VOL. 1, NO. 2 – DECEMBER 2012

VMWARE TECHNICAL JOURNAL

Editors: Steve Muir, Rita Tavilla

TABLE OF CONTENTS

1 **Introduction**
Pat Gelsinger, CEO

2 **VMware Collaborations**
VMLab: Infrastructure to Support Desktop Virtualization Experiments for Research and Education
Prasad Calyam, Alex Berryman, Albert Lai, Matthew Honigford

vQuery: A Platform for Connecting Configuration and Performance
Ilari Shafer, Snorri Gylfason, Gregory R. Ganger

Elastic Resource Allocation in Datacenters: Gremlins in the Management Plane
Mukil Kesavan, Ada Gavrilovska, Karsten Schwan

3 **VMware Intern Projects**
Toward a Paravirtual vRDMA Device for VMware ESXi Guests
Adit Ranadive, Bhavesh Davda

Intrusion Detection Using VProbes
Alex Dehnert

Design and Implementation of a Cloud Tenant UI
Louis Weitzman, Alister Lewis-Bowen, Elizabeth Li, Jason Fedor

4 **VMware Engineering**
FrobOS is turning 10: What can you learn from a 10 year old?
Stuart Easson

Storage DRS: Automated Management of Storage Devices In a Virtualized Datacenter
Sachin Manpathak, Ajay Gulati, Mustafa Uysal

A Social Media Approach to Virtualization Management
Ravi Soundararajan, Emre Celebi, Lawrence Spracklen, Harish Muppalla, Vikram Makhija

VMware View® Planner: Measuring True Virtual Desktop Experience at Scale
Banit Agrawal, Rishi Bidarkar, Sunil Satnur, Tariq Magdon-Ismail, Lawrence Spracklen, Uday Kurkure, Vikram Makhija

vSOM: A Framework for Virtual Machine-centric Analysis of End-to-End Storage IO Operations
Sandeep Uttamchandani, Wenhua Liu, Samdeep Nayak

vmware®

VMware Sponsored Academic Research Awards Request for Proposal (RFP)

Theme: Storage in support of Software Defined Datacenters (SDDC)

VMware® invites university-affiliated researchers worldwide to submit proposals for funding in the general area of Storage in support of an SDDC. Our research interests in the field of Storage for SDDC are broad, including but not limited to the following topics:

- End-to-end application policy-based management and automation
- Storage QoS enforcement
- Integrated resource management for storage, networking, and computation
- Storage services, e.g. compression, de-duplication
- Securing Storage in Multi-Tenant environments
- Programming models for Non-Volatile Memory
- Alternatives to Block IO interfaces
- Shared Nothing Storage Architectures
- Storage for hybrid clouds (public/private)
- Emerging storage workloads and applications
- Energy-efficient storage design
- Large-scale testing of distributed storage
- Scale-out file systems
- NoSQL storage architectures
- Cloud storage for mobile use cases

Initial submissions should be extended abstracts, up to two pages long, describing the research project for which funding is requested and must include articulation of its relevance to VMware. After a first review phase we will create a shortlist of proposals that have been selected for further consideration, and invite authors to submit detailed proposals, including details of milestones, budget, personnel, etc. Initial funding of up to $150,000 will be for a period of one year, with options to review and recommit to funding subsequently.

2013 Key Dates

- March 13 — Two page extended abstracts due
- April 5 — Notification of interest from VMware
- May 1 — Full proposals due
- May 27 — Announcement of final acceptance of proposals

Please contact Rita Tavilla, Research Program Manager (rtavilla@vmware.com) with any questions.
All submissions should be sent to vmap-rfp@vmware.com.

It's my pleasure to introduce the second volume of the VMware Technical Journal. As VMware CEO and throughout my career I have emphasized the importance of engagement between industry and academia as an integral part of the "golden triangle" of innovation—organic innovation within dynamic companies alongside innovative research developments with universities and the passion of start-up environments. This volume of the Journal highlights how VMware taps into this approach, with many voices coming together to demonstrate the relationship between our academic research collaboration, internship programs and organic R&D efforts.

The journal begins with perspectives shared by collaborators at Georgia Institute of Technology, Carnegie Mellon University and Ohio State University. Their contributions highlight research projects supported by VMware through a combination of financial sponsorship, software donations and participation by our engineers. VMware-sponsored projects such as these span multiple focus areas, such as cluster scheduling, performance monitoring and security in cloud and virtualized environments. We continue to identify new research opportunities, and our next RFP scheduled for Spring 2013 will provide a great opportunity for a number of researchers to begin new collaborative projects.

We then feature three papers written by past interns, describing the breadth of projects undertaken by these students while at VMware. Our internship program is a highly structured opportunity for outstanding students to work on a specially defined project, which often leads to significant output such as a research publication or product enhancement. The program fosters long-term relationships, with many interns participating in multiple internships and frequently joining VMware upon graduation. Our PhD interns often integrate their summer projects into their PhD relationships, which in turn contribute to our ongoing academic partnerships—an example of the golden triangle at work!

We close this edition with contributions from a number of VMware engineers. These perspectives share how innovation occurs in R&D, whether providing a historical perspective on a key component of VMware's development infrastructure - FrobOS - or describing a new idea for leveraging social networking to efficiently manage large datacenters. Many of our engineers act as mentors to interns and also participate in research partnerships, thus reinforcing the relationship between every aspect of innovation.

We hope you enjoy this edition as much as we've enjoyed the collaboration and impact highlighted in the stories we share with you here. We welcome your comments and ideas for future articles—keep the input and innovation coming!

Pat Gelsinger
CEO, VMware

VMLab: Infrastructure to Support Desktop Virtualization Experiments for Research and Education

Prasad Calyam
The Ohio State University
pcalyam@oar.net

Alex Berryman
The Ohio State University
berryman@oar.net

Albert Lai
The Ohio State University
albert.lai@osumc.edu

Matthew Honigford
VMware, Inc.
mhonigford@vmware.com

Abstract

In terms of convenience and cost-savings, user communities have benefited from transitioning to virtual desktop clouds (VDCs) that are accessible via thin-clients, moving away from dedicated hardware and software in "traditional desktops". Allocating and managing VDC resources in a scalable and cost-effective manner poses unique challenges to cloud service providers. User workload profiles in VDCs are bursty, such as in daily desktop startup, or when a user switches between text and graphics-intensive applications. Also, the user quality of experience (QoE) of thin-clients is highly sensitive to network health variations within the Internet.

To address the challenges associated with developing scalable VDCs with satisfactory thin-client user QoE, we developed a "VMLab" infrastructure for supporting desktop virtualization experiments in research and educational user communities. This paper describes our efforts in using VMLab infrastructure to support the following:

- Desktop virtualization sandboxes for system administrators and educators
- Research and development activities relating to VDC resource allocation and thin-client performance benchmarking
- Virtual desktops for classroom lab user trials involving faculty and students
- Evaluation of the feasibility to deploy computationally intensive interactive applications in virtual desktops, such as remote volume visualization
- Educational laboratory course curriculum development involving desktop virtualization exercises

I. Motivation and Significance

Today, common user applications such as email, photos, videos, and file storage are supported at Internet-scale by cloud platforms, including HP Cloud Assure, Google Mail, and Amazon S3. Even academia increasingly is adopting cloud infrastructures and related research themes to support scientific research and education communities, such as the National Science Foundation Cluster Exploratory (NSF CluE) and the Department of Energy's (DOE) Magellan project. The next frontier for these user communities is to transition traditional distributed desktops with dedicated hardware and software installations into virtual desktop clouds (VDCs) that are accessible via thin-clients.

Moreover, in the not so distant future, we can envisage home users signing up for virtual desktops (VDs) with a VDC service provider providing Desktop-as-a-Service (DaaS) as a utility. With such a utility service, a thin-client such as a settop box can be shipped to a residential user to access a VD in a manner similar to what we have today for other common computing and communication needs, such as VoIP and IPTV. The settop box can be connected to television monitors or computer monitors, and multiple residential users can have their own unique login through this box to personalized VDs.

The drivers for transitioning traditional desktops to VDCs are obvious in terms of user convenience and cost-savings:

- Easier management of desktop support in terms of operating system, application and security upgrades
- Reduction in the number of underutilized distributed desktops unnecessarily consuming power
- Wider access to applications and data by mobile users

Allocating and managing VDC resources in a scalable and cost-effective manner poses unique challenges for service providers. User workload profiles in VDCs are bursty, such as in daily desktop startup, or when a user switches between text and graphics-intensive applications. Also, the user quality of experience (QoE) of thin-clients is highly sensitive to network health variations within the Internet. Unfortunately, existing solutions focus mainly on managing server-side resources based on utility functions of CPU and memory loads [1–4] and do not consider network health and thin-client user QoE. There is surprisingly little work being done [5–6] on resource adaptation coupled with measurement of network health and user QoE. Investigations such as [6] and [7] highlight the need to incorporate network health and user QoE factors into VDC resource allocation decisions.

It is self-evident that any cloud platform's capability to support large user workloads is a function of both server-side desktop performance as well as remote user-perceived QoE. In other words, *a CSP can provision adequate CPU and memory resources to a VD in the cloud, but if the thin-client protocol configuration does not account for network health degradations and application context, the VD is unusable for the user.* Another real-world scenario that motivates intelligent resource allocation is the fact that: *CSPs today do not have frameworks and tools that can estimate how many concurrent VD requests can be handled on a given set of system and network resources within a data center such that resource utilization is maximized, and at worst, the minimum user QoE is guaranteed as negotiated in service-level agreements (SLAs).* Resource allocations without combined utility-directed information of system loads, network health, and thin-client user experience in VDC platforms inevitably results in costly guesswork and over-provisioning, even for as few as tens of users. Also, due to lack of tools to measure the user experience from the server-side of VDCs, management functions in VDCs, such as configuring thin-client protocol parameters, often are performed using guesswork, which in turn impacts user QoE.

2. VMlab Infrastructure

A. Resources and Setup

To address the research and development challenges in developing scalable VDCs with satisfactory thin-client user QoE, we developed a "VMLab" infrastructure [8] that supports desktop virtualization experiments for research and education user communities. Initially funded by the Ohio Board of Regents, VMLab now is supported by the VMware End-user Computing Group, VMware Academic Program, Dell Education Cloud Services, and the National Science Foundation (under award numbers NSF CNS-1050225 and NSF CNS-1205658).

In the current VMLab infrastructure, an IBM® BladeCenter® S Chassis acts as a VDC data center that can concurrently support up to approximately 50 VDs. The BladeCenter has two IBM HS22 Intel® blade servers each with two quad-core CPUs, 32 GB of RAM, and four network interface cards (NICs). The storage resource is approximately 9 TB of shared SAS storage. The client-side uses a mix of several physical thin-clients from IBM, HP, and Wyse.

A netem network emulator [18] on a Linux Kernel is used for laboratory experiments to introduce network latency and loss and constrain end-to-end available bandwidth between the client and server sides. The VMware View™ desktop virtualization solution is used primarily to provision resources and broker virtual desktops, and has prerequisites such as VMware vSphere®. A web portal (http://vmlab.oar.net) enables information sharing about VMLab resources, capabilities, and salient project results. The web portal also provides information for VMLab users to gain hands-on access to run desktop virtualization experiments in their own sandboxes.

Each sandbox in VMLab has a dedicated virtual network with a separate blade allocation, as well as storage and firewall resources. Resource provisioning is performed based on user requirements and experiment plans. VPNs are set up using the OpenVPN™ server [19] for WAN connections to avoid using public IP address for VDs, and to accept VPN connections from external IP addresses. A virtual pfSense® server is used as a firewall appliance to handle all traffic between VDs and the Internet. Firewall rules are set or modified to restrict access to certain ports and addresses based on user sandbox requirements. The hypervisor, Active Directory, web portal and other supporting infrastructure are hosted in individual virtual machines within the VMLab infrastructure.

For experimentation involving distributed, multi-data center VDCs with realistic settings, VMLab resources are augmented with additional data center and thin-client resources from the NSF-supported Global Environment for Network Innovations (GENI) [17] infrastructure. The GENI infrastructure is a federated cloud of system (Emulab [20], PlanetLab [21]) and network resources (Internet2® and National LamdaRail (NLR)) for controlled as well as real-world experiments. It also provides a sliceable Internet infrastructure with wide area network (WAN) programmability using OpenFlow technologies that enable the dynamic allocation and migration of virtual machines in experiment slices. A multi-domain test bed with extended VLAN connectivity is set up between two data centers: VMLab at The Ohio State University and Emulab at the University of Utah. Distributed GENI nodes located at several university campuses (Stanford University, Georgia Institute of Technology, University of Wisconsin, Rutgers University) are used as thin-client sites.

B. Users and Activities

Over the last three years, the VMLab infrastructure has supported:

- Desktop virtualization sandboxes for system administrators and educators [8]
- Research and development activities relating to VDC resource allocation and thin-client performance benchmarking [9]–[13]
- Virtual desktops for classroom lab user trials involving faculty and students [14] [15]
- Evaluation of the feasibility to deploy computationally-intensive interactive applications, such as remote volume visualization, in virtual desktops [14] [16]
- Educational laboratory course curriculum development involving desktop virtualization exercises [13]

The desktop virtualization sandboxes were set up for several campus system administrators in Ohio, Michigan, and Texas for a variety of experiments involving VMware Virtual Desktop Infrastructure (VDI) technologies, web portal and electronic lab notebook staging, and thin-client video streaming performance testing. Educators in three departments (Dept. of Chemistry, Dept. of Industrial and Systems Engineering, Small Animal Imaging Shared Resource (SAISR)) at The Ohio State University (OSU), as well as in Polymer Ohio have experimented with VMLab resources to set up VDs for classroom labs. The classroom lab applications within VDs ranged from common applications such as Microsoft Word and Microsoft Windows Media Player to remote volume visualization applications, such as surgical simulation and polymer injection-flow modeling, that are computationally-intensive, have massive datasets, and are highly interactive in nature.

In addition, the Ohio Board of Regents CIO Advisory Board Members recently sponsored a VDPilot project, a feasibility study of a VDC for classroom labs. This VMLab related study leverages universities' pre-existing high-speed access to the OARnet network and to national networks such as Internet2 and NLR in order to assess the user QoE of accessing desktops remotely compared to physically going to a computing lab, as well as analyzing the challenges and cost savings due to shared resources amongst collaborating institutions.

Development of novel, dynamic VDC resource allocation schemes and an OpenFlow controller are ongoing, integrating them into a VDC-Sim simulator. The VDC-Sim can act as a *cloud resource broker* to "control and manage routing flows", as well as "measure and monitor user QoE delivery" in a "Run Simulation" mode, or can actually interact with VDC components in a GENI slice in a "Run Experiment" mode. VDC-Sim is being adapted to develop a graduate course curriculum involving desktop virtualization exercises in VMLab-GENI infrastructures through collaboration with the Department of Computer Science at Purdue University. These exercises include a study of resource allocation schemes, and comparing and optimizing thin-client protocol performance.

Lastly, a project involving underserved communities is being led by researchers in the Department of Computer Science and Engineering at Ohio State University. Here, researchers are working to equip the Linden community around Columbus, Ohio with VDCs as part of their emerging STEM education and other critical community support programs that can benefit from virtual desktop access.

3. Exemplar Use Cases, Experiments, and Results

A. Thin-client Performance Benchmarking Toolkit Development

Clearly, instrumentation and measurement are needed on the server and thin-client sides to gather performance data for making the best resource adaptations in VDC platforms around system loads, network health, and thin-client user QoE. VMLab resources are being used to develop a novel virtual desktop benchmarking toolkit, called VDBench, [9] to create application and user group profiles based on CPU, memory, and network bandwidth measurements.

The current VDBench prototype can measure user QoE of atomic and aggregate application tasks in terms of interactive response times or timeliness metrics such as application launch time, web page download time, "Save As..." task time. Tasks are executed via different thin-client protocol configurations, such as RDP, RGS, and PCoIP, under synthetic system loads and network health impairments. It uses the concept of "marker packets" to correlate thin-client user events with server-side resource performance events in packet captures. It also leverages measurements from built-in memory management techniques in VMware® ESX®, such as ballooning under heavy loads [22], and earlier research on slow-motion benchmarking of thin-client performance under varying network health conditions [23]. Figure 1 shows the VDBench Java client prototype. The software can run on Windows and Linux platforms, and has capabilities for NIC selection for test initiation as well as interactions with the benchmarking engine for reporting test results. Enhancements to the VDBench Java client prototype are ongoing, including the ability to install and configure the software to run on physical desktops or commercial Windows or Linux operating system embedded thin-clients.

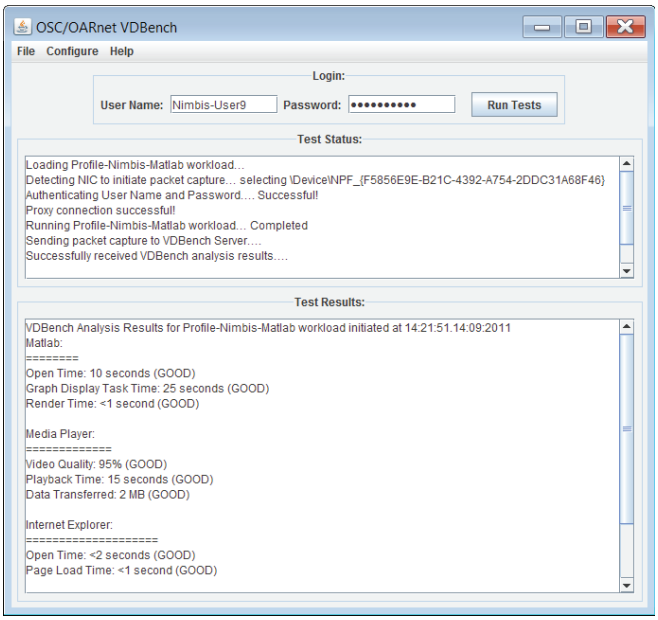

Figure 1. VDBench Java Client Prototype User Interface

B. Utility-directed Resource Allocation Scheme Development

In another set of salient research activities, application and user group profiles obtained through VDBench in VMLab are being used to develop utility-directed resource allocation schemes for VDCs at Internet scale. More specifically, we developed a utility-directed resource allocation model (U-RAM) [10] that uses offline benchmarking-based utility functions of system, network, and human components to dynamically (online) create and place VDs in resource pools at distributed data centers, while optimizing resource allocations along timeliness and coding efficiency quality dimensions. We showed how this resource allocation problem approximates to a binary integer problem whose solution is NP-hard.

To solve this problem, we proposed an iterative algorithm with fast convergence that uses combined utility-directed decision schemes based on Kuhn-Tucker optimality conditions [24]. The ultimate optimization objective was to allocate resources (CPU, memory, network bandwidth) to all VDs such that the global utility is maximized under the constraint that each VD at least meets it minimum quality requirement along timeliness and coding efficiency dimensions.

To assess the VDC scalability that can be achieved by U-RAM provisioning, simulations were conducted to compare U-RAM performance with other resource allocation models:

- Fixed RAM (F-RAM), where each VD is over provisioned, something that is common in today's cloud platforms due to a lack of system and network awareness
- Network-aware RAM (N-RAM), where allocation is aware of required network resources yet over provisions system resources (RAM and CPU) due to a lack of system awareness information
- System-aware RAM (S-RAM), where allocation is the opposite of N-RAM
- Greedy RAM (G-RAM), where allocation is aware of system and network resource requirements based purely on conservative rule-of-thumb information rather than the objective profiling used by U-RAM

Several data center sites were considered, assuming each site had 64 GB of RAM, a 100 Mbps duplex network bandwidth interface, and a scalable number of 2 GHz CPU cores. Several factors were varied during the simulation runs, including the number of data center sites, the number of CPU cores at each site, and the type of desktop pools to which incoming VD requests belonged. The simulation results clearly showed that U-RAM outperforms other schemes by supporting more VDs per core and allowing a greater number of user connections to the VDC with satisfactory user QoE.

In addition, U-RAM and F-RAM are implemented in the VMLab-GENI test bed, as illustrated in Figure 2 [11]. Using Matlab-based animation of a horse point-cloud as the thin-client application, we demonstrated that U-RAM provides improved performance and increased scalability in comparison to F-RAM under realistic settings.

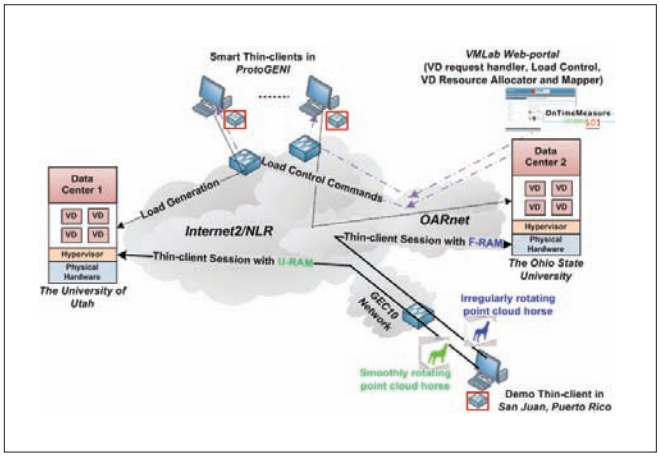

Figure 2. VMLab-GENI provisioning experiment to compare U-RAM and F-RAM schemes

The U-RAM work is being extended for provisioning VDs by investigating salient problems with subsequent placement of VDs across distributed data centers [12]. Placement decisions are influenced by session latency, load balancing, and operation cost constraints. In addition, placement decisions need to be changed over time for proactive defragmentation of resources for improved performance and scalability, as well as reactive VD migrations for increased resilience and sustained availability. Proactive defragmentation of resources is performed using global optimization schemes to overcome the resource fragmentation problem in VDCs that results from placements being done opportunistically to reduce user wait times for initial VD access. We refer to opportunistic placements as those that are performed using local schemes that use high-level information about resource status in data centers.

Over time, resource fragmentation due to careless packing of VDs on resources and changing application workloads leads to the "tetris effect" that decreases scalability (VDs per core) and performance (user QoE), thereby affecting the VDC Net-Utility. In contrast, reactive VD migrations are triggered by cyber attack or planned maintenance events, and should be performed in a manner that does not drastically affect the VDC Net-Utility. Not all VD migrations suggested by proactive or reactive schemes generate positive benefit in VDC Net-utility, since VD migration is an expensive and disruptive process. Therefore, we model the cost of migration and normalize it to utility of VDs, and migrate only the VDs (positive pairs) that generate positive Net-benefit in the VDC.

C. VDPilot: Virtual Classroom Lab User Trials

Providing access to expensive, computational software such as Matlab® and SPSS has always been a logistical and licensing challenge for professors who want to train their students with industry-standard software. Although universities have labs with pre-licensed versions of the software available, lab access for some students is inconvenient. Furthermore, many students need pervasive access to the software and have trouble obtaining a license and installing the software correctly on their home computers. Professors who want to manage lab exercises, assignments, and exams use e-mail to send and receive large files, and are limited in their ability to access and assist in the work-in-progress of students.

To address these problems, the Ohio Board of Regents CIO Advisory Board Members recently commissioned a VDPilot feasibility study for hosting virtual desktops and shared storage for classroom labs within the Ohio-based university system. The study was initiated to investigate the use of federated shared infrastructure resources that would simplify classroom lab computing for faculty and students, and reduce costs for universities.

As part of the VDPilot study, VMLab was reconfigured to support subjective testing for approximately 50 faculty and students with secure remote access to lab software using thin-clients over the Internet [15]. User trials were conducted with professors and students, as well as some IT administrators, who were asked to compare going to a physical lab versus using the remote thin-clients while performing tasks in the virtual desktops using applications such as Microsoft Excel®, Matlab, SPSS, Windows Media Player, and Internet Explorer®.

Figure 3 shows the VDPilot survey (screenshot) that participants completed after following subjective testing instructions provided to them through the VDPilot web portal.

Figure 3. VDPilot survey participants completed after subjective testing

The survey results showed that 50 percent of participants found the virtual desktop user experience to be comparable to their home computer's user experience, while 17 percent could not decide which user experience was better. Interestingly, 8 percent found the virtual desktop user experience to be better than their home computer user experience, particularly in the case of resource-intensive applications such as SPSS. Quotes recorded from faculty and students indicated they liked the virtual computer lab access in the pilot project in its current form. Two of the professors who participated in the study were eager to have their students use the pilot project test bed immediately as part their ongoing course offering. This confirms there is a real and current need for hosting virtual desktops and shared storage for classroom labs at universities.

D. Remote Interactive Volume Visualization for Researchers

VMLab resources have been used in experiments for the evaluation and support of a Remote Interactive Volume Visualization Infrastructure for Researchers (RIVVIR) [14] [16] to serve an increasing user base in the Small Animal Imaging Shared Resource (SAISR) at the Ohio State University and Polymer Ohio communities. RIVVIR provides an environment in which users can access VDs that host computationally-intensive interactive applications and their related massive data sets.

Given the growing trend of users of data-intensive remote volume visualization applications that deal with gigabyte to petabyte sized data sets, it is impractical to carry or download these data sets and run computational analyses. Users of such applications in communities such as the ones supported by SAISR and Polymer Ohio inevitably must use VDs that have high-performance computing (HPC) capabilities at the data location and high-speed intermediate networks. Moreover, VDs allow a visual interpretation of large data sets, a powerful medium that can foster science and engineering innovations. Further, RIVVIR allows users to access their computationally-intensive interactive applications using handheld devices. They can jointly collaborate on the application steering with researchers at other institutions for visualization and analytics tasks related to their research and development efforts.

In our RIVVIR development efforts, we are interested in exploration that is beyond the classical thin-client model. We are exploring hybrid computing models and advanced multimedia stream content processing schemes, where execution is apportioned between thin-clients and the back-end server. For high-motion session output or computationally-intensive rendering such as 3D, we are investigating thin-client protocol optimizations that leverage increased server processing power to improve frame rate. The session switches to a general thin-client protocol configuration for low-motion video and other routine rendering.

In addition, we are evaluating the potential of caching repetitive video blocks, such as desktop backgrounds and menu items, to reduce server-side processing, bandwidth consumption, and interaction delays. Furthermore, there has been a trend lately in the use of thin-clients with high-resolution displays and significant computing power, especially with the latest generation of Apple iPads and similar products. To support these emerging thin-client platform applications, we are exploring related hybrid computing issues, where computing is distributed between the client and server ends, depending upon application context.

Figure 4 shows the RIVVIR configurations within VMLab for SAISR and Polymer Ohio community users for high-end demonstrations [14] [16]. A remote SAISR or Polymer Ohio user accesses a VD application similar to other users of typical applications, such as Microsoft Word or Internet Explorer, while their computationally-intensive interactive applications in the back-end rely on HPC infrastructure at the Ohio Supercomputer Center (OSC).

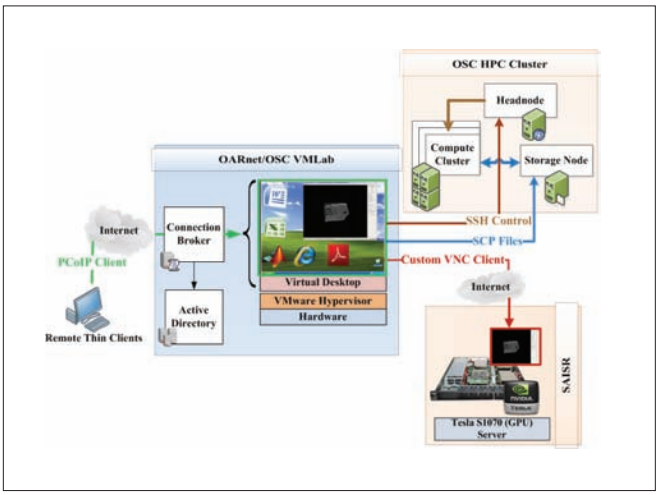

Figure 4. RIVVIR configurations for SAISR and Polymer Ohio community users

There are several challenges in configuring VDCs to support such applications due to their computational, storage, and network resource requirements for optimal user QoE. As in the case of SAISR users, custom applications need to be deployed that may rely on legacy thin-client protocols such as VNC, which are unsuitable for large delay or lossy networks that are common on the commercial Internet, as shown from our previous studies [25] [26]. We plan to

experiment with various reconfigurations of the enhanced VMLab infrastructure to study optimizations in thin-client protocols, such as tunneling and network-awareness, that can deliver satisfactory remote volume visualization user QoE. In the case of tunneling experiments, working through campus firewalls is a challenge. Additionally, we plan to address challenges in tunneling legacy protocols within the latest thin-client protocols, such as PCoIP to support remote users with large delay or lossy networks between thin-clients and servers.

4. Conclusion

Our VMLab deployment efforts have provided valuable operations experience to support a wide variety of research and education use cases relating to desktop virtualization experimentation. These experiences are helping significantly the engineering and operations of production services for desktop virtualization at the Ohio Supercomputer Center and OARnet, and new service models for research and education are being developed for user communities. Developers of the GENI community within infrastructure groups and instrumentation and measurement services groups have developed new capabilities by supporting unique experiment requirements of our VDC Future Internet application. We expect future desktop virtualization experimenters to benefit from these advancements.

The number of researchers and educators desiring to use VMLab resources is increasing rapidly, and several new initiatives are looking to use VMLab to scale to a large number of actual users in classroom labs and underserved communities. Our ongoing research and development projects on VDC resource allocation and thin-client performance benchmarking are ramping up for more extensive experiments. As a result, the VMLab infrastructure is being enhanced to support the concurrent provisioning up to 150 VDs. In addition, more than 20 physical thin-clients are being deployed. These units can be shipped to end-user sites for VDC experiments with a diverse group of geographically distributed users. This will enable us to validate successful DaaS offerings, where service providers can observe performance and control the end-to-end components to consistently meet SLAs and deploy an economically viable service delivery model.

We believe research and development outcomes from VMLab enable the realization of the next frontier—one that transforms end-user computing capabilities in the future Internet and enables society to derive benefits from computer and network virtualization. As a result, VMLab infrastructure enhancements are focused on collecting valuable real-world data sets to gain a better understanding of workload profiles for diverse applications and user groups in VDCs. This is the kind of understanding, missing in today's research literature, that could, in turn, fuel the development of new frameworks and tools to allocate and manage resources for improved performance, increased scalability, and cost-effective VDCs.

Acknowledgment

This material is based upon work supported by the VMware Academic Program and the National Science Foundation under award numbers CNS-1050225 and CNS-1205658. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of VMware or the National Science Foundation.

The following students and colleagues at The Ohio State University have contributed towards the various VMLab projects described in this paper: Rohit Patali, Aishwarya Venkataraman, Mukundan Sridharan, Yingxiao Xu, David Welling, Saravanan Mohan, Arunprasath Selvadhurai, Sudharsan Rajagopalan, Rajiv Ramnath, and Jayshree Ramanathan.

References

1 D. Gmach, S. Krompass, A. Scholz, M. Wimmer, A. Kemper, "Adaptive Quality of Service Management for Enterprise Services", *ACM Transactions on the Web,* Vol. 2, No. 8, Pages 1-46, 2008.

2 P. Padala, K. Shin, et. al., "Adaptive Control of Virtualized Resources in Utility Computing Environments", *Proc. of the 2nd ACM SIGOPS/EuroSys, 2007.*

3 B. Urgaonkar, P. Shenoy, et. al., "Agile Dynamic Provisioning of Multi-Tier Internet Applications", *ACM Transactions on Autonomous and Adaptive Systems*, Vol. 3, No. 1, Pages 1-39, 2008.

4 H. Van, F. Tran, J. Menaud, "Autonomic Virtual Resource Management for Service Hosting Platforms", *Proc. of ICSE Workshop on Software Engineering Challenges of Cloud Computing,* 2009.

5 K. Beaty, A. Kochut, H. Shaikh, "Desktop to Cloud Transformation Planning", *Proc. of IEEE IPDPS,* 2009.

6 N. Zeldovich, R. Chandra, "Interactive Performance Measurement with *VNCplay*", *Proc. of USENIX Annual Technical Conference,* 2005.

7 J. Rhee, A. Kochut, K. Beaty, "DeskBench: Flexible Virtual Desktop Benchmarking Toolkit", *Proc. of Integrated Management (IM),* 2009.

8 P. Calyam, A. Berryman, A. Lai, R. Ramnath, "VMLab Testbed for Desktop Virtualization to Support Research and Education", *MERIT Desktop Virtualization Summit,* 2010, http://vmlab.oar.net.

9 A. Berryman, P. Calyam, A. Lai, M. Honigford, "VDBench: A Benchmarking Toolkit for Thin-client based Virtual Desktop Environments", *IEEE Conference on Cloud Computing Technology and Science (CloudCom),* 2010.

10 P. Calyam, R. Patali, A. Berryman, A. Lai, R. Ramnath, "Utility-directed Resource Allocation in Virtual Desktop Clouds", *Elsevier Computer Networks Journal (COMNET),* 2011.

11 P. Calyam, M. Sridharan, Y. Xiao, K. Zhu, A. Berryman, R. Patali, "Enabling Performance Intelligence for Application Adaptation in the Future Internet", *Journal of Communications and Networks (JCN)*, 2011.

12 M. Sridharan, P. Calyam, A. Venkataraman, A. Berryman, "Defragmentation of Resources in Virtual Desktop Clouds for Cost-Aware Utility-Optimal Allocation", *IEEE Conference on Utility and Cloud Computing (UCC)*, 2011.

13 P. Calyam, A. Venkataraman, A. Berryman, M. Faerman, "Experiences from Virtual Desktop Cloud Experiments in GENI", *GENI Research & Educational Experiment Workshop (GREE)*, 2012.

14 P. Calyam, D. Stredney, A. Lai, A. Berryman, K. Powell, "Using Desktop Virtualization to access advanced Educational Software", *Internet2/ESCC Joint Techs*, Columbus, OH, 2010.

15 P. Calyam, A. Berryman, D. Welling, S. Mohan, R. Ramnath, J. Ramnathan, "VDPilot: Feasibility Study of Hosting Virtual Desktops for Classroom Labs within a Federated University System", *International Journal of Cloud Computing*, 2012.

16 A. Lai, D. Stredney, P. Calyam, B. Hittle, T. Kerwin, D. Reed, K. Powell, "Remote Interactive Volume Visualization Infrastructure for Researchers", *AMIA Annual Symposium*, 2011.

17 Global Environment for Network Innovations, http://www.geni.net

18 The Linux Foundation Netem, http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

19 OpenVPN, http://openvpn.net/

20 Emulab/ProtoGENI Infrastructure, http://www.emulab.net

21 PlanetLab- GENI, http://groups.geni.net/geni/wiki/PlanetLab

22 C. Waldspurger, "Memory Resource Management in VMware ESX Server", *ACM SIGOPS Operating Systems Review*, Vol. 36, Pages 181 - 194, 2002.

23 A. Lai, J. Nieh, "On The Performance Of Wide-Area Thin-Client Computing", *ACM Transactions on Computer Systems*, Vol. 24, No. 2, Pages 175-209, 2006.

24 R. Rajkumar, C. Lee, J. Lehoczky, D. Slewlorek, "A Resource Allocation Model for QoS Management", *Proc. of IEEE RTSS*, 1997.

25 P. Calyam, A. Kalash, A. Krishnamurthy, G. Renkes, "A Human-and-Network Aware Encoding Adaptation Scheme for Remote Desktop Access", *Proc. of IEEE MMSP*, 2009.

26 P. Calyam, A. Kalash, R. Gopalan, S. Gopalan, A. Krishnamurty, "RICE: A Reliable and Efficient Remote Instrumentation Collaboration Environment", *Journal of Advances in Multimedia's special issue on Multimedia Immersive Technologies and Networking*, 2008.

vQuery: A Platform for Connecting Configuration and Performance

Ilari Shafer
Carnegie Mellon University

Snorri Gylfason
VMware, Inc.

