Server-Storage Virtualization: Integration and Load Balancing in Data Centers

Aameek Singh
IBM Almaden Research Center
Email: Aameek.Singh@us.ibm.com

Madhukar Korupolu
IBM Almaden Research Center
Email: madhukar@us.ibm.com

Dushmanta Mohapatra
Georgia Tech
Email: dmpatra@cc.gatech.edu

***Abstract*—We describe the design of an *agile* data center with integrated server and storage virtualization technologies. Such data centers form a key building block for new cloud computing architectures. We also show how to leverage this integrated agility for non-disruptive load balancing in data centers across multiple resource layers - servers, switches, and storage. We propose a novel load balancing algorithm called VectorDot for handling the hierarchical and multi-dimensional resource constraints in such systems. The algorithm, inspired by the successful Toyoda method for multi-dimensional knapsacks, is the first of its kind.**

We evaluate our system on a range of synthetic and real data center testbeds comprising of VMware ESX servers, IBM SAN Volume Controller, Cisco and Brocade switches. Experiments under varied conditions demonstrate the end-to-end validity of our system and the ability of VectorDot to efficiently remove overloads on server, switch and storage nodes.

I. Introduction

With increasing scale and complexity of modern enterprise data centers, administrators are being forced to rethink the design of their data centers. In a traditional data center, application computation and application data are tied to specific servers and storage subsystems that are often over-provisioned to deal with workload surges and unexpected failures. Such configuration rigidity makes data centers expensive to maintain with wasted energy and floor space, low resource utilizations and significant management overheads.

Today, there is significant interest in developing more *agile* data centers, in which applications are loosely coupled to the underlying infrastructure and can easily share resources among themselves. Also desired is the ability to migrate an application from one set of resources to another in a non-disruptive manner. Such agility becomes key in modern cloud computing infrastructures that aim to efficiently share and manage extremely large data centers. One technology that is set to play an important role in this transformation is virtualization.

A. Integrated Server and Storage Virtualization

Virtualization technologies enable application computation and data to be hosted inside virtual containers (e.g., virtual machines, virtual disks) which are decoupled from the underlying physical resources. For example, server virtualization technologies such as VMware [1] and Xen [2] facilitate application computation to be packaged inside virtual machines (VMs) and enable multiple such virtual machines to be run alongside each other on a single physical machine. This allows extensive sharing of physical resources across applications. Additionally, the new live-migration advancements [3], [4] allow VMs to be migrated from one server to another without any downtime to the application running inside it.

Storage virtualization technologies[1] on the other hand, virtualize physical storage in the enterprise storage area network (SAN) into *virtual* disks that can then be used by applications. This layer of indirection between applications and physical storage allows storage consolidation across heterogeneous vendors and protocols, thus enabling applications to easily share heterogeneous storage resources. Storage virtualization also supports live migration of data in which a virtual disk can be migrated from one physical storage subsystem to another without any downtime. Many storage virtualization products such as IBM SAN Volume Controller (SVC) [5] and EMC Invista [6] are increasingly becoming popular in data centers [7].

While the server and storage virtualization technologies may have existed independently for the last few years, it is their integration in function and management that is truly beneficial. Integrated server and storage virtualization with their live-migration capabilities allows applications to share *both* server and storage resources, thus consolidating and increasing utilizations *across* the data center. Integrated management of the two improves efficiency by ensuring that the right combination of server and storage resources are always chosen for each application. It also facilitates data center optimizations like load balancing (Section-I-B).

In this paper, we describe our system HARMONY that integrates server and storage virtualization in a real data center along with a dynamic end-to-end management layer. It tracks application computation (in the form of VMs) and application data (in the form of Vdisks) and continuously monitors the resource usages of servers, network switches, and storage nodes in the data center. It can also orchestrate live migrations of virtual machines and virtual disks in response to changing data center conditions. Figure 1 shows the data center testbed. The testbed and HARMONY system are explained in greater detail in Sections II and III.

[1]We focus on block-level storage virtualization (see Section- II) in this work.

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

B. Load Balancing in Data Centers: Handling Hierarchies and Multi-Dimensionality

An important characteristic for a well managed data center is its ability to avoid hotspots. Overloaded nodes (servers, storage or network switches) often lead to performance degradation and are vulnerable to failures. To alleviate such hotspots, load must be migrated from the overloaded resource to an underutilized one[2]. Integrated server and storage virtualization can play a key role by migrating virtual machines or virtual disks without causing disruption to the application workload. However, intelligently deciding which virtual items (VM or Vdisk) from all that are running on the overloaded resource are to be migrated and to where can be a challenging task.

Even if we knew which item to move, deciding where to move it to needs to address the multidimensionality of the resource requirements. For example, assigning a virtual machine to a server requires cpu, memory, network and io bandwidth resources from that server. So the multidimensional needs have to be carefully matched with the multidimensional loads and capacities on the servers. Second, the hierarchical nature of data center storage area network topologies (e.g. Figure 1) implies that assigning a VM to a server node or a Vdisk to a storage node puts load not just on that node but also on all the nodes on its I/O data path (referred to as flow path). So the attractiveness of a node as a destination for a VM or Vdisk depends on how loaded each of the nodes in its flow path is.

In this paper, we describe a novel *VectorDot* algorithm that takes into account such hierarchical and multi-dimensional constraints while load balancing such a system. The *VectorDot* algorithm is inspired by the Toyoda method for multi-dimensional knapsacks and has several interesting aspects. We describe the algorithm in Section- IV.

We validate HARMONY and *VectorDot* through experiments on both a real data center setup as well as simulated large scale data center environments. Through our experiments, we are able to demonstrate the effectiveness of *VectorDot* in resolving multiple simultaneous overloads. It is highly scalable producing allocation and load balancing recommendations for over 5000 VMs and Vdisks on over 1300 nodes in less than 100 seconds. It also demonstrates good convergence and is able to avoid oscillations.

II. HARMONY Data Center Testbed: Setup and Configuration

In this section, we describe the data center testbed set up for HARMONY and also overview important issues related to integrating server and storage virtualization technologies. We start with a quick overview of a data center storage area network.

[2]Hotspot alleviation is often considered to be a more practical approach in contrast to *pure* load balancing which requires maintaining an equal distribution of load at all times as the latter may require expensive migrations even when the system is not under stress.

A. Storage Area Network (SAN)

The storage area network in the data center is composed of servers (hosts), switches and storage subsystems connected in a hierarchical fashion mostly through a Fibre Channel (FC) [8] network fabric. Each server has one or more Host Bus Adapters (HBAs) with one or more Fibre Channel ports each. These ports are connected to multiple layers of SAN switches which are then connected to ports on storage subsystems. In a large enterprise data center, there can be as many as thousands of servers, hundreds of switches and hundreds of storage subsystems.

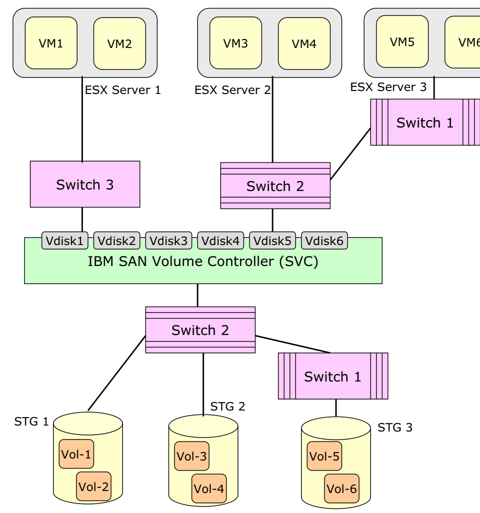


Fig. 1. HARMONY Testbed Setup

Figure- 1 shows the HARMONY testbed. We will discuss various components of the testbed in the coming sections. Other than this hierarchical physical connectivity, there are two important logical configurations in a SAN – **Zoning** dictates which storage subsystem ports can be accessed by any given host and **Logical Unit Number (LUN) mapping/masking** configuration defines the storage volumes on the storage subsystem that can be accessed by a particular host [8].

