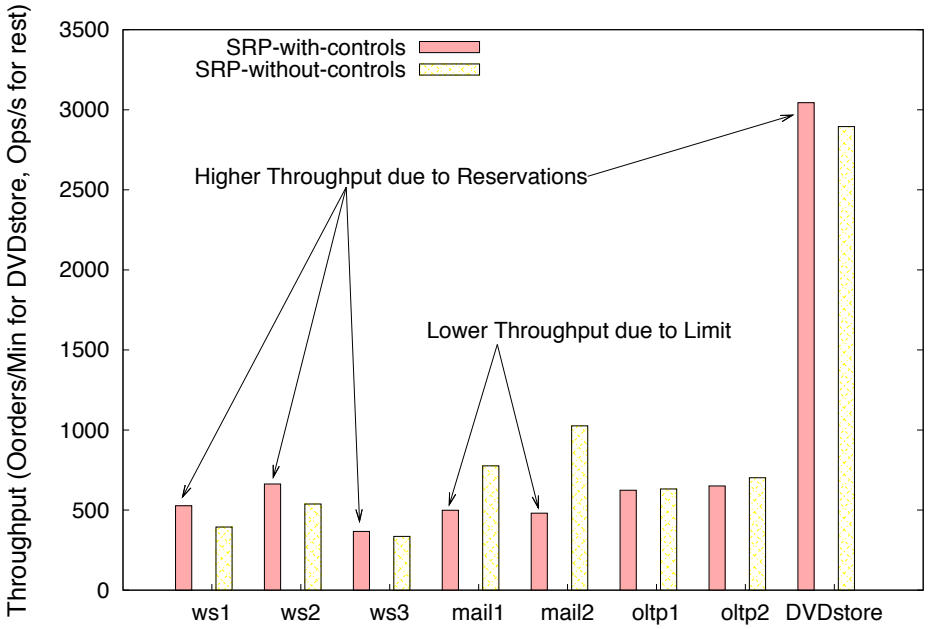
Capacity is allocated to VMs within the same pool
VM4 (in the same RP) get R=1500

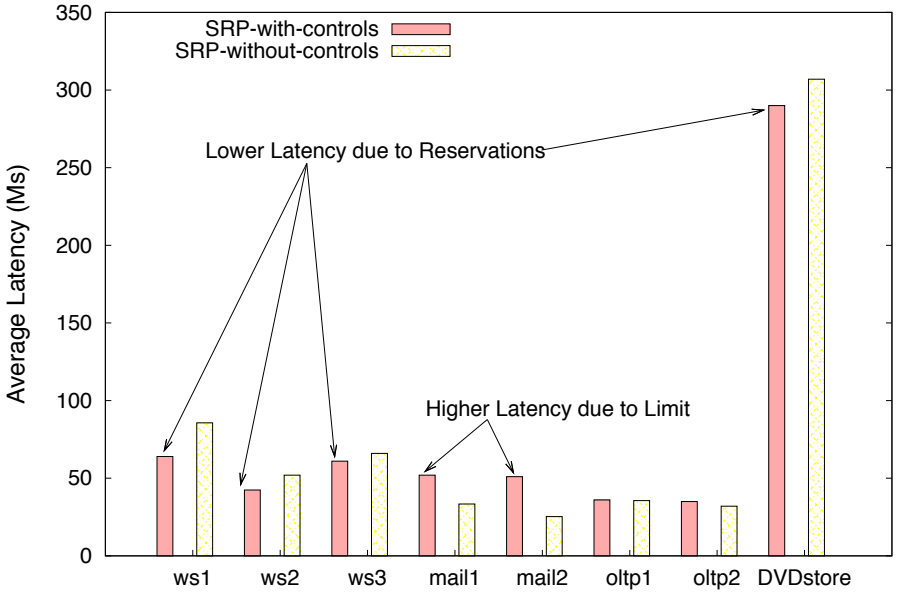
Experiment 2 Setup



Experiment 2: Results

- We ran this with & without SRPs (R=1,L=Max and S=1, for all)





Setting pool level controls helps VMs receive higher/lower performance

Summary

- **Storage resource pools provide a powerful abstraction to the user**
- **SRPs are able to provide isolation between pools & sharing within pools**
- **SRPs enforces controls across hosts in a distributed manner**

- **Future work:**
  - Automatically set R,L,S values based on latency requirements or other input
  - Test on SDD-based and multi-tiered storage devices

Thank You!

Q & A

Backup slides

Implementation Details

- SRP is implemented as user-world on ESX
- Steps followed by SRP module:

1. Publish per-VM demands on a shared file
2. Read this shared file and aggregate VM demand up the tree

3. Do R, L, S, queue depth divvy on the full tree
4. Set R,L,S values of local VMs in mClock

5. Compute “array queue depth” using the control equation
6. Set host level LUN queue depth using queue depth divvy

How to do QueueDepth divvy?

- Convert Q(t+1) to IOPS using Little's law:

$$Qiops(t+1) = \frac{Q(t+1)}{\mathcal{L}}$$

- Divvy ***Qiops*** using R,L,S and demand across the tree
- Let VM entitlement E_i = divvied value

$$Q_{host}(t+1) = \frac{\sum_{VMsonhost} E_i}{Qiops} * Q(t+1)$$

- Host queue depth is set in proportion to its shares of VM entitlements

Break Cluster-level Tree into per Datastore Tree

- Customers specify resource pools at datastore cluster level
- SDRS can periodically splits them into per datastore resource pools