Gregory R. Ganger
Carnegie Mellon University

Abstract

Discovering the causes of performance problems in virtualized systems is often more difficult than without virtualization, because they can be caused by changes in infrastructure configuration rather than the user's application. vQuery is a system that collects, archives, and exposes configuration changes alongside fine-grained performance data, so the two can be correlated. It gathers configuration change data without modifying the systems it collects from and copes with platform-specific details within a general, graph-based model of Infrastructure-as-a-Service (IaaS) infrastructures. Configuration data collected from two VMware® vSphere™ environments reveals that configuration changes are frequent and involved, opening interesting new directions for configuration-aware performance diagnosis techniques.

1. Introduction

Consolidating computing activities onto shared infrastructures, such as in cloud computing and other virtualized data centers, offers substantial efficiency benefits for providers and consumers alike. But, it also introduces complexities when trying to understand the performance behavior of any given activity, since it can depend on many factors not present when using dedicated infrastructure. For example, the VMs used for the activity can migrate or be resized, or new VMs for other activities can be instantiated on shared hardware. As on-demand resource allocation (as in cloud computing) and automated configuration optimization (e.g., via VMware DRS[1]) grow more common, such factors increasingly create potentially confusing performance effects.

Traditionally, to understand application performance and diagnose performance problems, administrators and application engineers rely on resource usage instrumentation data from infrastructure runtime systems, such as time-sampled CPU utilization, memory allocated, and network packets sent/received. In VM-based infrastructures, the same data types can be captured for each VM as well. But, while such data exposes how resource usage changed at a given point in time, it offers little insight into why. Deducing why, so that one can decide what (if any) reactive steps to take, often is left entirely to the intuitions and experience of those involved in the diagnosis.

We believe an invaluable additional source of information should be captured and explicitly correlated with resource utilization data: the configuration history. Of course, any runtime infrastructure maintains its current configuration, and many log at least some configuration changes. Since configuration changes often cause performance changes, purposefully or otherwise, correlating the two should make it possible to highlight root causes of many problems automatically. In addition, the combination of the two offers the ability to expose powerful insights for system management and automation, such as which configuration changes usually improve performance and how particular problems were overcome in the past.

This paper describes our prototype system (called vQuery) for configuration change tracking and mining, together with initial experiences. vQuery collects time-evolving configuration state alongside fine-grained resource usage data from a VMware vCloud™-based infrastructure, stores it, and allows it to be queried. Configuration changes are captured by listening to vSphere's API and vCloud's internal update notifications. They are stored as a time-evolving graph of entities (e.g., VMs and physical hosts) as vertices and relationships as edges. This general approach avoids changes to the infrastructure software, accommodates a range of IaaS systems, and allows a range of configuration history queries.

We deployed vQuery on a local VMware software based private cloud (referred to as Carnegie Mellon's vCloud) as well as a VMware testbed, with positive initial results. We illustrate some of the power of configuration change history with interesting anecdotes and data from these deployments, and discuss challenges still ahead on this line of research.

The remainder of this paper is organized as follows. Section 2 explains what we mean by "configuration data" and configuration changes in more detail, including examples from VMware systems. Section 3 describes the design and current implementation of vQuery, focusing on how configuration data is captured, stored, and queried. Section 4 presents some data, early experiences, and anecdotes from vQuery deployments. Section 5 discusses our ongoing research on vQuery and exploiting configuration change data. Section 6 discusses related work.

2. Configuration Data and Changes

In a distributed computing infrastructure, various types of configuration are spread across files, databases, and within software. The word "configuration" often is used for concepts that include command-line flags to programs, OS-level settings, and the layout of virtual machines across computing resources. In this work, we focus on the last type: infrastructure-level properties that affect how virtualized environments, such as vSphere and vCloud, function and that reflect their current state.

Even this type of configuration is very heterogeneous. Some data are as simple as key-value pairs, but other data encodes lists, objects, and hierarchies. Some is controlled by end users (e.g., the choice of guest operating system for a VM), some is primarily automated by the computing infrastructure (e.g., which IP address a VM is assigned), and some can be managed by both (e.g., the choice of physical resources that back a VM). More concretely, Table 1 shows a selection of configuration properties in a VMware-based environment, ranging from simple descriptive properties to relationships with other entities.

ENTITY TYPE	CONFIGURATION PROPERTIES
VM	**vSphere**: *host system*, *networks*, *datastore*, name, annotation, memory, vCPUs, CPU allocation (reservation/limit/shares), memory allocation (reservation/limit/shares), virtual disk layout (chain length), power state, guest OS type, guest OS state, guest OS screen dimensions, guest NIC (IPs, network, state), guest disk (capacity, free space, path), IP address, VMware tools state + version **vCloud**: *vApp*, *networks*, name, vCPUs, memory, guest OS, status, storage
vApp	**vCloud**: *owner*, *vDC*, *networks*, status
User	**vCloud**: name, VM quota
Host	**vSphere**: *network*, CPU (frequency, number of cores and packages), memory size, power state
Network	**vSphere**: name **vCloud**: fence mode, parent, DNS (addresses, suffix), netmask, IP ranges
Datastore	**vSphere**: name, capacity, free space, type, url

Table 1: selected configuration properties in vSphere and vCloud. The properties that represent other entities are shown in italic.

Modeling configuration consistently is one focus of vQuery. In addition to the different types and meanings of configuration within vSphere and vCloud, different virtualized infrastructures expose different configuration properties. For example, where vSphere has a platform-independent datastore abstraction, the OpenStack infrastructure platform separates storage into block storage, local storage, and a separate VM image service. We would like to represent configuration in a sufficiently general way to model such different environments.

A key aspect of configuration on which we focus is that much of it changes over time. Some configuration properties may change very slowly (e.g., the amount of RAM on a physical host is seldom adjusted), while others are increasingly dynamic (e.g., the placement of a VM on a physical host is adapted by DRS). In tracking configuration as it relates to performance, we focus on recording changes in order to ask questions such as "was there a relevant configuration change around the same time as a given performance change?" and "what were all the configuration changes associated with a given VM?" Beyond diagnosis, maintaining a change history can also help us understand how and why systems evolve[2].

Additionally, infrastructure configuration is not a collection of unrelated facts. Configuration properties are associated with entities, whether physical (hosts and physical networks) or virtual (VMs and users), and these entities are meaningfully related. For example, VMs are placed on physical hosts, and users own vApps, which contain VMs. Examples of these "relational" properties are shown in italic in Table 1. We believe maintaining information about the relational structure of configuration—and how it changes—is important for diagnosis. It is intuitively important to be able to ask questions, such as "which VMs were on a given host when there was a performance problem?" Additionally, a variety of diagnosis approaches have taken advantage of the fact that the effects of changes often propagate through causal dependencies among the components of a distributed system[3 4 5], many of which are directed along these relations.

3. vQuery: Design

vQuery is designed to track fine-grained configuration data in a way that maintains the features described above. At a high level, the problems we need to solve are the same as those for performance monitoring: how to collect, store, and access configuration data. A simplified overview of our approach is shown in Figure 1, and this section describes each component in turn.

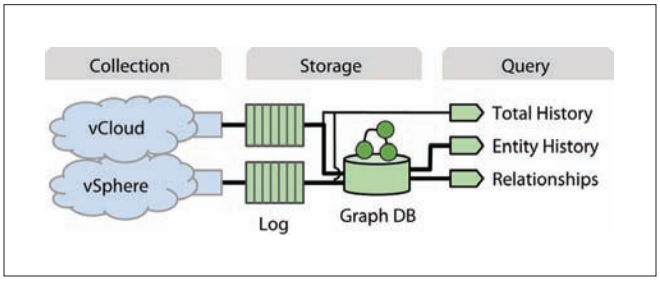

Figure 1: high-level overview of the vQuery configuration tracking system. It collects data from vSphere and vCloud, stores them in separate logs, and ingests them into a graph database to be queried.

3.1 Configuration Collection

Changes to configuration occur from both human and automated sources, and they clearly do not happen only at a fixed interval. It is insufficient for just the current configuration to be exposed by infrastructure APIs. For the collection process to be more efficient

and accurate than polling, there must be some way of obtaining updates. Ideally, the mechanism should require minimal, if any, modification of the infrastructure. We built our prototype without modifying code in vSphere or vCloud.

For vSphere, we build on the existing interface for subscribing to update notifications. Specifically, we use a PropertyCollector and its WaitForUpdates method to receive changes to a set of configuration properties of interest.

Although vCloud does not currently offer such an interface, we collect configuration from vCloud by listening to internal messages as a signal for when to query its configuration API. As an example, vCloud sends a message to start an action (e.g, start a VM, (1) in Figure 2), which results in a message sent to an AMQP message bus (1) and actions in vSphere (2). When the task is finished, a completion message is posted to the message bus. Our configuration collector listens to the same AMQP message bus (3), filters to listen to only task completion messages, and queries an appropriate API to find details about configuration change after a task completes (4).

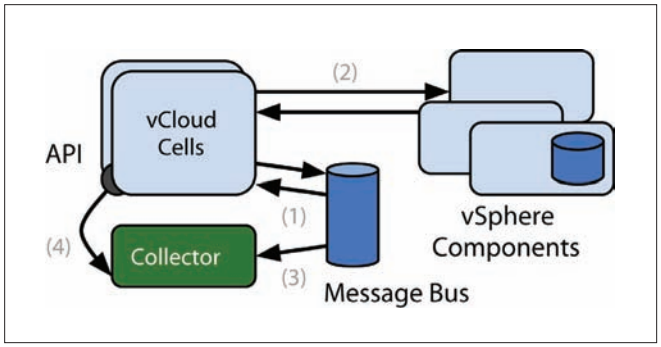

Figure 2: Configuration collection strategy for vCloud and OpenStack. The configuration collector listens to messages on the vCloud message bus and polls an appropriate API upon intercepting a task completion message.

This design pattern is not restricted to vCloud. The recently popular OpenStack IaaS also uses an AMQP message bus for inter-node communication[6]. We built enough of a collector for OpenStack to confirm that messages can be intercepted for configuration event notification. The same technique is not limited to message bus transports. For example, systems based on bare Remote Procedure Calls (RPC) could be instrumented similarly, albeit with a lower-level interceptor. In addition to vSphere, we have started collecting limited configuration data from an instance of the Tashi cluster management system[7].

In an ideal world, we would capture changes to all types of configuration that might affect performance, including those from the application layer. For example, recent research describes mechanisms for capturing changes to configuration files within guest VMs without modifying guest software[8]. Integrating such changes with those accessible from vCloud and vSphere is a direction for future work.

3.2 Configuration Storage

A primary challenge in storing configuration is how to represent it. Here, we describe a time-evolving representation of configuration information that is designed to support historical and relational queries using a general model. The representation is a graph with a loose schema—formally a typed, directed, attributed multigraph that also tracks time.

Infrastructure entities (VMs, physical hosts, storage nodes, networks, and so on) are vertices of this graph. Each vertex is associated with three mandatory fields: a unique identifier (id), a type (VM, host, and so on), and a valid-time interval[9]. Infrastructure entities can be created and removed over time (as in Figure 3, VMs can be allocated and deleted), but their historical presence must be remembered to support retrospective analytics. Each vertex also has a map of attribute names to a list of time-changing values ordered by time. The intuition behind this format is that each infrastructure attribute can change over time (e.g, a user changing the allocation of a VM, as shown as vRAM in Figure 3). The after-image of each value is appended to the list. Our implementation currently supports primitive types (strings, integers, floating-point numbers) and arrays thereof.

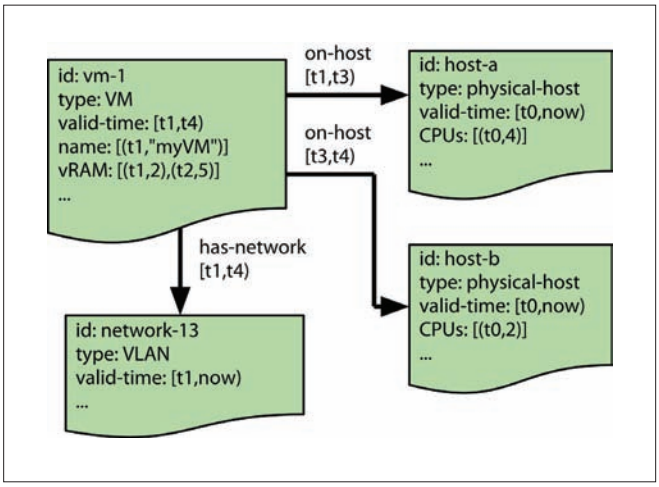

Figure 3: property graph of changing configuration. Each entity (VM, host, etc.) is associated with a number of time-evolving properties, in addition to time-evolving relationships with other entities. A unique identifier (id) and type of entity (type) are the only two required properties. The properties that track relationships (e.g. host) are specified by a user.

Edges between entities have two mandatory fields: a type of relationship and a valid-time interval. Similar to entities, such a relationship often exists only for a given time interval. For instance, in Figure 3, **vm-1** moved from **host-a** to **host-b** at time **t3** and was removed at time **t4**. In this way, the graph captures events such as VM migration not simply as events but as changes that relate entities. These edges—akin to foreign keys—are the schema of the configuration graph. Administrators must specify which configuration attributes have semantic meaning as dependencies (and must contain identifiers as values).

Maintaining configuration in this format also allows for the use of existing graph databases as an underlying persistence layer—in particular, those that store property graphs and have the ability to build indexes on properties.

Updating the graph, while relatively straightforward in principle, requires care in practice. The principle of the algorithm is reminiscent of that used to update a transaction-time state table in a temporal database[10], as applied to a property graph. Unfortunately, when receiving configuration updates from different layers of infrastructure, dependencies can be reported before the entities to which they refer. Consider the following ordered sequence of observations, similar to events observed in practice:

1. The VLAN **network-1** is created
2. vCloud reports that **vm-1** is connected to **network-1**
3. vSphere reports the existence of **network-1**

The final desired graph should contain a **has-network** edge from **vm-1** to **network-1**. If updates are applied in the given order, the graph will contain an invalid edge after step #2, since the existence of **network-1** is not yet known. We maintain a set of these "pending edges," which are scanned as new updates arrive. If one matches a newly-created entity the dependency is added with the original valid-time. As a beneficial side effect, this technique allows the update algorithm to operate with insertion batches atop the transactional graph database used (Neo4j[11]).

One drawback of storing configuration so generally is that we push the problem of forming meaningful queries to the querier. For example, retrieving a list of VMs requires selecting entities with the VM type rather than scanning a table named "VMs." Also, we assume loosely synchronized timestamps across different reporters of entity information, a property provided by the underlying VMware infrastructure.

3.3 Configuration Query

To ask questions about configuration history, we build a few abstractions on top of the graph database to supplement its query language[12]. Here, we focus on a few that align with our primary goals of historical queries that provide the history of an entity or the system, and relational queries that discover entities that likely depend on or influence each other.

- Historical: get-backlog(t_{start}, t_{end}): obtain all configuration changes to any entity between times t_{start} and t_{end}.
- Historical: get-property-names(*E*): get a list of properties associated with entity *E*, followed by get-property (*E*, *name*) to get a time-ordered list of changes to the property with name *n*.
- Relational: get-subgraph (*E*, *d*): do a breadth-first traversal of entities connected to entity *E*, up to a maximum depth *d* (or, with get-subgraph(*E*, *n*), up to a maximum number of entities *n*).

3.4 Performance Collection

In addition to the technique for storing configuration data described above, a source of performance data is necessary to connect configuration with performance. The performance data we consider consists of time series streams of metrics reported by the hypervisor and aggregated by management software. In contrast to configuration data, many mature systems exist for collecting and archiving this data at the infrastructure level[13 14 15].

For performance data collection, we use the StatsFeeder prototype described in more detail in the first issue of the VMware technical journal[16]. We collect nine metrics from each virtual machine and 15 metrics from each physical host every 20 seconds. These performance metrics are described briefly in Table 2.

VIRTUAL MACHINE	
CPU	**usage**: time used by this VM **system**: time spent in the VMkernel **wait**: time spent waiting for hardware/kernel locks **ready**: time spent waiting for a CPU (e.g., on an oversubscribed host) **guaranteed**: time used of the total guaranteed to the VM **extra**: time used beyond what the VM was originally assigned
Memory	**swapped**: amount of VM memory swapped out to disk **swaptarget**: amount of memory the VMkernel is aiming to swap **vmmemctl**: size of the memory balloon
HOST	
CPU	**usage**: aggregated time the CPU was used **idle**: time the CPU was idle
Disk	**usage**: average disk throughput **read**: average read throughput **write**: average write throughput **commands**: disk commands issued **commandsAborted**: disk commands aborted **busResets**: SCSI bus reset commands **numberRead**: number of disk reads **numberWrite**: number of disk writes
Network	**packetsRx**: packets received **packetsTx**: packets transmitted **usage**: average transmit + receive KB/s **received**: average receive KB/s **transmit**: average transmit KB/s

Table 2: collected performance data. All metrics are times, averages, or sums over a sample period (20s).

4. Early experiences with vQuery

A full evaluation of the vQuery framework would assess whether it can answer real diagnosis and monitoring queries. Although the project is still in the preliminary stage, this section provides some early experiences with configuration data collection and synthetic relational queries.

4.1 Historical

A functional configuration monitor collects and stores configuration changes over extended periods. This section describes some of the output from the two vSphere instances to which vQuery has been connected. The Virtual SE Lab (vSEL)[17] is an environment at VMware that is used for events at VMworld, training, and demos. We collected configuration changes from it a month prior to VMworld 2011. The vCloud at Carnegie Mellon (CMU) is a cloud we deployed to support academic workloads from courses, individual researchers, and groups with large research computing demands submitted via batch schedulers. Table 3 lists a few basic metrics of configuration change for each environment.

The last row of Table 3 highlights the diversity of configuration—and the need to be somewhat selective in what is collected and retained. One of the configuration properties exposed by vSphere and collected in the CMU dataset was datastore free space, a frequently updated property that accounted for over 63% of the configuration changes we observed. Although free space changes can be important to monitor, either collecting them infrequently or treating them as time series metrics (rather than as configuration changes) is more appropriate.

	vSEL	CMU
Collection period	12 days, starting 21 July 2011	75 days, starting 12 July 2012
Number of physical hosts	100	15
Number of changes	27888	63820
Number of configuration properties gathered	11	36
Number of changes, less free space changes	27888	23466

Table 3: Basic metrics of configuration change from two vSphere instances.

To better understand configuration changes that have occurred, visualization is crucial. As one example view, Figure 4 shows the configuration changes that occurred in the 75-day CMU dataset. Since there are so many types of configuration changes, we only show the top 10 types of change in the legend (by number of changes). A number of noteworthy events are visible from temporal and spatial groups on the chart. (See the caption for detail.)

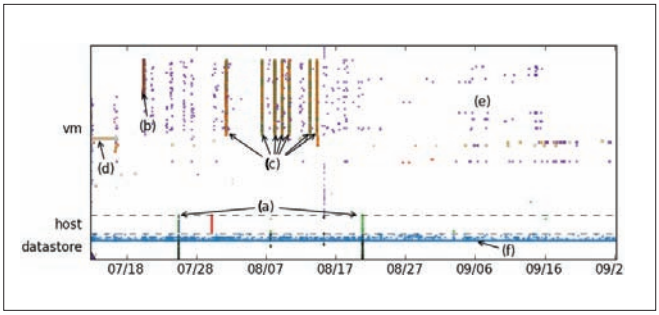

Figure 4: Configuration changes to a small datacenter. Dots represent configuration events to VMs, hosts, and datastores (spread on the vertical axis) across the horizontal time axis. The labeled periods are:

a) changes to many hosts and datastores around the time of a switch outage (first event) and switch replacement (second event) in another virtual datacenter

b) a user adding 30 VMs to an existing set of VMs to run experiments

c) the same user restarting the entire set of VMs when they became unresponsive

d) a user setting up a Windows VM, including many restarts

e) many points in this region (and between (b) and (c)) are VM migrations

f) this row of changes is primarily changes to datastore free space. (The VM disk free space changes shown in Table 3 are filtered out of this image.)

An additional observation we make about the snapshot of configuration in Figure 4 is that many configuration changes co-occur. For example, when VMs are restarted (e.g., the events marked as (c)), their power state changes along with the status of VMware tools in the guest OS and the status of its connection to a virtual network. Together, these changes represent the event "VM restart". Attributing its performance effects to a single one of these changes (particularly a change such as the state of VMware tools) would be misleading. Together with the observation in Table 3 that some configuration events are less meaningful than others, distilling semantically meaningful changes from the noise in configuration will be an important step forward.

4.2 Relational

One important aspect of vQuery is providing query access to related entities, which builds on database support for rapid neighborhood queries in the spatio-temporal configuration graph. We use a graph database (Neo4j); these databases are typically optimized for fast constant-time adjacency lookup[18]. This feature is one key way to manage queries across large graphs: the entities that are closer through dependency traversal are those that are more likely related.

For example, when performing a diagnosis query involving the performance of VM v, likely culprits include configuration changes to its resources (e.g., compute, networking, storage), which are within a traversal distance of 1. Furthermore, other VMs contending for those resources are also of interest, and are within a distance of 2. Although infrequent, relationships with a distance of 3 also arise: VMs in vCloud are modeled as abstract entities that are backed by VMs in vSphere.

Correlating configuration changes from a vCloud VM to a colocated vSphere VM needs 3 hops. If one needs to connect configuration changes to another vCloud VM, the distance would be 4 hops. Most cases involve just 1-2 hops.

To demonstrate that queries in common cases are relatively fast, Figure 5 shows the time required to run a query starting over the largest portion of the vSEL configuration graph. We run queries starting from a random entity in the 1821-entity graph up to a given depth. One can observe that querying for entities separated by a distance of 1 is fast (typically less than two milliseconds), and queries to distance 2 are typically under 10ms.

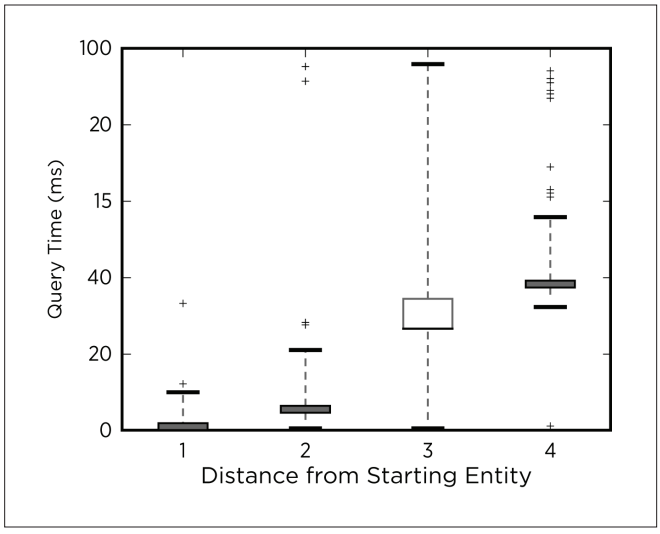

Figure 5: time taken to retrieve related entities to a random starting entity. 1,000 queries were performed at each distance, and boxes extend to the interquartile range.

5. Next Steps

As described above, vQuery forms an infrastructure for collecting, storing, and querying fine-grained configuration and performance data. Moving forward, we plan to use these augmented sources of monitoring data to perform more accurate diagnoses than with traditional black-box performance data alone. In particular, our next aim is to find configuration changes that are the root cause of performance problems. Three concrete examples are:

- **Short-term changes**. VM migration performed by DRS and virtual disk migration performed by storage DRS require network bandwidth and physical host resources. By monitoring performance, we hope to observe the short-term impact of these mechanisms and attribute it to the host and storage configuration changes we observe. We believe the fine-grained performance information we collect will be important to distinguish these performance variations, in addition to recent historical configuration
- **Contention**. virtualized workloads contend for resources, and perfect isolation is not yet a reality across resources, such as caches and disks. Migrating or starting workloads that use a host, datastore, or network can be a source of performance variation for VMs sharing that resource. The configuration changes we measure include migrations and power state changes, which we hope to correlate with performance monitoring data of contending entities. We believe relational queries will be necessary to identify configuration changes that occur to "neighbors," which are potential sources of contention.
- **Explaining parameters**. simply understanding which performance metrics are influenced by a configuration change can be a valuable source of guidance when identifying configuration-related problems, since the impact of configuration parameters often is unclear from name or documentation alone. Identifying performance changes related to configuration could allow us to annotate configuration parameters with the metrics they affect, providing guidance towards how they behave.

6. Related Work

6.1 Configuration Management

Recognition of the complexity of deploying and managing applications across clusters has spurred many configuration management efforts. Tools that have received recent attention include Chef[19] and Puppet[20], which focus on automated application deployment and configuration. CFengine[21] was among the first such tools, designed to reduce the burden of manually scripting policies and configurations across Unix workstations. It has since added support for deploying policies across the cloud computing environments we consider.

These tools primarily facilitate the creation of configuration rather than monitoring changes over time. That is, most focus on actuating configuration rather than monitoring what exists. CFengine is notable among the examples above for also incorporating a familiar-sounding notion of "knowledge management," which is a collection of facts about infrastructure and the relationships between them.

6.2 Correlating Configuration with Performance

Much work on understanding the connection between configuration and performance is focused on tuning configuration to optimize application performance. At least a few techniques, though, focus on our primary motivating use case: finding configuration changes that are the root cause of performance changes.

Many of these techniques have emerged from work on diagnosis in large-scale networks. MERCURY[22] considers an instance of the problem in ISP networks, and identifies the impact of upgrades and routing configuration changes on time series performance indicators, such as CPU utilization and packet loss. Whereas MERCURY considers mostly long-term changes in performance, PRISM[23] operates in the same setting and focuses instead on shorter time-scale changes, such as "spikes." WISE[24] also operates on ISP configuration and performance, but uses it to answer questions of the form "what would be the performance impact of making a configuration change?"

In the context of distributed applications, although NetMedic[25] uses two known snapshots in time as "good" and "problematic" points for diagnosing application-level errors, it uses some of the same concepts discussed here—notably, inference based on system performance data and an (automatically generated) dependency graph. ASDF[26] also correlates multiple time evolving measurements, similar to the black-box monitoring data described here, to perform root-cause diagnosis of performance problems.

6.3 Problem Diagnosis

Our work shares high-level goals with efforts to diagnose problems in distributed systems using widely available black-box performance metrics, such as CPU time and network throughput. For instance, Kasick et al. use statistical comparison across multiple machines to perform root-cause diagnosis in parallel file systems[27]. At the application level, work focused on multi-tier distributed systems has used time series CPU performance metrics to localize faults to individual machines[28], and domain-specific counters in IP networks[29].

By taking advantage of deeply instrumented "white-box" systems, a broader range of distributed system diagnosis techniques have been used for finding the sources of performance problems. For example, end-to-end traces , which track activity as it moves across system components, can be a rich source of insight[30]. Spectroscope[31] is one such tool that leverages these traces for root-cause performance problem diagnosis.

7. Summary

In virtualized environments, such as VMware vSphere, the additional indirection between workloads and the resources they use can lead to additional challenges when finding the source of performance problems. Infrastructure configuration changes can be a hidden source of performance variation. Identifying such effects requires configuration change capture and analysis. vQuery is a system for tracking configuration changes so that we can correlate them with traditional performance data, and early experiences with it are promising. Moving forward, we plan to integrate the data we collect to automatically produce insight about configuration-related performance problems in virtualized infrastructures.

References

1 A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu. "VMware distributed resource management: design, implementation, and lessons learned." VMware Technical Journal, vol. 1, no. 1, pp. 47–64, 2012.

2 H. Kim, T. Benson, and A. Akella. "The Evolution of Network Configuration: A Tale of Two Campuses," IMC 2011.

3 A. A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and J. Emmons, "Detecting the performance impact of upgrades in large operational networks." ACM SIGCOMM Computer Communication Review, vol. 40, no. 4, pp. 303–314, 2010.

4 P. Bahl, R. Chandra, and A. Greenberg, "Towards highly reliable enterprise network services via inference of multi-level dependencies," ACM SIGCOMM 2007.

5 M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," NSDI 2004.

6 OpenStack, http://www.openstack.org.

7 M. A. Kozuch, M. P. Ryan, R. Gass, S. W. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. López, M. Stroucken, and G. R. Ganger, "Tashi: location-aware cluster management," ACDC 2009.

8 W. Richter, M. Satyanarayanan, J. Harkes, and B. Gilbert, "Near-Real-Time Inference of File-Level Mutations from Virtual Disk Writes," Technical Report CMU-CS-12-103, 2012.

9 C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass, "A Glossary of Temporal Database Concepts," ACM SIGMOD Record, vol. 21, no. 3, pp. 35–43, Sep. 1992.

10 R. T. Snodgrass, Developing time-oriented database applications in SQL. Morgan Kaufmann Publishers Inc., 1999.

11 Neo4j, http://neo4j.org/

12 Neo4j cypher query language, http://docs.neo4j.org/chunked/stable/cypher-query-lang.html

13 Nagios, http://www.nagios.org

14 Zenoss, http://www.zenoss.com/

15 IBM Tivoli, http://www.ibm.com/developerworks/tivoli/

16 V. Soundararajan, B. Parimi, and J. Cook, "StatsFeeder: An Extensible Statistics Collection Framework for Virtualized Environments," VMware Technical Journal, vol. 1, no. 1, pp. 32–44, 2012.

17 F. Donald. "cim1436 - Virtual SE Lab (vSEL): Building the VMware Hybrid Cloud." VMworld 2011.

18 M. Rodriguez. "MySQL vs. Neo4j on a Large-Scale Graph Traversal," http://java.dzone.com/articles/mysql-vs-neo4j-large-scale

19 Opscode Chef, http://www.opscode.com/

20 Puppet, http://puppetlabs.com/

21 M. Burgess, "A site configuration engine," Computing systems, vol. 8, no. 2, pp. 309–337, 1995.

22 A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, and J. Wang, "Detecting the Performance Impact of Upgrades in Large Operational Networks," SIGCOMM 2010.

23 A. Mahimkar, Z. Ge, J. Wang, and J. Yates, "Rapid detection of maintenance induced changes in service performance," CoNEXT 2011.

24 M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with WISE," in ACM SIGCOMM Computer Communication Review, 2008, vol. 38, no. 4, pp. 99–110.

25 S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," ACM SIGCOMM Computer Communication Review, vol. 39, no. 4, p. 243, Aug. 2009.

26 K. Bare, S. Kavulya, J. Tan, X. Pan, E. Marinelli, M. Kasick, R. Gandhi, and P. Narasimhan, "ASDF: An Automated, Online Framework for Diagnosing Performance Problems," in Architecting Dependable Systems VII, A. Casimiro, R. de Lemos, and C. Gacek, Eds. Springer Berlin / Heidelberg, 2010, pp. 201–226

27 M. Kasick, J. Tan, R. Gandhi, and P. Narasimhan, "Black-box problem diagnosis in parallel file systems," FAST 2010.

28 K. A. Bare, S. Kavulya, and P. Narasimhan, "Hardware performance counter-based problem diagnosis for e-commerce systems," NOMS 2010.

29 S. P. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan, "Draco : Statistical Diagnosis of Chronic Problems in Large Distributed Systems," DSN 2012.

30 E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger, "Stardust: Tracking activity in a distributed storage system," SIGMETRICS 2006.

31 R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," NSDI 2011.

Elastic Resource Allocation in Datacenters: Gremlins in the Management Plane

Mukil Kesavan
Center for Experimental
Research in Computer
Systems (CERCS)
Georgia Institute of Technology
mukil@cc.gatech.edu

Ada Gavrilovska
Center for Experimental
Research in Computer
Systems (CERCS)
Georgia Institute of Technology
ada@cc.gatech.edu

Karsten Schwan
Center for Experimental
Research in Computer
Systems (CERCS)
Georgia Institute of Technology
schwan@cc.gatech.edu

Abstract

Virtualization has simplified the management of datacenter infrastructures and enabled new services that can benefit both customers and providers. From a provider perspective, one of the key services in a virtualized datacenter is elastic allocation of resources to work-loads, using a combination of virtual machine migration and per-server work-conserving scheduling. Known challenges to elastic resource allocation include scalability, hardware heterogeneity, hard and soft virtual machine placement constraints, resource partitions, and others. This paper describes an additional challenge, which is the need for IT management to consider two design constraints that are systemic to large-scale deployments: failures in management operations and high variability in cost. The paper first illustrates these challenges, using data collected from a 700-server datacenter running a hierarchical resource management system built on the VMware vSphere platform. Next, it articulates and demonstrates methods for dealing with cost variability and failures, with a goal of improving management effectiveness. The methods make dynamic tradeoffs between management accuracy compared to overheads, within constraints imposed by observed failure behavior and cost variability.

1. Introduction

Virtualization of physical datacenter resources enables a fluid mapping in which resource allocations can be varied elastically in response to changes in workload and resource availability. This is critical to realizing the benefits of utility computing environments like cloud computing systems, which can dynamically grow and shrink the resources allocated to customer workloads based on actual and current demands. Such elasticity of resources results in operational efficiency for cloud providers and in potential cost savings for customers.

IT managers face a number of challenges when implementing elastic resource allocation in current-generation virtualized datacenters that are often populated with tens of thousands of machines [10]. These challenges include scalability, hard-ware heterogeneity, hard and soft virtual machine (VM) placement constraints, resource partitions, and others, to which the research community has responded with novel techniques and associated system support [5, 8, 14, 13, 11].

This paper highlights the importance of two additional factors posing challenges to elastic resource management for large-scale datacenter and cloud computing systems. First, management operations may fail because the majority of these higher-level services are implemented in a best effort management plane. Second, there can be large variations in the costs of these management operations. For example, consider these three elastic resource allocation scenarios: 1) dynamic virtual machine placement to address long-term virtual machine demands, 2) live virtual machine migration or offline placement during power-on, and 3) using per-server resource schedulers for finer-grained allocation [2]. Prior work has shown that these management plane operations, including live virtual machine migration, can fail and that they exhibit varying resource costs [12]. These failures decrease the effectiveness of elastic resource allocation and variable costs complicate dealing with management overhead, relative to the benefits derived from elastic resource management. These facts, then, contribute to a 'glass ceiling' in the management plane that limits the improvements achievable by elastic resource allocation services [7, 9].

This paper illustrates the effects and importance of understanding management plane operations and their behavior, including empirical evidence of the operation failure rates and cost variability, observed in a 700-server datacenter running VMware vSphere. In this system, the base functionality of the vSphere platform has been extended with Cloud Capacity Manager (CCM), a scalable, hierarchical, elastic resource allocation system that is built on top of VMware's DRS. CCM consists of three hierarchical levels: (i) clusters (small groups of hosts as defined by DRS), (ii) superclusters (groups of clusters), and (iii) cloud (a group of superclusters). In addition to the load balancing and re-source allocation performed for virtual machines of a single cluster by DRS, CCM dynamically shuffles *capacity* be-tween clusters and superclusters in response to aggregate changes in demand.

This paper quantifies the impact of management operation failures and cost variability on CCM, and presents simple methods for coping with these issues. It concludes with a brief discussion of the broader implications this poses when designing and constructing large-scale datacenter infrastructure services. Future research points out that design for management operation failures and cost variability, explored in the context of elastic resource allocation, is more

broadly applicable to higher-level services pertaining to high availability, power management, virtual machine backups/disaster recovery, virtual machine environment replication, and more generally, to adaptive systems and control.

2. CCM Overview

The overall architecture of CCM is shown in Figure 1. Demand-aware load balancing is periodically per-formed by capacity managers at the cluster and supercluster levels, and at the overall cloud level. Capacity managers operate independently, but share with the level above (if present) the combined resource demand information of all virtual machines on the hosts they manage. Sharing, as well as load balancing, operates at progressively larger time scales when moving up the hierarchy. Based on the combined virtual machine resource demand information (including some additional headroom), an *imbalance* metric is computed at each manager, as the standard deviation of the normalized demand of sub-entities. Load balancing is triggered when this imbalance is above an administrator-specified threshold during an invocation of the algorithm, and capacity is moved from entities with low-normalized demand to those with high-normalized demand.