B. Virtualization Technologies

Virtual Machine technology, first introduced in the 1960s [9], has been widely exploited in recent years for consolidating hardware infrastructure in enterprise data centers with technologies like VMware [1] and Xen [2]. While server virtualization has garnered a lot of attention from the academic and research communities, storage virtualization has received considerably less attention, even though IDC estimates it to be growing at over 47.5% annually in revenue [7].

Storage virtualization refers to the process of abstracting physical storage into virtualized containers called *virtual disks* (*Vdisks*) that can be used by applications. The layer of indirection between applications and physical storage allows aggregating heterogeneous physical storage into logical pools with the characteristic that Vdisks abstracted from a pool are unaffected even if pool membership changes. Storage virtualization provides a number of benefits like consolidation, dynamic growth or shrinking of storage (thin provisioning) and performance optimizations like striping that provide faster I/O access through parallel I/Os. It also provides the critical capability of non-disruptive data migration from one subsystem to another analogous to VM live-migration.

Storage virtualization can be at different granularities, for example, in *block-level virtualization*, data *blocks* are mapped to one or more storage subsystems, but appear to the application as residing on a single volume whereas in *file virtualization* multiple filesystems can be made to appear as a single filesystem with a common namespace. In this work, we focus on more popular block-level storage virtualization technologies.

C. HARMONY **Setup and Configuration**

In HARMONY, we used the market leading in-band block virtualization appliance called IBM SAN Volume Controller (SVC) [5]. As shown in Figure- 1, we set up the SVC in the I/O paths of the three ESX servers. Configuring the virtualization appliance into the data path between host servers and storage subsystems allows aggregation across the entire storage infrastructure and provides features like data caching, I/O access optimizations using striping and parallelization, replication services and data migration. In the testbed, the SVC is connected to the servers through three SAN switches from Brocade (Switch-1 and Switch-2) and Cisco (Switch-3). Three enterprise class storage controllers from the IBM DS4000 series provide physical storage to the virtualization appliance. Of the resources, switches and storage are shared with other users of the data center. Observe that two switches *above* the SVC are the same as ones *under* it. This is a typical configuration as virtualization appliances are introduced into existing storage fabric and available ports on the switches are configured for its connectivity to both host servers and storage subsystems.

SAN Configuration: From a SAN perspective, the SVC appears as a storage subsystem to servers and as a server to storage subsystems. Thus, the zoning and LUN mapping configurations (as discussed earlier) can be split into two parts - the SVC to storage connectivity and hosts to SVC connectivity. For Vdisk live-migration to work, we ensure that SVC has access to all storage subsystems and their volumes by creating a zone with SVC and storage subsystems' ports and LUN mapping each storage volume to the SVC. For VM live-migration to work between the three ESX servers, we ensure that all three servers can access all SVC Vdisks (during migration, the destination server loads the image of the migrating virtual machine from SVC directly) by zoning server HBA ports with the SVC ports and LUN mapping Vdisks to all hosts. Additional configuration for hosts that can be used for VM migration includes a gigabit ethernet connection (required for fast transmission of VM state during live migration) and IP addresses in the same L2 subnet.

III. **End-to-End Management Layer**

HARMONY extracts an end-to-end view of the SAN including performance and usage characteristics. Figure 2 describes the eco-system of HARMONY. It interacts with multiple external components:

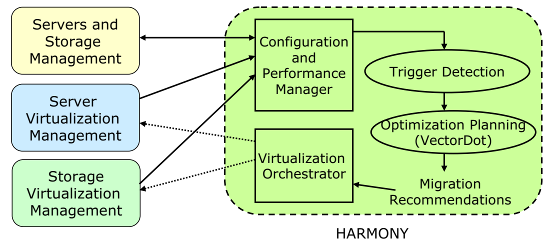


Fig. 2. HARMONY Architecture

- *Server and Storage Management* software like HP Systems Insight Manager [10] and IBM TotalStorage Productivity Center [11]. HARMONY obtains information about resources, SAN topology and performance statistics about servers and storage area network from this component. Most management software use standards like CIM [12] to collect resource configuration and performance information. While we had direct access to [11]'s databases, there are also open source efforts on standardizing how such information is maintained [13].
- *Server Virtualization Management*: HARMONY also interacts with a server virtualization manager like VMWare Virtual Center [14] to (a) obtain configuration and **performance** information about VMs and physical servers, and (b) to **orchestrate** functions like live VM migration. There does not exist a single popular management system for Xen yet, though a combination of techniques allow similar functionality [15]. HARMONY uses VMWare Virtual Center's Software Development Kit (SDK) [16] for communication and orchestration of VM migration.
- *Storage Virtualization Management*: HARMONY also interacts with a storage virtualization manager to obtain virtual storage configuration and to orchestrate non-disruptive data migration. In our implementation we used IBM SVC's CLI interface using a Java implementation of ssh.

Internally, HARMONY consists of a *Configuration and Performance Manager*, which is responsible for maintaining an end-to-end view of the storage area network by correlating configuration and performance information from individual server and storage management, server virtualization management and storage virtualization management components. Once a trigger is identified in the collected data (for example, a switch node is overloaded above a configurable threshold), HARMONY initiates the optimization planning component – *VectorDot*. The latter generates recommendations for migrating one or more VMs or Vdisks to alleviate the hotspot. These are then processed by the *Virtualization Orchestrator* by using appropriate server and storage virtualization managers.

Note that HARMONY's migration of VMs and Vdisks need not be for the sole purpose of load balancing. Such actions may also be orchestrated for other optimizations like performance improvements (migrating to a better server/storage node), reducing power utilization (migrating VMs to lesser number of servers) or high availability (migrating storage from a subsystem that is being de-commissioned).

A. Migration Orchestration and Application Impact

In this section, we give an example orchestration of live VM and storage migration on the testbed through HARMONY. For this example, we migrate VM 2 (Figure 1) from Server-1 to Server-2 and its physical storage (Vol-2) from STG-1 to STG-2. VM 2 is configured with 3 GHz CPU, 1.24 GB memory and its storage volume is of size 20 GB. During the migration, the application gets continued access to its resources (computation resources from VM 2 and storage from Vdisk2). Internally, integrated migration changes the entire virtual to physical resource mapping including VM's host system, physical storage subsystem and I/O path for data access.

Even though migrations cause no downtime in applications, they do impact application performance. In order to illustrate this, we ran the popular Postmark benchmark [17] in the VM. Postmark simulates an email server workload by creating a large number of small files and performing random transactions on those files, generating both CPU and I/O load.

State	Throughput	Duration (s)	Overhead
Normal	1436 trans/s	–	–
Virtual Machine Migration	1265 trans/s	50	11.9%
Virtual Storage Migration	1327 trans/s	549	7.6%

TABLE I
MIGRATION IMPACT ON APPLICATION PERFORMANCE

As shown in Table I, during VM migration Postmark transaction throughput drops by around 11.9%. This happens due to the CPU congestion at source and destination servers caused by live migration (that requires maintaining changes in application state and transmitting those changes over the ethernet network). Storage live migration through the virtualization appliance does not cause CPU congestion and has a smaller overhead of 7.6%, though it occurs for a much more prolonged period of time, as 20 GB of data is read from one storage subsystem and written to another. This example is only an illustration of migration impact on application performance and may vary with configuration and size of VM and I/O traffic loads on the SAN. In practice, many situations warrant taking this performance hit to avoid application downtime (for example, scheduled maintenance or upgrade of physical servers) or when the potential benefits of migration outweigh these costs (for example, eliminating a hotspot). Additionally, it is possible to make migration decisions based on the performance impact or application priority.

Next, we describe how HARMONY can be used for integrated load balancing in enterprise data centers using a novel methodology called *VectorDot*.

IV. Dealing with Overloads: Practical Load Balancing

Suppose a node in the system gets overloaded, i.e., goes over its configured threshold along one of the dimensions. The node could be a server node, storage node or switch node. A server overload can occur due to excessive CPU, memory, network or disk I/O usage. A storage node overload can occur due to excessive storage space usage or disk I/O rates. A switch node (or a switch port) overload can occur due to a large amount of I/O being pushed through it. The node that gets overloaded first is dictated by the node capacities and the node location in the SAN topology. To address the overload, the goal of the load balancing algorithm in HARMONY is to move one or more VMs or Vdisks from under the overloaded node to suitable underloaded nodes.

System State Input: The input to the load balancing algorithm comprises the current state of the virtualized data center: including the status of the server nodes, storage nodes, switch nodes, the storage virtualization appliance SVC, the topology connections and the set of VMs and Vdisks in the system. A sample set of resource types and parameters for each of the nodes is shown in Table II. For server nodes, the resources of interest are cpu, memory, network bandwidth, and storage I/O bandwidth. For each of these, there are three relevant parameters on each node: current usage value, a capacity value and a suggested threshold fraction between 0 and 1. The threshold fraction is a hint to the load balancer to keep the usage of the resource below that fraction. For a storage node the parameters we currently monitor are I/O rate (Mbps) and storage space (GB), while for switches we use the amount of I/O flowing through them (Mbps). The algorithm and system are flexible enough to accommodate other parameters if needed.

Node/Item	Resource Type	Parameter (Units)
Server Node	CPU	cpuU, cpuCap, cpuT (0..1)
	Memory	memU, memCap, memT (0..1)
	Net. bandwidth	netU, netCap, netT (0..1)
	I/O bandwidth	ioU, ioCap, ioT (0..1)
Storage Node	Space	spaceU, spaceCap, spaceT (0..1)
	I/O rate	ioU, ioCap, ioT (0..1)
Switch Node	I/O rate	ioU, ioCap, ioT (0..1)

TABLE II
SAMPLE SET OF PARAMETERS FROM EACH NODE

If all the nodes in the system are below their thresholds then the system is considered to be in a good state and there is nothing for the load balancer to do. If however a node exceeds the threshold along any of its resource dimensions, the node is considered to be an *overloaded node* or *trigger* *node*, and the goal of the load balancing algorithm is to move one or more items (VMs or Vdisks) to bring the overloaded nodes below their threshold as much as possible. However, deciding what item to move and to where is a challenging problem.

Multidimensionality: Even if we knew which item to move, deciding where to move it to needs to address the multidimensionality of the VM, Vdisk resource requirements. For example, assigning a virtual machine to a server requires certain amount of cpu, memory, network and I/O bandwidths on the destination node. Similarly assigning a Vdisk to a storage node requires certain amount of space capacity (GB) and I/O capacity (Mbps) on the destination node. So the multidimensional needs of the VM (resp., Vdisk) have to be carefully matched with the multidimensional loads and capacities on the node.
Example: VM A requires 100 MHz of CPU, 50 MB RAM, 0.5 Mbps of network bandwidth and 0.2 Mbps of storage bandwidth. If a server node S of capacity 2GHz of CPU, 512 MB RAM, 2 Mbps network bandwidth and 2 Mbps of storage bandwidth is loaded at 50% along each of the dimensions then it can still accommodate the new VM. However if it is, say, more than 75% loaded along network bandwidth dimension then it cannot accommodate the new VM.

FlowPath, Hierarchy Constraints: Furthermore, different switches in the system will be at different load levels depending on the nature of the workloads going through them and the dynamic variations in the workloads due to time-of-day fluctuations, seasonality fluctuations etc. Assigning a VM or Vdisk to a server or storage node induces load not just on the end server or storage node but also on the switch nodes along its I/O data path, referred to as the flow path. Even if a physical server has enough spare cpu, memory, network and storage bandwidth capacities to host a particular VM, if the switch it is connected to is overloaded then it cannot be a good destination for the VM. Thus the attractiveness of a *leaf* node for a VM or Vdisk needs to consider how loaded each of the nodes on the flow path is.
Example: In Figure 1, assigning VM4 to Server 3 induces load not just on Server 3 but also on Switches 1 and 2. Even though Server 3 has sufficient spare resources to host VM4, if Switch 1 is overloaded and does not have enough spare capacity to accommodate VM4's I/O traffic, then server3 would not be a feasible destination for VM4.

Our solution: In this paper, we propose a novel *VectorDot* algorithm to address these hierarchical and multidimensional constraints that arise when deciding what items to move and to where. The *VectorDot* algorithm is inspired by the successful Toyoda heuristic [18] for multidimensional knapsacks [19]. While the original Toyoda heuristic was limited only to selecting items into a single knapsack (item-comparison only), here we extend it to a collection of knapsacks (node-comparisons also) and for dynamic load-balancing reassignment of items among the knapsacks, in the presence of additional constraints mentioned above. The *VectorDot* algorithm is the first of its kind for addressing such hierarchical multi-dimensional situations.

Before we describe the algorithm we introduce the notion of node load fraction vectors and item node load fraction vectors.

A. Node Load Fraction Vectors

Given the state of each node, as in Table II, the node load fraction vector, $NodeLoadFracVec(u)$, for a node u is defined as the multidimensional vector representing the usage fractions (between 0 and 1) for each resource at the node u. Similarly the threshold vector, $NodeThresholdVec(u)$, for a node u is the multidimensional vector representing the corresponding threshold fractions at node u. Note that these two vectors at a node will have the same number of dimensions. For example,

- For a server node u, the node load fraction vector is given by $\left\langle \frac{cpuU}{cpuCap}, \frac{memU}{memCap}, \frac{netU}{netCap}, \frac{ioU}{ioCap} \right\rangle$ and the node threshold vector by $\langle cpuT, memT, netT, ioT \rangle$.
- For a storage node u, the two vectors are given by $\left\langle \frac{spaceU}{spaceCap}, \frac{ioU}{ioCap} \right\rangle$ and $\langle spaceT, ioT \rangle$ respectively.
- For a switch node u, they are given by $\left\langle \frac{ioU}{ioCap} \right\rangle$ and $\langle ioT \rangle$ respectively.

Overloads and Imbalance Scores: To measure the degree of overload of a node, and of the system, we use the notion of an imbalance score. The imbalance score allows us to penalize nodes based on how high they are above their threshold. Linear scoring functions do not suffice, for example, to distinguish between a pair of nodes at $3T$ and T and a pair of nodes both at $2T$ each. Both configurations get the same score with a linear scoring function, hence we use an exponential weighting function.

$$IBscore(f, T) = \begin{cases} 0 & \text{if } f < T \\ e^{(f-T)/T} & \text{otherwise} \end{cases}$$

where f is the load usage fraction and T is the corresponding threshold for a resource. This function penalizes the $3T$ and T configuration more than the $2T$ and $2T$ configuration, thus encouraging the system to move towards the latter configuration. The imbalance score for a node u is obtained by summing over the imbalance scores along all the dimensions of its vector, i.e.,

$$IBscore(u) = \sum_i IBscore\big(NLFVec_i(u), NTVec_i(u)\big)$$

where $NLFVec_i(u)$ is a short for the ith component of $NodeLoadFracVec(u)$ and $NTVec_i(u)$ is a short for the same along $NodeThresholdVec(u)$. The total imbalance score of the system is obtained by summing over all the nodes.

$$TotalIBScore = \sum_u IBscore(u)$$

The $TotalIBScore$ is a measure of the total imbalance in the system relative to the thresholds, and the goal of the load balancing system is to reduce this as much as possible by migrating one or more VMs or Vdisks.