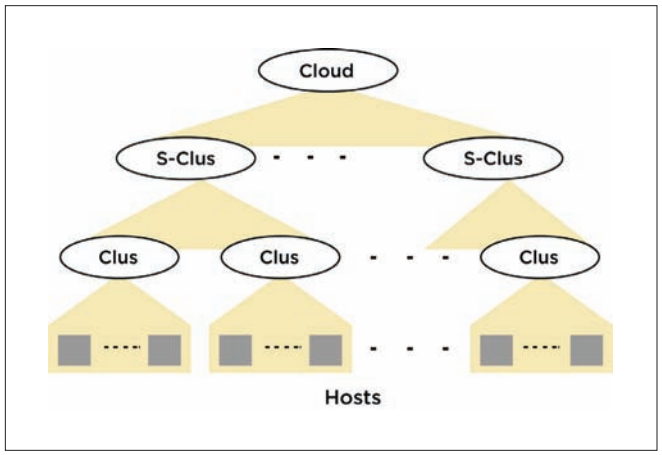

Figure 1. Hierarchical resource management architecture.

At the cluster level, VMware DRS is used to make independent and localized decisions to balance loads by migrating virtual machines using per-virtual machine demand estimates. At the supercluster and cloud levels, coarse-grained allocation changes are carried out by *logically re-associating capacity*. This process migrates individual evacuated hosts, rather than individual virtual machines, across clusters and superclusters. All virtual machines running on a host to be re-associated are migrated to other hosts that are part of the same cluster to which the host currently belongs. This is done to seamlessly integrate with DRS and to minimize the amount of state that must be moved between capacity managers during each migration. DRS automatically adapts to increased and decreased capacity in a cluster without requiring any changes.

CCM is implemented in Java, using the vSphere Java API [3] to collect metrics and enforce management actions in the vSphere provisioning layer. DRS is used in its standard vSphere server form. Both the cloud manager and supercluster managers are implemented as part of a single, multithreaded application running on a single host to make it simple to prototype and evaluate.

The remainder of this paper treats CCM as a black-box system that ingests monitoring information and emits management actions. The paper studies the management actions carried out in the management plane, or management enactment, and focus on enactment failures and variation in enactment cost.

Host-move action: A *host-move* is one of the basic actions performed by CCM at the supercluster and cloud levels, the purpose being to elastically allocate resources in the datacenter. There are two significant types of moves: host-move between clusters, and host-move between super-clusters. Each host-move is composed of a series of *macro* operations in the management plane that must be executed in order for an inter-cluster move, as shown in Figure 2. Each macro operation may be implemented by one or more lower level *micro* management plane operations.

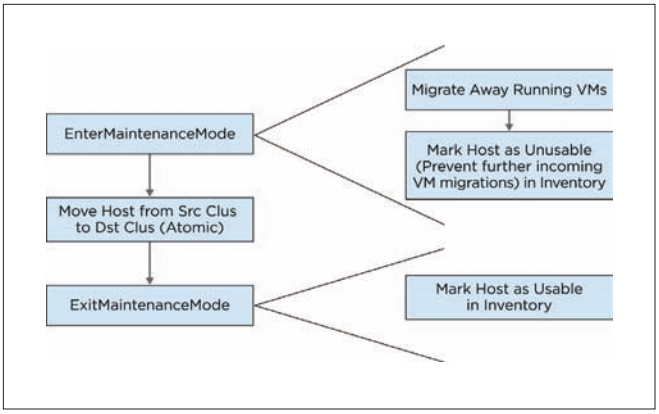

Figure 2. Inter-cluster host-move.

The "EnterMaintenanceMode" operation, for instance, places a host into a *maintenance* state, in which no virtual machines currently use it, so that this host can be moved from one cluster to another. This is potentially the most resource-intensive of these management plane operations because all virtual machines currently running on the host must be evicted. The time to complete this operation depends on factors like virtual machine size and active virtual machine memory footprints [4]. DRS selects other hosts in the source cluster and moves evicted virtual machines to those hosts.

The important point to note about these composite host-move operations is that the failure of a macro operation always results in complete failure of the whole host-move, whereas a failure of a micro-operation may or may not result in total failure depending on whether it is a pre-requisite for future operations (e.g., "Getting un-collected stats" need not result in total failure). As discussed further below, this classification of management plane operations helps when devising ways to cope with failures.

3. Failure and Cost Variability Analysis

This section outlines the experimental setup and presents data on virtual machine migration and host-move operation failures for representative large runs across 256 hosts of the 700-host datacenter. It also shows how these failures impact the performance of CCM, by defining a 'goodput' metric termed *effective management action throughput (emat)*.

Testbed: Each datacenter host has two dual-core AMD Opteron 270 processors, 4 GB of memory, and runs the VMware vSphere Hypervisor (ESXi) v4.1. The hosts are all connected to each other and four shared storage arrays (4.2 TB total capacity) via a Force 10 E1200 switch over a flat IP space. The common shared storage is exported as NFS stores to make it easy to migrate virtual machines across the datacenter machines. The open source Cobbler installation server runs on a dedicated host for DNS, DHCP, and PXE booting needs. The VMware vSphere server and client are used to provision, monitor, and partially manage the cloud.

Workload and setup: Realistic datacenter-wide load patterns are generated by replaying the resource usage traces released by Google Inc. [1]. The first four hours of the resource usage pattern of the jobs in the trace are replayed on 1024 virtual machines, 256 per job. There are a total of 16 clusters covering the 256 hosts, with each cluster initially containing 16 hosts and 64 virtual machines. A supercluster is composed of a set of 4 clusters, separate from other superclusters, with a total of 64 hosts and 256 virtual machines. This results in a total of four superclusters in the cloud. A given job's virtual machines fit within a single supercluster. A more detailed explanation of the load generation framework and trace replay appears in [6].

The load-balancing algorithm at the cluster level runs once every 5 minutes, at the supercluster level once every 20 minutes, and at the cloud level once every hour. Results for four different configurations of CCM are shown. The first configuration attempts to carry out *all* of the host-move actions recommended by the balancing algorithms during their respective invocations. Further, all of the moves are also carried out in parallel, with the intent to reduce the amount of time the system is in a state of flux. This sort of a configuration is not uncommon in practice where system developers assume a low and stable cost for enforcing management. This configuration is named CCM_nr, short for CCM with "no restrictions". In CCM_nr and the rest of the configurations, DRS is set to carry out priority 1, 2, and 3 virtual machine migration recommendations, with at most eight virtual machines per host in parallel.

Figure 3 shows the number of successful host-move operations. All of the attempted host-moves are inter-cluster moves, which are determined by the nature of the work-load. The results showed a 38% host-move failure rate and low number of successful host-moves that result in little to no effect on work-load performance (data not shown here). In addition to outright failures, there were also a large number of extremely slow operations that did not complete during the course of the experiment. In the figure, these are also counted as failures. Figure 4 presents the proportion of failures due to a failure of each of the three macro management plane operations in the inter-cluster move. It can be seen that more than 90% of the failures are due to a failure of the "EnterMaintenanceMode" operation, or in other words, a failure to evict (by migrating them away) all of the running virtual ma-chines on the hosts in question.

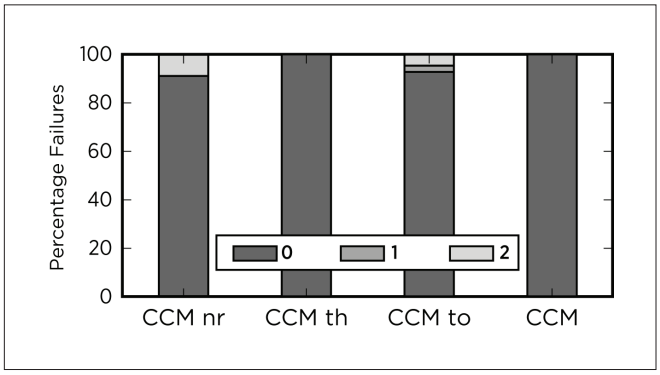

Figure 4. Fraction of inter-cluster host-move failure due to failure of each macro operation. Key: 0=EnterMaintenance-Mode, 1=Move-Host, 2=ExitMaintenanceMode.

In Figure 5, CCM_nr shows a noticeable 29% virtual machine migration failure over the course of the experiment. Note that the migration failures reported here include those due to host-moves and those performed by DRS during its load balancing. However, it still gives a general link between high migration failure rates and host-move failures given that a failure to evacuate a single running virtual machine would result in the failure of the entire operation.

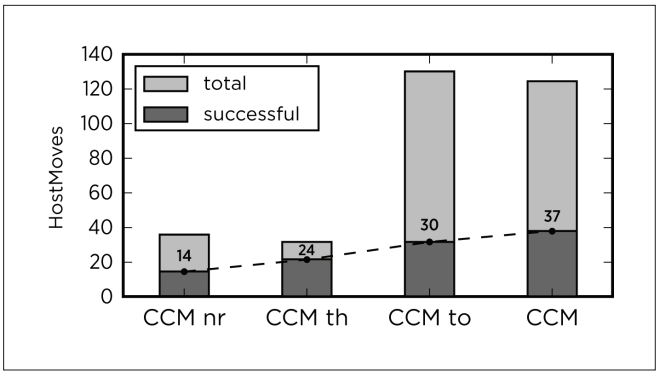

Figure 3. Number of successful host-moves.

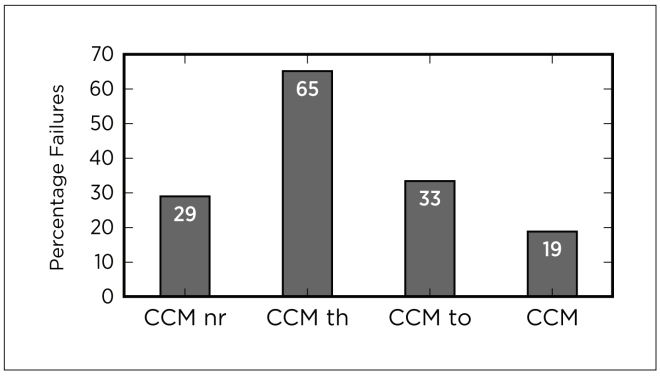

Figure 5. Migration failures.

Table 1 shows some of the major causes of VM migration failures collected from the vSphere layer and their percentage contribution to the overall number of migration failures. Some of the causes appear to be random or transient in nature, possibly due to software bugs or configuration issues, whereas others point to the fact that migrations may have failed directly or indirectly due to high resource pressure given the way CCM_nr enforces actions (e.g., "Operation timed out").

ABBV. CAUSE	nr	th	to	CCM
General system error	31	22	10	21
Failed to create journal file	45	7	63	54
Operation timed out	4	49	18	13
Operation not allowed in cur state	12	22	4	12
Insuf. host resources for VM reserv.	0	0	3	0
Changing mem greater than net BW	0	0	1	0
Data copy fail: already disconnected	8	0	0	1
Error comm. w/ dest host	0	0	1	0

Table 1: VM migration failure causes breakdown for each con-figuration. Values denote percentages.

Given this intuition, three different configurations of CCM are presented in which the degree of resource pressure imposed by management actions is controlled. Pressure is changed by: (1) explicit throttling of management enactions, i.e., the number and parallelism of host-moves (CCM_th), (2) automatically aborting long-running actions using timeouts (CCM_to) and, (3) a combination of both action throttling and timeouts (CCM). Configurations differ in their use of values for throttling host-moves, i.e., by limiting the maximum number of host-moves per balancing period to eight and reducing the parallelism in moves to no more than four at a time for each supercluster. In addition, a static timeout setting of 16 minutes is used for a single host-move operation for CCM_to and CCM. Experimental results with these configurations test the hypothesis that the resource pressure induced by management causes the observed failures.

The CCM_th, CCM_to, and CCM configurations exhibit higher success rates (41%, 53%, and 62%, respectively, compared to CCM_nr) in the host-move operations performed, as shown in Figure 3. This success points to the fact that there is a quantifiable benefit in managing the resource pressure of management action enforcement. Note that the monitoring and load-balancing algorithms, and the workload, remain the same for all configurations. In the case of CCM_to and CCM, both configurations achieve a much higher success rate while also attempting almost three times as many host moves as CCM_nr and CCM_th.

This is because a timeout allows stopping long running host-moves where the cost-to-workload benefit ratio is unfavorable. If the load imbalance in the workload continues to persist, the balancing algorithms recommend a fresh set of moves during the next round that may be more cost effective. In this fashion, the higher-level service could explore more efficient host-move alternatives than those available at a prior point in time.

To better quantify the behavior exhibited in the experiments discussed above required a new to more objectively compare the different configurations: *effective management action throughput (emat)*—the ratio of the number of successful host-moves to the total amount of time spent in making all moves (successful and failed). Table 2 shows the total minutes spent performing the host-moves and the emat metric for each of the four configurations. The total time is summed over all host-moves attempted by each configuration, counting parallel moves in serial order. Even though CCM and CCM_to perform nearly three times as many moves (see Figure 3) as the other two configurations, the total time spent enforcing the moves was significantly lower. This, combined with the fact that these configurations also delivered a higher host-move success rate, results in a two-to-six fold advantage in terms of the *emat* metric.

METRIC	CCM_nr	CCM_th	CCM_to	CCM
Total Host-move Mins	4577	703	426	355
emat (moves/hr)	0.18	2.05	4.23	6.25

Table 2: Management Metrics

Note that for the CCM_nr and CCM_th configurations, the balancing algorithms cannot start recommending and enforcing new host-moves while the previous moves are still in progress and the system is in an unstable state. This is the reason for the rather smaller number of host-move attempts in both of these cases. The large proportion of host-move failures for CCM_to and CCM are due to a combination of explicit aborts and the fact that the configurations are still afflicted by a non-negligible fraction of virtual machine migration failures as shown in Figure 5.

In addition to the host-move failures, it is also important to consider the cost-to-benefit ratio of the enforced management actions. Figure 6 depicts the average, minimum, and maximum values of successful host-move action times for all four configurations. Given that the resource cost of an action is directly proportional to

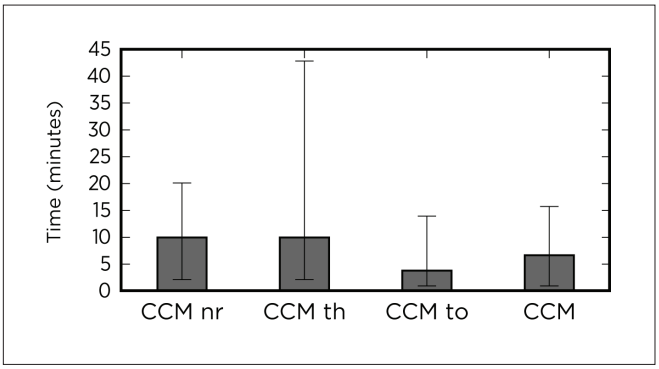

Figure 6. Host-move times.

the amount of time the action takes, it is important to abort overly long-running host-moves, which is illustrated with the more stable host-move times observed with CCM_to and CCM, each of which

are more likely to deliver a favorable cost-to-benefit ratio. Further, since the most resource-intensive stage of a host-move operation is the "EnterMaintenanceMode" operation, with its need to evict all of the current virtual ma-chines running on a host, this operation is particularly affected by high variations in virtual machine migration times. Variability in these times is evident from the fact that the 90th percentile values of virtual machine migration times computed for CCM_nr, CCM_th, CCM_to, and CCM show high values of 377s, 320s, 294s, and 335s, respectively. These variations have two causes: those attributable to management pressure and natural causes due to differences in virtual behavior, leading to runtime variations in the active memory footprints. The latter are beyond the control of the management layer, but demonstrate that host-move times can vary widely, and as a result, the abort/search method of exploring more efficient moves is still relevant. Alternatively, the system can also explicitly track and predict management costs and make moves when the moves will produce the most favorable cost-to-benefit ratio.

4. Conclusions

This paper draws attention to the importance of designing for failures, not only for the applications running in large-scale datacenter systems, but also and perhaps even more importantly, for the management actions that are in-tended to improve application performance. Intuitively, this is because management failures can have a substantial effect on the efficiency of datacenter operation. First, because failed actions consume resources without contributing to the desired improvements and second, because the resource pressure induced by such actions can lead to action failures or undue delays. Therefore, it is important to design a data-center's management plane that considers management failures as well as variability in the costs of enforcing actions in the management plane.

The experimental results shown in this paper illustrate the efficacy of simple methods for improving otherwise cost-variable and/or failure-intolerant management action. Methods include explicit action throttling to reduce the resource pressure imposed by such actions, and aborts that prevent undue resource consumption by actions experiencing delays. Results shown in the paper use static settings to test the usefulness of the action throttling and early aborts. Future work will develop techniques to dynamically derive these parameters. Additional experiments are in process with alternative strategies to hedge against failures that cannot be controlled, in order to minimize overall management cost, reduce failure rates, and maximize the bene-fits to application workloads.

References

1 googleclusterdata: traces of google tasks running in a production cluster, http://code.google.com/p/googleclusterdata/

2 VMware DRS, http://www.vmware.com/products/DRS

3 VMware VI (vSphere) Java API, http://vijava.sourceforge.net/

4 C. Clark et al. Live migration of virtual machines. In NSDI'05.

5 D. Gmach et al. Workload analysis and demand prediction of enterprise datacenter applications. In IISWC '07.

6 M. Kesavan et al. Xerxes: Distributed load generator for cloud-scale experimentation. Open Cirrus Summit, 2012.

7 S. Kumar et al. vManage: loosely coupled platform and virtualization management in datacenters. ICAC '09.

8 X. Meng et al. Efficient resource provisioning in compute clouds via virtual machine multiplexing. In ICAC '10.

9 R. Moreno-Vozmediano et al. Elastic management of cluster-based services in the cloud. ACDC '09.

10 D. Schneider et al. Under the hood at Google and Facebook. Spectrum, IEEE, 48(6):63 –67, June 2011.

11 Z. Shen et al. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In SoCC '11.

12 V. Soundararajan et al. The impact of management operations on the virtualized datacenter. In ISCA '10.

13 T. Wood et al. Black-box and gray-box strategies for virtual machine migration. In NSDI'07.

14 X. Zhu et al. 1000 islands: Integrated capacity and workload management for the next generation datacenter. In ICAC '08.

Toward a Paravirtual vRDMA Device for VMware ESXi Guests

Adit Ranadive[1]
Georgia Institute of Technology
adit.ranadive@gatech.edu

Bhavesh Davda
VMware, Inc.
bhavesh@vmware.com

Abstract

Paravirtual devices are common in virtualized environments, providing improved virtual device performance compared to emulated physical devices. For virtualization to make inroads in High Performance Computing and other areas that require high bandwidth and low latency, high-performance transports such as InfiniBand, the Internet Wide Area RDMA Protocol (iWARP), and RDMA over Converged Ethernet (RoCE) must be virtualized.

We developed a paravirtual interface called Virtual RDMA (vRDMA) that provides an RDMA-like interface for VMware ESXi guests. vRDMA uses the Virtual Machine Communication Interface (VMCI) virtual device to interact with ESXi. The vRDMA interface is designed to support snapshots and VMware vMotion® so the state of the virtual machine can be easily isolated and transferred. This paper describes our vRDMA design and its components, and outlines the current state of work and challenges faced while developing this device.

Categories and Subject Descriptors
C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks- *High-speed*; C.4 [**Performance of Systems**]: Modeling techniques; D.4.4 [**Operating Systems**]: Communication Management- *Network Communication*;
General Terms
Algorithms, Design, High Performance Computing, Management
Keywords
InfiniBand, Linux, RDMA, Subnet Management, Virtualization, virtual machine

1 Adit was an intern at VMware when working on this project.

1. Introduction

Paravirtualized devices are common in virtualized environments [1-3] because they provide better performance than emulated devices. With the increased importance of newer high-performance fabrics such as InfiniBand, iWARP, and RoCE for Big Data, High Performance Computing, financial trading systems, and so on, there is a need [4-6] to support such technologies in a virtualized environment. These devices support zero-copy, operating system-bypass and CPU offload [7-9] for data transfer, providing low latency and high throughput to applications. It is also true, however, that applications running in virtualized environments benefit from features such as vMotion (virtual machine live migration), resource management, and virtual machine fault tolerance. For applications to continue to benefit from the full value of virtualization while also making use of RDMA, the paravirtual interface must be designed to support these virtualization features.

Currently there are several ways to provide RDMA support in virtual machines. The first option, called passthrough (or VM DirectPath I/O on ESXi), allows virtual machines to directly control RDMA devices. Passthrough also can be used in conjunction with single root I/O virtualization (SR-IOV) [9] to support the sharing of a single hardware device between multiple virtual machines by passing through a Virtual Function (VF) to each virtual machine. This method, however, restricts the ability to use virtual machine live migration or to perform any resource management. A second option is to use a software-level driver, called SoftRoCE [10], to convert RDMA Verbs operations into socket operations across an Ethernet device. This technique, however, suffers from performance penalties and may not be a viable option for some applications.

With that in mind, we developed a paravirtual device driver for RDMA-capable fabrics, called Virtual RDMA (vRDMA). It allows multiple guests to access the RDMA device using a *Verbs API*, an industry-standard interface. A set of these *Verbs* was implemented to expose an RDMA-capable guest device (vRDMA) to applications. The applications can use the vRDMA guest driver to communicate with the underlying physical device. This paper describes our design and implementation of the vRDMA guest driver using the VMCI virtual device. It also discusses the various components of vRDMA and how they work in different levels of the virtualization stack. The remainder of the paper describes how RDMA works, the vRDMA architecture and interaction with VMCI, and vRDMA components. Finally, the current status of vRDMA and future work are described.

2. RDMA

The Remote Direct Memory Access (RDMA) technique allows devices to read/write directly to an application's memory without interacting with the CPU or operating system, enabling higher throughput and lower latencies. As shown on the right in Figure 1, the application can directly program the network device to perform DMA to and from application memory. Essentially, network processing is pushed onto the device, which is responsible for performing all protocol operations. As a result, RDMA devices historically have been extremely popular for High Performance Computing (HPC) applications [6, 7]. More recently, many clustered enterprise applications, such as databases, file systems and emerging Big Data application frameworks such as Apache Hadoop, have demonstrated performance benefits using RDMA[6, 11, 12].

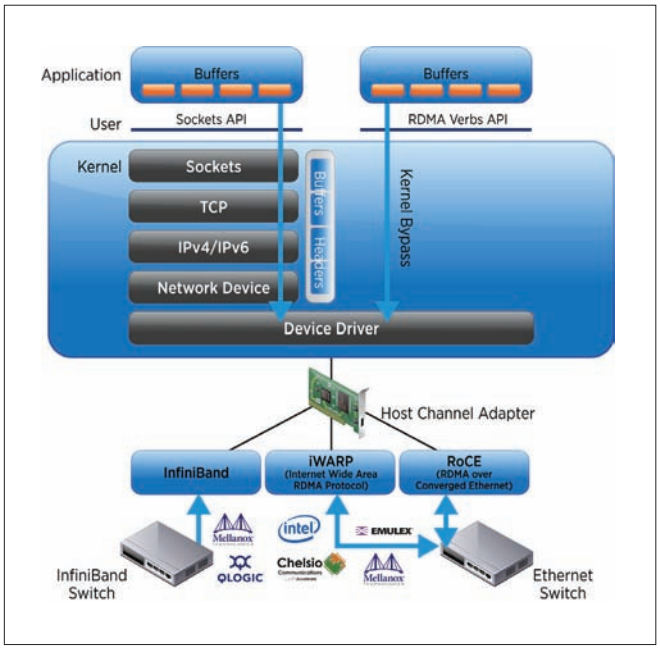

Figure 1. Comparing RDMA and Sockets

While data transfer operations can be performed directly by the application as described above, control operations such as allocation of network resources on the device need to be executed by the device driver in the operating system for each application. This allows the device to multiplex between various applications using these resources. After the control path is established, the application can directly interact with the device, programming it to perform DMA operations to other hosts, a capability often called *OS-bypass*. RDMA also is said to support *zero-copy* since the device directly reads/writes from/to application memory and there is no buffering of data in the operating system. This offloading of capabilities onto the device, coupled with direct user-level access to the hardware, largely explains why such devices offer superior performance. The next section describes our paravirtualized RDMA device, called Virtual RDMA (vRDMA).

3. vRDMA over VMCI Architecture

Figure 2 illustrates the vRDMA prototype. The architecture is similar to any virtual device, with a driver component at the guest level and another at the hypervisor level that is responsible for communicating with the physical device. In the case of our new device, we include a modified version of the OpenFabrics RDMA stack within the hypervisor that implements the core Verbs required for RDMA devices. Using this stack allows us to be agnostic with respect to RDMA transport, enabling the vRDMA device to support InfiniBand (IB), iWARP, and RoCE.

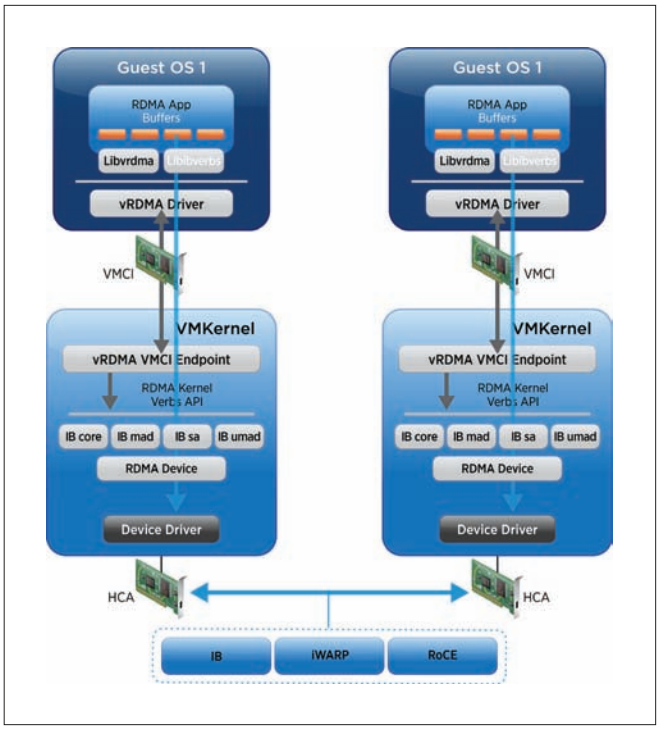

Figure 2. vRDMA over VMCI Architecture[2].

We expose RDMA capabilities to the guest using VMCI [13], a virtual PCI device that supports a point-to-point bidirectional transport based on a pair of memory-mapped queues and datagrams serving as asynchronous notifications. Using VMCI, we construct communication endpoints between each guest and an endpoint in the hypervisor called the *vRDMA VMCI endpoint*, as shown in Figure 2. All guests connect with the hypervisor endpoint when the vRDMA driver is loaded in the guests.

To use RDMA in a virtual environment, guest operating systems use the standard OpenFabrics Enterprise Distribution (OFED) RDMA stack, along with our guest kernel vRDMA driver and a user-level library (**libvrdma**). These additional components could be distributed using VMware Tools, which already contain the VMCI guest virtual device driver. The OFED stack, the device driver, and library provide an implementation of the industry-standard Verbs API for each device.

2 The "RDMA Device" shown here and in other figures refers to a software abstraction in the I/O stack in the VMKernel.

Our guest kernel driver communicates with the vRDMA VMCI endpoint using VMCI datagrams that encapsulate Verbs and their associated data structures. For example, a 'Register Memory' Verb datagram contains the memory region size and guest physical addresses associated with the application.

When the VMKernel VMCI Endpoint receives the datagram from the guest, it handles the Verb command in one of two ways, depending on whether the destination virtual machine is on the same physical machine (intra-host) or a different machine (inter-host).

The vRDMA endpoint can determine if two virtual machines are on the same host by examining the QP numbers and LIDs[14] assigned to the virtual machines. Once it has determined the virtual machines are on the same host, it emulates the actual RDMA operation. For example, when a virtual machine issues an RDMA Write operation, it specifies the source address, destination address, and data size. When the endpoint receives the RDMA Write operation in the Post_Send Verb, it performs a memory copy into the address associated with the destination virtual machine. We can further extend the emulation to the creation of queue pairs (QPs), CQs, MRs (see [15] for a glossary of terms) and other resources such that we can handle all Verbs calls in the endpoint. This method is described in more detail in Section 4.4.

In the inter-host case, the vRDMA endpoint interacts with the ESXi RDMA stack as shown in Figure 2. When a datagram is received from a virtual machine, it checks to see if the Verb corresponds to a creation request for communication resources, such as Queue Pairs. These requests are forwarded to the ESXi RDMA stack, which returns values after interacting with the device. The endpoint returns results to the virtual machine using a VMCI datagram. When the virtual machine sends an RDMA operation command, such as RDMA Read, RDMA Write, RDMA Send, or RDMA Receive, the endpoint directly forwards the Verbs call to the RDMA stack since it already knows it is an inter-host operation.

4. Components of vRDMA

This section describes the main components of the vRDMA paravirtual device and their interactions.

4.1 libvrdma

The **libvrdma** component is a user-space library that applications use indirectly when linking to the **libibverbs** library. Applications (or middleware such as MPI) that use RDMA link to the device-agnostic **libibverbs** library, which implements the industry-standard Verbs API. The **libibverbs** library in turn links to **libvrdma**, which enables the application to use the vRDMA device.

Verbs are forwarded by **libibverbs** to **libvrdma**, which in turn forwards the Verbs to the main RDMA module present in the guest kernel using the Linux /**dev** file system. This functionality is required to be compatible with the OFED stack, which communicates with all underlying device drivers in this way. Figure 3 illustrates the RDMA application executing a '*Query_Device*' Verb.

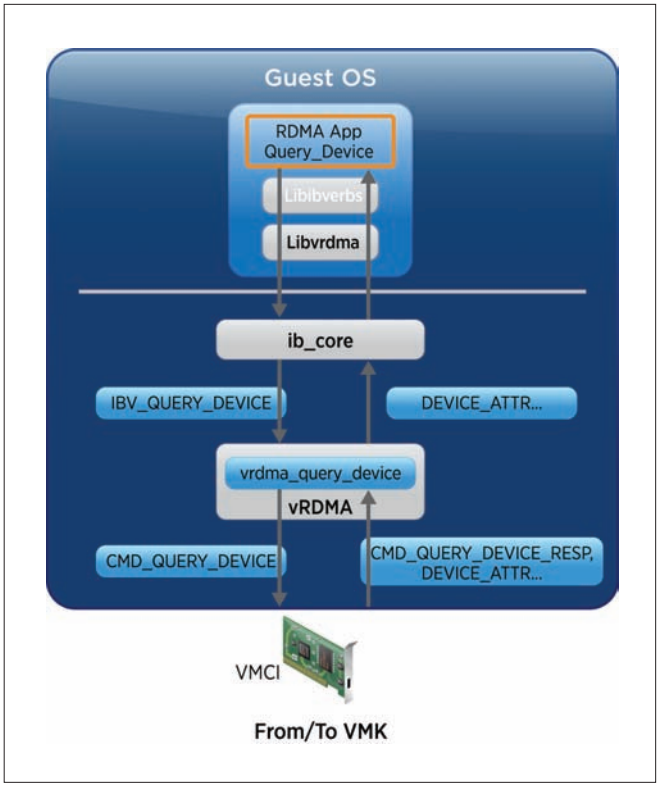

Figure 3. Guest vRDMA driver and libvrdma

4.2 Guest Kernel vRDMA Driver

The OFED stack provides an implementation of the kernel-level Verbs API called the **ib_core** framework. The framework allows device drivers to register themselves using callbacks for each Verb. Each device driver must provide implementations of a mandatory list of Verbs (Table 1). Therefore, our vRDMA guest kernel driver must implement this list of Verbs to register successfully with the OFED stack. Underneath these Verbs calls we communicate with the vRDMA endpoint to ensure valid responses for each Verb. Next, using the **Query_Device** Verb as an example, we describe how the Guest kernel driver handles Verbs.

Query_Device	Modify_QP
Query_Port	Query_QP
Query_Gid	Destroy_QP
Query_Pkey	Create_CQ
Create_AH	Modify_CQ
Destroy_AH	Destroy_CQ
Alloc_Ucontext	Poll_CQ
Dealloc_Ucontext	Post_Send
Alloc_PD	Post_Recv
Dealloc_PD	Reg_User_MR
Create_QP	Dereg_MR

Table 1: Verbs supported in the vRDMA prototype.

As shown in Figure 3, the **Query_Device** Verb is executed by the application. It is passed to the **ib_core** framework, which calls the **Query_Device** function in our vRDMA kernel module using the registered callback. The vRDMA guest kernel module packetizes the Verbs call into a buffer to be sent using a VMCI datagram. This datagram is sent to the vRDMA VMKernel VMCI Endpoint that handles all requests from virtual machines and issues responses.

4.3 ESXi RDMA stack

The ESXi RDMA stack is an implementation of the OFED stack that resides in VMKernel and contains device drivers for RDMA devices. In our prototype, the ESXi RDMA stack mediates access to RDMA hardware on behalf of our vRDMA device in the guest. Other hypervisor services, such as vMotion and Fault Tolerance, could use this stack to access the RDMA device.

4.4 VMKernel vRDMA VMCI Endpoint

The main role of the VMKernel vRDMA VMCI endpoint is to receive requests from virtual machines and send appropriate responses back by interacting with the ESXi RDMA stack. Most requests from virtual machines are in the form of Verbs calls. Responses depend on the location of the destination virtual machine. As mentioned in Section 3, the vRDMA endpoint handles the Verb command in two ways depending on whether the destination virtual machine is on the same host. To decide whether the virtual machine is on the same host, the endpoint consults a list of communication resources used by the virtual machine: QP numbers, CQ number, MR entries, and LIDs. These identify the communication taking place and, therefore, the virtual machines.

For example, the endpoint can identify the destination virtual machine when the source virtual machine issues a Modify QP Verb, which includes the destination QP number and LID[14]. It matches these with its list of virtual machine communication resources, connecting the two virtual machines in the endpoint when it finds a match. Once connected, the Modify QP/CQ Verb is not forwarded to the RDMA stack. Instead, the endpoint returns values to the virtual machine, stating the Verb completed successfully. When the source virtual machine subsequently executes an RDMA data transfer operation, the vRDMA endpoint performs a memory copy based on the type of operation. For example, when a virtual machine issues an RDMA Write, it specifies the source, destination address, and data size. The endpoint performs the memory copy when it receives the Verb. Figure 4 shows the vRDMA architecture when virtual machines reside on the same host.

When virtual machines are determined to be on different hosts based on a previous Modify QP Verb call, the vRDMA endpoint forwards any Verbs received to the RDMA stack in the VMKernel. After this check, the endpoint forwards all Verbs calls to the RDMA stack. The RDMA stack and the physical device are responsible for completing the Verb call. Once the Verb call completes, the endpoint accepts the return values from the RDMA stack and sends them back to the source virtual machine using VMCI datagrams.