B. Virtual Item Load Fraction Vectors

The resource requirements of each virtual item, VM or Vdisk, are captured by a similar vector. For example, for a VM the load usage vector is a 4-dimensional tuple $\langle cpuU, memU, netU, ioU \rangle$ describing the resource usage requirements of the VM. For Vdisks, the load vector is 2-dimensional: $\langle spaceU, ioU \rangle$ capturing the amount of storage space and I/O required. Each virtual item is mapped to a physical resource based on its resource requirements.

When a virtual item vi is being considered to be mapped to a particular node u (ignoring hierarchical constraints for now), then a useful concept is that of a item node load fraction vector, $ItemNodeLoadFracVec(vi, u)$. This captures the resource requirements of the item as a fraction of u's capacity. For example, for a VM vi and node u, this is constructed as:

$$\left\langle \frac{cpuU(vi)}{cpuCap(u)}, \frac{memU(vi)}{memCap(u)}, \frac{netU(vi)}{netCap(u)}, \frac{ioU(vi)}{ioCap(u)} \right\rangle$$

Note that this captures the different capacities on different nodes, i.e., heterogeneity among machines. Similar item node load fraction vectors are defined for Vdisks on storage nodes and for VMs, Vdisks on switch nodes.

C. Path Vectors

As mentioned earlier, however, assigning a VM or Vdisk to a server or storage node requires resources not just on that end node but also on all the (switch) nodes along its flow path. To capture this, we extend the above load fraction vectors to path load fraction vectors by concatenating the node vectors along the flow paths.

The $FlowPath(u)$ of a node is the path from a leaf node u to the StorageVirtualizer node SVC. The path vectors $PathLoadFracVec(u)$, $PathThresholdVec(u)$, and $ItemPathLoadFracVec(u)$ are natural extensions of the corresponding single node vectors by concatenating along the flow path of u. Table III gives a formal definition.

Example: In Figure 1, the FlowPath for Server 3 is (Server3 $\rightarrow$ Switch1 $\rightarrow$ Switch2). If the $NodeLoadFracVec(.)$ for Server3 is $\langle x_1, x_2, x_3, x_4 \rangle$, for Switch1 is $\langle y_1, y_2 \rangle$, and for Switch2 is $\langle z_1, z_2 \rangle$, then the $PathLoadFracVec(.)$ for Server3 will be $\langle x_1, x_2, x_3, x_4, y_1, y_2, z_1, z_2 \rangle$.

D. VectorDot: Overview and Extended Vector Products

Now to evaluate the attractiveness of node u for virtual item vi, we use an extended vector product, $EVP(vi, u)$. This product $EVP(vi, u)$ is essentially the dot product of $PathLoadFracVec(u)$ and the $ItemPathLoadFracVec(vi, u)$, along with a few other modifications to account for thresholds, imbalance scores, and the need to avoid oscillations during load balancing. If multiple destination nodes are available as a candidate for a virtual item we then take the node u that minimizes the $EVP(vi, u)$ measure as the chosen candidate. Recall that the regular dot product of two equal dimension vectors, A and B is given by,

$$dotproduct(A, B) = A.B = \sum_{i:1 \leq I \leq |A|} a_i * b_i$$

Example: Suppose a virtual item vi has to choose between two nodes u and w (u and w could be, for example, Server 1 and Server 2 in Figure 1) and suppose that concatenated load fraction vector for the flow path of u, A_u is $\langle 0.4, 0.2, 0.4, 0.2, 0.2 \rangle$ and that for w, A_w is $\langle 0.2, 0.4, 0.2, 0.4, 0.2 \rangle$. Further suppose that the requirements vector of vi along the flow path of u, $B_u(vi)$, and that along the flow path of w, $B_w(vi)$ are both the same, $\langle 0.2, 0.05, 0.2, 0.05, 0.2 \rangle$. Now in comparing u and w, the $EVP(vi, u)$ turns out to be more than that of $EVP(vi, w)$ and hence w is preferred over u which is in fact the right choice, since u is more loaded along the dimensions where vi's requirements are also high.

The dot product, among other things, helps distinguish nodes based on the item (VM or Vdisk) requirements. A node that is highly loaded along a dimension where the item requirements are high is penalized more than a node whose high load is along other dimensions. The EVP computation based on these intuitions and other enhancements to account for thresholds, imbalance scores and the need to avoid oscillations is given in Figure 3. This is the basis of our algorithm.

The **extended-vector product**, $EVP(vi, u)$ is defined with three vector arguments. In addition to the above two, it uses a third argument, the $PathThresholdVec(u)$. The idea is that the components of the first vector, the $PathLoadFracVec(u)$ are first smoothed w.r.t the components of the threshold vector and the result then participates in a regular dot-product with the second vector as illustrated in Figure 3. The idea of the smoothing step is to weigh each component relative to its threshold as in the imbalance scores. So a component at 0.6 when the threshold is 0.4 gets a higher smoothed value than a component at 0.6 when the threshold is 0.8, for example.

Another augmentation in the $EVP(vi, u)$ computation is to avoid oscillations. If vi was already on node u, then $EVP(vi, u)$ essentially takes the above kind of dot product of the node's path load fraction vector and vi's path load fraction vector with smoothing. However, if vi were not already on u, then we first compute what the load fraction vector would be for the path of u if vi were added to u. This is called the $AdjustedPathLoadFracVec(vi, u)$ and is needed for two reasons. First, it gives a more realistic assessment of the loads on the path after the move. Second, it helps avoid oscillations.

Example. Suppose there are two nodes u and v, with node u at load fraction 0.8 and v at 0.6, assuming single dimension. Now if we are considering moving an item of size 0.2 from u and compare with v's pre-move load of 0.6 then v will look preferable (the dot product will also be lower), so we will end up moving it to v. But once it reaches v, the situation

Symbol	Short Description	Applies to	Formula
$NodeLoadFracVec(u)$	Load Fraction Vector	All nodes u	e.g. $\left\langle \frac{cpuU}{cpuCap}, \frac{memU}{memCap}, \frac{netU}{netCap}, \frac{ioU}{ioCap}, \cdots \right\rangle$
$NodeThresholdVec(u)$	Threshold Vector	All nodes u	e.g. $\langle cpuT, memT, netT, ioT \rangle$
$ItemNodeLoadFracVec(vi, u)$	Vector of vi's usage as fraction of u's capacities	All nodes u VM/Vdisk item vi	e.g. $\left\langle \frac{cpuU(vi)}{cpuCap(u)}, \frac{memU(vi)}{memCap(u)}, \frac{netU(vi)}{netCap(u)}, \frac{ioU(vi)}{ioCap(u)} \right\rangle$
$FlowPath(u)$	Path of flow from $u \rightarrow SVC$	Leaf nodes u	Path of flow from $u \rightarrow SVC$
$PathLoadFracVec(u)$	Concatenated Load Fraction Vectors along flowpath of u	Leafnodes u	$\cup_{v \in FlowPath(u)} NodeLoadFracVec(v)$
$PathThresholdVec(u)$	Concatenated Threshold Vectors along flowpath of u	Leafnodes u	$\cup_{v \in FlowPath(u)} NodeThresholdVec(v)$
$ItemPathLoadFracVec(vi, u)$	Concatenation of above along flowpath of u	Leafnodes u VM/Vdisk item vi	$\cup_{v \in FlowPath(u)} ItemNodeLoadFracVec(vi, v)$

TABLE III
NOTATION AND TERMINOLOGY

Algorithm1 *VectorDot*: COMPUTING EVP

EVP(**Vitem** vi, **leafNode** u) {
  if (vi already on u) **then**
    $LVec \leftarrow PathLoadFracVec(u)$
    $VVec \leftarrow ItemPathLoadFracVec(vi, u)$
    $TVec \leftarrow PathThresholdVec(u)$
    return $EVP2(LVec, VVec, TVec)$
  else
    $LVec \leftarrow AdjustedPathLoadFracVec(vi, u)$
    $VVec \leftarrow ItemPathLoadFracVec(vi, u)$
    $TVec \leftarrow PathThresholdVec(u)$
    return $EVP2(LVec, VVec, TVec)$
  end if
}

Fig. 3. *VectorDot*: Extended-Vector Product

Algorithm2 *VectorDot*: EVP2

$EVP2(LVec, VVec, TVec)$ {
  Assert($LVec.size() = VVec.size()$)
  Assert($LVec.size() = TVec.size()$)
  $val \leftarrow 0$
  for $i = 1 \ldots LVec.size()$ **do**
    $val+ = VVec[i] * Smooth(LVec[i], TVec[i])$
  end for
}

$Smooth(frac, T)$ {
  return $e^{\alpha \frac{frac - T}{T}}$
}

Fig. 4. *VectorDot*: EVP2

is reversed and it will want to come back to u. This results in an oscillation and can happen also at multiple dimensions and larger granularities. Comparing with post-move score of v which will be 0.8 makes it a more stable comparison.

The adjusted path load fraction vector for vi and u is obtained as before by concatenating the $AdjustedNodeLoadFracVec(vi, w)$ for each node w on the $FlowPath(u)$. The latter for an individual node w on the $FlowPath(u)$ is computed as what the load fraction vector at w will be if vi were also assigned to go through it. Note that for nodes w that are above the least common ancestor of u and $vi.currentphysical$, this does not need any adjustment because the contribution of vi to these nodes was already included.

E. Handling Overloads and Selecting Destinations

Based on the above $EVP(vi, u)$ measure, we define different selection criterion for the destination node for a virtual item vi. A node u is said to be feasible for a item vi if and only if placing vi at u does not violate the multidimensional capacities not only at node u but also at all the nodes v on the flow path of u. The $FirstFit(vi)$ is the first feasible node that is found, traversing nodes in a static order either by name or index. The $BestFit(vi)$ is the node that minimizes $EVP(vi, u)$ among all the feasible nodes u. $WorstFit(vi)$ is the node that maximizes $EVP(vi, u)$ among all the feasible nodes u. The purpose of the $WorstFit(vi)$, though it is not expected to do well, is to serve as a comparison candidate for evaluating the validity of the EVP measure. We also introduce a variant of $BestFit$ called $RelaxedBestFit$, which incorporates randomness and visits nodes in a random order until a certain configurable number of feasible nodes are found (e.g., 5) and returns the one with the minimum EVP measure among them. This can be much faster in practice especially when visiting all nodes is not necessary.

The algorithm begins by identifying trigger nodes if any in the system and computing their imbalance scores. A node u is said to be a *trigger* if its load fraction exceeds its threshold along any of the dimensions. Formally,

$$trigger(u) = \begin{cases} 1 & \text{if } \exists i : NLFVec_i(u) > NTVec_i(u) \\ 0 & \text{otherwise} \end{cases}$$

where $NLFVec_i(u)$ is a short for the ith component of $NodeLoadFracVec(u)$ and $NTVec_i(u)$ is a short for the ith component of $NodeThresholdVec(u)$. The load balancing algorithm considers trigger nodes in decreasing order of their imbalance scores $IBScore(u)$. If there are no trigger nodes, it is done. Otherwise it picks the node with the highest score and looks for the best extended vector product based move from the node.

Leaf Node Overload: Both server and storage node overloads are handled in a similar fashion since they are both leaf nodes in the SAN hierarchy. If the trigger node u in question is a leaf node, we sort all the items (VMs or Vdisks) on the node in decreasing order of their extended-vector products with u, $EVP(vi, u)$. For each item, beginning with the first one, we look for its $BestFit$ destination besides u. The items vi are evaluated based on the EVP improvements they provide and whether they can remove the trigger. The EVP delta $delta(vi, u)$ is calculated as

$$delta(vi, u) = EVP\big(vi, u\big) - EVP\big(vi, BestFit(vi)\big)$$

The candidate item moves are ranked first by whether they can remove the trigger and then by decreasing EVP deltas. The move with the highest rank is returned once found. Using $RelaxedBestFit$ in place of $BestFit$ gives us a faster alternative especially since it doesn't have to visit all the nodes. In Section V we evaluate both combinations in terms of their speed and quality.

Switch Node Overload: If the trigger node u in question is an internal node, i.e., a switch node, then the algorithm first orders the items (VMs or Vdisks) in the decreasing order of their contributions to the load on switch node u. This can be done by taking the dot products along just the switch load dimensions. Then for each of these items, it proceeds as in the previous part, looking for a BestFit destination outside the subtree of u.

Note that the items here will be VMs or Vdisks or a combination of both depending on how much they contribute to load on the switch. The items are ranked based on their extended-vector product deltas and whether they can remove the trigger. The move with the highest rank is returned once found. If distinguishing candidate moves based on size is desired, especially when choosing between VM and Vdisk moves, the ranking of the moves can be done by extended-vector product deltas per unit size and whether they can remove the trigger. The process repeats until all the triggers have been removed or no further moves are possible.

Note that only feasible moves are considered and recommended throughout the algorithm. In the next section we evaluate the performance of *VectorDot* on both a real data center testbed and also on large synthetic data centers.

V. Experimental Evaluation

In this section, we experimentally evaluate *VectorDot* and HARMONY. The evaluation is composed of two parts. First in Section V-A, we evaluate the efficacy and scalability of *VectorDot* on simulated data center environments of various sizes. This establishes the usefulness of the *VectorDot* measures and its characteristics like convergence. Second in Section V-B, we implement our techniques on the data center testbed and thus, validate its operations in a real enterprise setup.

A. Synthetic Experiments on Larger Sizes

First, we evaluate the efficacy and scalability of *VectorDot* on various problem sizes using simulated data center topologies. Our goal is four-fold: (1) To verify whether the extended vector product is always a good measure for selecting candidates, (2) The number of moves *VectorDot* would take to get the system to a balanced state when starting from different initial states, (3) Does it always converge to a close to balanced state? Or does it get stuck in local optima or oscillate between configurations, and (4) The running time for large input SANs with large numbers of VMs and Vdisks.

For this purpose we built a simulator to generate topologies of varying sizes and configurations. As in any realistic SAN environment, the size of the topology is based on the size of the application workload (number of VMs). We used simple ratios for obtaining the number of host servers, storage subsystems and switches. For example, for a workload with 500 virtual machines each with one virtual disk, we used 100 host servers and 33 storage nodes. We used the industry best-practices based core-edge design [8] of the SAN to design the switch topology. For the above example, we had 10 edge switches, 4 core switches that connect to the virtualization appliance. The appliance was then connected to storage nodes using two levels of 4 core switches.

The multidimensional load capacities for the servers, (e.g., $\langle cpuC, memC, netC, ioC\rangle$), storage and network nodes are generated according to a Normal distribution, $N(\alpha, \beta)$ where α is the mean and β the standard deviation of the distribution. Varying α allows us to control the degree of overload in the system, while varying β allows us to control the variances among the different instances. Similarly, for VMs and Vdisks, their resource usage requirements ($\langle cpuU, memU, netU, ioU\rangle$ and $\langle spaceU, ioU\rangle$ resp.) are generated using a different Normal distribution. The α, β values for each of these parameters is separate so they can be controlled independently. The default is 0.55 for each of these values.