Figure 5 shows the **Query_Device** Verb being received by the vRDMA endpoint. As an optimization, the values for such Verbs can be cached in the endpoint. Therefore, the first Verb call is forwarded to the RDMA stack, which sends a Management

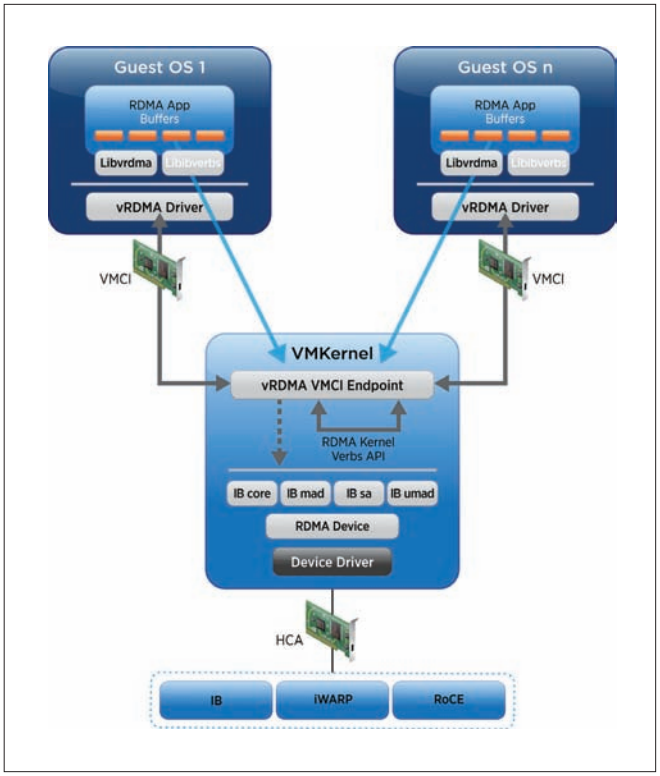

Figure 4. vRDMA architecture for virtual machines on the same host

Datagram (MAD) to the device to retrieve the device attributes. Additional **Query_Device** Verb calls from the virtual machine can be returned by the endpoint using the cached values, reducing the number of MADs sent to the device. This can be extended to other Verbs calls. The advantage of this optimization: mimicking or emulating the Verbs calls enables RDMA device attributes to be provided without a physical RDMA device being present.

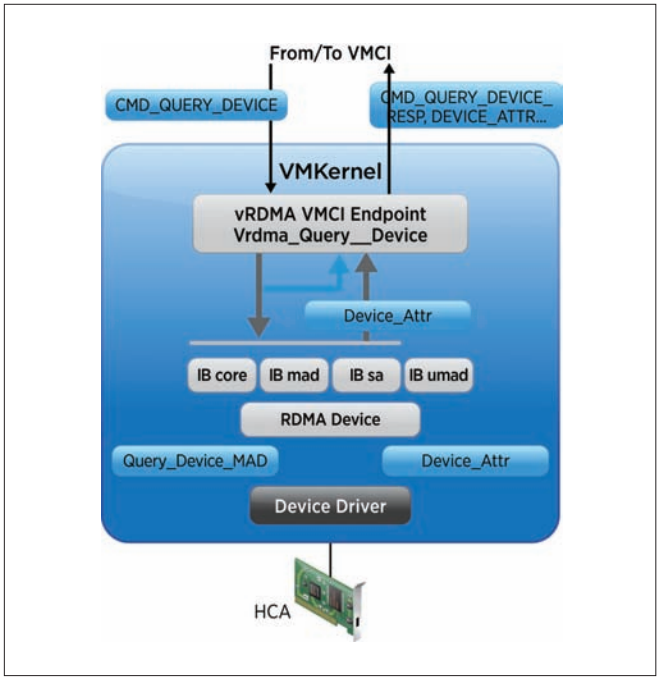

Figure 5. VMKernel vRDMA Endpoint and ESXi RDMA stack

Emulating these Verbs calls enables us to store the actual device state, containing the QP, CQ, and MR structures within the vRDMA endpoint, which is extremely useful in allowing RDMA-based applications to run on machines without RDMA devices and without modifying the applications to use another transport. This is an important attribute of our vRDMA solution.

5. Current Status and Future Directions

The vRDMA prototype is feature complete, with some testing and debugging remaining. We expect half-roundtrip vRDMA latencies to be about 5µs, lower than the SoftRoCE over vmxnet3[19] option, but higher than what one can achieve in the bare-metal, passthrough, or SR-IOV VF cases. We intend to measure and report latencies and bandwidths for our prototype when testing is completed.

5.1 Supporting RDMA without RDMA Devices

The Verbs API is an abstraction for the actual functionality of the RDMA device, and device drivers provide their own implementation of these verbs to register with **ib_core**. It is possible for the device driver to emulate Verbs by returning the expected response to the **ib_core** framework without interacting with the device. In our prototype, the guest vRDMA driver acts as the RDMA device driver and the vRDMA endpoint acts as the RDMA device, emulating Verbs calls. This layering and abstraction enables the vRDMA endpoint to use any network device and support Verbs-based applications without a physical RDMA device being present.

5.2 Support for Checkpoints and vMotion

One of the main advantages of a paravirtualized interface is the ability to support snapshots and vMotion. Because the state of the vRDMA device is fully contained in guest physical memory and VMCI device state, features such as checkpoints, suspend/resume, and vMotion can be enabled. Additional work will be required to tear down and rebuild underlying RDMA resources (QPs and MRs) during vMotion operations. This is work we are interested in exploring.

5.3 Subnet Management

One of the bigger challenges is to integrate paravirtual RDMA interfaces with subnet management. Consider the InfiniBand case in which the Subnet Manager (SM) assigns Local IDs (LIDs) [14] to IB ports and Global IDs (GIDs) to HCAs. One way to maintain addressability is to let ESXi query the IB SM for a list of unique LIDs and GIDs assignable to the virtual machines. In a large cluster with multiple virtual machines per host, the 16-bit range limits the number of virtual machines with unique LIDs. We might need to modify the SM to provide more LIDs in the subnet with virtual machines. Another alternative is to extend VMware® vCenter™ to be the "subnet manager" for virtual RDMA devices, and assign unique LIDs and GIDs within the vCenter cluster.

6. Related Work

While virtualization is very popular in enterprises, it has not made significant inroads with the HPC community. This can be attributed to the lack of support for high-performance interconnects and perceived performance overhead due to virtualization. There has been progress toward providing access to high-performance networks such as InfiniBand [4, 16] to virtual machines. With our prototype, we do not expect to meet the latencies as shown in [4]. We can, however, offer all the virtualization benefits at significantly lower latencies than alternative approaches based on traditional Ethernet network interface cards (NICs).

While there has been work to provide the features of virtualization [17, 18] to virtual machines, these approaches have not been widely adopted. Therefore, the disadvantage of this virtual machine monitor (VMM)-bypass approach is the loss of some of the more powerful features of virtualization, such as snapshots, live migration, and resource management.

7. Conclusion

This paper describes our prototype of a paravirtual RDMA device, which provides guests with the ability to use an RDMA device while benefiting from virtualization features such as checkpointing and vMotion. vRDMA consists of three components:

- **libvrdma**, a user-space library that is compatible with the Verbs API
- Guest kernel driver, a Linux-based module to support the kernel-space Verbs API
- A VMkernel vRDMA Endpoint that communicates with the Guest kernel driver using VMCI Datagrams

A modified RDMA Stack in the VMKernel is used so the vRDMA endpoint can interact with the physical device to execute the Verbs calls sent by the guest. An optimized implementation of the vRDMA device was explained, in which the data between virtual machines on the same host is copied without device involvement. With this prototype, we expect half round trip latencies of approximately 5µs since our datapath passes through the vRDMA endpoint and is longer than that of the bare-metal case.

Acknowledgements

We would like to thank Andy King for his insight into our prototype and for lots of help in improving our understanding of VMCI and vmware-tools. We owe a deep debt of gratitude to Josh Simons, who has been a vital help in this project and for his invaluable comments on the paper.

References

1. *The VMWare ESX Server*. Available from: http://www.vmware.com/products/esx/
2. Barham, P., et al. *Xen and the Art of Virtualization*. In Proceedings of SOSP, 2003.
3. *Microsoft Hyper-V Architecture*. Available from: http://msdn.microsoft.com/en-us/library/cc768520.aspx
4. Liu, J., et al. *High Performance VMM-Bypass I/O in Virtual Machines*. In Proceedings of *USENIX Annual Technical Conference*, 2006.
5. Ranadive, A., et al. *Performance Implications of Virtualizing Multicore Cluster Machines*. In Proceedings of *HPCVirtualization Workshop*, 2008.
6. Simons, J. and J. Buell, *Virtualizing High Performance Computing*. SIGOPS Oper. Syst. Rev., 2010.
7. Liu, J., J. Wu, and D.K. Panda, *High Performance RDMA-Based MPI Implementation over InfiniBand*. International Journal of Parallel Programming, 2004.
8. Liu, J. *Evaluating Standard-Based Self-Virtualizing Devices: A Performance Study on 10 GbE NICs with SR-IOV Support*. In Proceedings of *International Parallel and Distributed Processing Symposium*, 2010.
9. Dong, Y., Z. Yu, and G. Rose. *SR-IOV Networking in Xen: Architecture, Design and Implementation*. In Proceedings of *Worksop on I/O Virtualization*, 2008.
10. SystemFabricWorks. *SoftRoCE*. Available from: http://www.systemfabricworks.com/downloads/roce
11. Sayantan Sur, H.W., Jian Huang, Xiangyong Ouyang and Dhabaleswar K. Panda. *Can High-Performance Interconnects Benefit Hadoop Distributed File System?* In Proceedings of *Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds*, 2010.
12. Mellanox Technologies. *Mellanox Unstructured Data Accelerator (UDA)*. 2011; Available from: http://www.mellanox.com/pdf/applications/SB_Hadoop.pdf
13. VMware, Inc. VMCI API; Available from: http://pubs.vmware.com/vmci-sdk/
14. Wickus Nienaber, X.Y., Zhenhai Duan, *LID Assignment In InfiniBand Networks*. IEEE Transactions on Parallel and Distributed Systems, 2009.
15. InfiniBand Trade Association. *InfiniBand Architecture Specification, Release 1.2.*
16. Huang, W., J. Liu, and D.K. Panda. *A Case for High Performance Computing with Virtual Machines*. In Proceedings of *International Conference on Supercomputing*, 2006.
17. Huang, W., et al. *High Performance Virtual Machine Migration with RDMA over Modern Interconnects*. In Proceedings of *IEEE Cluster*, 2007.
18. Huang, W., et al. *Virtual Machine Aware Communication Libraries for High Performance Computing*. In Proceedings of *Supercomputing*, 2007.
19. VMware KB 1001805, "Choosing a network adapter for your virtual machine": http://kb.vmware.com/kb/1001805

Intrusion Detection Using VProbes

Alex Dehnert
VMware, Inc./Massachusetts Institute of Technology
adehnert@mit.edu

Abstract

Many current intrusion detection systems (IDS) are vulnerable to intruders because they run under the same operating system as a potential attacker. Since an attacker often attempts to co-opt the operating system, the IDS is vulnerable to subversion. While some systems escape this flaw, they generally do so by modifying the hypervisor. VMware® VProbes technology allows administrators to look inside a running virtual machine, set breakpoints, and inspect memory from a virtual machine host. We aim to leverage VProbes to build an IDS for Linux guests that is significantly harder for an attacker to subvert, while also allowing the use of a common off-the-shelf hypervisor.

1. Introduction

A common mechanism for defending computers against malicious attackers uses intrusion detection systems (IDSes). Network IDSes monitor network traffic to detect intrusions, while host-based IDSes monitor activity on specific machines. A common variety of host-based IDSes watches the kernel-application interface, monitoring the system calls that are used [2][4][7][8]. Based on the sequences of system calls used and their arguments, these IDSes aim to determine whether or not an attack is underway.

While intrusion detection systems are not fully effective, they have proven to be useful tools for catching some attacks. Since a host-based IDS runs on the host it is protecting, it is vulnerable to a virus or other attacker that seeks to disable it. An attacker might block network connectivity the IDS requires to report results, disable hooks it uses to gather information, or entirely kill the detection process. This is not a theoretical risk. Viruses in the wild, such as SpamThru, Beast, Win32.Glieder.AF, or Winevar[6] directly counter anti-virus software installed on their hosts.

The robustness of a host-based IDS can be improved by running it on the outside of a virtual machine, using capabilities exposed by the hypervisor to monitor the virtual machine, and gather information the agent in the guest would ordinarily use.

2. Design

VMware ESXi™ supports VProbes, a mechanism for examining the state of an ESXi host or virtual machine, similar to the Oracle Solaris Dynamic Tracing (DTrace) facility in the Oracle Solaris operating system. VProbes allows users to place user-defined probes in the ESXi kernel (VMkernel), the monitor, or within the guest. Probes are written in a C-like language called Emmett, and perform computation, store data, and output results to a log on the ESXi host. While primarily used to diagnose performance or correctness issues, VProbes also can be used to supply data to the IDS.

Our IDS is architected as two components:

- The **gatherer** uses VProbes to retrieve system call information from the guest virtual machine. This allows it to run outside of the guest while still gathering information from within the guest.
- The **analyzer** uses the data gathered to decide whether or not a system call sequence is suspicious. The analysis component is similar to components in other intrusion detection systems, and can use the same types of algorithms to identify attacks.

One advantage of splitting the gatherer from the analyzer is modularity. Two major variants of the gatherer exist currently: one for Linux and one for Microsoft Windows, with specialized code for 32-bit versus 64-bit Linux and the different operating system versions. All of these variants produce the same output format, enabling attack recognition strategies to be implemented independently in the analysis component. The gatherer does not need to change based on the attack recognition strategy in use. The analysis component can be oblivious to the operating system, architecture, or version. In addition, it is possible to run several analyzers in parallel and combine results. Running the analyzers with saved data rather than live data could be useful for regression testing of analyzers or detecting additional past exploits as improved analyzers are developed.

The division is as strict as it is for a different reason: language. VProbes provides quite limited support for string manipulations and data structures. Additionally, the interpreter has relatively low limits on how much code probes can include. While these limitations likely are solvable, separating them required significantly less work and allows the analyzer to use the functionality of Python or other modern languages without reimplementation.

The gatherer essentially outputs data that looks like the output of the Linux **strace** utility, with names and arguments of some system calls decoded. Depending on what seems most useful to the analysis component, this may eventually involve more or less symbolic decoding of names and arguments.

The gatherer also is responsible for outputting the thread, process, and parent process IDs corresponding to each system call, as well as the program name (**comm** value or Microsoft Windows equivalent, **ImageFile**Name, and binary path). Analysis scripts use this data to

build a model of normal behavior and search for deviations. Generally, these scripts build separate models for each program, as different programs have different normal behavior.

3. Implementation

3.1 Gatherer

The data gathering component uses VProbes to gather syscall traces from the kernel. Gatherers exist for 32-bit and 64-bit Linux (across a wide range of versions), and 64-bit Microsoft Windows 7. The Linux gatherers share the bulk of their code. The Microsoft Windows gatherer is very similar in structure and gathers comparable data, but does not share any code.

To run the gatherer, VProbes sets a breakpoint on the syscall entry point in the kernel code. When a syscall starts to execute the breakpoint activates, transferring execution to our callback. The callback extracts the system call number and arguments from the registers or stack where it is stored. In addition, the probe retrieves data about the currently running process that the analysis components need to segregate different threads to properly sequence the system calls being used and associate the syscalls with the correct per-program profile.

Optionally, the gatherer can decode system call names and arguments. The Linux callback has the name and argument format for several system calls hardcoded. For numeric arguments, the probe simply prints the argument value. For system calls that take strings or other types of pointers as arguments, it prints the referenced data. It also prints the name of the system call in use. Since the current analysis scripts do not examine syscall arguments, this capability was not implemented for the Microsoft Windows gatherer.

Another optional feature reports the return values from system calls. As with argument values, current analysis scripts do not utilize this data. Consequently, while support is available for Linux, it was not implemented for Microsoft Windows.

Writing a gatherer requires two key steps. First, relevant kernel data structures storing the required information must be identified. Second, the offsets of specific fields must be made available to the Emmett script. While the general layout changes little from one version of the kernel to another, the precise offsets do vary. As a result, an automated mechanism to find and make available these offsets is desirable.

3.1.1 Relevant Kernel Data Structures

The first step in implementing the gatherer is to find where the Linux or Microsoft Windows kernel stores the pertinent data. While a userspace IDS could use relatively well-defined, clearly documented, and stable interfaces such as system calls, or read /**proc** to gather the required information, we are unable to run code from the target system. As a result, we must directly access kernel memory. Determining the relevant structures is a process that involves reading the source, disassembling system call implementations, or looking at debugging symbols.

In Linux, the key data structure is the **struct task_struct**. This contains pid (the thread ID), tgid (the process ID), and pointers to the parent process's **task_struct**. We output the thread and process IDs, as well as the parent process ID, to aid in tracking fork calls. On Microsoft Windows, broadly equivalent structures exist (Table 1).

	LINUX	MICROSOFT WINDOWS
Key structure	**task_struct**	**ETHREAD** **EPROCESS**
Breakpoint at	**syscall_call, sysenter_do_call (32-bit)**; **system_call (64-bit)**	**nt!KiSystemServiceStart**
Thread ID	**syscall**	**Cid.UniqueThread**
Process ID	**tgid**	**Cid.UniqueProcess**
Parent PID	**parent->tgid**	**InheritedFromUniqueProcessId**
Program name	**comm**	**ImageFileName**
Program binary	**mm->exe_file**	**SeAuditProcessCreationInfo**

Table 1: Key fields in the Linux and Microsoft Windows process structures

Identifying the running program is surprisingly difficult. The simplest piece of information to retrieve is the **comm** field within the Linux **task_struct**. This field identifies the first 16 characters of the process name, without path information. Unfortunately, this makes it difficult to distinguish an **init** script (which tends to use large numbers of **execve** and **fork** calls) from the program it starts (which may never use **execve** or **fork**). Hence, full paths are desirable.

Path data is available through the **struct mm_struct**, referenced by the **mm** field of the **task_struct**. By recursively traversing the **mount** and **dentry** structures referenced through the **mm_struct exe_file** field, the full path of the binary being executed can be retrieved. Since **exe_file** is the executed binary, the entry for shell scripts tends to be **/bin/bash**, while Python scripts typically have a **/usr/bin/python2.7** entry, and so on. Therefore, it is important to identify the current program based on both the **comm** and **exe_file** fields—simply using **comm** is insufficient because of **init** scripts, while **exe_file** cannot distinguish between different interpreted programs.

In Microsoft Windows, finding this data poses different challenges. The program name (without a path) is easy to find. It is stored in the **ImageFileName** field of the **EPROCESS** structure. As with Linux, the full path is harder to find. On Windows, the **EPROCESS** structure **SeAuditProcessCreationInfo.ImageFileName->Name** field is a **UNICODE_STRING** containing the path to the process binary. Unlike Linux, recursive structure traversal is not required to read the path from the field. However, Microsoft Windows stores this path as a UTF-16 string. As a result, ASCII file names have alternating null bytes, which means Emmett's usual string copy functions do not work. Instead, we individually copy alternating bytes of the Unicode string into an Emmett variable. This converts non-ASCII

Unicode characters into essentially arbitrary ASCII characters. We believe these will be rare in program paths. Additionally, since the current analysis scripts treat the path as an opaque token, a consistent, lossy conversion is acceptable.

3.1.2 Accessing Fields from Emmett

Even after correct structures and fields are found a challenge remains. Although instrumentation has the structure address, it needs to know the offset of each field of interest within the structure so it can access appropriate memory and read data. Unfortunately, kernels are not designed to make the format of internal data structures easily accessible to third-party software. Emmett has two main features that make this feasible: the **offat*** family of built-in functions and **sparse** structure definitions.

3.1.2.1 Finding Offsets at Runtime

The **offat*** family of functions allows finding these offsets at runtime. Each function scans memory from a specified address, checking for instructions that use offsets in particular ways. Frequently, symbol files are used to find the appropriate address.

Emmett supplies three functions in this family. The first is **offatret**, which finds the ret instruction and returns the offset loaded into **rax** to be returned. By passing the address of a simple **accessor** function such as the Windows **nt!PsGetProcessImageFileName** to **offatret**, we can find the offset of a structure field such as the **ETHREAD ImageFileName**. The second is **offatseg**, which finds the first offset applied to the **FS** or **GS** segment registers. These registers are commonly used for thread-local state, making them helpful for finding thread-specific structures such as the **task_struct** in Linux or the Microsoft Windows **ETHREAD**. With a Microsoft Windows guest, **offatseg(&nt!PsGetCurrentProcess)** finds the offset of the **CurrentThread** pointer within the **KPCR** structure. Finally, **offatstrcpy** searches for calls to a specific function address and returns the offset used to load **RSI** for the call. This could be used to find the offset of a string element in a structure, but is not currently used by any gatherers.

The **offat*** functions offer the advantage of allowing a single Emmett script to be used against kernel binaries with different offset values. As a result, the VProbes distribution includes library code that uses **offat*** to find the current PID and certain other fields, which was used for the Microsoft Windows gatherer. However, **offat*** requires finding an appropriate accessor in which to search for offsets and is dependent on the precise assembly code generated by the compiler. Consequently, another mechanism was desirable for new gatherer code.

3.1.2.2 Encoding Offsets in Scripts

Emmett also supports sparse structure definitions, allowing required offsets to be conveniently encoded within the script. A sparse structure is defined similar to a normal C structure, except that some fields can be omitted. Prefixing a field definition with an @ character and an offset enables Emmett to use the specified offset instead of computing an offset based on the preceding fields in the structure and their size. Given a way to find offsets, this allows only relevant fields to be specified, ignoring those that are not needed.

For the Linux gatherer, a Linux module assembles the necessary offsets. When loaded, it exposes a file in the **/proc/** directory. The file contains offsets of a number of important fields in the kernel, formatted as **#define** statements. The file can be copied to the host and **#included** in a script that uses those offsets directly or to define sparse structures. Currently, users must compile the module, copy it to a machine running the correct Linux version, and load it into the kernel. In the future, we plan to provide a script to extract the relevant offsets directly from debug symbols in the module binary, instead of needing to load the module.

In the Microsoft Windows gatherer, the requisite offsets are extracted directly from the debugging symbols. A script uses the **pdbparse** Python library[5] to convert **ntkrnlmp.pdb**, which Microsoft makes available for use with debuggers, into Emmett structure definitions containing the fields of interest. The output file can be **#included** from the gatherer, and the structures can be traversed just as they would be in kernel code.

While this technique requires updating the script for different kernel versions, we find it more convenient. One advantage is that files with offsets can be generated automatically, and the Emmett preprocessor used to incorporate the current offsets into the rest of the code. Using sparse structures allows the Emmett compiler to perform type checking. The process of writing scripts is less prone to error when the compiler distinguishes between pointer and non-pointer members, or an offset in the **ETHREAD** and **EPROCESS** structures. In addition, code is much clearer when the same names and syntax can be used as is present in the source or debugger output. Therefore, while the Microsoft Windows gatherer uses both **offat*** and sparse structures, the Linux gatherer uses only the latter.

3.2 Analyzer

While our work focused on gatherers, we wrote two simple analyzers to validate the technique was sound. Both analyzers build profiles on a per-program basis. Programs are identified by their **comm** value and binary path (or Microsoft Windows equivalent). Recall that using only the former causes an **init** script and the daemon it runs to share a profile, while using only the latter combines all programs in a given interpreted language into one profile.

Both analyzers are passed logs of a normally executing system, as well as a potentially compromised system. They use this information to build a profile of normal behavior. If the potentially compromised system deviates from the profile, they report a potential attack.

3.2.1 Syscall whitelist

The simplest form of a profile is a simple whitelist of allowed system calls. As the normal log is read, the analyzer notes which system calls are used, such as **open**, **read**, **write**, and so on. When the potentially compromised log is read, the analyzer sees if any new system calls are used. If any are, it reports a possible intrusion.

While extremely simple, this analyzer can detect some attacks. We installed a **proftpd** server that was vulnerable to **CVE-2010-4221** and attacked it using Metasploit Project's exploit[3]. Under normal operation, an FTP server has a very simple system call pattern: it mostly just opens, reads, writes, and closes files. An attack, however,

often uses the **execve** function. Since the daemon does not normally use **execve**, it does not appear in the whitelist and the analyzer immediately triggers.

One advantage of this technique is that it requires little tuning and has few false positives. The profile is simple enough that building a complete "normal" profile is quite manageable. A disadvantage, of course, is that it detects relatively few intrusions. For example, an attack on a web server running CGI scripts is quite hard to detect, since a web server uses a much wider variety of system calls.

3.2.2 stide

A mechanism commonly used in the academic intrusion detection literature is sequence time-delay embedding[8], or *stide*. In this technique, the profile consists of all n-tuples of consecutive system calls. For example, with n=3 and a system call sequence of **open**, **read**, **write**, **read**, **write**, **close** the 3-tuples would be **(open, read, write)**, **(read, write, read)**, **(write, read, write)**, and **(read, write, close)**. To scan a trace for an intrusion, the analyzer checks for tuples that are not found in the training data. If enough such tuples are found in a sliding window an intrusion is likely. The length of the tuples, size of the sliding window, and threshold of how many tuples are required to trigger the system can be tuned to achieve an acceptable level of false positives and negatives.

We had difficulty getting this technique to produce reasonable error rates. One refinement that may help is to build tuples on a per-thread basis, so that multi-threaded programs or programs with multiple instances running at once do not interleave system calls. The runs were performed with only a handful of system calls reported to the analyzer. Using a more complete set might be more successful. Finally, we could try larger sets of training data and further tweak the parameters.

4. Performance

One concern for security software such as this is the performance overhead it entails. Since VProbes primarily causes a slowdown when the probe triggers, we expect performance to depend largely on how often system calls occur. In a workload involving a large number of system calls, we expect performance to suffer. In a largely computational workload, however, we expect performance to be roughly the same as without instrumentation.

To measure the performance overhead of instrumentation, we ran an Apache **httpd** instance in a virtual machine and used **ab**[1] to measure how long it took to make a large number of requests to the server. Several different file sizes were tested, as well as static and dynamic content to get a sense of how different system call patterns could affect performance. For this test, a version of the Linux instrumentation was used that printed approximately a dozen system calls and decoded their arguments. We found that for small static files (a couple hundred or thousand bytes), performance was prohibitive. Larger files were more reasonable: a 13 KB file had roughly 5 percent overhead compared to an uninstrumented server, and 20 KB or larger files had overhead well under 1 percent. We also considered a common web application, Mediawiki, to see how dynamic sites compared. An 11 KB page had approximately 13 percent overhead, while a 62 KB page saw 3 percent overhead.

Our current instrumentation is not optimized. To get a better sense of what portions of the instrumentation are slow, we enabled printing of all system calls and experimented with removing various parts of the instrumentation. With a 10 KB file, we found downloading it 10,000 times took 4.7 seconds on average. With full instrumentation, downloads took approximately 14.4 seconds. Of that 9.7 second increase, it appears that setting the breakpoint accounts for approximately 28 percent and computing the path to the binary is approximately 42 percent. With 70 percent of the runtime cost apparently due to these two components, optimizing either could have a significant impact.

For the breakpoint, VProbes supplies two ways to trigger probes. Users can set a breakpoint at an arbitrary guest address (as we currently do) or trigger when a certain limited set of events occurs, such as taking a page fault, changing CR3, or exiting hardware virtualization. This latter type of probe is significantly faster. If one of these probe points can be used instead of the current breakpoint, it could nearly eliminate the 28 percent of cost of the breakpoint.

For computing the path to the binary, more data is gathered then necessary. Simply retrieving the base name of the binary (**bash, python27**, and so on) is likely to be sufficient in combination with the **comm** value, and should be substantially faster. Alternatively, the binary can be cached and only recomputed when the CR3 or another property changes.

5. Conclusion

VProbes provides a viable replacement for an in-guest agent for a system call-based intrusion detection system. Our system is divided into two components: a gatherer that uses VProbes and an analyzer that determines whether a sequence of system calls is suspicious. The unoptimized performance of the gatherer likely is acceptable, depending on the workload, and several opportunities exist for further optimization.

While we focused on data gathering rather than analysis, we have an end-to-end proof of concept that uses whitelisted system calls to successfully detect certain attacks on a simple FTP server. More elaborate analysis techniques have been amply studied in the literature and could be combined with our instrumentation.

References

1 Apache. ApacheBench, http://httpd.apache.org/docs/2.2/programs/ab.html

2 Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In Proc. Network and Distributed Systems Security Symposium, pages 163–176, 2003.

3 jduck. ProFTPD 1.3.2rc3 - 1.3.3b Telnet IAC Buffer Overflow (Linux), http://www.metasploit.com/modules/exploit/linux/ftp/proftp_telnet_iac

4 Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. IEEE Softw., 14(5):35–42, September 1997.

5 pdbparse, http://code.google.com/p/pdbparse/

6 Raghunathan Srinivasan. Protecting anti-virus software under viral attacks. Master's thesis, Arizona State University, 2007.

7 Kymie M. C. Tan, Kevin S. Killourhy, and Roy A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In RAID, pages 54–73. Springer-Verlag, 2002.

8 Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In IEEE Symposium on Security and Privacy, pages 133–145. IEEE Computer Society, 1999.

Design and Implementation of a Cloud Tenant UI

Louis Weitzman
VMware, Inc.
weitzman@vmware.com

Alister Lewis-Bowen
VMware, Inc.
alister@vmware.com

Elizabeth Li
Carnegie Mellon University
etli@andrew.cmu.edu

Jason Fedor
Brown University
jfedor@cs.brown.edu

Abstract

This paper documents the design and implementation of SilverLining, a student intern project to create a simplified user experience to the cloud. vCloud Director® (vCD) provides a full-featured interface for system administrators and others to configure and control their cloud computing resources. The SilverLining project streamlines user workflows and interactions, making it easier to find virtualized applications and add them into a personal workspace. To support this effort, we created a JavaScript SDK to communicate with vCD through its API.

1. Introduction

The current vCloud Director interface allows a user to work with VMs in a very sophisticated way, enabling the management of storage, networks and compute resources. However, this can be very challenging for end users doing straightforward tasks. SilverLining is an implementation of an interface designed for an end user with minimal requirements who just wants to create and start predefined virtualized applications.As a summer project, we had very specific objectives to create a simple interface in the short period of an internship. To accomplish these goals, we needed to make some basic assumptions that could be relaxed going forward.

Objectives

Typically, a user would search a library of application templates and add them to their workspace. This process of instantiating templates creates working virtual applications (vApps). vCloud Director places these workloads into virtual datacenters (VDCs) where the appropriate resources, for example, CPU, storage and networks, are available to run the vApp. Unfortunately, this process can be complex for the casual user. SilverLining provides the basic functionality to find, instantiate and manage these cloud workloads in a simple and effective manner appropriate for a consumer.

The main objectives of SilverLining are two-fold; 1) demonstrate how easy it could be for a consumer to use the cloud, and 2) provide a JavaScript SDK that helps web developers build customized browser applications that communicate with VMware's cloud implementation using familiar technologies.

The interface should not only be easy to use, but also should demonstrate a responsive design that automatically adapts to different display devices and output formats. This makes it possible to show the same information at different resolutions and in different layouts as dictated by the device on which it is rendered.

The JavaScript SDK enables developers to exercise standard web technologies at a lower cost of entry. This allows them to easily create their own branded interface to the cloud. Hopefully, through this process, we can demonstrate concepts and guidance for the future development of VMware consumer UIs.

Assumptions

We made a few assumptions to simplify the end-user interaction. To scope the amount of work for this summer project and provide simplified workflows, we assumed reasonable defaults for user interactions. These defaults, including which virtual datacenter and networks to use in the organization, are saved as user preferences in local storage on the client-side. This allowed us to create the option of a *1-click* instantiation instead of forcing the user through a more complicated wizard workflow. Also, we added a setting to automatically power on templates when they are added to the workspace. A long-term solution might be to save these user preferences in server storage so that these become available to the user across workstations.

We started the project by assuming one VM per vApp but later relaxed this restriction to allow multiple VMs, adding a bit more flexibility. We wanted to create interactions that make the abstractions of the vApp layer simpler to the end-user. To further simplify the user interface, we hid the internal states of vApps and VMs that could confuse the user.

vCloud Director's User Interface

The current vCloud Director's interface presents three primary areas to end users: Home, My Cloud and Catalogs. Home shows a 'card view' of the vApps available to the user and appropriate vApp actions. My Cloud, also referred to as the workspace, does the same using a 'list view' adding navigation to all VMs within these vApps and logs of tasks performed by the user. The Catalog area provides a way to navigate through libraries of available templates that can be instantiated into the user's workspace. Figure 1 shows the current My Cloud page listing all of the user's vApps.

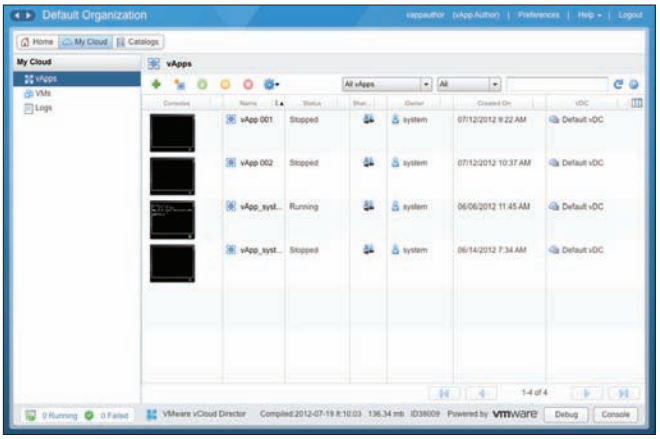

Figure 1. Current vCloud Director showing the My Cloud view, a workspace where users manage vApps and VMs.

2. User Workflows

For our summer project, we focused on a reduced set of workflows that provide the end user with the basic functionality to find, instantiate and manage vApps in their Cloud workspace. These include:

- Authenticate with vCD to gain access to an account within a predefined organization on a VMware Cloud installation.
- Browse a library, choose a template, instantiate it, and power it on. Do this by one-click with a limited number of customized user settings.
- Navigate through the user interface revealing different representations of vApps and Library templates.
- Perform power operations on vApps and VMs.
- Sort/Filter/Search vApps and templates by attributes in a faceted search.
- Show VM console thumbnails and provide console access.
- Use extremely simplified workflows for network configuration and VDC selection.
- Utilize tagging with metadata.
- Handle long running tasks and display notifications to the user.

3. User Experience Design

A few design principles guided us through this project. As we iterated over design solutions, we focused on the end-user workflows. Starting with sketches and wireframe mockups, we produced working examples of our design and refined these implementations, iterating to continually improve our solution.

Design Principles

Some basic design principles informed our decision making throughout the project, including the following:

- **Keep it simple** – Keep it as simple as possible but no simpler.
- **Use direct manipulation** – Objects are represented in the interface and the user should take actions directly on those objects.
- **Use simple hierarchies** – Reduce complexity, but allow future implementations the capability to drill into more detail as needed. For example, use search/filtering to find appropriate templates rather than digging deep into libraries. Also allow implementations on different devices to take advantage of similar navigation.
- **Simplify the workflows** – Limit the use of confirmation dialogs, use undo where appropriate, don't allow the user to get into dead-end or error conditions, and use strong defaults to avoid unnecessary user input.
- **Provide a 'sense of place'** – For both the cloud workspace and library, create a unique look to distinguish the space and the tasks to be done there.

Design Iterations

Initially, our designs were highly influenced by the existing interface. We began with sketches and refined them in low-resolution wireframes.

In our first implementation, we recycled the notion of representing vApps using cards, but added a "card flip" interaction as a means of showing more details. We also planned other interactions such as sorting and drag-and-drop. Each vApp was represented as a distinct object that could be directly manipulated. Any information not needed right away could be accessed in a single click (Figures 2 and 3).