1. Validating Extended Vector Product Measure: For this purpose we use the three different algorithms BestFit (BF), FirstFit (FF), WorstFit (WF) introduced in Section IV along with the BestFit optimization – RelaxedBestFit (RBF) described there. To validate the measure, we create topologies of different sizes ranging up to 1300 nodes with high load (α=0.7) and use the above extended vector product based methods to see how they pack them on to the multidimensional nodes. Figure 5 shows the imbalance score achieved by each of these methods with increasing problem sizes. As described in Section IV, lower the score, more balanced is the system.

As the graph illustrates, both BF and RBF algorithms yield placements with low imbalance scores. The average system load in this experiment is high (70%) so triggers cannot be fully avoided and a score of zero cannot be attained. In other

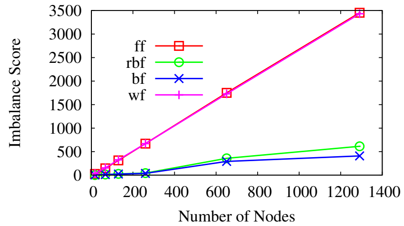


Fig. 5. Validating Extended Vector Product Measure

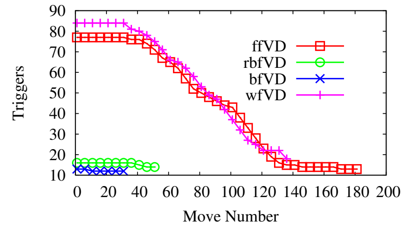


Fig. 6. Number of Moves: Trigger Comparison

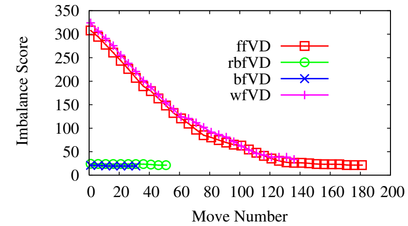


Fig. 7. Number of Moves: Score Comparison

experiments with low load scenarios, not shown here for space constraints, the BF and RBF achieve an imbalance score of close to 0 and almost no triggers. The WorstFit algorithm incurs a high penalty score confirming that the extended vector product measure does capture the contention of vitems on flow paths well – picking higher values of $EVP(vi, u)$ consistently leads to greater imbalance. This difference in performance of BestFit and WorstFit is an indication of the validity of extended vector product metric.

The good performance of RBF in Figure 5 is also an encouraging sign. It suggests that in practice it may not be necessary to look through all nodes to find the absolute best.

2. Number of Moves From Different Initial States: Next, to study the effect of *VectorDot* starting from different initial states – we begin with an empty setup, and use the four fit methods described above to place all the VMs and Vdisks among the servers and storage nodes and then invoke *VectorDot* to balance the resulting configuration. Figure 6 (resp., Figure 7) shows a summary of the number of trigger nodes in the system (resp, imbalance score) as a function of the number of moves made by *VectorDot*. The WF and FF cases begin with a higher degree of imbalance and hence *VectorDot* requires more moves.

For the BF and RBF cases, the system is already in a good balance so *VectorDot* needs fewer moves. It still makes a few moves when it finds better dot product improvements: the RBF case is likely to have more of improvements since RBF earlier looked only at the first 5 feasible candidates and hence may leave scope for more improvement. In both cases *VectorDot* does not increase the number of trigger nodes or score at any stage. The number of moves is roughly linear in the number of trigger nodes. The number of trigger nodes achieved by the BF placement strategy is an approximate indication of the best achievable for the system, even if we were to reassign all vitems afresh.

3. Convergence, Avoiding Local Optima and Oscillations: Figures 6 and 7 showed that *VectorDot* was able to balance the different initial states towards a similar final state in terms of imbalance score, without getting stuck in intermediate states or oscillating between high score configurations. These are desirable convergence properties for large-scale systems. Figure 8 shows the post-load balancing score for each of the above methods with varying problem sizes. It is worth comparing the imbalance scores in this graph with those in Figure 5 which shows the scores before load-balancing. The y-axis scales being different by a factor of 10 visually, the graph does not convey the full picture. However a closer examination reveals that while the FF and BF scores were 3500 and 300 earlier they are now much closer within 50 to 100. This shows that *VectorDot* is able to converge well in experiments in spite of starting from different initial states.

4. Running Time for Large Inputs: Figure 9 shows the running time for the different allocation mechanisms. Even though the graph shows the BestFit scheme as the slowest, it is not that bad in absolute terms. At the maximum point, it is taking 35s to allocate around 5000 VMs and Vdisks on a simulated data center topology of over 1000 nodes which seems adequate for HARMONY kind of system.

An interesting finding from our experiments is the performance of the RelaxedBestFit strategy in solution quality and running time. By using the same EVP measure as BestFit it is able to get balanced placements, and by using the approach of looking only until a few feasible candidates are found it is able run much faster than BestFit. This suggests that in practice looking through the candidates and finding the absolute best may not be worth the effort. Figure 10 shows the running time for the *VectorDot* load balancer invoked after each of the four allocation methods. Even though FF was the faster one in the previous graph, FFLB takes a long time because of the imbalance that FF created. For example, at 1300 nodes, FF had more than 90 triggers to be resolved by the Load Balancer. Even at that state *VectorDot* took around 12 minutes.

VectorDot in combination with BF and RBF is much faster. In practice the system is usually in a balanced state as *VectorDot* is continuously monitoring and fixing it, so there will only be a few triggers from one round to next. Hence we do not expect the running time of *VectorDot* to be a big concern in real systems such as HARMONY. Also, natural enhancements like caching and dynamic programming are expected to give further runtime speedups.

B. Data Center Testbed Validation

For end-to-end validation of HARMONY in a real data center setup, we created four scenarios in our testbed that cause overloads on multiple dimensions of servers, storage

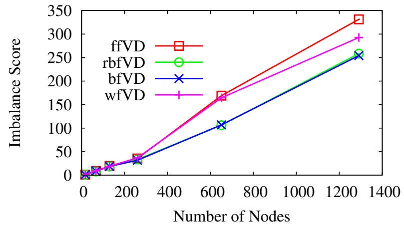


Fig. 8. Convergence: Scores *after* Allocation + Load Balancing

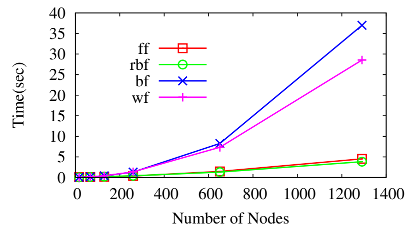


Fig. 9. Running Time For Basic Allocation

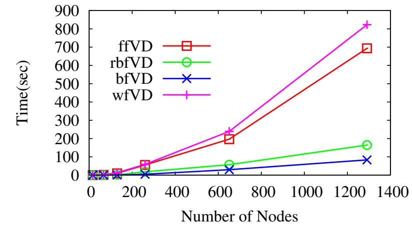


Fig. 10. Running Time For Allocation + Load Balancing

and switches. The data center testbed used in the experiments has been described in Section II. For the testbed experiments, we created six virtual machines (3 GHZ CPU, 1.24 GB RAM running RedHat Enterprise Linux 4.0) and distributed them equally between the three ESX servers. Each ESX Server has 1 HBA with 1 active port of 2GB I/O capacity and gigabit ethernet. As our setup required complex instrumentation to saturate the disk I/O and network capacities of host servers, we primarily focus on the other two dimensions of CPU and memory in our experiments (though our approach will easily apply to more dimensions). The physical storage for these virtual machines was derived from six storage volumes (of size 20 GB each) from three controllers. Each physical storage volume was virtualized into a single *Vdisk* and associated with one virtual machine. We disabled caching at the virtualization appliance to evaluate the direct effect on physical storage. The description of all resources including virtual machines to host and storage mappings are shown in Figure 11(a)[3].

We used publicly available workload generator *lookbusy* [20] to generate CPU and memory loads. The disk I/O loads were generated using Iometer [21]. All thresholds were set at 0.75. When any of the overloads are detected by HARMONY Performance Manager, it uses *VectorDot* to determine virtual resources that can be migrated and their destinations. It then uses the HARMONY Virtualization Orchestrator to execute suggested server and/or storage migrations by *VectorDot*.

1. Single Server Overload: As a first scenario, we overloaded Server-2 on the CPU dimension by creating a high CPU workload on VM 4. Figure 11(b) shows the CPU and memory utilizations for all three servers with elapsed time. HARMONY performance collection for resources (servers, storage and network) takes around 15 seconds. At t=15 seconds, HARMONY detects an overload on Server-2 and executes *VectorDot* to identify the right migration(s). The *VectorDot* scheme, chooses Server-3 even though Server-1 had greater CPU availability. Server-1 resulted in higher EVP score as its memory utilization was high. This illustrates multi-dimensional load balancing captured by *VectorDot*. HARMONY then executes the VM migration.

Notice that when the migration is executed, it causes a temporary CPU usage spike on the source and destination servers (Server-2 and Server-3). When the migration finishes this temporary spike drops and the overload has been resolved.

2. Multiple Server Overload: As part of our next validation experiment, we demonstrate ability of *VectorDot* to suggest multiple migrations and ability of HARMONY to execute these migrations. We created a CPU overload on Server-2 and a memory overload on Server-1 by increasing usages of VM 4 and VM 6 respectively. Figure 11(c) shows utilizations with elapsed time.

VectorDot first chooses migration of VM 4 from Server-2 to Server-3 (greater threshold overshoot) and then suggests migrating VM 6 from Server-1 to Server-2. HARMONY executes these migrations in the same order. As before, there are temporary CPU spikes during the migrations. However, after the two migrations complete, all server loads come below configured thresholds. Overall it takes about two minutes to execute both migrations and balance the load among servers.

3. Integrated Server and Storage Overload: In the next experiment, we demonstrate a storage migration integrated with a server migration. As mentioned before, the storage resources are shared with other users in the data center and we did not have the liberty to overload any of the storage controllers. As a result, we crafted the experiment by fixing the capacity of the storage controller to a smaller number (100 Mbps). Thus, now a migration is triggered if utilization caused by our storage accesses exceeds $0.75 * 100$ Mbps.

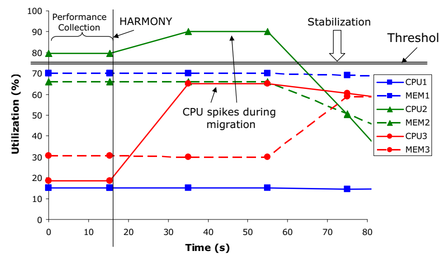
We generated a high I/O workload on VM 5 which accesses Vdisk1 (from Vol-1 on STG-1). We also added a memory workload to create an overload on VM 5's host server Server-1. Based on *VectorDot* recommendations, HARMONY is able to resolve both overloads using a combination of VM and Vdisk migrations. To show utilizations for multiple resource types, we tabulate the CPU and memory utilization of servers, and storage I/O utilization of storage controllers in Figure 12; VM-5 migration to Server-3 resolves the memory overload of Server-1 and storage migration of Vol-1 to STG-2 resolves the I/O throughput overload of STG-1. Overall this process takes little under 10 minutes to complete.

4. Switch Overload: In our final validation experiment we demonstrate how HARMONY handles a switch overload.

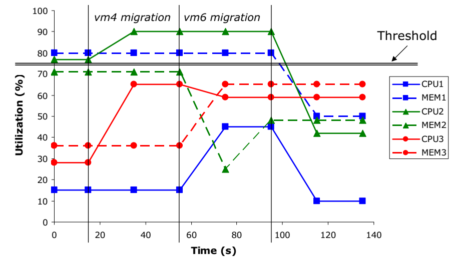
[3]Fig 1 showed different virtual to physical mappings for ease in exposition.

Resource	Description		
Host Servers (VMWare ESX 3.0.1)	Server	CPU	Memory
	Server-1	6 GHz (2x3GHz)	2.6 GB
	Server-2	3 GHz (1x3GHz)	4 GB
	Server-3	4 GHz (2x2GHz)	4 GB
Storage Volumes (20 GB, RAID 5)	Volume	Physical Controller	
	Vol-1, Vol-2	STG-1	
	Vol-3, Vol-4	STG-2	
	Vol-5, Vol-6	STG-3	
Virtual Storage	Vdisk 1-6	Vol 1-6 (resp.)	
Virtual Machines (3 GHz CPU, 1.2 GB RAM, RedHat EL4.0)		Server	Storage
	VM-1	Server-3	Vdisk5
	VM-2	Server-3	Vdisk3
	VM-3	Server-2	Vdisk4
	VM-4	Server-2	Vdisk2
	VM-5	Server-1	Vdisk1
	VM-6	Server-1	Vdisk6

(a)



(b)



(c)

Fig. 11. (a) Testbed Resource Description (b) Single Server Overload Resolution. Solid and dashed lines represent CPU and memory utilizations resp. (c) Multiple Server Overload Resolution. Solid and dashed lines represent CPU and memory utilizations resp.

	Servers (%) CPU, Mem	Storage % I/O
Initial Configuration	Server-1: 49.3, **82.4**	STG-1: **77.8**
	Server-2: 62.7, 58.9	STG-2: 9.8
	Server-3: 39.9, 47.5	STG-3: 54.9
After VM-5 Migration	Server-1: 33.3, 54.8	″
	Server-2: 59.1, 58.3	
	Server-3: 67.3, 71.8	
After Vol-5 migration	″	STG-1: 59.1
		STG-2: 26.5
		STG-3: 56.9

Fig. 12. Integrated Server and Storage Overload Resolution

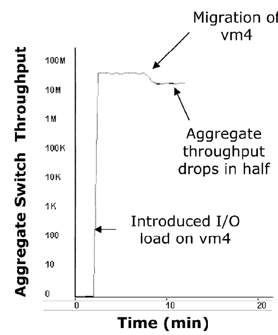


Fig. 13. Switch Overload (*Y-axis on log scale*)

Similar to the storage experiments, as a courtesy to users sharing the same switch, we crafted the experiment to create an overload scenario with less actual I/O. We fixed the capacity of Switch-2 to 50 MBps. Then we generated an I/O workload on VM 4 (that uses Vdisk2 virtualized from Vol-2 on STG-1) at the rate of 20 MBps. Since Switch-2 receives I/O from both Server-2 and the virtualization appliance (it is both *above* and *under* the appliance – Figure-1), the workload generate 40 MBps on Switch-2.

Figure 13 shows the **actual** performance monitor screenshot from the switch management interface (Y-axis on log scale). As switch utilization goes to 40MBps – 80% of (pseudo) capacity – HARMONY identifies it to be an overload trigger and initiates *VectorDot*. *VectorDot* analyzes Switch-2 load balancing through both server and storage virtualization. However, since all I/O from virtualization appliance to storage nodes goes through Switch-2, a storage migration will not yield any improvement. Hence *VectorDot* identifies the migration of VM 4 from Server-2 to Server-1, which reduces the I/O utilization at the switch by half (I/O from virtualization appliance to STG-1 still flows through Switch-2).