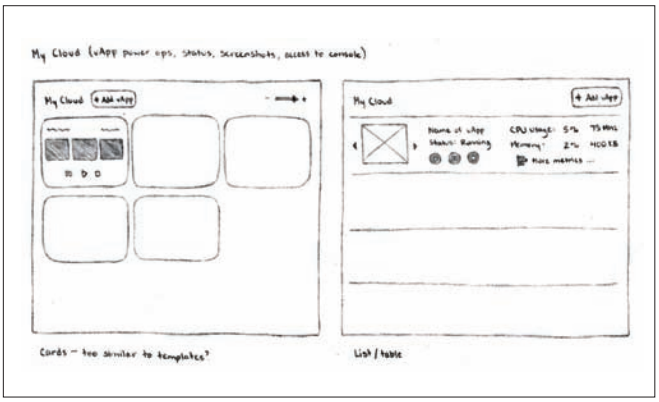

Figure 2. Initial sketches exploring card and list views of vApps and templates similar to what exists in the current UI.

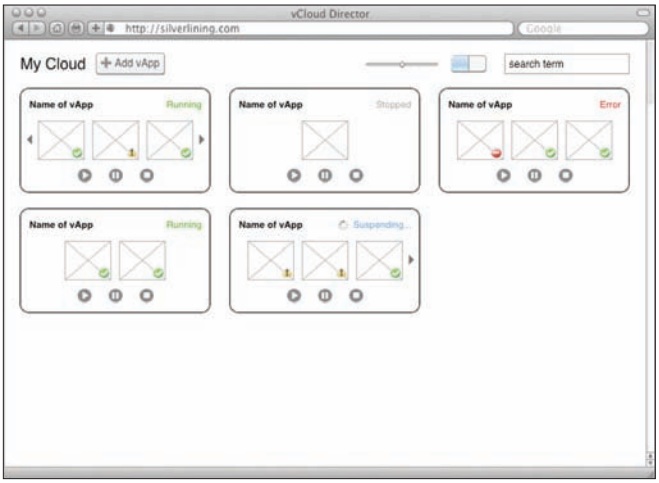

Figure 3. Wireframe mockup of the My Cloud as cards.

We designed templates to be displayed in a simple table that can be filtered using a powerful, faceted search syntax (Figure 4). For example, to find vApps with a name containing the string "windows", the user could type "name:windows" and immediately filter the view. An early implementation of this design allowed the user to specify searchable values for several object attributes. We expanded this capability in the final implementation to include non-string attributes, such as memory size, as well. This search is very quick and effective, and after using it to narrow down options, templates could quickly be scanned down the list for comparison.

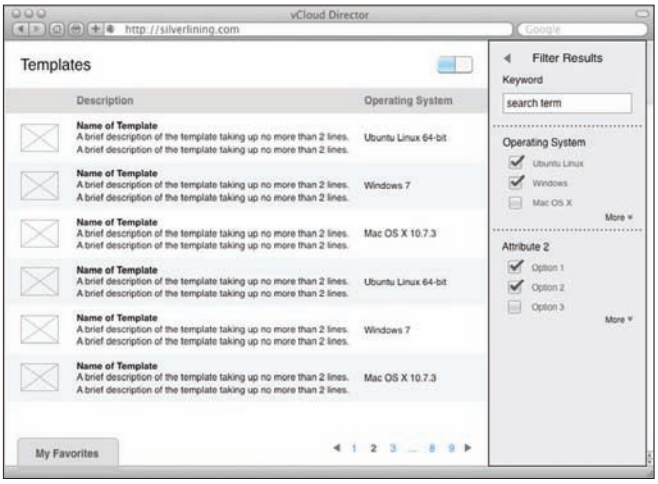

Figure 4. Wireframe mockup of the Library displayed as a list with faceted search.

4. User Feedback

To get feedback on the initial designs, we conducted a small focus group and demonstrated SilverLining to the vCD team in Cambridge. From these sessions we gained two key insights. First, there needed to be better representation and navigation of the object hierarchy. The vApp cards did not suggest an intuitive way to drill into details about vApps and VMs and their associated resources.

Second, choosing library templates needed to be more of a *shopping experience*; template descriptions, ratings, number of instantiations and cost information should be shown in a way that helps users make informed decisions about what to choose.

Also, it became obvious that our original designs did not provide for different layouts while maintaining a common interaction pattern. So we went back to the drawing board to focus on how the UI would behave in various situations (Figures 5 and 6).

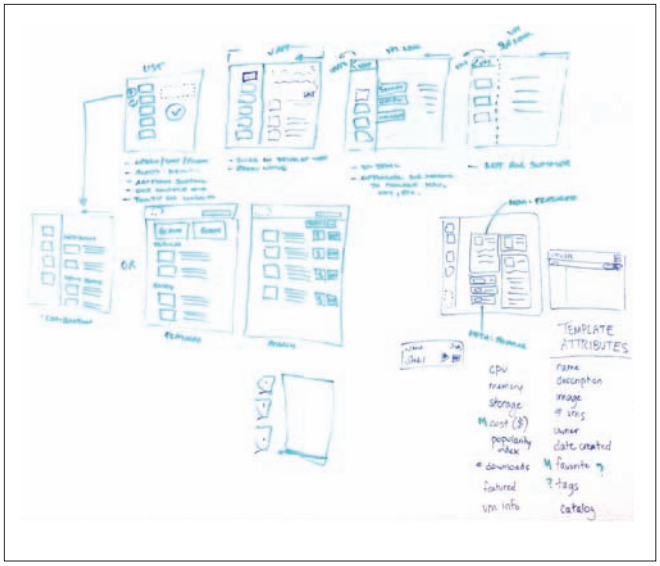

Figure 5. Back to the drawing board (literally) with a focus on the dynamic interaction between outline and detail views.

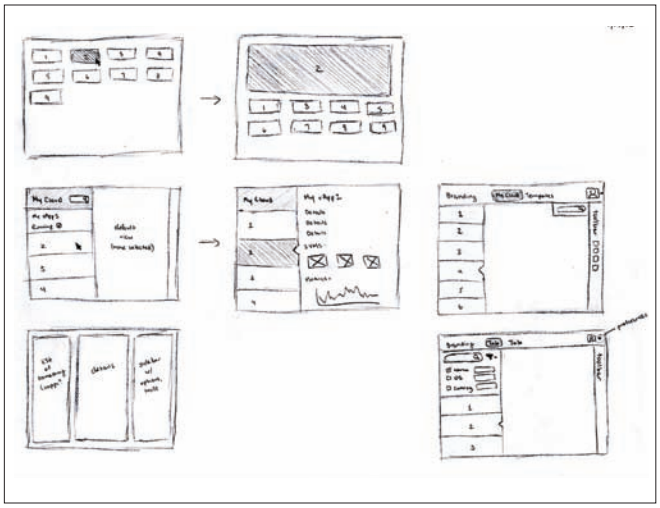

Figure 6. More sketches with expanding cards and multi-view layouts using a left navigation pane.

5. Design Implementation

After exploring many alternatives, we converged on an implementation utilizing sliding panels. This design maintains context, enables drag and drop between the library and the cloud, and supports layouts in other devices while maintaining the application's interaction pattern. The panels are divided into two columns, with a list of vApps on the left and details on the right. The panels slide left and right to navigate linearly through the hierarchy and can be extended to drill deeper through any number of levels (Figures 7 and 8).

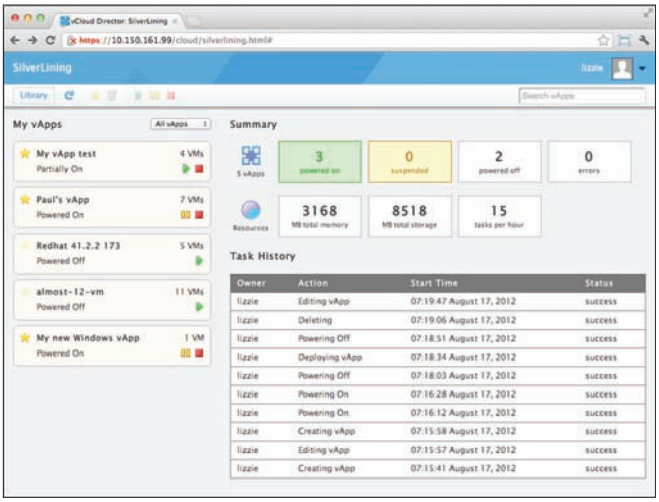

Figure 7. SilverLining's home page showing vApps, general status and recent tasks.

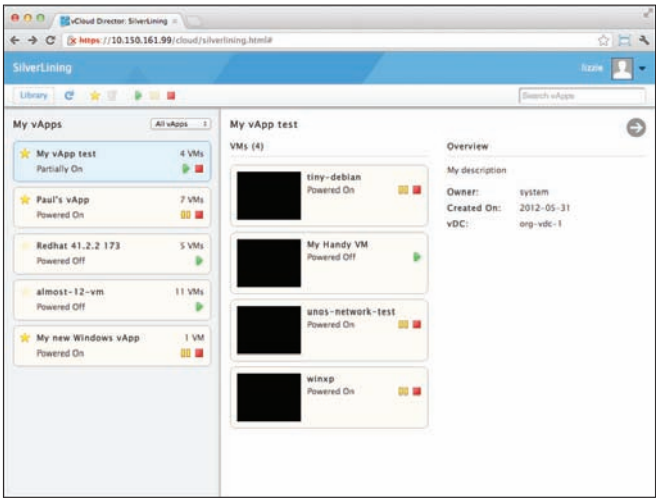

Figure 8. Navigating into the details of a vApp displays all its VMs and other associated metadata.

The template library was designed as a separate, distinctly branded layout for browsing with a shopping catalog feel, showing featured templates and information about popularity (Figure 9). The library slides open next to the user's list of vApps so that the user knows what context they are in and how to go back. These sliding panels provide an affordance that worked well with the 'swipe' gesture on touch devices. Figure 10 shows an interaction in the library filtering the content with a faceted search. Figure 11 illustrates an instantiation workflow allowing the user to customize a few parameters. In comparison, the instantiation workflow in the current vCloud Director presents a multi-step wizard requiring the user to choose many options, some of which they may not understand or even care about.

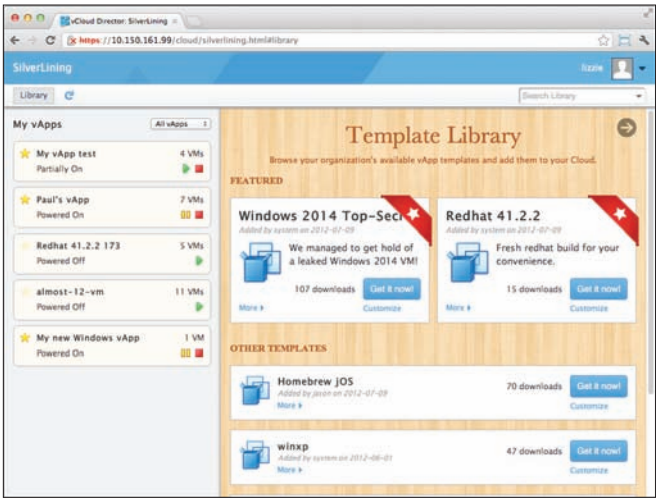

Figure 9. With vApps still visible, the Library displays all templates in a distinctive layout, promoting featured ones.

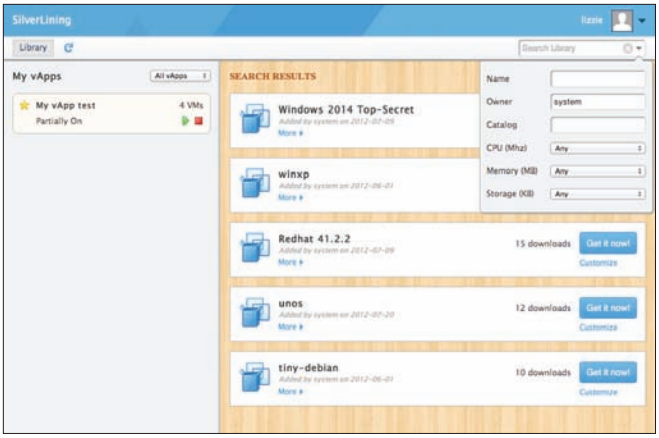

Figure 10. Exploring the Library with faceted search filtering templates to view those owned by "system".

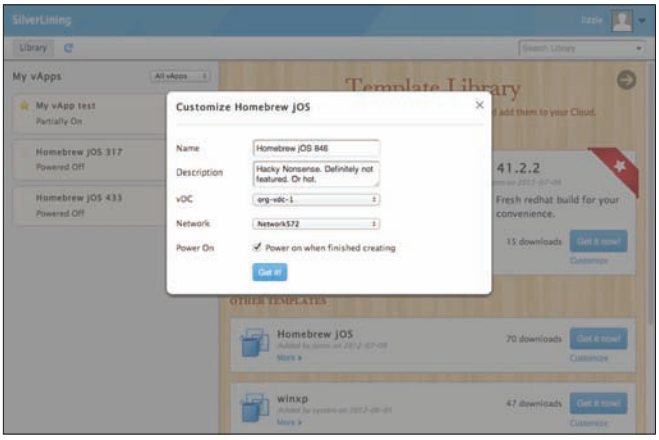

Figure 11. One-click instantiation immediately adds the template to your workspace while the customize option, shown here, provides the ability to override the default values.

Testing our implementation using the panel layouts on the different standard device sizes validated this last design iteration. As the screen real estate was reduced, the layout degraded gracefully but still maintained the same interaction pattern. The layout for a mobile phone can be seen in Figure 12. In the end we had a fully functional application written in HTML5, CSS3 and JavaScript.

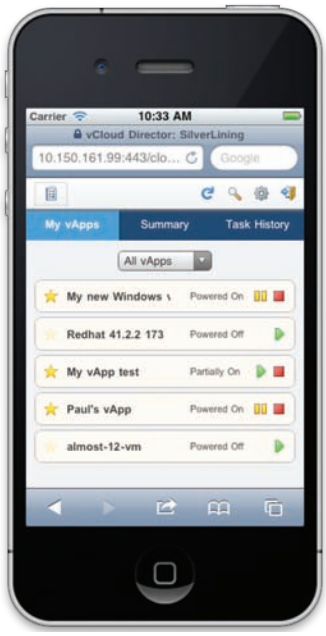

Figure 12. Alternative layout of the same content for smartphones illustrates responsive design organizing information in tabs when limited space is available.

6. Implementation and Technology

Our versatile JavaScript SDK powers the SilverLining project. This library communicates with VMware vCloud Director (vCD) v5.1 through its API. It assumes that a vCD Organization and a user account already exist. This latest version of vCloud Director includes enhanced metadata support and access to server side support for an HTML5 console to VMs. We limited the scope of the SDK development to provide just the functionality that our defined end user would require. This scoping effort helped encapsulate the requirements to produce a demonstrable deliverable for our summer project.

The SDK functions can fetch information on a certain object from vCD, or parse information already stored in the local cache. After authentication, the SDK fetches basic information about the user's workloads—the list of vApps and their corresponding VMs, basic network and VDC info, catalogs of vApp templates, and user information. Then, based on these starting objects, the web developer can pull specific details based on what they want to display. For example, to learn about metadata for a certain vApp, you call a function, 'cloud.metadata.get(vAppObject)', and the SDK will perform the sequence of calls necessary to learn about the metadata for that vApp. Prefetching provides some speed benefit and allows the rendering logic to operate independently from the SDK.

Using the credentials after logging in, the JavaScript SDK has full access to vCloud Director's REST API. It performs a number of AJAX calls in order to navigate the object hierarchy. If the SDK ran on the user's local host or some other website, it could trigger browser security measures designed to prevent cross-site scripting (XSS) when talking to a remote server [9]. Initially, we remedied the situation by running each request through a local proxy server, which then relayed the calls through HTTPS to the vCD instance and back. However, this additional proxy code added to the size of the SDK and limits the ability for this library to be dropped into existing web applications. We decided to install this SDK directly into the vCD cell thus increasing the likelihood that the API's IP address and the resulting web application's IP address are the same, thus thereby eliminating any cross-site scripting limitations.

Metadata

We used the powerful metadata feature in vCD 5.1 to extend the functionality in useful ways. Metadata provides typed, key-value pairs for various vCD objects including vApps, VMs and templates. Once defined, objects can be filtered and sorted to identify important aspects of these objects.

This metadata feature created many new possibilities for our design, several of which we implemented. For example, for the notion of a *shopping experience*, we labeled some templates as *featured* to allow the rendering logic to display those templates more prominently in the Library section. In an actual use-case, we could imagine the vCD Organization Admin role promoting some particular templates for a specific project. We also provide live data about the number of downloads with a counter that is incremented on every instantiation. *Favorites* is another metric stored as metadata for an individual user. Once tagged, the user can then filter for these metadata enhanced vApps or templates. Socializing the popularity of shared objects, in this case templates, provides users with ways to gauge which templates might be best to add to their workspace.

Local Storage

Network latency can occasionally cause the vCD cell to take a very long time to respond to requests. To make our system more responsive, we decided to use a caching system so we could display data immediately at startup. Since the rendering logic is decoupled from the SDKs logic to fetch the data, the UI can display cached information while the live data loads. Then, once the SDK has live data, it can present this to the UI seamlessly. As a result, the user has immediate access to their data as soon as they log in, even though it might be a little old. They can navigate around the hierarchy, even queue up instantiation requests or power something on before any information has even transferred to the client. In effect, this is an implicit *Offline Mode*, creating the illusion of connectivity before it has been established. To store the cache, we are compressing the cloud's data, and storing it as a JSON object in local storage, an HTML5 feature. This approach is appropriate for the type of user we have in mind for this design. But we are aware that storing data this way could become un-scalable when managing one or several vCD organizations or an entire cloud.

Notifications

To support long running operations, the SDK takes advantage of the Task system within the vCloud API. Whenever an action is submitted, a task object is returned. This object can be polled to get the current status until completion. This ensures that the SDK only pulls information as needed. If any task results in an error while the user is logged in, a notification is issued and noted in the task log. Also, the data model maintains the status of all objects, including any errors, thereby allowing the interface to render object status as it sees fit.

Separation of Form and Content

As mentioned earlier, we decoupled fetching data and the presentation of that data when rendering the UI. To do this we used a Model-View-View Model (MVVM) architecture through a popular JavaScript library plugin called Knockout.js [7]. With this separation, we can use the same responsive CSS3 design to provide a polished presentation and adaptive layout depending on the device used.

7. Intern Coordination

Having two summer interns created some challenges as well as opportunities for SilverLining. We needed to plan the project and provide some groundwork before they arrived, and, once onboard, bring them up to speed on the problem and current system. Also, we needed to clearly separate responsibilities and deliverables. Luckily, a separation around JavaScript SDK development, UI design, and implementation was straightforward. The design helped create the requirements for the SDK, and the UI implementation provided a testing harness for SDK development.

Recognizing that communication is a critical part of any successful project, we made it a priority. Throughout the summer we met regularly to discover any roadblocks and to map current progress. Also, we made an effort to communicate our results to other interested parties, including the key product drivers within the organization. The timely feedback received helped move the design and implementation forward. Specifically, we created and maintained a wiki with sections for requirements, schedules, processes, and results for all to view, including video screen-casts showing our progress. Weekly meetings reviewed past work and planned future work. Additional meetings were held to resolve on-going issues and critique the latest design iterations.

8. Conclusion

With two interns providing critical mass, we were able to make significant progress over a very short time. As a result, SilverLining was a great success and met all of our expectations. We were able to demonstrate a UI to vCloud Director that was both simple and effective for the end user we defined. The JavaScript SDK we created now provides a building block for future efforts using standard web technologies. Our responsive design provides a glimpse at how future efforts can adapt to the ever-increasing number of display devices.

We plan to solicit more user feedback beyond our immediate group to help guide our implementation. Once completed, we plan to distribute SilverLining as a Fling [3] to allow developers both inside and outside VMware to exercise the JavaScript SDK and provide us with feedback. The goal is to allow third parties to easily create custom interfaces uniquely branded for their own organizations, and scoped to the needs of specific classes of users wanting to use vCloud Director.

Acknowledgments

We would like to acknowledge and thank those who helped us along the way, including our managers: Brad Meiseles, Vinod Johnson; our colleagues: Stephen Evanchik, Jeff Moroski, Zach Shepherd, Simon Threasher; user research: Amy Grude, Peter Shepherd; our other interns: Paul Furtado, Dan Trujillo and Shivam Tiwari; our product managers: Catherine Fan, Maire Howard, and Dmitri Zimine and our reviewers: Eric Hulteen and Steve Strassmann.

References

1. CSS, http://www.w3.org/TR/CSS/
2. Krug, S. “Don’t make me think, A Common Sense Approach to Web Usability,” Macmillan, 2000.
3. Fling, http://labs.vmware.com/flings
4. HTML5, http://www.w3.org/TR/html5/
5. “HTML5 Leads a Web Revolution,” Anthes, G., Communication of the ACM, Vol 55, No. 7, July 2012.
6. jQuery, http://docs.jquery.com/Main_Page
7. Knockout, http://knockoutjs.com/
8. VMware vCloud® API., http://communities.vmware.com/community/vmtn/developer/forums/vcloudapi
9. XSS, http://en.wikipedia.org/wiki/Cross-site_scripting

FrobOS is turning 10:
What can you learn from a 10 year old?

Stuart Easson
VMware, Inc
stuart@vmware.com

Abstract

The Virtual Machine kernel, and the virtual machine monitor (VMM) in particular, have a difficult but critical task. They need to present a set of virtual CPUs and devices with enough fidelity that the myriad supported guest operating systems (GOS) run flawlessly on the virtual platform. Producing a very-high-quality VMM and kernel requires all low-level CPU and device features to be characterized and tested, both on native hardware and in a virtual machine.

One way to work toward this goal is to write tests using a production operating system, such as Linux. Because an operating system wants to manage CPU and device resources, it ends up hiding or denying direct access to CPU and low-level device interfaces. What is desired is a special-purpose operating system that enables easy manipulation of machine data structures and grants direct access to normally protected features. This article presents the main features of FrobOS, a test kernel and associated tools that directly address this development goal. Written in C, FrobOS provides access to all hardware features, both directly and through sets of targeted APIs. Using a built-in boot loader, a FrobOS test is run by being booted directly on hardware or in a virtual machine provisioned within a hypervisor. In either case, FrobOS scales very well. Tests are simple to run on systems ranging from a uniprocessor 32-bit with 128MB of memory to a 96 thread, 64-bit, 1TB Enterprise server.

Despite FrobOS maturity,[1] information about FrobOS has not been widely disseminated. Today, only a handful of teams actively use the system, with the greatest use occurring within the virtual machine monitor team. This article describes FrobOS for the broader development community to encourage its use.

1. Introduction

The virtualization of a physical computer poses a number of difficult problems, but the Virtual Machine kernel (VMX) and virtual machine monitor (VMM) in particular have both a difficult and functionally critical task: present a set of virtual CPUs with their associated devices as a coherent virtual computer. This pretense must be executed with enough fidelity that the myriad supported (GOS) run flawlessly on the virtual platform. The task is made more difficult by the requirement that the virtual CPU must be able to change some of its behaviors to conform to a different reference CPU, sometimes based on the host system, sometimes defined by the cluster of which it is a part, or based on the users preferences and the GOS expectations. Despite these conflicting requirements it must do its work with little or no overhead.

Great software engineering is an absolute requirement to implement a high-quality virtual machine monitor. The best engineering is ultimately only able to rise to greatness through the rigors of thorough testing. Some testing challenges can be mitigated through focused testing with reference to a high-quality model. Unfortunately, these represent two major difficulties: both the reference model and focused testing are hard to find.

For many purposes, the behaviors one wants in the virtualized CPU are described “in the manual”. Yet there are problems:

- Vendor documentation often contains omissions, ambiguities, and inaccuracies.
- Occasionally CPUs do not do what manuals suggest they should.
- Some well-documented instructions have outputs that are ‘undefined’, a potential headache for a ‘soft’ CPU that is required to match the underlying hardware.
- Different generations of x86 hardware do not match for undefined cases, even those from the same vendor.

1 Perforce spelunking suggests the first check-in was more than 10 years ago.

For the cases where reference documents fail to provide a complete answer, the only recourse is to execute the instructions of interest and study what happens in minute detail. These details can be used to make sure the behavior is modeled correctly in the virtual environment. This brings us to the second difficulty: how do you execute the instructions of interest in an appropriate way? Commodity operating systems (COS) are not helpful in this regard. Despite offering development tools, they use CPU protection mechanisms to stop programs from running many interesting instructions.

This article describes FrobOS, a testing tool designed specifically to address the issues described above. The Monitor team uses FrobOS to build their unit tests, performing checks on the virtual hardware that would be difficult or impossible using a COS.

The FrobOS test development process is hosted on Linux and uses familiar tools (Perl, gcc, gas, make, and so on) provided by the VMware® tool chain. The result of building a FrobOS test is a bootable image that can be started from floppy, PXE, CD, or other disk-like device. FrobOS startup is quite efficient, and is limited essentially by the host BIOS and boot device. As a bootable GOS, FrobOS can be run easily in a virtual machine, and virtual machine-specific customizations to the run-time environment can be made at build or test execution time.

This article presents the use of FrobOS on VMware hosted products such as VMware Workstation™. However, it also runs on VMware® ESX®. See the end of this article for links to more information about this important tool.

2. The Problem

Two types of problems need to be addressed to make a high-quality VMM: the generation of reference models, and testing.

1. **Generating reference models.** What do you expect your virtual CPU and devices to do? When vendor manuals fail to provide an adequate answer, all that is left is to execute code and see what the CPU does under the conditions of interest.
2. **Testing.** In general, testing has a number of issues:
   - • Development efficiency: How hard is it to create a test?
   - • Run-time efficiency: How much overhead does running a test incur?
   - • Environmental accessibility: What can one touch and test before crashing the world?
   - • Determinism: If you do the same steps again, do you get the same results?
   - • Transparency: Can a test result be interpreted easily?

(COS) such as Linux and Microsoft Windows offer tools, documentation, and support for making end-user programs. They seem like a good vehicle for testing. Unfortunately, running a test requires a very different environment than the one used for creating it. When considering an operating system such as Linux or Microsoft Windows for running tests, a number of problems emerge.

- Typical COS work hard to stop programs from accessing anything that would interfere with fairness, stability, or process protection models.
- While it is possible to use an operating system-specific extension to gain supervisor privileges, the environment is fragile and very unpredictable.
- Many desired tests are 'destructive' to their run-time environment, requiring the machine to be rebooted before testing can continue.
- The long boot time for a typical COS makes for very low run-time efficiency.
- Inevitably, the boot of a COS grossly pollutes the lowest level machine state.
- Scheduling in a GOS makes the exact reproduction of machine state difficult or impossible.

Given these problems what is needed is a special-purpose operating system that does not protect or serve. It should be largely be idle unless kicked into action. It needs to boot quickly, and ideally does not do anything outside the test author's notice.

3. The Idea

Our approach is to create an operating system with a kernel specifically designed for efficient generation and completely unfettered execution of low-level CPU and device tests. Ideally, development for this kernel would use familiar tools in a stable run-time environment. While all levels of x86 hardware and devices need to be accessible, users should not be required to know every detail of the hardware to use features in a normal way. Where possible, our operating system provides high-level programmatic access to CPU features, a set of C language APIs that encapsulate the complexities of the x86 architecture and its long list of eccentricities. Done well, it would perhaps mitigate the inherent pitfalls in ad-hoc coding. Properly realized, the benefits of FrobOS are many, including:

- **Ease of use.** FrobOS tests are cross-compiled from Linux, using normal development tools and processes. Running the output in VMware Workstation is automatic, while doing so in VMware ESX requires only a few more command-line parameters. Running natively requires the boot image to be DD'ed onto a floppy disk, or better yet, setting up an image for PXE booting.
- **Efficiency.** The early development of FrobOS was driven by the recognition that unit tests being built by Monitor engineers necessarily shared a lot of code—code that was tricky to write—so why not do it once, and do it right? Over time, this library of routines has grown to include the previously mentioned, as well as PCI device enumeration and access, ACPI via Intel's ACPICA, a quite complete C RTL, an interface to Virtual Performance counters, a complete set of page management routines, and more.

- **Quick results.** Shortest runtime has always been a philosophy for FrobOS. Monitor engineers are an impatient bunch, and the net effect is a very fast boot time. In a virtual machine, a FrobOS test can boot, run, and finish in less than 10 seconds. In general, the boot process is so fast it is limited by the BIOS and the boot device, virtual or native.
- **Flexibility.** By default, the output of a test build is a floppy disk image that can be booted natively (floppy, CD, PXE...), in VMware Workstation (default), or VMware ESX (in a shell or via remote virtual machine invocation). If needed, the build output can be a hard disk image. This is the default for a UEFI boot, and might be needed if the test generates large volumes of core files.
- **Footprint.** A typical FrobOS test run-time footprint is small. The complete test and attendant run-time system fit on a single 1.44MB floppy disk, with room for a core file if the test crashes. FrobOS hardware requirements are small. Currently, it boots in ~128Mb, but a change of compile time constants can reduce it to the size of an L2 cache.
- **Restrictions.** FrobOS imposes very few restrictions. All CPLs and CPU features are trivially available. There is no operating system agenda (fairness, safety, and so on) to interfere with test operation.
- **Testing.** FrobOS addresses the need for low-level CPU and device testing in a way not seen in a COS. Since FrobOS is cross-compiled from Linux, the programming environment for FrobOS has a familiar feel. Engineers have their normal development tools at hand. The FrobOS kernel, libraries, and tests are built with GCC and GAS. While the majority of code is written in C, the initial bootstrap code and interrupt handlers are assembly coded.
- **Scalability.** FrobOS is eminently scalable. It runs on a Pentium III system with less 128Mb of memory, yet offers complete functionality on systems with 96+ CPUs and more than 1TB of RAM. A recent check-in to improve the efficiency of the SMP boot required statistics to be gathered. The results highlighted just how quick this process is—all 80 threads in a server with an Intel® Westmere processor can be booted and shutdown by a FrobOS SMP test in approximately 1 second.
- **Booting.** For characterization purposes, FrobOS tests can be booted directly on hardware from floppy, CD-ROM, and PXE, and there has been success with USB sticks. There are several suites populated with tests that are expected to function usefully in a non-virtual machine (native environment).

4. Welcome to the Machine

What is FrobOS? Depending on the specific interest one has, FrobOS can be seen in a number of different ways: For a test developer, FrobOS is perhaps most accurately described as a GOS construction kit. For someone performing a smoke test of a new build of VMware Workstation, FrobOS is a catalog of unit tests. While for an engineer working on enabling x86 Instruction set extensions in the Monitor FrobOS is an instruction level characterization tool, providing many convenient interfaces to low level details.

Looking around the FrobOS tree within the bora directory, one finds a set of scripts, sources, and libraries. The purpose of the pieces is to build a bootable image designed to execute the test(s) in a very efficient way. The tests are stored in the **frobos/test** directory. Each test is stored in an eponymously named directory and represents a unit test or regression test for a particular bug or area of the monitor.

At present, FrobOS has three teams developing new tests: the Monitor, SVGA, and Device teams have all generated great results with the platform. Most recently, an intern in the Security team made a USB device fuzzer with great results. He filed several bugs as a result of his work, and added basic USB functionality to FrobOS. Later sections of this article present a real example of a device test, specifically demonstrating testing proper disablement of the SVGA device.

FrobOS tests are defined in the **suite.def** file. This file can be found, along with the rest of the FrobOS infrastructure, under the bora directory **vmcore/frobos**. The **doc** subdirectory contains documentation about FrobOS. The test subdirectory contains the tests and **suite.def**. The **runtime/scripts** subdirectory contains scripts, such as **frobos-run**, for running FrobOS.

Most FrobOS operational functions are controlled with an executive script called **frobos-run**. Written in Perl, the script uses the catalog of tests defined in **suite.def** to build each requested test's bootable image(s) and then, by default, starts the execution of the images in VMware Workstation as virtual machines. As each test is built and run, **frobos-run** provides several levels of test-specific parameterization. At build time, a handful of options control the compiler's debug settings (**-debug**), whether to use a flat memory model (**-offset**), and which BIOS to use when booting (**-efi**), and so on.

At run time, **frobos-run** creates a unique configuration file to control VMM-specific parameters, such as the number of virtual CPUs (VCPUs), memory size, mounted disks, and so on. Additionally, other test or VMM-specific options can be applied via the command line. These options are applied to the current run, and can override settings in **suite.def**. FrobOS uses GRUB as its boot loader and supports reading parameters from the GRUB command line, so options can be passed to a test to control specific behaviors. Ordinarily, a FrobOS test runs to some level of completion. In normal cases a test can Pass, Fail, or Skip. The final states of Pass and Fail are easily understood. Skip is slightly unusual—it means either **frobos-run** or the test discovered an environmental issue that would make running the test meaningless. An example might be running a test for an Intel CPU feature on a VIA CPU, or running an SMP test with only one CPU. In the event something happens to terminate the test prematurely, **frobos-run** inspects the logs generated and notices the absence of Pass messages and Fails the test.

So what do you need to make a FrobOS test? The minimal 32-bit FrobOS test can be assembled from the following four items:

1. Two additional lines in **suite.def**, describing how **frobos-run** should find, build, and run the test. Example: **legacymode**
2. A directory in **../frobos/test/** whose name matches the entry in **suite.def**.

3. A file named **frobostest.mk** that describes the source files required to build the test.

```
CFILES = main.c
```

4. The source file (main.c) containing the test code. Example:

```
#define ALLOW_FROBOS32
#include <frobos.h>

TESTID(0, "Journal example test");

void
Frobos_Main(void)

{
EXPECT_TRUE(1 == 1);
}
```

Assuming a VMware Workstation development tree and tool chain are already established, the test is built and invoked using frobos-run as follows:

```
frobos-run -mm bt legacymode:example
```

This invocation produces the following output:

```
Found 1 matching test...

Building tests....
Launching: legacymode:example (BT) (PID 27523), using 1 VCPU
PowerOn

Random_Init: Using random seed: 0x34a37b99379cfef2

TEST: 0000: Journal example test CHANGE: 1709956
PASS: Test 0000: Journal example Test (1 cases)
Frobos: Powering off VM.
      PASS: legacymode:example (BT) (PID 27523) after 5s.

Hostname:       shanghai
Command Line:   legacymode:example -mm bt
Environment:    /vmc/bora:ws:obj
Client:         vmc, synced on 2012/02/07, change number 1709956
Suite spec:     legacymode:example
Monitor modes:  BT
Start time:     Tue Feb  7 12:38:00 2012
End time:       Tue Feb  7 12:38:07 2012
Duration:       0h:00m:07s

Tests run:      1
Passes          1
Skipped Tests:  0
Test Failures:  0
Log file: .../build/frobos/results/shanghai-2012-02-07.5/frobos-runlog
------------------------------
```

While there is a lot of bookkeeping information, the lines starting "PASS:..." show the test booted and ran successfully. The total time of 7 seconds includes starting VMware Workstation, booting FrobOS, and running the test. If the test were run natively, the lines from "**Random_Init:...**" to "**PASS: legacymode:...**" would be identical. Such log lines are copied to **com1** as well.

Code reuse is critical for productivity and reliability. FrobOS makes testing across three major CPU modes relatively trivial. The test example is part of the legacymode suite and runs in 'normal' 32-bit protected mode. It can be made into a 64bit test with the addition of one line (#define ALLOW_FROBOS64), seen here in situ:

```
#define ALLOW_FROBOS32
#define ALLOW_FROBOS64
#include <frobos.h>

TESTID(0, "Journal example test");
...
```

One line in **suite.def** also is needed:

```
...
example:
        legacymode,
        longmode
...
```

The 64-bit (longmode) version of the test is invoked with the following command:

```
frobos-run -mm bt longmode:example
```

While not shown here, the output looks very similar, and as expected the test passes again. To run the test in compatibility mode, use **#define ALLOW_FROBOS48**.