These validation experiments demonstrate the feasibility of our integrated load balancing approach in a real deployment.

VI. Related Work

Server virtualization [1], [2], [22] and its live migration advancements [3], [4] have spawned a new era of resource management in enterprise data centers. The VMWare Distributed Resource Scheduler [23] performs such migrations based on CPU and memory load on servers. Wood et al [15] describe an enhanced load balancing system for Xen that accounts for network usage as well. [24], [25] describe other resource management and consolidation techniques for servers using virtualization. However such work is restricted only to the server level in data centers and does not take into account the hierarchical data center topology spanning servers, storage and switch nodes.

There are many commercially available storage virtualization products [5], [6] that are being increasingly deployed in enterprises. Research in this area has primarily included design of virtualization systems [26], and addressing Quality-of-Service issues for applications with projects like Facade [27] and Stonehenge [28]. Techniques for storage migration have also received considerable attention, ranging from efficient data migration methods to reducing application impact [29], [30]. Such techniques can be used for performing load balancing at the storage level. However, none of these efforts address any integrated server and storage virtualization techniques.

SoftUDC [31] describes a vision for a virtualization based data center combining server, storage and network virtualization. Parallax [32] also describes an integrated server storage virtualization technique to scale to large amounts of storage but only for direct attached non-SAN environments. Both efforts did not report any performance analysis and also lack a load balancing scheme like *VectorDot* that addresses multi-dimensional loads in a hierarchical data center. Additionally, their solution is based inside host Virtual Machine Monitor (VMM) using host based storage virtualization – which requires changes to server virtualization environment for deployment and also incurs significant overheads for doing host-based storage migrations (Section II). A recent commercial effort Cisco VFrame [33] also describes an integrated virtualization platform. However, with the limited public information it is unclear as to their orchestration capabilities for non-disruptive VM and storage migration. Also, no clear load balancing strategy has been reported.

Load balancing schemes with multi-dimensional loads have an underlying similarity to the well known multi-dimensional knapsack problems [34], [19], [35]. However, the problem is famously NP-Hard [35]. Our method is inspired by the successful Toyoda heuristic [18]. While the original Toyoda heuristic was limited to selecting items into a single knapsack (item-comparison only), here we extend it to a collection of knapsacks (node-comparisons also) and dynamic load balancing among them in hierarchical SAN topologies. [36] reports another use of Toyoda heuristic for resource management in storage systems, however it neither addresses the hierarchy, nor has been evaluated for performance and quality.

VII. Conclusions

In this paper, we presented our design of an agile data center with integrated server and storage virtualization along with the implementation of an end-to-end management layer. We showed how to leverage this for non-disruptive practical load balancing in the data center spanning multiple resource layers – servers, storage and network switches. To this end, we developed a novel *VectorDot* scheme to address the complexity introduced by the data center topology and the multi-dimensional nature of the loads on resources. Our evaluations on a range of synthetic and real data center testbeds demonstrate the validity of our system and the ability of *VectorDot* to effectively address the overloads on servers, switches, and storage nodes.

As part of future work, we plan to incorporate proactive migrations as well into the system, based on predicting workload needs from historical trends [15] and statistical models [37]. Another direction is to incorporate application priorities and preferential selection of applications to migrate.

References

[1] VMware, http://www.vmware.com/.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of Symp. on Operating Systems Principles (SOSP)*, 2003.

[3] M. Nelson, B. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *USENIX Annual Technical Conference*, 2005.

[4] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI*, 2005.

[5] "IBM Storage Virtualization: Value to you," IBM Whitepaper, May 2006.

[6] EMC Invista, http://www.emc.com/products/software/invista/invista.jsp.

[7] IDC, "Virtualization across the Enterprise," Nov 2006.

[8] T. Clark, *Designing Storage Area Networks*. Addison-Wesley, 1999.

[9] R. Goldberg, "Survey of virtual machine research," *IEEE Computer*, vol. 7, no. 6, pp. 34–45, June 1974.

[10] Hewlett Packard Systems Insight Manager, http://h18002.www1.hp.com/products/servers/management/hpsim/index.html.

[11] IBM TotalStorage Productivity Center, http://www-306.ibm.com/software/tivoli/products/totalstorage-data/.

[12] DMTF Common Information Model Standards, http://www.dmtf.org/standards/cim.

[13] Aperi Storage Management Project, http://www.eclipse.org/aperi.

[14] VMWare Virtual Center, http://www.vmware.com/products/vi/vc/.

[15] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and Gray-box Strategies for Virtual Machine Migration," in *Proceedings of Symp. on Networked Systems Design and Implementation (NSDI)*, 2007.

[16] VMWare Infrastructure SDK, http://www.vmware.com/support/developer/vc-sdk/.

[17] J. Katcher, "PostMark: A New File System Benchmark," *Network Appliance Technical Report TR3022*, 1997.

[18] Y. Toyoda, "A simplified algorithm for obtaining approximate solutions to zero-one programming problems," *Management Science*, vol. 21, no. 12, pp. 1417–1427, 1975.

[19] A. Freville, "The multidimensional 0–1 knapsack problem: An overview," *European Journal of Operational Research*, vol. 155, no. 1, pp. 1–21, 2004.

[20] Devin Carraway, "lookbusy – A Synthetic Load Generator," http://devin.com/lookbusy.

[21] Iometer, http://www.iometer.org.

[22] Microsoft Virtual Server, http://www.microsoft.com/windowsserversystem/virtualserver.

[23] "VMware Infrastructure: Resource management with VMware DRS," VMware Whitepaper, 2006.

[24] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase, "Virtual machine hosting for networked clusters: Building the foundations for "autonomic" orchestration," vol. 0, p. 7, 2006.

[25] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic Placement of Virtual Machines for Managing SLA Violations," in *Proceedings of the 10th IEEE Symposium on Integrated Management (IM)*, 2007.

[26] A. Brinkmann, M. Heidebuer, F. M. auf der Heide, U. Rckert, K. Salzwedel, and M. Vodisek, "V:drive - costs and benefits of an out-of-band storage virtualization system," in *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST)*, pp. 153 – 157.

[27] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Facade: Virtual storage devices with performance guarantees," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[28] L. Huang, G. Peng, and T. cker Chiueh, "Multi-dimensional storage virtualization," in *SIGMETRICS '04/Performance '04: Proceedings of International Conference on Measurement and Modeling of Computer Systems*, 2004.

[29] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A. Karlin, J. Saia, R. Swaminathan, and J. Wilkes, "An experimental study of data migration algorithms," in *Proceedings of International Workshop on Algorithm Engineering*, 2001, pp. 28–31.

[30] C. Lu, G. Alvarez, and J. Wilkes, "Aqueduct: Online Data Migration with Performance Guarantees," in *Proceedings of USENIX Conference on File and Storage Technologies*, 2002.

[31] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler, "SoftUDC: A software based data center for utility computing," *IEEE Computer*, 2004.

[32] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand, "Parallax: managing storage for a million machines," in *HOTOS'05: Proceedings of the 10th Conference on Hot Topics in Operating Systems*, 2005.

[33] Cisco, "Data Center Virtualization and Orchestration: Business and Financial Justification," July 2007.

[34] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley, 1990.

[35] "Knapsack problem: Wikipedia," http://en.wikipedia.org/wiki/Knapsack_problem.

[36] R. A. Golding and T. M. Wong, "Walking toward moving goalpost: agile management for evolving systems," in *First Workshop on Hot Topics in Autonomic Computing*, 2006.

[37] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger, "Storage device performance prediction with CART models," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 412–413, 2004.