By default, **frobos-run** launches a test three times, once for each of the monitor's major execution modes: Binary Translation (BT), Hardware Execution/Software MMU (HV), and Hardware Execution/Hardware MMU (HWMMU). I used the **-mm bt** switch to override this since I did not want all that output. Note that **-mm** is the short form of the **–monitorMode** option. After your test is ready, **frobos-run** allows all instances for a particular test to be **run** using the all pseudo suite:

```
frobos-run all:example
```

This runs all the entries in **suite.def** for the test named **example** using all three monitor modes. In this case, it runs six tests, the product of the CPU modes and monitor execution modes: (32, 64) x (BT, HV, HWMMU). Because the number of tests can explode quickly, **frobos-run** is SMP-aware. It knows how many CPUs each test needs (from **suite.def**) and determines how many are available on the host. Using the **-j nn** command line option, it schedules multiple tests to run in parallel. As a result, the six tests can be run much more quickly on a 4-way host:

```
frobos-run all:example -j 4
```

This results in the following output:

Found 6 matching tests...

```
Building tests....

Launching: legacymode:example (BT) (PID 16853), using 1 VCPU
Launching: longmode:example (HV) (PID 16855), using 1 VCPU
Launching: legacymode:example (HWMMU) (PID 16856), using 1 VCPU
Launching: legacymode:example (HV) (PID 16858), using 1 VCPU
        PASS: legacymode:example (HWMMU) (PID 16856) after 4s.
Launching: longmode:example (HWMMU) (PID 17009), using 1 VCPU
        PASS: legacymode:example (BT) (PID 16853) after 5s.
Launching: longmode:example (BT) (PID 17048), using 1 VCPU
        PASS: longmode:example (HV) (PID 16855) after 5s.
        PASS: legacymode:example (HV) (PID 16858) after 5s.
        PASS: longmode:example (HWMMU) (PID 17009) after 3s.
        PASS: longmode:example (BT) (PID 17048) after 4s.

Duration:       0h:00m:12s

Tests run:      6
Passes:         6
Skipped Tests:  0
Test Failures:  0
Log file: .../build/frobos/results/shanghai-2012-02-07.9/frobos-runlog

-----------------------------------
```

As you can see, **frobos-run** starts four tests and waits. As each test finishes, **frobos-run** starts another test. The total run time is about twice as long as individually run tests, for a net speed increase of approximately 300 percent. Additional host CPUs allow more tests to execute in parallel. Since some tests require more than one CPU, **frobos-run** keeps track and schedules accordingly. In addition, the **frobos-run** scheduler is quite sophisticated. By default, the scheduler attempts to keep the host fully committed—but not over committed—so tests are scheduled according to CPU and memory requirements.

The following shows how to create a test that would be tricky, perhaps impossible, in an operating system such as Linux or Microsoft Windows. It first writes code to touch an unmapped page, generating a page fault. Once the fault is observed, it checks a few important things that should have occurred or been recorded by the CPU as a result of the page fault exception.

1. A page fault occurs. (This is possible in Linux and Microsoft Windows!)
2. The error code reported for the page fault is correct.
3. The address reported for the page fault is correct.

```
#define ALLOW_FROBOS32
#include <frobos.h>

TESTID(1, "Journal example Test 1");

void
Frobos_Main(void)
{
        // MM_GetPhysPage() returns the address of an unmapped physical
page
        PA physAddr = MM_GetPhysPage();

        // There is no mapping for physAddr, this must #PF...

        EXPECT_ERR_CODE(EXC_PF, PF_RW, *(uint8 *)physAddr = 0);
        EXPECT_INT(CPU_GetCR2(), physAddr, "%x");
}
```

Running the test results in the following:

```
TEST: 0001: Journal example Test 1 CHANGE: 1709956
PASS: Test 0001: Journal example Test 1 (1 cases)
Frobos: Powering off VM.
        PASS: legacymode:example1 (BT) (PID 22715) after 4s.
```

With a few extra lines in the C source, and two additional entries in **suite.def**, we can test for the same conditions in both compatibility and 64-bit modes.

```
/* C source */
#define ALLOW_FROBOS32
#define ALLOW_FROBOS48
#define ALLOW_FROBOS64

/* suite.def */
...
example:
        legacymode,
        compatmode,
        longmode
...
```

The **frobos-run** tool automatically runs the test in the BT, HV, and HWMMU monitor modes.

```
...
        PASS: legacymode:example1 (HWMMU) (PID 23171) after 4s.
        PASS: compatmode:example1 (HWMMU) (PID 23175) after 4s.
        PASS: longmode:example1 (HWMMU) (PID 23192) after 4s.
        PASS: legacymode:example1 (BT) (PID 23165) after 5s.
        PASS: compatmode:example1 (HV) (PID 23167) after 5s.
        PASS: legacymode:example1 (HV) (PID 23172) after 5s.
        PASS: longmode:example1 (HV) (PID 23168) after 5s.
        PASS: longmode:example1 (BT) (PID 23195) after 5s.
        PASS: compatmode:example1 (BT) (PID 23177) after 6s.
...
Duration:       0h:00m:10s
```

That is a lot of testing for 5 (or 3?) real lines of 'new' code.

This test uses a couple of **EXPECT** macros, wrappers for code to check for a certain expected value or behavior, and a lot more

code to catch all sorts of unexpected behaviors. In the event the values sought are not presented, that fact is logged and the test is flagged as a failure. Execution continues unless overridden, since there may be other tests of value yet to run. The **EXPECT** macros can hide a lot of very complex code. If they cannot perform the task needed, the underlying implementation is available as a try/fail macro set for more generality.

5. So just how much do I have to do?

We saw earlier that making test code start in 32-bit, compatibility, or 64-bit mode is easy. To enable cross-mode testing, the FrobOS runtime library is as processor mode agnostic as possible. Calling the **MM_MapPage()** function achieves the same result in 32-bit/PAE or 64-bit modes. The following, slightly more elaborate, code fragment retrieves an unused page, identity maps, zeros it, and announces it did so—and it works as expected in all processor and paging modes.

```
...
PA physAddr = MM_GetPhysPage();
ASSERT(physAddr != NULL);
MM_MapPage(physAddr, physAddr, PTE_P | PTE_RW | PTE_US | PTE_A);
memset(PA_TO_PTR(physAddr), 0, PAGE_SIZE);
Log("zeroed page at %p\n", PA_TO_PTR(physAddr));
...
```

I hesitate to claim that every RTL routine achieves complete register size and mode agnosticism, but it is a very good percentage. If the library-provided types, accessors, and the like are used, tests tend to be easily moved to all modes with only a little extra effort.

6. Let's Get Real

While it is easy for me to say that writing tests in FrobOS is simple, perhaps the point is better amplified with a real example: a FrobOS test written in response to a bug. Let's dive into device testing. The purpose of the test is to make sure the VMware SVGA device is disabled and invisible when turned off in the virtual machine configuration file, and remains off across a suspend and resume operation.

As discussed earlier, there are three pieces:

1. The entry in **suite.def** that includes the configuration option that turns the SVGA device off, specifies which hardware version to use, and so on:

```
...
835729-svga-not-present: # Basic testing when SVGA device is removed
        all  (-passthru "svga.present=FALSE"
                -passthru "virtualhw.version=8"
                -bits 32 -cpus 1),
        svga
        device
```

2. The **frobostest.mk file**, which is the same as shown previously
3. The test source

```
/*****************************************************
 * Copyright 2012 VMware, Inc.  All rights reserved. -- VMware Confidential
 *
*****************************************************/

/*
 * main.c --
 *
 *      Test basic functionality with the svga device removed.
 */

#define ALLOW_FROBOS32
#include "frobos.h"
#include "vm_device_version.h"

TESTID(835729, "SVGA-not-present");
SKIP_DECL(SK_NATIVE);

/*
 *-----------------------------------------------------
 *
 * Frobos_Main --
 *
 *      This is the main entry point for the test.
 *
 * Results:
 *      None
 *
 *-----------------------------------------------------
 */
void
Frobos_Main(void)
{
        PCI_Device *pci;

        PCI_Init();

        Test_SetCase("Verify SVGA device not present");

        pci = PCI_Search(PCI_VENDOR_ID_VMWARE,
PCI_DEVICE_ID_VMWARE_SVGA2, NULL);
        EXPECT_TRUE(pci == NULL);

        pci = PCI_Search(PCI_VENDOR_ID_VMWARE, PCI_DEVICE_ID_
VMWARE_SVGA, NULL);
  EXPECT_TRUE(pci == NULL);

        pci = PCI_Search(PCI_VENDOR_ID_VMWARE, PCI_DEVICE_ID_
VMWARE_VGA, NULL);
        EXPECT_TRUE(pci == NULL);

        if (Test_IsDevelMonitor()) {
                Test_SetCase("Suspend resume with no SVGA device");
        Mon_SuspendResume();
        }
}
```

A couple of new features are used here. The (optional) **SKIP_DECL** makes the test refuse to run (SKIP) if it is booted directly on hardware. The PCI library is used to gain access to the device. The source calls **PCI_Init()** and searches for the various device IDs that have been used by our SVGA device, and tests to make sure it is not found. The special backdoor call to force a suspend/resume

sequence only is supported on a developer build, so we protect the call appropriately. This test reproduces the bug on an unfixed tree, and runs in less than 10 seconds.

The author of the test tells me this test took him 30 minutes to write, including a coffee break. A similar test, if possible at all, would take much longer in any other operating system. Plus, programmers can convert this test to run in compatibility mode and 64-bit mode with the addition of two lines of source and two lines in **suite.def**. These additional lines are seen at the start of the next and prior examples.

7. What about SMP?

By convention, the x86 architecture distinguishes the first CPU to boot from those CPUs that boot later. The first is called the Boot Strap Processor (BSP), and those that follow are called auxiliary processors (AP). Assuming you want to test CPU features in an SMP environment, FrobOS offers APIs to bring APs into the action. Additional CPUs configured in a virtual machine, or those present natively, are booted and then 'parked' looping on a shared variable.*

So what do we need to make an SMP FrobOS test? Following in our minimalist vein, here are the source and differences from the simplest possible test:

• The suite.def file adds an option to enable more CPUs:

```
...
smpexample:
        smp (-cpus 0)      # use all available CPUs
...
```

• The source file main.c is as follows:

```
#define ALLOW_FROBOS32
#define ALLOW_FROBOS48
#define ALLOW_FROBOS64
#include <frobos.h>

SKIP_DECL(SK_SMP_2)

TESTID(2, "Journal SMP example test");

static void
SayHello(void * unused)
{
        Log("Hello from CPU %u\n", SMP_MyCPUNum());
}

void
Frobos_Main(void)
{
        int i;

        SayHello(NULL);
        for (i = 0; i < SMPNumCPUs(); i++) {
                SMP_RemoteCPUExecute(i, SayHello, NULL);
        }

        SMP_WaitAllIdle();
}
```

I made the test able to run in all three basic modes, and introduced a couple of new features. This particular **SKIP_DECL(SK_SMP_2)** makes the test skip when there is only one CPU present. In addition, I used **SMP_RemoteCPUExecute()** to kick each AP into action, while **SMP_WaitAllIdle()**, as its name suggests, waits for all APs to become idle.

The test simply iterates through all available CPUs, asking each CPU to execute the **SayHello()** function. Each AP polls for work and executes the function as soon as it can, resulting in the following output:

```
...
TEST: 0002: Journal SMP example Test CHANGE: 1709956
Hello from CPU 0
Hello from CPU 6
Hello from CPU 5
Hello from CPU 4
Hello from CPU 2
Hello from CPU 3
Hello from CPU 7
Hello from CPU 1
PASS: Test 0002: Journal SMP example Test
Frobos: Powering off VM.
...
```

This was performed on an 8-way virtual machine. As one might hope, the code functions without modification in environments with any number of CPUs. Plus, the same code runs in 32-bit, compatibility, and 64-bit modes.

* Using a shared variable might seem like a strange choice, after all there are other mechanisms such as Inter-Processor Interrupts (IPI) that are specifically designed for one CPU to send messages to another CPU. Using an IPI requires both the sender and receiver to have their APIC enabled, and the receiver must be ready and able to receive interrupts. While this is certainly a valid set of test conditions, it is not reasonable to impose them as a limitation on all SMP tests.

The SMP environment provided is not overly complex. Once booted into a GCC compatible runtime environment, APs do nothing unless asked. Data is either SHARED or PRIVATE (default). The run-time library provides basic locks, simple CPU coordination routines, and reusable barriers. For almost all runtime functions, the AP can do whatever the BSP can do.

8. There's More

When **frobos-run** runs a test in a virtual machine, it has the means to set a number of important options in the configuration file, including the Virtual Hardware version. This allows (or denies, depending on your perspective) guest visibility for certain hardware features, including instruction set extensions, maximum memory, maximum number of CPUs, and so on. If needed, a particular FrobOS test can inspect a variable to determine the specific hardware value and thereby tailor error checking or feature analysis as needed.

9. Conclusion

FrobOS is a great tool for writing low-level x86 and hardware tests on the PC platform, both for virtual machines and native operation. For test authors, it offers a rich run-time environment with none of the blinkers and constraints so typical of COS running on this platform. For product testing, FrobOS offers an ever-growing catalog of directed tests, with extensive logging and failure reporting capabilities. For most test running purposes, **frobos-run** hides the differences between VMware ESX and VMware Workstation, allowing the test author to check the behavior of either environment.

By its nature, FrobOS is a great characterization tool. The simple run-time model and very low requirements mean almost any x86-compatible machine that can boot from a floppy disk, PXE, USB disk or CD-ROM can be tested. It is a particularly easy fit for developers running Linux that wish to make or run tests, FrobOS uses the normal tools (make, gcc, gas, and so on) to build the kernel and a large catalog of tests. It has a very capable executive script that hides most of the complexities of building and running the tests: individually, as specific collections, or as a whole.

We are extending the use of FrobOS in several areas, including:

- Building fuzzers, with three already built (CPU, USB device, and VGA FIFO)
- Dynamic assemblers, because sometimes you just cannot make up your mind
- RTPG, a CPU fuzzer
- UEFI, a new BIOS
- Coverage, examining who executes what and why

Now, it is your turn. In addition to the documentation and sources in the **bora/vmcore/frobos** directory, I encourage you to explore the following resources:

- https://wiki.eng.vmware.com/Frobos
- The output of the **frobos-run** –help command
- Existing test and **suite.def** examples

Join my team in helping better test and understand our hypervisor.

Acknowledgements:

Thanks to present and past authors of the tests, libraries, and tools that have become what we call FrobOS. The list is long, but includes the past and present Monitor team, FrobOS maintainers, and Monitor reliability: Rakesh Agarwal, Mark Alexander, Kelvin Fong, Paul Leisy, Ankur Pai, Vu Tran, and Ying Yu.

Many thanks to Alex Garthwaite and Rakesh Argarwal for their guidance in writing this article, the reviewers without whom there would be many more mistakes of all sorts, and Mark Sheldon for the real SVGA enablement test.

Storage DRS: Automated Management of Storage Devices In a Virtualized Datacenter

Sachin Manpathak
VMware, Inc.
smanpathak@vmware.com

Ajay Gulati
VMware, Inc.
agulati@vmware.com

Mustafa Uysal
VMware, Inc.
muysal@vmware.com

Abstract

Virtualized datacenters contain a wide variety of storage devices with different performance characteristics and feature sets. In addition, a single storage device is shared among different virtual machines (*VMs*) due to ease of VM mobility, better consolidation, higher utilization and to support other features such as VMware Fault Tolerance (*FT*) [19] and VMware High Availability (*HA*) [20], that rely on shared storage. According to some estimates, the cost of managing storage over its lifetime is much more expensive as compared to initial procurement costs. It is highly desirable to automate the provisioning and runtime management operations for storage devices in such environments.

In this paper, we present Storage DRS as our solution for doing automated storage management in a virtualized datacenter. Specifically, Storage DRS handles initial placement of virtual disks, runtime migration of disks among storage devices to balance space utilization and IO load in a unified manner, and respect business constraints while doing so. We also present how Storage DRS handles various advanced features from storage arrays and different virtual disk types. Many of these advanced features make the management more difficult by hiding details across different mapping layers in the storage stack. Finally, we present various best practices to use Storage DRS and some lessons learned from initial customer deployments and feedback.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques.
C.4 [**Performance of Systems**]: Measurement techniques.
C.4 [**Performance of Systems**]: Performance Attributes.
D.4.8 [**Operating Systems**]: Performance—*Modeling and Prediction*
D.4.8 [**Operating Systems**]: Performance—*Measurements*
D.4.8 [**Operating Systems**]: Performance—*Operational analysis*

General Terms

Algorithms, Management, Performance, Design, Experimentation.

Keywords

VM, Virtualization, Resource Management, Scheduling, Storage, Hosts, Load Balancing

1. Introduction

Virtualized infrastructures provide higher utilization of physical infrastructure (servers and storage), agile IT operations, thereby reducing both capital and operating expenses. Virtualization offers unprecedented control and extensibility over consumption of compute and storage resources, allowing both VMs and their associated virtual disks to be placed dynamically based on current load and migrated seamlessly around the physical infrastructure, when needed. Unfortunately, all this sharing and consolidation comes at the cost of extra complexity. A diverse set of workloads that are typically deployed on isolated physical silos of infrastructure now share a bunch of heterogeneous storage devices with a variety of capabilities and advanced features. Such environments are also dynamic—as new devices, hardware upgrades, and other configuration changes are rolled out to expand capacity or to replace aging infrastructure.

In large datacenters, the cost of storage management captures the lion's share of the overall management overhead. Studies indicate that over its lifetime, managing storage is four times more expensive than its initial procurement [9]. The annualized total cost of storage for virtualized systems is often three times more than server hardware and seven times more than networking-related assets [13]. Due to inherent complexity and stateful nature of storage devices, storage administrators make most provisioning and deployment decisions in an ad-hoc manner trying to balance the space utilization and IO performance. Administrators typically rely on rules of thumb, or risky and time-consuming trial-and-error placements to perform workload admission, resource balancing, and congestion management.

There are several desirable features that a storage management solution needs to provide, in order to help administrators with the automated management of storage devices:

- Initial placement: Find the right place for a new workload being provisioned while ensuring that all resources (storage space and IO) are utilized efficiently and in a balanced manner.
- Load balancing: monitor the storage devices continuously to detect if free space is getting low on a device or if IO load is imbalanced across devices. In such cases, the solution should take remediation actions or recommend resolution to the administrators.

- Constraint handling: Handle a myriad of hardware configuration details and enforce business constraints defined by the administrator, such as anti-affinity rules for high availability.
- Online data gathering: All of the above needs to be done in an online manner by collecting runtime stats and without requiring offline modeling of devices or workloads. The solution should automatically estimate available performance in a dynamic environment.

Toward automated storage management, *VMware introduced Storage Distributed Resource Scheduler (Storage DRS)*, the first practical, automated storage management solution that provides all of the functionality mentioned above. Storage DRS consists of three key components: 1) a continuously updated storage performance and usage model to estimate performance and capacity usage growth; 2) a decision engine that uses these models to place and migrate storage workloads; and 3) a congestion management system that automatically detects overload conditions and reacts by throttling storage workloads.

In this paper, we describe the design and implementation of Storage DRS as part of a commercial product in VMware's vSphere management software [17]. We start with some background on various storage concepts and common storage architectures that are used in VMware based deployments in Section 2. Then we present a deep-dive in to the algorithms for initial placement and load balancing that are used for generating storage recommendations while balancing multiple objectives such as space and IO utilization (Section 3). We provide various use case scenarios and how Storage DRS would handle them along with the description of these features.

In practice, arrays and virtual disks come with a wide variety of options that lead to different behavior in terms of their space and IO consumption. Handling of advanced array or virtual disk features and various business constraints is explained in Sections 4 and 5. Given any solution of such complexity as Storage DRS there are always some caveats and recommended ways to use them. We highlight some of the best practices in deploying storage devices for Storage DRS and some of the key configuration settings in Section 6.

Finally, we present several lessons learned from real world deployments and outline future work in order to evolve Storage DRS to handle the next generation of storage devices and workloads (Section 7).

We envision Storage DRS as the beginning of the journey to provide software-defined storage, where one management layer is able to handle a diverse set of underlying storage devices with different capabilities and match workloads to the desired devices, while doing runtime remediation.

2. Background

Storage arrays form the backbone for any enterprise storage infrastructure. These arrays are connected to servers using either fiber channel based SAN or Ethernet based LANs.

These arrays provide a set of data management features in addition to providing a regular device interface to do IOs.

In virtualized environments, shared access to the storage devices is desirable because a bunch of features such as live migration of VMs (*vMotion* [8]), VMware HA and VMware FT depend on the shared storage to function. Shared storage also helps in arbitrating locks and granting access of files to one of the servers only, when several servers may be contending. Some features such as *Storage IO Control (SIOC)* also use shared storage to communicate information between servers in order to control IO requests from different hosts [16].

2.1 Storage Devices

Two main interfaces are used to connect storage devices are with VMware ESX hypervisor: block-based interface and file-based interface. In the first case, the storage array exposes a storage device (also called as LUN) as a set of blocks on which one can do regular IO operations using SCSI commands. ESX hypervisor installs a clustered file system called VMFS [14] on that LUN. VMFS allows every ESX host to see the same file system and all changes, in a consistent manner. In the second case, a storage device is exported by as a mount point by NFS server. ESX hypervisor accesses the device using NFS protocol. Currently ESX supports and implements NFSv3 protocol.

In both cases, ESX creates a concept of a datastore and exposes that to the administrator as a management and provisioning entity. Typically, a datastore is backed by a single LUN or NFS mount point, but we also allow a VMFS file system to extend across two devices. Such configuration is not supported by some of the features and is uncommon in customer deployments. We use the term datastore to denote a LUN or NFS mount point in this paper.

At the storage array, the storage controllers pool a set of underlying physical devices to create a volume or a RAID-group on them. Different vendors use different terms but the overall concept of creating a pool of underlying physical resources is pervasive. This pool is governed typically by similar properties in terms of reliability, fault handling, and performance sharing across the underlying devices. On top of this volume or a RAID-group, one can create a LUN, which is striped across the underlying devices. Finally these LUNs are exposed as a block device or a mount point over NFS.

This virtualization of underlying devices is hidden from the hypervisor and the exact performance and device level characteristics are not known outside the array. Since there are multiple layers of mappings across the storage stack from virtual disks to all the way down to physical disks, it is often quite difficult to discern where exactly a given block is stored.

In order to manage these devices, a solution such as Storage DRS needs to infer the performance characteristics in an online manner.

Storage controllers leverage the mapping of LUN address space to physical disk pool in many different ways to provide space savings and performance enhancing functionality. One of the widely used features is thin provisioning, where a LUN with a fixed reported capacity is backed by a much smaller address space from among the physical drives. For example, a 2 TB datastore can be backed up by a total of 500 GB physical drive pool. Storage controllers only map the used (i.e., written) blocks in the datastore to the allocated space in the backing pool.

The capacity of the backing pool can be adjusted dynamically by adding more drives when the space demand on the datastore increases over time. The available free capacity in the backing pool is managed at the controller, and controllers provide dynamic events to the storage management fabric when certain threshold in the usage of the backing pool is reached. These thresholds can be set individually at storage controllers. The dynamic block allocation in thin provisioned datastores allow storage controllers to manage its physical drive pool in a much more flexible manner by shifting the available free capacity to where it is needed.

In addition, storage controllers also offer features like compression and de-duplication to compress and store the identical blocks only once. Common content, such as OS images, copies of read-only databases, and data that does not change frequently can be compressed transparently to generate space savings. A de-duplicated datastore can hold more logical data than its reported capacity due to the transparent removal of identical blocks.

Storage controllers also offer integration with hypervisors to offload common hypervisor functions for higher performance. VM cloning, for example, can be performed efficiently using copy-on-write techniques by storage controllers: a VM clone can continue to use the blocks of the base disk until it writes new data, and new blocks are dynamically allocated during write operations. Storage controller keeps track of references to each individual block so that the common blocks are kept around until the last VM using them is removed from the system. Clones created natively at the storage controllers are more efficiently stored as long as they remain under the same controller. Since the independent storage arrays can't share reference counts, all the data of a clone needs to be copied in case a native clone is moved from one controller to another.

Storage controllers also take advantage of the mapping from logical to physical blocks to offer dynamic performance optimizations by managing the usage of fast storage devices.

Nonvolatile storage technologies such as solid-state disks (SSDs), flash memory, and battery backed RAM can be used to transparently absorb a large fraction of the IO load to reduce latency and increase throughput. Controllers remap blocks dynamically across multiple performance tiers for this purpose. This form of persistent caching or tiering allows fast devices to be used efficiently depending on the changing workload characteristics.

2.2 Virtual Disks

The datastores available to the hypervisor are used to store virtual disks belonging to a VM and other configuration files (e.g., snapshots, log files, swap files, etc.). Hypervisor controls the access to physical storage by mapping IO commands issued by VMs to file read and write IOs on the underlying datastores. This extra mapping layer allows hypervisors to provide different kinds of virtual disks for increased flexibility.

In the simplest form, a virtual disk is represented as a file in the underlying datastore and all of its blocks are allocated. This is called a thick-provisioned disk. Since the hypervisor controls the block mapping, not all blocks in a virtual disk have to be allocated at once. For example, VMs can start using the virtual disks while the blocks are being allocated and initialized in the background. These are called lazy-zeroed disks and allow a large virtual disk to be immediately usable without waiting for all of its blocks to be allocated at the datastore.

VMFS also implements space saving features that enable virtual disk data to be stored only when there is actual data written to the disk. This is similar to thin provisioning of LUNs at the array level. Since no physical space is allocated before a VM writes to its virtual disk, the space usage with thin-provisioned disks starts small and increases over time as the new data is written to the virtual disk. One can overcommit the datastore space by provisioning substantially larger number of virtual disks on a datastore, as there is no need to allocate unwritten portions of each virtual disk. This is also known as space over commitment. Figure 1 depicts the virtual disk types. ESX also provides support for VM cloning where clones are created without duplicating the entire virtual disk of the base VM. Instead of a full replica, a delta disk is used to store the modified blocks by the clone VM. Any block that is not stored in the delta disk is retrieved from the base disk. These VMs are called linked clones as they share base disks for the data that is not modified in the clone. This is commonly used to create a single base OS disk storing the OS image and share that across many VMs. Each VM only keeps the unique data blocks as delta disk that contains instance specific customizations.

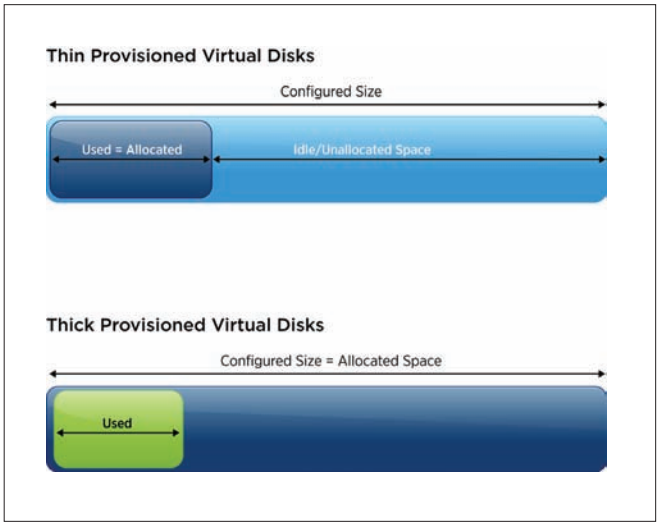

Figure 1. Virtual Disk Types

Finally, we use the key primitive of live storage migration (also called *Storage vMotion* [8]) provided by ESX that allows an administrator to migrate a virtual disk from one datastore to another without any downtime of the VM. We rely heavily on this primitive to do runtime IO and space management.

2.3 Storage Management Challenges

Despite the numerous benefits of virtualization in terms of extensible and dynamic allocation of storage resources, the complexity and large number of virtual disks and datastores calls for an automated solution. It is hard for an administrator to keep track of the mappings and sharing at various levels, in order to make simple decisions such as finding the best datastore for an incoming VM. The design may need to consider several metrics as explained below:

- **Space requirements**: One might think that using a datastore with the most available space is the best option. As we have described, determining the datastore with the most available space is harder in the presence of thin provisioning, de-duplication, linked clones, and unequal data growth from thin provisioned virtual disks. For example, it is often more space efficient to utilize an existing base disk for a linked clone than to create a full clone on another datastore - the former will use a fraction of the space as compared to the full clone.
- **Performance requirements**: Using a datastore with the most available performance headroom (IOPS or latency) seems to be a good option, but it is hard to determine the available headroom in the presence of rampant sharing of underlying physical resources behind many layers of mapping. Estimating the performance of a storage system and dynamically detecting the sharing among multiple datastores are hard problems that need to be solved to efficiently manage performance.
- **Multiple-dimensions**: It is quite possible that the best datastore for available space is not the same as the best datastore for available performance. Determining the relative goodness of a placement when there are multiple optimization criteria is needed in any solution.
- **Constraints**: Administrators can also provide constraints such as anti-affinity rules (e.g., keeping multiple VMs on different datastores), datastore compatibility (e.g., using a native datastore feature), connectivity constraints, or datastore preference (e.g., requiring a certain RAID level) that need to be taken into account for placement. In these cases, a provisioning operation might have to move other virtual disks to be successful.

Beyond just the initial placement, storage resources must be continuously monitored to detect changes in capacity consumption and IO loads. It is common practice to relocate workloads in the server infrastructure using vMotion or Storage vMotion to carry out various remediation actions.

This level of flexibility requires that the management layer be sufficiently intelligent to automatically detect resource imbalance, formulate actions that will fix the issue before it can develop into a service disruption, and carry out these actions in automated fashion.

3. Solution: Storage DRS

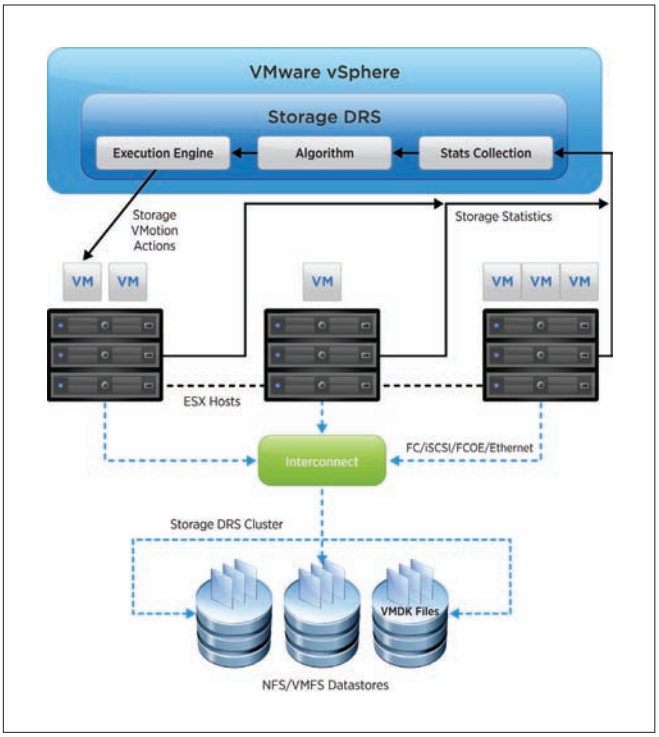

Figure 2. Storage DRS as Part of VMware vSphere

Figure 2 describes vSphere architecture with Storage DRS. As illustrated in the figure, Storage DRS runs as part of vCenter Server Management Software [18]. Storage DRS provides a new abstraction to the administrator, which is called datastore cluster. This allows administrators to put together a set of datastores into a single cluster and use that as the unit for several management operations. Within each datastore cluster, VM storage is managed in the form of Virtual Disks (*VMDKs*). The configuration files associated with a VM are also pooled together as a virtual disk object with space requirements only. When a datastore cluster is created, the user can specify three key configuration parameters:

- **Space Threshold**: This threshold is specified in percentage and Storage DRS tries to keep the space utilization below this value for all datastores in a datastore cluster. By default this is set to 80%.
- **Space-Difference Threshold**: This is an advanced setting that is used when all datastores have space utilization higher than the space threshold. In that case a migration is recommended from a source to a destination datastore only if the difference in space utilization is higher than this space-difference value. By default, this is set to 5%.
- **IO Latency threshold**: A datastore is considered overloaded in terms of IO load only if its latency is higher than this threshold value. If all datastores have latency smaller than this value, no IO load balancing moves are recommended. This is set to 10 ms by default. We compute 90th percentile stats in terms of datastore IO latency to compare with this threshold.

- **Automation level**: Storage DRS can be configured to operate in fully automated or manual modes. In fully automated mode, it will not only make recommendations but execute them without any administrator intervention. In manual mode, administrator intervention is needed to approve the recommendations.

All these parameters are configurable at any time. Based on these parameters, the following key operations are supported on the datastore cluster:

1. Initial placement API for VMDKs of a VM can be called on a datastore cluster instead of a specific datastore. Storage DRS implements that API and provides a ranked list of possible candidate datastores based on space and IO stats.
2. Out-of-space situations are avoided in the datastore cluster by monitoring the space usage and comparing that to the user-set threshold. If there is any risk that a datastore may cross the space utilization threshold, Storage DRS recommends Storage vMotion to handle that case.
3. Monitor and balance IO load across datastore, if the 90th percentile latency is higher than the user-set threshold for any datastore, Storage DRS tries to migrate some VMDKs out of that datastore to lightly loaded ones.
4. Users can put a datastore in the maintenance mode. Storage DRS tries to evacuate all VMs off of the datastore to suitable destinations within the cluster. The user can then perform maintenance operations on the datastore without impacting any running VMs.

All resource management operations performed by Storage DRS respect user set constraints (or rules). Storage DRS allows specification of anti-affinity and affinity rules between VMDKs in order to fulfill business, performance and reliability considerations.

As depicted in Figure 2, Storage DRS architecture consists of three major components - (1) Stats Collection: Storage DRS collects space and IO statistics for datastores as well as VMs. VM level stats are collected from the corresponding ESX host running the VM and datastore level stats are collected from SIOC, which is a feature that runs on each ESX host and computes the aggregated datastore level stats across all hosts. For example, SIOC can provide per datastore IOPS and latency measures across all hosts accessing the datastore. (2) Algorithm: This core component does computation of resource entitlements and recommends feasible moves. Multiple choices could be possible, and recommendations carry a rating indicating their relative importance/benefit. (3) Execution Engine: This component monitors and executes the Storage vMotion recommendations generated by the algorithm.

In this paper, we focus on handling of space, user constraints and virtual disk features such as linked clones. To handle IO, Storage DRS performs online performance modeling of storage devices. In addition, it continuously monitors IO stats of datastores as well as VMDKs. Details of IO modeling and corresponding algorithms are described in earlier papers [4][5]. We do not discuss those topics here.

In the remaining section, we first describe automated initial placement of VMs in a Storage DRS cluster. Then we go into details of load balancing, which form the core of storage resource management followed by the discussion on constraint handling.

3.1 Initial Placement

Initial placement of VMs on datastores is one of the most commonly used features of Storage DRS. Manually selecting appropriate storage for a VM often leads to problems such as poor IO performance and out of space scenarios. Storage DRS automatically selects most fitting datastore to place a VM based on space growth modeling and IO load history of the datastores. During initial placement, Storage DRS does not have any history of VMs IO profile or its future space demand. So, it uses conservative estimates for space and IO: space is assumed to be full disk size for thick provisioned disks and a fixed small size for thin provisioned ones. In terms of IO load we use the average load from other existing VMDKs on the datastore.

In addition, Storage DRS gives preference to datastores connected to as many hosts as possible. This allows solutions like Distributed Resource Scheduler (DRS) [1] to do load balancing of VMs across hosts more effectively. Storage DRS supports all virtual disk formats for placement—thin, thick eager zeroed as well as thick lazy zeroed. Thin disks in particular, start off with a small size; but can consume all the space up to their configured size over time. This poses a risk of running out of space on a datastore. Such placements are done with future space growth considerations.

Many aspects of Storage DRS such as constraints, prerequisite moves, pending recommendations are common to initial placement and load balancing. We discuss them in detail when we describe load balancing. Figure 3 below describes the initial placement used by Storage DRS.

Data: A set of datastores D, on which VM *v* can be placed.
Result: A set of recommendations R
For *datastore d ∈ D*; **Do**
  If *d has capacity for v and affinity rules allow the placement*, then
    Prop = [place v on d];
    Goodness(prop) = Imbalance(before) – Imbalance(after) placement;
    Make sure prop does not conflict with pending recommendations
  Add Prop to R
Sort R in descending order of goodness

Figure 3. Initial Placement Algorithm

Not all available datastores can be used for initial placement. (1) A datastore may not have sufficient resources to admit the VM (2) Inter-VM anti-affinity rules may be violated due to particular placement or (3) VM may not be compatible with some datastores

in the datastore cluster. Storage DRS applies these filters before evaluating VM placement on a datastore. For each datastore that passes the filters, the placement is checked for conflicts with pending recommendations. Then Storage DRS computes datastore cluster imbalance as if the placement was done on that datastore. Goodness is measured in terms of change in imbalance as a result of the placement operation. After evaluating VM placements, they are sorted based on their goodness values. Datastore with highest goodness value is overall the best datastore available for VM placement.

3.2 Load Balancing

Load balancing ensures that datastores in a datastore cluster do not exceed their configured thresholds as space consumption and IO loads change during runtime. Unlike DRS, which minimizes the resource usage deviation across hosts in a cluster, Storage DRS is driven by threshold trigger. Its load balancing mechanism moves VMs out of only those datastores, which exceed their configured threshold values. Figure 4 gives the outline of load balancing as used by Storage DRS. For each pass of Storage DRS, the algorithm is invoked first for datastores that exceeded their space threshold and later for those violating IO threshold, effectively fixing space violations followed by IO violations in the datastore cluster.

```
Data: Inventory snapshot S representing datastores,
      VMs in Storage DRS cluster
Data: Set of datastores D from S which exceed given threshold
Result: A set of recommendations R
R ← {}"
while D ≠ empty do
    Sort D by descending order of excess usage;
    bestMove ← None;
    for datastore d ∈ D do
        V ← Set of VMs running on d;
        for VM v in Set V do
            move ← [Move v from d to d1 where d1 ∈ D and d1 ≠ d];
            if CostBenefit(move) < 0 then
                continue
            if Goodness(Move) > Goodness(bestMove)
            then
                bestMove ←move;
    if bestMove = None then
        break;
    R ←RU bestMove;
    Update S;
    D ← Set of datastores from S which exceed threshold;
```

Figure 4. Storage DRS Load Balancing

Storage DRS is able to factor in multiple resources (space, IO and connected compute capacity) for each datastore when generating a move proposition. It uses weighted resource utilization vector to compare utilizations. Resources that are closer to their peak utilization get higher weights compared to others. Following example best illustrates the effectiveness of Storage DRS in multi-resource management.

Consider a simple setup of two (DS1 and DS2) datastores with 50GB space each and same IO performance characteristics. Both datastores are part of same Storage DRS cluster.

Two types of VMs are to be placed on these datastores. (1)

High IO VMs, which run a workload of 5-15 outstanding IOs for the duration of the test. (2) Low IO VMs with IO workload of 1-3 outstanding IOs for the duration of test.

The VMs have pre-allocated thick disk of 5GB each. During the experiment, high IO VMs are placed on DS1 and low IO VMs on DS2. The initial setup is as described in table 1 below:

DATASTORE	VM TYPE	NUMBER OF VMs	SPACE USED	LATENCY
DS1	High IO	4	20GB	28ms
DS2	Low IO	5	25GB	<10ms

Table 1: Initial Placement—Space and IO

The latency numbers are computed using a device model, which dictates the performance of the device as a function of workload. Storage DRS is invoked to recommend placement for a **new** High IO VM, making the total number of VMs to ten. Placement on DS1 had rating of 0, while placement on DS2 received rating of 3, even though DS1 has more free space than DS2. This is because DS1 is closer to exhausting its IO capacity. By placing the VM on DS2, Storage DRS ensures that IO bottleneck can be avoided. Furthermore, another Storage DRS pass over the datastore cluster recommended moving out one of the High IO VM from DS1 to DS2. The balanced cluster, at the end of evaluation was as in table 2 below:

DATASTORE	VM TYPE	NUMBER OF VMs	SPACE USED	LATENCY
DS1	High IO	3	15GB	<10ms
DS2	Low IO High IO	5 2	35GB	<10ms

Table 2: Final Datastore Cluster State

Note that the space threshold for datastores is set to 80% of their capacity (40GB). So from space perspective, the final configuration is still balanced.

Storage DRS is equally effective in balancing multiple resources while load balancing VMs in a cluster. Consider the two datastores (DS1, DS2) setup as before. This example uses 9 VMs with high IO workload with 1.2GB disk and 8 VMs with low IO workload and 5GB disk. All VMs were segregated on datastores based on their workload and space profiles, so the initial setup looks as described in Table 3.

DATASTORE	VM TYPE	NUMBER OF VMs	SPACE USED	LATENCY
DS1	High IO, Low Space	9	10.8GB	40ms
DS2	Low IO, High Space	8	40.0GB	<10ms

Table 3: Imbalanced Initial Configuration

The space threshold was configured at 75% datastore capacity (37.5GB). As is evident from the description, the setup is imbalanced from both space and IO perspective. Storage DRS load balancing was run multiple times on this cluster, during which 5 moves were proposed. The final configuration looked as in Table 4.

DATASTORE	VM TYPE	NUMBER OF VMs	SPACE USED	LATENCY
DS1	High IO, Low Space	6	17.2GB	31ms
	Low IO, High Space	2		
DS2	High IO, Low Space	3	33.6GB	19ms
	Low IO, High Space	6		

Table 4: Balanced Final Configuration

Note that final configuration does not perfectly balance space as well as IO, but performs only the moves sufficient to bring the space consumption below the configured threshold and balance IO load more evenly.

3.2.1 Filters

Load balancing is done in two phases. First, moves balancing space usage are generated followed by IO balancing. Space moves are evaluated for future space growth. A move which violates space or IO threshold or that may cause destination datastore to run out of space in future is rejected. Similarly, IO load balancing move of a VM to datastore with higher latency than source is filtered out.

3.2.2 Cost Benefit Analysis

Although a move is useful in balancing resource usage of datastore cluster, Storage vMotion is lengthy and costly operation. After a VM is moved to its destination, it should make cluster resource usage fair for a long time. Otherwise, changes in VM workload/growth rate can cause ping-pong moves. So, in addition to goodness, each move is evaluated per its cost and resulting benefit. For Storage, the benefit is computed as net reduction in normalized resource usage for space and IO on source with respect to increase on destination. The cost is computed in terms of total storage transferred as part of Storage vMotion and the number of IOs, which experience increased latency for that period. Linked clones indirectly affect cost benefit analysis. A move to a datastore, which has larger portion of clone disk chain is preferred, because it results in greater space savings as well as less costly Storage vMotion.

3.3 Pending Recommendations

Provisioning of new VMs and Storage vMotion of existing VMs can take several minutes to complete in a typical case. New invocation of Storage DRS under such transient conditions may perceive available free space and IO incorrectly and generate recommendations, based on stale information. Execution Engine tracks lifecycle of recommendations and corresponding actions. Prior to load balancing run, an accurate snapshot is generated in order to produce valid recommendations. This ability is important for group deployments such as vApp and test/dev environments. Similarly, during the algorithm run Storage DRS maintains a snapshot of resource state of datastores and VMs in the datastore cluster. After generating a valid recommendation, its impact is applied to the snapshot and resource values are updated before searching for next load balancing move. That way, each subsequent recommendation does not conflict with prior recommendations.

4. Constraint Handling

Storage capacity and IO performance are the primary resources Storage DRS considers in its initial placement and load balancing decisions. In this section, we describe several additional constraints that are taken into account when a virtual disk is placed on a datastore. There are two types of constraints considered by Storage DRS:

- **Platform constraints**: Some features that are required by a virtual disk may not be available in certain storage hardware or operations may be restricted due to firmware revision, connectivity, or configuration requirements.
- **User specified constraints**: Storage DRS provides a rule engine that allows users to specify affinity rules that restrict VM placement. In addition, Storage DRS behavior is controlled by a number of configuration options that can be used to relax restrictions or specify user preferences on a variety of placement choices. For example, the degree of space overcommit on a datastore with thin provisioned virtual disks can be adjusted dynamically using a configuration setting.

Storage DRS considers all constraints together when it is evaluating virtual disk placement decisions. Constraint enforcement can be strict or relaxed for a subset of constraints.

In strict enforcement, no placement or load balancing decision that violates a constraint is possible, even though they may be desirable from performance point of view. In contrast, relaxed constraint enforcement is a two-step process. In the first step, Storage DRS attempts to satisfy all constraints using strict enforcement. If this step is successful, no further action is necessary. In the second step, relaxed constraints can be violated when Storage DRS considers placement decisions. The set of constraints that can be relaxed and under what conditions is controlled by user configuration settings.

4.1 Platform Constraints

In this section, we describe platform constraints enforced by Storage DRS. The first group of platform constraints are defined as part of VMware APIs for Storage Awareness [15] (VASA) that enable storage devices to communicate their configuration constraints to Storage DRS. As we have described earlier, storage controllers determine the amount of actual capacity available for thin provisioned datastores. When the available backing space for a thin provisioned datastore runs low, storage controllers send events to the vCenter management server using the VASA API so that Storage DRS can adjust its placement decisions accordingly.

In these cases, even though the actual available space is running low, the datastore may appear to have a much larger free space. In response, Storage DRS does not place new virtual disks or move

other virtual disks to datastores that are identified as running low on capacity. These restrictions are lifted when the storage administrator provisions new physical storage to back a thin provisioned datastore. This is an example of a dynamic constraint declared entirely by the storage controllers.

VASA APIs are also used to control whether Storage DRS can move virtual disks from one datastore to another in response to imbalanced performance. Since relocating a virtual disk to a different datastore also shifts the IO workload to that datastore, a performance improvement is possible only when independent physical resources back two datastores. In a complex storage array, two different datastores may be sharing a controller or physical disks along the IO path. As a result, their performance may be coupled together. Storage DRS considers these relationships and attempts to pick other, non-correlated datastores to correct performance problems.

If virtual disks are created using storage-specific APIs (native cloning or native snapshots), Storage DRS restricts the relocation of such virtual disks, as native API must be used for their movement. Alternatively, if new provisioning operations require storage-specific APIs to be used, Storage DRS constrains the candidate pool of datastores that has the necessary capabilities. For example, thin-provisioned virtual disks are only available in recent versions of VMFS. In these cases, the capabilities are treated as strict constraints on actions generated by Storage DRS.

Platform specific constraints can also be specified through VMware Storage Policy Based Management (*SPBM*) framework, by using storage profiles. Storage profiles associate a set of capabilities generated by users, system administrators, or infrastructure providers. For example, a high performance, highly available datastores can be tagged as "*Gold storage*", whereas other datastores with lower RAID protection can be tagged as "*Silver storage*". Storage DRS clusters consist of datastores with identical storage profiles. That way, load balancing and initial placement operations satisfy SPBM constraints. In the future, storage clusters can be more flexible and allow for datastores with different storage profiles as we discuss later.

Storage DRS is compatible with VMware HA. HA adds a constraint that the VM's configuration files can only be moved to a datastore visible to the host running HA master. Storage DRS checks for such conditions before it proposes moves for HA protected VMs.

4.2 User-specified Constraints

Storage DRS provides a rich rule enforcement engine that allows users to specify affinity rules that restricts VM placement. Storage DRS supports the following rules:

- **Virtual Machine Anti-Affinity**: If one or more VMs are part of an anti-affinity rule, Storage DRS ensures that they are placed on different datastores. This is useful to identify a set of VMs that should not fail together so that services supported by those VMs are always available. In enforcing anti-affinity rules, Storage DRS also considers physical resource sharing as reported through VASA APIs so that anti-affine VMs are not placed on datastores where storage controllers identified as sharing resources.
- **Virtual Disk Anti-Affinity**: If a single VM has more than one virtual disk, Storage DRS supports placing them on different datastores. Anti-affinity is useful for managing storage of I/O intensive applications such as databases. For example, log disks and data disks can be automatically placed at different datastores, enabling better performance and availability.
- **Virtual Disk Affinity**: Using this rule, all virtual disks of a virtual machine can be kept together on the same datastore. This is useful for majority of the small servers and user VMs as it is easier for administrators when all the files comprising a VM are kept together. Furthermore, this simplifies VM restart after a failure.

Storage DRS also supports relaxation of affinity-rule enforcement during rare maintenance events, such as when VMs are evacuated from a datastore. Since affinity-rule enforcement might constrain the allocation of available resources, it is useful to temporarily relax these constraints when available resources will be reduced intentionally for maintenance.

Storage DRS respects constraints for all of its regular operations such as initial placement and load balancing. User specified constraints could be added or removed at any given time. If there is any constraint violation after the rule set is modified, Storage DRS immediately generates actions to fix the rule violations as a priority. User-specified constraints can also be added using certain advanced configuration settings. For example, the degree of space over provisioning due to thin provisioning can be explicitly controlled by instructing Storage DRS to keep some reserve space for virtual disk growth. This is done by using a certain fraction of unallocated space for thin disks as consumed. This fraction is controlled by an advanced option.

5. Estimating Space Usage

IIn this section, we describe how Storage DRS estimates the space requirements of virtual disks for scenarios where space consumption is highly dynamic. Thin provisioned virtual disk grows over time as the new data is written for the first time, up to the provisioned capacity of the virtual disk. The rate of growth of a virtual disk is variable depending on the applications running inside the VM. In addition, frequent creation and deletion of virtual machine snapshots also contribute to the variable growth rates. This is because there is no separate placement step for snapshot creation; VM's current datastore is used for the newly created snapshots. Finally, linked clones themselves use delta disks that contain only the different content from the base disk using which the clone is created. Since the delta disk grows over time similar to a thin provisioned disk, different datastores experience varying space usage growth rates. It is important to keep track of datastore space usage automatically and take actions with the unequal growth rates taken into account. This prevents a datastore from running out of space and causing a service disruption to the running VMs.

Storage DRS avoids out of space scenarios and extraneous moves by modeling space growth. It maintains a running average of datastore space usage over a period of time and predicts the space growth based on this running average. Using the model,

Storage DRS avoids placing VMs on datastores where space will be running out faster than other datastores for a fixed time in the future. Following example illustrates space modeling. The initial setup is as described in table 5 below:

DATASTORE	FREE SPACE	NUMBER OF VMs	TIME TO FULL SIZE
DS1	50GB	23	80Hrs
DS2	50GB	20	30Hrs

Table 5: Initial Placement with Growing Disk

Each of the VMs starts with 100 MB and eventually grows to 2 GB in size. This is analogous to typical lifecycle of a VM with thin provisioned virtual disks. VMs on DS1 grow slowly, and attain a size of 2 GB in 80 hours, while VMs on DS2 grow to 2 GB in 30 hours. After 80 hours, DS1 will have used 46 GB of space, while in 30 hours; DS2 will hit 40 GB space usage. Next time Storage DRS places a VM; it chooses DS1 even though it has less free space at the time of placement. Since growth rate of VMs on DS1 is slower, over time the space usage across these datastores will be balanced.

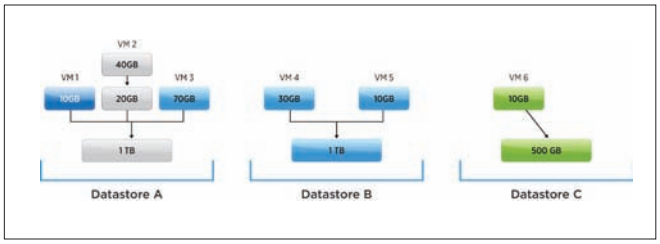

Figure 5. Space Usage with Linked Clones

Linked clones have a different type of complexity for space usage estimation: relocation of a linked clone will consume different amounts of space depending on the availability of base disks. Consider the setup as outlined in Figure 5. Datastore A and Datastore B have both an identical base disk that are used by VM1, VM2, VM3, VM4, and VM5. VM1 currently uses 10 GB in its delta disk plus the 1 TB base disk at Datastore A. Similarly, VM4 is using 30 GB of delta disk and the 1 TB base disk at Datastore B. If VM1 is to be relocated to Datastore B, only 10 GB of the delta disk will have to be moved, since VM1 can start using the identical base disk at Datastore B. However, if VM1 is to relocate to Datastore C, a copy of a 1 TB base disk has to be made as well, and as a result, both the base disk and the delta disk have to be moved. Note that relocating VM1 to Datastore B not only results in a smaller total space being used but also will complete faster since a much smaller amount of data needs to be moved.

The base disk of linked clones is retained so long as there is at least one clone using the base disk. For example, unless both of Vm4 and Vm5 are relocated to a different datastore, the 1 TB base disk will continue to occupy space at Datastore B. Moving either of Vm4 or Vm5 alone will result in space savings of only 30 GB and 10 GB respectively, leaving the 1 TB base disk intact.

6. Best Practices

Given the complexity and diversity of storage devices available today, it is often hard to design solutions that work for a large variety of devices and configurations. Although we have tried to set the default knobs based on our experimentation with a couple of storage arrays, it is not possible to procure and test many of the devices out there. In order to help system administrators to get the best out of Storage DRS, we suggest the following practices in real deployments:

(1) Use devices with similar data management features: Storage devices have two types of properties: data management related and performance related. The data management related properties include RAID level, back up capabilities, disaster recovery capabilities, de-duplication etc. We expect all datastores in a storage DRS cluster to have similar such data management properties so that the virtual disks can be migrated among them without violating any business rules. Storage DRS handles the performance variation among devices but assumes that virtual disks are compatible with all devices based on the data management features. This is something that we plan to relax in future, by using storage profiles and placing or migrating virtual disks only on the datastores with compatible profile.

(2) Use full or similar connectivity to hosts: We suggest keeping full connectivity among datastores to hosts. Consider a datastore DS1 that is only connected to one host as an extreme case. The VM whose virtual disk is placed on DS1 can not be migrated to other hosts in case that host has high CPU and memory utilization. In order to migrate the virtual machine, we will also have to migrate the disks from that datastore. Basically, poor connectivity constraints the movement of VMs needed for CPU and memory balancing to a few set of hosts.

(3) Correlated datastores: Different datastores exposed via a single storage array may share the same set of underlying physical disks and other resources. For instance, in case of EMC ClaRiiON array, one can create RAID groups using a set of disks, with a certain RAID level and carve out multiple LUNs from a single RAID group. These LUNs are essentially sharing the same set of underlying disks for RAID and it doesn't make sense to move a VMDK from one to another for IO load balancing. In storage DRS, we try to find such correlation and also allow storage arrays to tell us about such performance correlation using VASA APIs. In future, we are also considering exposing an API for admins to provide this information directly, if the array doesn't support VASA API to provide correlation information.

(4) Ignoring stats during management operations: Storage DRS collects IO stats for datastores and virtual disks continuously and computes online percentiles during a day. These stats are reset once a day and a seven-day history is kept although only one-day stats are used right now. In many cases, there are nightly background tasks such as back-up and virus scanners that lead to a very different stats profile during the night as compared to actual workday. This can also happen for tasks with a different periodicity, such as a full backup on a weekend. These tasks can distort the view of load on a datastore for Storage DRS. We have provided API support to declare such time periods during which the stats should not be collected and integrated in to the daily profile. We suggest storage administrators

to use this API, to avoid any spurious recommendations. Furthermore, it is a good idea to ignore recommendations after a day of the heavy management operation such as RAID rebuild or something, unless it is actually desirable to move out of the datastore that went through the rebuild process and provided high latencies, to protect against future faults.

(5) Use the affinity and anti-affinity rules sparingly: Using too many rules can constrain the overall placement and load-balancing moves. By default we keep all the VMDKs of a VM together. This is done to keep the same failure domain for all VMDKs of a VM. If this is not critical, consider changing this default, so that storage DRS can move individual disks if needed. In some cases using rules is a good idea, since only the user is aware of the actual purpose of the VMDK for an application. One can use VMDK-to-VMDK anti-affinity rules to isolate data and log disks for a database on two different datastore. This will not only improve performance by isolating a random IO stream from a sequential write stream, but also provide better fault isolation. To get high availability for a multi-tier application, different VMs running the same tier can be placed on separate datastores using VM-to-VM anti-affinity rules. We expect customers to use these rules only when needed and keeping in mind that in some cases, the cluster may look less balanced due to the limitations placed by such rules on our load balancing operation.

(6) Use multiple datastore instead of using extents: Extents allow the admin to extend a single datastore to multiple LUNs. These are typically used to avoid creating a separate datastore for management. Extent based datastores are hard to model and reason about. For instance, the performance of that datastore is a function of two separate backing LUNs. IO stats are also a combination of the backing LUNs and are hard to use in a meaningful way. Features like SIOC are also not supported on datastores with multiple extents. With Storage DRS the management problem with multiple datastores is already handled. So we suggest storage administrators to use separate datastores per LUN and use storage DRS to manage them, instead of using extents to increase the capacity of a single datastore.

(7) Storage DRS and SIOC threshold IO latencies: Users specify threshold IO latency of datastore while enabling Storage DRS on a datastore cluster. Storage DRS uses threshold latency value as a trigger. If datastore latency exceeds this threshold, VMs are moved out of such datastore(s) in order to restore latency value below threshold.

When Storage DRS is enabled on a datastore cluster, SIOC is also enabled on individual datastores. SIOC operation is controlled by its own threshold latency value. By default SIOC threshold latency is higher than that of Storage DRS. SIOC operates at much higher time frequency than Storage DRS. Without SIOC, there could be situations where IO workloads kick in for short durations and IO latency can shoot up for those time intervals. Until Storage DRS can remedy the situation, SIOC acts as a guard and keeps IO latency in check. It also ensures that other VMs on that datastore do not suffer during such intervals and get their proportional IO share. It is important to make sure that SIOC threshold latency is higher than that of Storage DRS. Otherwise, datastore latency will always appear lower than the threshold value to Storage DRS and it will not generate moves to balance IO load in the datastore cluster.

7. Discussion and Future Work

7.1 Storage DRS in the Field

VMware DRS technology influenced Storage DRS to a large extent. They both use similar concepts such as cluster, load balancing domain, recommendations, rules, and faults.

Over time, as Storage DRS deployments increased in the field, a few key differences have emerged.

Unlike DRS, it is critical for Storage DRS to choose the right storage for virtual disks. While placing a VM, users want simplicity and policy driven placement. As discussed previously, we plan to expand Storage DRS to make VM placements work seamlessly across all storage in a vSphere environment. Another key difference is the scale. Ideally, Storage DRS users want aggregation of all storage in a single pool, and let Storage DRS automatically manage VM placements, IO performance, space usage, and policy compliance over the entire pool. In this regard, we are working towards improvements in scale and performance.

Storage architectures have evolved significantly since the initial design of Storage DRS. As storage controllers become more intelligent, the existing black box performance models [5] are less capable of covering all salient aspects of device performance. We are exploring new interfaces for performance modeling so that storage devices can compactly report their performance capabilities that can be used by Storage DRS. In addition; we are exploring combinations of active [5] and passive [4] performance modeling approaches.

Storage DRS best practices recommend datastores with similar data management features and even similar disk types in a datastore cluster. Many customers want the capability to include different storage types in terms of capacity, performance, protocols, protection level, etc. in a single pool. These not only require improvements in automation, but also more fine grain controls when managing datastores in a cluster.

7.2 Future Directions

Storage technologies and storage system designs are evolving at a very rapid pace. With the introduction of multi-core CPUs, high-bandwidth interconnects and solid-state disks, many new storage architectures are coming to market. Some of the common new designs include: multi-tiered storage, scale-out storage and compute-storage converged architectures.

In case of multi-tiered storage, SSDs are being used as a primary storage media either as a first tier or as a caching layer. Both designs make it harder to model the datastore from outside as a black box. This is because an outside observer cannot know the hit rate of IOs in the SSD tier.

The scale-out storage paradigm is very useful for cloud service providers. They can buy a unit of storage and scale out as demand grows. In this case, the servers may see a single connection point but the IOs can get served from one of many backend storage controllers via internal routing. Some examples of this architecture include EMC Isilon [3], NetApp ONTAP GSX [2], IBM SoNAS [7] etc. These architectures make it hard to determine the amount of available performance left on the storage device.

The server-storage converged architectures such as Nutanix [11], Simplivity [12], HP Lefthand [6] etc. take the scale-out to the next level by coupling together local storage across servers to form a shared datastore. These solutions provide high-speed local access for most of the IOs and do remote IOs only when needed or for replication. In all these cases, it is quite challenging to get a sense of performance available in a datastore. We think a good way to manage these emerging storage devices is by creating a common API that brings together the virtual device placement and management with that of the internal intelligence of the storage devices. We are working towards such a common language and incorporating it as part of VASA APIs.

So far we have talked about modeling IO performance, but similar problems arise for space management as well. Thin provisioning, compression and de-duplication for primary storage are becoming common features for space efficiency in arrays. This makes it harder to estimate the amount of space that will get allocated to store a virtual disk on a datastore. Currently Storage DRS uses the actual provisioned capacity as reported by the datastore. In reality the space consumed may be different: this has the effect of space usage estimations being slightly inaccurate. We are working on modifying storage DRS to handle such cases and also planning to add additional APIs for better reporting of space allocations.

Overall, building a single storage management solution that can do policy based provisioning across all types of storage devices and perform runtime remediation when things go out of compliance is the main goal for Storage DRS.

8. Conclusion

In this paper, we presented the design and implementation of a storage management feature called Storage DRS from VMware. The goal of Storage DRS is to make several management tasks such as initial placement of virtual disks, out of space avoidance and runtime load balancing of space consumption and IO load on datastores. Storage DRS also provides a simple interface to specify business constraints using affinity and anti-affinity rules, and it enforces them while making provisioning decisions. We also highlight some of the complex array and virtual disk features that make the accounting of resources more complex and how Storage DRS handles that. Based on our initial deployment in the field, we have gotten a very positive response and feedback from customers. We consider Storage DRS as the beginning of the journey to software managed storage in virtualized datacenters and we are planning to accommodate the newer storage architectures and storage disk types in future to make Storage DRS even more widely applicable.

References

1 Resource Management with VMware DRS, 2006, http://vmware.com/pdf/vmware_drs_wp.pdf

2 M. Eisler, P. Corbett, M. Kazar and D. S. Nydick. Data Ontap GX: A scalable storage cluster. In *5th USENIX Conference on File and Storage Technologies*, pages 139-152, 2007.

3 EMC, Inc. EMC Isilon OneFS File System. 2012, http://simple.isilon.com/doc-viewer/1449/emc-isilon-onefs-operating-system.pdf

4 A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated IO Load Balancing across Storage Devices. In *USENIX FAST, Feb 2010*.

5 A. Gulati, G. Shanmuganathan, I Ahmed, M. Uysal and C. Waltspurger. Pesto: Online Storage Performance Management in Virtualized Datacenters. In *Proc. Of the 2nd ACM Symposium on Cloud Computing*, Oct 2011.

6 Hewlett Packard, Inc. HP LeftHand P4000 Storage. 2012, http://www.hp.com/go/storage

7 IBM Systems, Inc. IBM Scale Out Network Attached Storage, 2012. http://www-03.ibm.com/systems/storage/network/sonas/index.html

8 A. Mashtizadeh, E. Celebi, T. Garnkel, and M. Cai. The Design and Evolution of Live Storage Migration in VMware ESX. In Proc. USENIX Annual Technical Conference (ATC'11), June 2011 (to appear).

9 D. R. Merrill. Storage Economics: Four Principles for Reducing Total Cost of Ownership. May 2009, http://www.hds.com/assets/pdf/four-principles-for-reducing-total-cost-of-ownership.pdf

10 M. Nelson, B. H. Lim, and G. Hutchins. Fast Transparent Migration of Virtual Machines. In Proc. USENIX, April 2005.

11 Nutanix, Inc. The SAN free datacenter. 2012, http://www.nutanix.com

12 Simplivity, Inc. The Simplivity Omnicube Global Federation. 2012, http://www.simplivity.com

13 N. Simpson. Building a data center cost model. Jan 2010, http://www.burtongroup.com/Research/DocumentList.aspx?cid=49

14 S. Vagnani. Virtual machine le system. In ACM SIGOPS Operating Systems Review 44.4, pages 57-70, 2010.

15 VMware, Inc. VMware APIs for Storage Awareness. 2010, http://www.vmware.com/technical-resources/virtualization-topics/virtual-storage/storage-apis.html

16 VMware, Inc. VMware vSphere Storage IO Control. 2010, http://www.vmware.com/files/pdf/techpaper/VMW-vSphere41-SIOC.pdf

17 VMware, Inc. VMware vSphere. 2011, http://www.vmware.com/products/vsphere/overview.html

18 VMware, Inc. VMware vCenter Server. 2012, http://www.vmware.com/products/vcenter-server/overview.html

19 VMware, Inc. VMware vSphere Fault Tolerance. 2012, http://www.vmware.com/products/datacenter-virtualization/vsphere/fault-tolerance.html

20 VMware, Inc. VMware vSphere High Availability. 2012, http://www.vmware.com/solutions/datacenter/business-continuity/high-availability.html

A Social Media Approach to Virtualization Management

Ravi Soundararajan ravi@vmware.com
Emre Celebi ecelebi@vmware.com
Lawrence Spracklen lspracklen@vmware.com
Harish Muppalla harishm@vmware.com
Vikram Makhija vmakhija@vmware.com

Performance Group and Product Management, VMware, Inc.

Abstract

The average virtualization administrator finds it difficult to manage hundreds of hosts and thousands of virtual machines. At the same time, almost anyone on earth with a smartphone is adept at using social media sites like Facebook for managing hundreds of friends. Social media succeeds because a) the interfaces are intuitive, b) the updates are configurable and relevant, and c) the user can choose arbitrary groupings for friends. Why not apply the same techniques to virtualized datacenter management?

In this paper, we propose combining the tenets of social media with the VMware® vSphere® management platform to provide an intuitive technique for virtualized datacenter management. An administrator joins a social network and "follows" a VMware vCenter™ server or another monitoring server to receive timely updates about the status of an infrastructure. The vCenter server runs a small social media client that allows it to "follow" hosts and their status updates. This client can use the messaging capabilities of social media (posting messages to lists, deleting messages from lists, sending replies in response to messages etc.) to apprise the administrator of useful events. Similarly, hosts contain a small client and can "follow" virtual machines and be organized into communities (clusters), and virtual machines can be organized into "communities" based on application type (for example, all virtual machines running Microsoft Exchange) or owner (for example, all virtual machines that belong to user XYZ). By creating a hierarchy from an administrator to the host to the virtual machine, and allowing each to post status updates to relevant communities, an administrator can easily stay informed about the status of a datacenter. By utilizing message capabilities, administrators can even send commands to hosts or virtual machines. Finally, by configuring the types of status that are sent and even the data source for status updates, and by using social media metaphors like hash tags and 'likes', an administrator can do first-level triaging of issues in a large virtualized environment.

Categories and Subject Descriptors

D.m [**Miscellaneous**]: virtual machines, system management, cloud computing.

General Terms

Performance, Management, Design.

Keywords

virtual machine management, cloud computing, datacenter management tools.

1. Introduction

One of the most challenging problems in virtualized deployments is keeping track of the basic health of the infrastructure. Operators would like to know quickly when problems occur and would also like to have guidance about how to solve issues when they arise. These problems are exacerbated at scale: it is already difficult to visualize problems when there are 100 hosts and 1000 virtual machines (virtual machines), but what about in setups with 1000 hosts and 10,000 virtual machines? Conventional means for monitoring these large environments focus on reducing the amount of data to manageable quantities. Reducing data is difficult, requiring two 2 distinct skill sets: first, knowledge of virtualization, in order to determine what issues are serious enough to be alerted and how to solve such issues; second, the ability to create intelligent visualizations that reduce data into manageable chunks.

Automated techniques for monitoring the health of an infrastructure [13] have become increasingly prevalent and helpful. Such approaches leverage the collection and analysis of a large number of metrics across an environment in order to provide a concise, simplified view of the status of the entire environment. However, despite the success of such tools, significant training is often required in order to obtain proficiency at understanding and using the output of such tools.

In this paper, we approach the problem of virtualization monitoring from a different perspective. We observe that while the average administrator might find it difficult to monitor 100 hosts and 1000 virtual machines, the same administrator might find it relatively easy to keep abreast of his hundreds of Facebook[1] friends. The reason for this is that social networking sites allow many knobs to limit the information flow to a given user. Moreover, these knobs are extremely intuitive to use. For example, users of Google+ can organize friends into circles and limit status updates to various circles, or might choose to propagate status updates only to select listeners. The knobs are also designed with a keen understanding of the problem domain: for example, in a social network, birthdays are important events, so social networks create special notifications based solely on birthdays.

We propose organizing a virtualized environment into a social network of its own, including not just humans (system administrators), but also non-human entities (hosts, virtual machines, and vCenter servers). Each member of the community is able to contribute

status updates, whether manually (in the case of humans) or programmatically (through automated scripts running on virtual machines and hosts). We organize this social network according to our understanding of the hierarchy in a virtualized environment, and we limit the information flow so that only the most important updates reach an administrator. Moreover, the administrator is capable of performing commands within the social media client. By combining the reduction of information with the ability to perform basic virtualization management operations in response to such information, and by wrapping these features into the familiar UI of a social media application, we create an intuitive, platform-independent method for basic monitoring and management of a virtualized environment.

The outline of this paper is follows. In section 2, we explain how we map the constructs of a social network to a virtual hierarchy. In section 3, we describe a prototype design for such a monitoring scheme, leveraging one set of social networking APIs, the Socialcast Developer APIs. In section 4, we describe our initial experiences deploying such a system in a real-world environment. In section 5, we discuss related work. We provide conclusions and future directions in section 6.

2. Comparing Virtual Inventories and Social Media Networks

Figure 1 depicts a sample social network. A two-way arrow suggests a friend relationship. For example, A is friends with B and B is friends with G, but G is not friends with A. In addition, B, F, and G might choose to create a separate, private group, as indicated by the dotted rectangle in the figure. There is a distinction between physical entities, namely the members of the groups like A through G, and the logical entities, like the group consisting of B, F, and G.

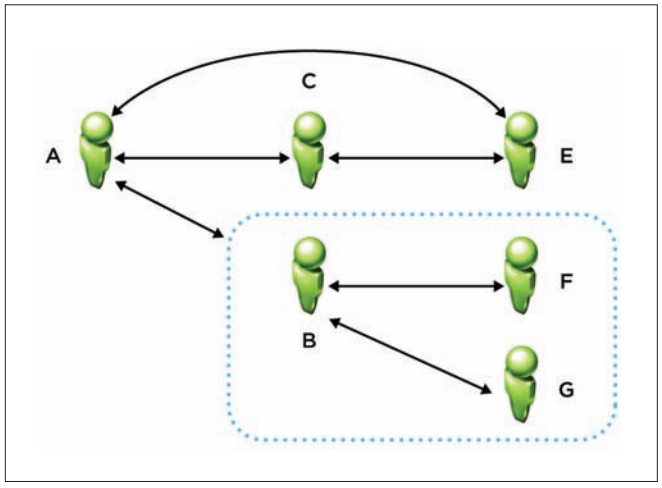

Figure 1: A sample social network, illustrating friend relationships, groups, and hierarchies.

Similar to a social network, a virtualized infrastructure also consists of 'physical' members and logical groups. For example, consider the sample VMware vSphere[15] inventory shown in Figure 2. As the figure indicates, vCenter server W1-PEVCLOUD-001 is composed of a datacenter named vCloud, which in turn consists of a cluster AppCloudCluster and a group of hosts. The cluster contains resource pools and virtual machines. This hierarchy can be mapped to a 'social' network of its own. An administrator can be a 'friend' of a vCenter server. A vCenter server can have hosts as friends, and hosts can have virtual machines as friends. Hierarchy is important because it is one method of limiting information flow. In a social network, a person like A might choose instead to only be friends with B, knowing that if anything interesting happens to F and G, that B will likely collect such information and share it with A. In a similar manner, the vCenter server need not choose to be friends with all virtual machines, but just with all hosts. If a host receives enough status updates from the virtual machines running on it, it might choose to signal a status change to vCenter. In a similar way, an administrator might choose to be friends only with vCenter, knowing that vCenter can accumulate status updates and propagate them to the administrator.

While hosts and VMs are entities in our virtualization social network, we currently do not add datacenters, clusters, and resource pools as entities. At present, this is because of a practical issue. Datacenters, clusters, and resource pools cannot be added as friends because they do not have a 'physical' manifestation. In other words, while an administrator can send and receive network packets to/from virtual machines and hosts, an administrator cannot send a message to a datacenter. Instead, datacenters, clusters, resource pools, and host/virtual machine folders are more similar to a 'group' in a social network. We extend the notion of a group to include not just virtualization hierarchy, but also allow arbitrary user-defined collections of entities. For example, it might be helpful to put all virtual machines that run a SpecJBB[9] in a group labeled SpecJBB, or it might be helpful to put all virtual machines under a given resource pool in a given group.

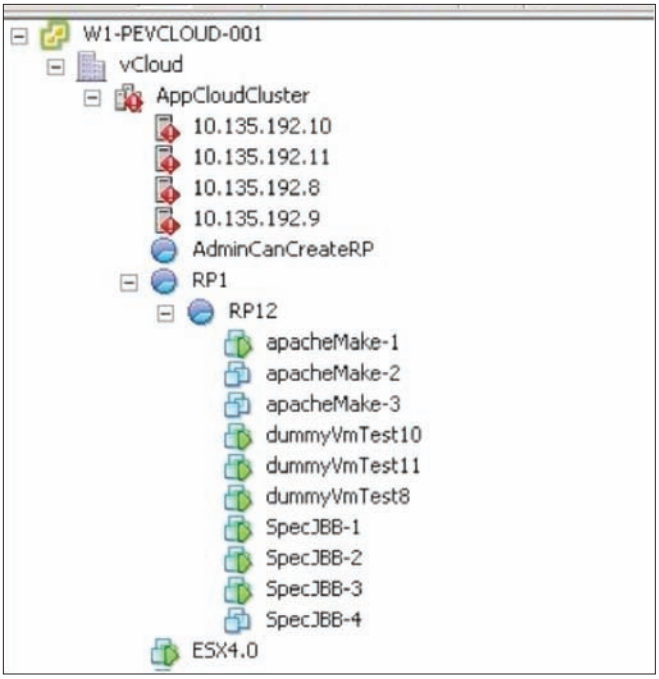

Figure 2: A Sample VC Inventory. Hosts, virtual machines and the vCenter server itself can be mapped to members of a social network, while datacenters, clusters, and resource pools are like groups.

3. Prototype Design

In this paper, we propose that virtual machines, hosts, and vCenter servers become nodes in a social network. Each one runs agents that publish various pieces of information to the social network. An administrator can then examine the web site (or her mobile device) in order to get interesting information about the virtualized infrastructure in a more intuitive manner than a standard management interface.

In order to validate this design approach, we discuss a proof-of-concept design based on using the Socialcast Developer API [7]. The verbs of social media messaging, as encapsulated in the Socialcast Developer API, map very well to the verbs required to build an efficient notification system for virtualized infrastructure. In the next two sections, we describe the Socialcast API and then indicate how it might be used to build a management infrastructure. Using a special on-premise Socialcast virtual appliance, we have been able to prototype and validate most of the proposals described in this section.

3.1 The Socialcast API

Before discussing our prototype design, we first give a brief description of social media messaging in Socialcast. There are several kinds of messages in Socialcast. There are *community streams*, in which a group of users can essentially subscribe to a given topic and see messages related to that topic. There are also *private messages*, which are messages directed to a particular user and not viewable by anyone else. There are *comments*, in which users can essentially respond to existing stream messages, and there are *private message replies*, which are similar to comments, but are responses to private messages. Messages and comments can be *liked* (in which other users express approval) or *un-liked* (in which a previously-posted 'like' is removed). Messages can be tagged with categories or filtered by content. Finally, users can be *followed*: if user A is followed by user B, then when user A makes comments, user B is notified of them. This allows user B to keep abreast of the events in A's life.

Based on this description of the message types in Socialcast, we can now take a brief look at the relevant portions of the API.

1. **Messages API**: The messages API allow users to read a single stream message or a group of stream messages, create new messages, update new messages, destroy messages, and search messages. A user can also specify the retrieval of messages since a certain date, the retrieval of messages that fit certain criteria, etc.
2. **Likes API**: The likes API allows a user to like a message or un-like a message.
3. **Comments API**: The comments API allows a user to retrieve comments, create comments, update comments, or delete comments. There is also a 'comment likes' API where a user can like or un-like a comment.
4. **Flagging API:** With flagging, a user tags a message that she has posted as being important to her, as a reminder to her to view it later.
5. **Private Messages API**: The private messages API allows users to perform all of the same actions as in the standard messages API, but for private messages.
6. **Users API**: The users API allows users to retrieve information about other users, search for users, deactivate users, and retrieve messages from a specific user.
7. **Follow/Unfollow API**: The follow API allows users to 'follow' other users (i.e., see comments or notifications by the other users).
8. **Groups API**: The groups API allows users to list groups, the members of groups, and group memberships of a given user.
9. **Attachments API**: The attachments API allows a user to create attachments (either separately or as part of a message).

These commands can be simple HTTP GET or POST requests. Simply by installing a library like libcurl[2] in virtual machines, we are able to have virtual machines programmatically send status and receive status. Adding this library to an ESX host further enables an ESX host to send/receive status. In essence, because of the ability to programmatically interact with messages, groups, etc., the hosts and virtual machines are able to be users in the virtualization social network in the same way that human beings are members of the virtualization social network.

3.2 Mapping the Socialcast API to virtualization monitoring

To understand how the Socialcast model fits in with virtualization monitoring, consider how virtualized environments are monitored. If important events happen, then notifications are sent to an administrator. These notifications are acknowledged and then cleared. Multiple similar issues might happen among a group of hosts or virtual machines, suggesting a common root cause. Messages can be flagged according to severity, and messages with common headers can be additional categorized.

Consider our canonical design in which an administrator follows the vCenter server. The vCenter server, in turn, follows hosts, as shown in Figure 3. The hosts follow virtual machines. Because hosts are often in clusters, it might be useful to organize a set of hosts into a group named after the parent cluster. Virtual machines might reside in folders or resource pools, so virtual machines might be placed in groups based on parent folder or resource pool. Moreover, other interesting groupings are possible. For example, perhaps every physical host that belongs in rack X can go into a group named X, or every virtual machine running Microsoft Exchange can go into a group named "Microsoft Exchange." An administrator might also decide to join such a group of virtual machines to view notifications related to these 'Microsoft Exchange' virtual machines.

Figure 3: Mapping virtualization relationships to social relationships. In this case, a vCenter server is managing a total of 16 hosts. From a social media perspective, vCenter is 'following' those hosts.

This simple "social network" of persons, hosts, and virtual machines forms a powerful monitoring service. For example, when a virtual machine encounters an issue like a virtual hard drive running out of space, the virtual machine can do a simple http POST request to indicate its status ("ERROR: hard drive out of space") using the messages API, as shown in Figure 4. In this case, a custom stream for Exchange virtual machines has been created, so the virtual machine sends the message to Socialcast and specifically to this custom group. If an administrator is periodically watching updates to this stream, he might notice a flurry of activity and choose to investigate the Exchange virtual machines in his infrastructure. Alternatively, a host can have an agent running that automatically reads messages to a given stream, parses them, and performs certain actions as a result. Finally, by creating a graph connecting vCenter to its end users and to administrators, it becomes easier to notify the relevant parties when an event of interest has occurred: for example, if a virtual machine is affected, then we can limit notifications to the followers of that virtual machine (presumably the users of that virtual machine).

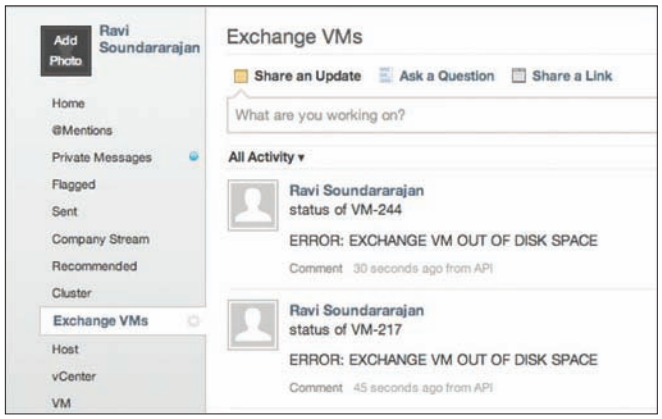

Figure 4: A community of Exchange virtual machines. The administrator has a stream for messages from Exchange virtual machines. The Exchange virtual machines send messages to the stream when they are running out of disk space.

Blindly sending messages to a stream can result in an overload of messages to a human. To avoid such an overload, we can take advantage of a helpful feature of the Socialcast API: the ability to read a stream before publishing to it. If several virtual machines are exhibiting the same issues (for example, hard drive failures), rather than each posting to the same stream and inundating an administrator with messages, the Socialcast agent on each virtual machine can programmatically read the public stream and find out if such a message already exists. If so, the virtual machine can 'like' the message instead of adding a new message to the stream. In this manner, an administrator that is subscribed to this group will not be overwhelmed with messages: instead, the administrator will see a single error message with a large number of 'like' messages. This might suggest to the administrator that something is seriously wrong with some shared resource associated with these virtual machines, like the datastore backing the virtual machines. An illustration of this is shown in Figure 5, in which a number of hosts lose connection to an NFS server. The first server that is affected posts a message, and the others 'like' that message, providing an at-a-glance view of the severity of the problem.

Figure 5: Using 'Like' as a technique for aggregating data. Each host has the same error (in this case, failure to connect to an NFS storage device), but rather than having each one post a separate message, the first affected host posts a message, and subsequent hosts 'like' that message, providing an at-a-glance view of the severity of the problem.

Along the same lines, consider a host that is following each of its virtual machines. The host can use a simple loop to poll for status updates by its virtual machines. When enough such ERROR messages are detected, the host might decide to post an aggregated "ERROR: VM disk failures" to its status. The host can also query Socialcast to find out the number of likes of a given message and thereby determine how many entities are affected by that error. The vCenter server that is following this host might then choose to update its status accordingly ("ERROR: HOST X shows VM disk failures"). The administrator, who is following this vCenter server, will then see the appropriate status notification and might decide to investigate the host. Notice that by utilizing the hierarchical propagation of messages, an administrator sees a greatly reduced set of error messages. The administrator might further decide to create a special group called a 'cluster', and put all hosts in that cluster in a group. The administrator might choose to occasionally monitor the messages in the cluster group. By seeing all messages related to the cluster in one place, the administrator might notice patterns that would not otherwise be obvious. For example, if the cluster group shows a single host disconnect message and a number of 'likes' for that single 'host disconnect' message from the other hosts, it might be the case that a power supply to a rack containing these hosts has failed, and all hosts are subsequently disconnected. Note here that the ability to read the group messages before publishing is crucial to reduce the number of messages: Rather then sending discrete messages for each error, the 'like' attribute is used. Depending on the type of power supply (managed or not), the power supply itself might be able to join a given social network of hosts and virtual machines and emit status updates.

As yet another technique for reducing information, a host or vCenter might utilize flags or comments. For example, depending on the content of the messages (for example, ERROR vs. WARNING), a host that is following its virtual machines might examine a message stream, choose the messages with ERRORS, and flag them or comment on them, indicating that they are of particular importance. The host can later programmatically examine flagged/commented messages and send a single update to the vCenter server. The vCenter server, in turn, can notify the administrator with a single message.

3.2.1 Flexibility and Extensibility

As noted earlier, because nearly all of these messages rely on simple HTTP GETs and POSTs, any virtual machine or ESX host or vCenter server can utilize the entire breadth of the Socialcast API. In each case, it is simple matter of writing a shell script that does rudimentary monitoring and invokes appropriate GET and POST requests. Moreover, more complex workflows and messaging are possible. For example, a simple script in a Linux virtual machine that monitors vmstat[5] might notice that the free memory has dipped below a predefined threshold. The virtual machine might decide to post a status update and use the attachment API to include as an attachment the output of vmstat. Alternatively, the virtual machine might decide to generate a graph and attach the graph to its status message. For Windows virtual machines running MS-SQL, perhaps an agent can periodically monitor disk activity using perfmon[6] and send the results of perfmon as an attachment to the appropriate administrator. Moreover, the architecture can be extended to include any data source that can generate POST/GET requests, ranging from physical devices like core network switches to change management software or procurement software, providing a single pane of glass for any activity related to an individual entity.

3.2.2 Sending and receiving commands via Socialcast

Messages do not have to be limited to static read-only content. For example, perhaps an administrator can send a private message to a virtual machine that includes the body of a script. When the virtual machine reads the message, it can execute the script. Similar such commands can be sent to hosts. For example, a primitive heartbeat mechanism can also be implemented: if each virtual machine and host is configured to send a message once a day, and if a host periodically checks to see if each virtual machine has issued an update, the host can potentially detect if a virtual machine has gone offline. The host could then send itself a command to power on the virtual machine, and if no response is detected from the virtual machine, a message can be posted by the host to the administrator's group. To prevent security issues with malicious users sending arbitrary commands to hosts and virtual machines, it is important to leverage the in-built features of a Socialcast community: namely, that only authorized community members and members of a given group (for example, an administrators group for system administrators) are allowed to send messages to other members.

3.2.3 Message archival and search

The preceding sections have demonstrated various advantages to using a social-media API for virtualization monitoring, including techniques for information reduction and simple interfaces for generating arbitrary types of status information in the form of attachments. An additional advantage of using an online community for virtualization monitoring is that these communities can be hosted in a private cloud, avoiding storage space concerns on any of the entities themselves. Moreover, the Administrator can periodically flush old messages or messages that have been acknowledged and acted upon. Socialcast stores messages in its database and allows full-text search as well as searching using database indexes. Thus, messages can easily be searched, providing a helpful audit trail.

3.3 Implementation Details

3.3.1 System Architecture

Based on the discussion above, one possible implementation involves installing agents in all ESX hosts, virtual machines, and vCenter instances, and having them directly post updates to a Socialcast server. Currently, this might be quite difficult, mainly because IT administrators are understandably risk-averse, so any changes to infrastructure applications (such as vCenter and the software running on ESX hosts) require significant levels of approval. As the approach matures and such an agent is more hardened and tested in the field, however, this is certainly a viable approach. To accommodate existing environments, we chose a slightly less disruptive approach that leverages only publicly available, secure interfaces.

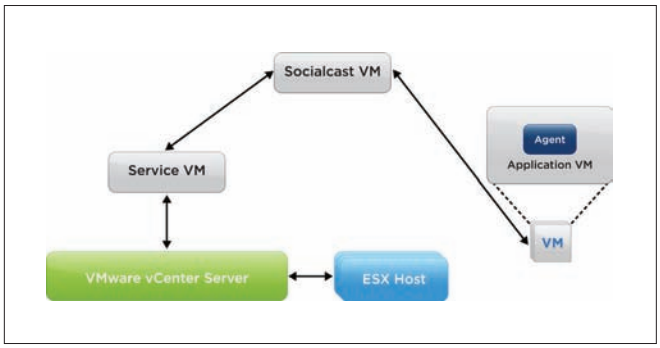

Figure 6: Initial Implementation. A Service virtual machines monitors vCenter and coordinates with vCenter to monitor ESX hosts, posting status to Socialcast on their behalf. Application virtual machines do not use the Serivce VM, and instead run monitoring agents that post status directly to Socialcast.

A logical block diagram of our initial implementation is shown in Figure 6. We use a Socialcast Virtual Machine to implement the social network. We install monitoring agents in each application virtual machine. In application virtual machines running Windows, the agent monitors WMI counters or perfmon counters and application-specific log files like Apache http logs, and it sends HTTP POST requests directly to the Socialcast Virtual Machine. For Linux virtual machines, the monitoring agent examines log files or the output of tools like iostat. For vCenter and for the ESX hosts, in our initial prototype, we chose not to install agents:* instead, we use a service virtual machine to bridge between the Socialcast Virtual Machine and vCenter. The service virtual machine communicates with vCenter over the vSphere API in order to read log data, event data, or statistics. In turn, vCenter is also able to retrieve similar data from each ESX host. In this manner, the service virtual machine only needs to authenticate to one server (vCenter) in order to retrieve information for any host in the infrastructure. The Service Virtual Machine, in turn, performs simple aggregations before posting the data to Socialcast on behalf of the appropriate ESX host or vCenter server. The Service Virtual Machine can also choose to like or comment on the data instead, on behalf of the appropriate ESX host or vCenter. Socialcast is then responsible for its own aggregations (for example, showing how many hosts are affected by a problem by displaying the number of 'likes', as shown in Figure 5). The service virtual machine monitors

* "In some early prototypes, we used agents on vCenter and ESX. However, in order to allow easier deployment in the VMware Hands-on-Lab (see section 4), we opted for this approach."

ESX hosts and vCenter and posts updates on their behalf to the Socialcast VM. While we could have used a Service Virtual Machine to probe application virtual machines in addition to vCenter and ESX, we made the tradeoff to install agents in application virtual machines for two main reasons. First, we were trying to locate application-specific behaviors, and the data we needed could not be retrieved via an API. For example, we initially wanted to probe virtual machines running VMware View [16] to detect latency issues, and such information is kept in certain log files rather than exposed publicly. Second, many application virtual machines already have existing monitoring agents, or export standard interfaces like SNMP and VMI, so there is a precedent for an administrator to admit new agents into a virtual machine. Finally, many administrators create virtual machines from templates or catalogs, so upon deployment, so it is relatively easy to automate the process of installing an agent.

3.3.2 Coupling Virtualization Management and Social Media

To effectively couple virtualization management to social media, our system must perform three functions:

1. Discover the relationships between these entities.
2. Create the appropriate mapping of these virtualization entities to entities in a social network.
3. Monitor each of these entities and post interesting status to Socialcast.

To discover the relationships between these entities, the Service Virtual Machine uses the VMware Web Services SDK to retrieve topology information about the virtualization infrastructure from the vCenter server. This topology information includes the hosts being managed by the vCenter server, the virtual machines running on each host, and the virtual datacenters and clusters. Once this topology information has been gathered, the Service Virtual Machine maps these entities to members of the social network by making calls to the Socialcast Virtual Machine to create users for the vCenter server, the virtual machines, and the hosts. The Service Virtual Machine also makes calls to the Socialcast Virtual Machine to create groups for the datacenters and clusters. To create the appropriate mapping of the relationships, we have nodes in a hierarchy 'follow' their descendants. For example, vCenter follows the hosts it is managing. The ESX hosts 'follow' the virtual machines that they are running. Virtual machines are joined to the datacenters or clusters they belong to, as are hosts. As a final step, we link users to their virtual machines, although this is currently a mostly manual step, unless the user has annotated virtual machines with user information in a structured manner amenable to auto discovery.* At this point, the graph database of the social network has a complete map of the virtualization infrastructure.

The off-the-shelf Socialcast Virtual Machine is architected primarily for human-to-human interaction and collaboration. Thus, user creation requires an administrator logging into a Socialcast instance in order to create user information and to send an email invitation, or it requires an import from another identity source like LDAP. Moreover, the joining of a group is also typically a manual operation performed by a human. As a result, the Socialcast API in its present form does not support creation of users and joining of groups. However, we have modified the API and the Socialcast Virtual Machine to support both of these operations in an automated way, allowing us to completely script the procedure of adding virtualization entities to the social network.

For monitoring these entities, we choose a variety of metrics. For vCenter itself, several factors are important. We monitor the performance metrics of the vCenter server itself like task latencies by using the vSphere API[11]. We also gather usage metrics like CPU, disk, memory and network: these can be gathered via standard interfaces like SNMP. We also examine the log files of the vCenter server itself: these log files are available via the API, given the user has appropriate permissions.

For ESX hosts, we examine performance statistics and kernel logs that are accessible via the vSphere API. The vSphere API allows administrator users with appropriate roles and permissions to login to the vCenter server and access the kernel logs of the ESX hosts. To reduce spew on Socialcast posts, we filter the kernel messages from the hosts and only post warnings and errors. To gain more insight into high vCenter task latencies, it is sometimes helpful to examine the communication logs between vCenter and the ESX hosts: we use the vSphere API to retrieve these logs. Finally, the vCenter server also has an API for retrieving performance statistics per host.

For virtual machine monitoring, we use a two-pronged approach: we collect resource usage statistics for the virtual machine (CPU, Disk, Memory, and network) using the vSphere API. In addition, our agents collect in-guest resource statistics and also examine log files. For example, certain applications like virtual desktops emit log statements when the frame rate of the desktop is low enough to cause user-perceived latencies. Our agent examines such log files and posts relevant message to Socialcast. For resource usage, we do preliminary trending analysis to see if a problem like high memory usage has occurred and then is resolved, and we post resolution of the message to Socialcast. Because these statistics are being gathered within the guest, we gain some visibility that might not be available by the virtual machine-level metrics collected using the vSphere API.

4. Monitoring Case Study: VMworld Hands-On-Labs

To validate our design and gain real-world feedback on our approach, we installed our monitoring service at the hands-on labs at VMworld 2012.[18][19][21]. The hands-on-lab allows VMworld attendees to experiment with various VMware products by following a scripted series of steps. There are over 20 different types of labs to showcase various VMware products, and there are nearly 500 users at a time. As a user enters the hands-on-lab area and indicates a preference for a lab topic, a provisioning portal determines whether a version of the requested lab is available. If not, the lab is provisioned. A lab consists of a number of virtual machines (between 10-17 virtual machines in most cases), and encompasses a mini virtual datacenter that the user can control.

* An alternate approach here could be to use commercially available application discovery tools and then provide an API to link the virtual machines to applications and applications to users.

The hands-on-lab posed unique challenges for our monitoring solution, and required modifications to our original design. The main issue is that the hands-on-lab represents a *high churn environment*, in which virtual machines are constantly being created and destroyed, existing for an hour on average. We use the vSphere API to track the creation and deletion of virtual machines and appropriately modify the relationships within Socialcast, and this environment stresses such code severely. Moreover, because the load on the ESX hosts is highly variable, virtual machines are migrated quite frequently. Our code uses the vSphere APIs to track the motion of virtual machines and update the relationships appropriately, but the frequency of updating such relationships is much higher than might be expected from a typical social network. For example, 2000 virtual machines might be created, destroyed, or moved every hour for 8 hours. In contrast, a company like VMware, with approximately 17,000 employees, might create a dozen new Socialcast users per day.

In light of these challenges and to provide adequate performance, our ultimate deployment architecture utilizes multiple Socialcast Virtual Machines organized using Socialcast clustering. We also employ multiple Service Virtual Machines and divide them among the multiple vCenter servers in the Hands-on-Lab. We also have a Service Virtual Machine for doing preliminary monitoring of the cloud management stack (VMware vCloud Director[14]) that is controlling the vCenter servers.

For our initial monitoring, we focused on a few key areas:

1. **Resource utilization of management components**. We monitored the resource usage of the management software so that we could feed the information back into our core development teams. We show an example in Figure 7, which helped us isolate a given management server that showed much higher CPU usage than others and therefore merited further investigation.

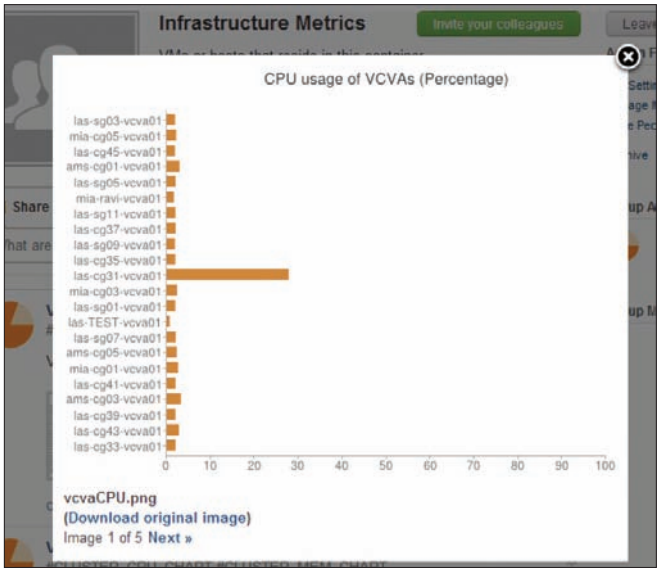

Figure 7: CPU Usage of vCenter servers across infrastructure. The chart is pushed periodically to the administrator's Socialcast stream. In this post, one of the vCenters is showing much higher CPU utilization than others and might need further investigation.

2. **Operational workload on management servers.** The hands-on lab represents an extreme of a cloud-like self-service portal. Creating a user's lab from the self-service portal ultimately results in provisioning operations on the vCenter server, including the cloning of virtual machines from templates, reconfiguring those virtual machines with the proper networking, and then destroying the virtual machines when they are no longer in use. Understanding the breakdown of operations helps developers determine which operations to optimize in order to improve infrastructure performance. We see this in Figure 8, in which we show the breakdown of tasks across all vCenter instances and notice a pattern in the workflow. Specifically, vCenter appears to perform multiple reconfigure operations per VM power operation. We can thus investigate reducing the number of reconfiguration operations as a possible orchestration optimization.

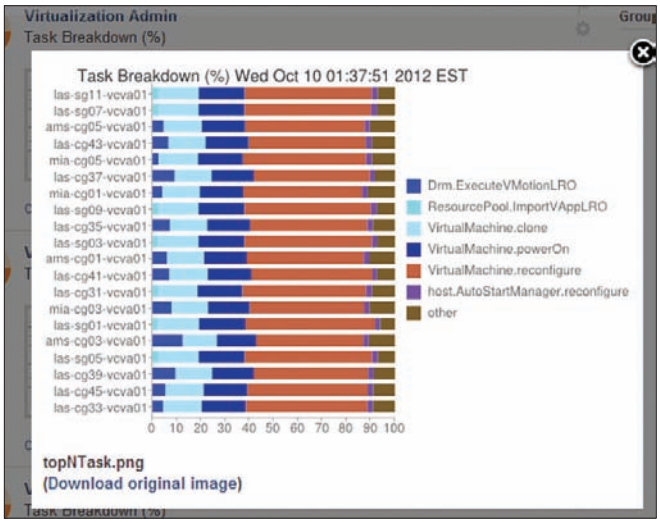

Figure 8: Operational Workload on Management Servers. The standard self-service workflow requires multiple virtual machine reconfigure operations before powering on the VM. This represents a potential optimization opportunity.

3. **Alarms/Errors on virtual machines, ESX hosts and vCenter servers.** We monitor kernel-level error events on ESX hosts and warning/error events on vCenter servers. We also monitor alarm conditions on virtual machines. An example of a kernel-level ESX host error event is loss of connectivity to shared storage. Alarm conditions on virtual machines include high CPU utilization and high disk utilization. For the alarms that are already built into the vCenter server (for example, high virtual machine CPU usage), we leverage vCenter's alarm mechanism, while for others (like high disk usage within a virtual machine), we utilize agents with the virtual machines to monitor and proactively alert Socialcast appropriately.

There were several areas in which our approach had notable advantages over conventional approaches. For example, while the resource utilization of the various management components is available via the API, it can be complex to retrieve this data across multiple installations. By collecting such data and putting it in a single pane of glass, we provided an at-a-glance view of the health of the infrastructure.

By dissecting the operational workload of the datacenter, we were able to determine some possible areas of optimization for our management stack. For example, certain operations require reconfiguration of virtual machines multiple times before the virtual machine is ultimately deployed. By tracking operational metrics and posting them for a group of management components, we were able to see the severity of this problem immediately.

By maintaining relationships between virtual machines and their hosts in an intuitive way, we were able to reason about certain operations more easily. For example, we had deployed our monitoring virtual machines across the infrastructure. At some point, we wanted to move some virtual machines from one host to another. We had created a special group consisting of just our monitoring virtual machines, and by listing the members of this group, we could easily find all of our monitoring virtual machines, and then by 'mousing over' those virtual machines, we could find the hosts on which they resided. Note that this is possible in a standard virtualized infrastructure, but the social network metaphor provides a natural way to perform such a search.

A final example illustrates an unintentional synergy between social media metaphors and virtualization management. One of the administrators wanted to change a cluster to allow automated virtual machine migrations and wanted to see how many migrations would result. Because we had tagged each migration with a hashtag[10] (#Success_drm_executevmotionlro in this case), we were instantly able to search for the number of instances of that hash tag within a given cluster and find out how many migrations resulted from changing the setting in two different clusters. This case is shown in Figure 9.

Figure 9: Synergy between social media and virtualization management with hashtags. By tagging successful tasks with a hashtag (#Success_drm_executevmotionlro), we were able to instantly determine the number of such tasks performed by 2 different vCenter servers (las-cg39, 909 times, and las-cg41, 534 times), without adding any customized aggregation code.

5. Related Work

Many corporations have used the Socialcast developer API to create real-time communication and collaboration tool. One example is an integration of Socialcast and Microsoft Sharepoint[7], in which a Socialcast community can be embedded into a SharePoint site. Communication within the site can be viewed outside of SharePoint, and newcomers to the group can view the archives of previous conversations. In this paper, we have extended the notion of a community to include not just humans but also entities like virtual machines and hosts. Virtual machines and hosts can use automated monitoring in order to 'communicate' with each other and with humans. We have also used the metaphors of social media to assist in virtual management. While adding non-humans to social networks is not a new idea [3], tying together these social media metaphors to virtualization management is novel.

As indicated in section 2, in some sense, virtualization management is already a form of an online community. Virtual machines might generate alarms that can be viewed by vCenter, and vCenter can email administrators in turn with the alarm information. What is missing is an easy-to-use aggregation system based on arbitrary tags. VMware vSphere already contains tagging capabilities, so an administrator can aggregate virtual machines and perhaps utilize analytics tools like vCenter Operations Manager™ [13] for generating alerts based on arbitrary aggregations.

Compared to standard virtualization management tools, our monitoring is more flexible: the monitoring can easily be customized for a particular type of virtual machine. Rather than trying to define a custom alarm for each type of application, a virtual machine owner can simply collect various application-level metrics within the virtual machine and then update status as required: an Exchange virtual machine owner might select messages processed per second; the owner of a virtual machine that does compilation jobs might choose to collect the build times and trigger a status change if the build times suddenly get worse; the owner of a virtual machine that is acting as an NFS datastore might trigger a notice if disk space within the virtual machine gets low. In each case, a simple shell-based script and GET/POST requests are all that are required to provide powerful notification capabilities.

6. Conclusions and Future Work

In this paper, we propose a social-media approach to monitoring virtualized environments. We draw an analogy between a social network and the network created by virtual machines, hosts, and vCenter servers. In a social network, users can follow each other's updates, send each other messages, and create closed groups within the social network for selective communication; in a similar way, we propose taking a virtualized environment and creating a 'community' that includes the various entities of a virtualized environment along with the system administrator. Each entity (whether human or not) is capable of using a simple API to communicate status updates. By judicious creation of hierarchies (for example, having administrators 'follow' vCenter servers, vCenter servers follow hosts, and hosts follow virtual machines) and by using information reduction techniques (for example, 'liking' various types of messages instead of posting the same messages repeatedly), we can avoid excess information flow to the administrator, while still allowing the administrator to view important status updates in the environment. Moreover, the simple API also allows administrators the ability to send commands to any entity, providing a technique for platform-independent remote management via any mobile device.

The approach described in this paper might be quite disruptive for an existing environment. For the user that is reluctant to turn an entire inventory into a social network, there are intermediate steps to validate the approach. The first step is to simply incorporate the

data streams from vCenter into a feed that is posted to a Socialcast site. This is the traditional use of a collaboration tool like Socialcast: administrators use a central repository for data storage, data sharing, and communication, and incorporate external data sources. The next step might be to add hosts but not virtual machines to the social network. The final stage would be to fully embrace the social networking model by adding all simple Socialcast clients to each virtual machine and host and allow virtual machines and hosts to follow and be followed.

There are many avenues for future work. First and foremost, we have learned that an integrated view of the relationships in a virtualization hierarchy is important, so we intend to continue adding more and more entities to the social network. For example, we can add virtual and physical networks, network switches, and intelligent storage devices[17]. We can also add more application awareness and associate virtual machines with each other based on whether they communicate with each other. We can also refine our algorithms for associating users with virtual machines and applications to make that process more automated. As we increase the number of entities and possible churn, we must continue to tune the performance of our service virtual machines and the Socialcast VM.

Another promising avenue for future work is integration with other data sources like vCenter Operations Manager or Zenoss[20]. Assuming modifications to the Operations Manager server to provide notifications on anomalies, the administrator can follow the Operations Manager and be apprised of anomalies or alarms. We can also envision even richer use of the data from this social network of virtualized entities. For example, the various community streams can be uploaded to off-line analytics engines to provide interesting statistics on a given environment, like the most commonly misbehaving virtual machines or hosts or the most common virtual machine error messages, or even the most common scripts that are run on a host. Socialcast itself has some basic analytics which are extremely valuable: we can perhaps imagine extending the architecture to allow plugins that provide virtualization-specific analytics.

While the use cases presented in this paper have focused on monitoring, we can also envision allowing simple commands to be sent via this interface. One simple example, as mentioned earlier, might be allowing a user to send private messages to an individual virtual machine to reboot or provide diagnostic information, although this would require strict security controls (limiting access to administrators or virtual machine owners, for example), or possibly enriching the interface to allow right-click actions on a given entity. This might require leveraging existing access control systems and coupling them to Socialcast's mechanisms for creating users and assigning permissions. Finally, we can also potentially embed the administrator's social media web page directly into the vSphere web-based client, providing a complete one-stop shop for virtualization management and monitoring, combining standard paradigms with an intuitive yet novel social media twist.

Acknowledgments

We thank Rajat Goel for providing an on-premise Socialcast virtual appliance that we could deploy for testing purposes. We thank Sean Cashin for numerous hints for using the Socialcast API effectively. We thank Steve Herrod and the office of the CTO for helpful comments on our work. Finally, we thank Conrad Albrecht-Buehler, for extremely helpful comments on our paper.

References

1 Facebook, www.facebook.com

2 Haxx. 'libcurl: the multiprotocol file transfer library,' http://curl.haxx.se/libcurl/

3 Holland, S.W. Social Networking with Autonomous Agents. United States Patent US 2012/0066301, http://www.google.com/patents/US20120066301

4 Linux iostat utility, http://www.unix.com/apropos-man/all/0/iostat/

5 Linux vmstat utility, http://nixdoc.net/man-pages/Linux/man8/vmstat.8.html

6 Microsoft. Perfmon, http://technet.microsoft.com/en-us/library-bb490957.aspx

7 Socialcast. Making Sharepoint Social: Integrating Socialcast and SharePoint Using Reach and API, http://blog.socialcast.com/making-sharepoint-social-integrating-socialcast-and-sharepoint-using-reach-and-api

8 Socialcast. Socialcast Developer API, http://www.socialcast.com/resources/api.html

9 SPEC. SpecJBB2005, http://www.spec.org/jbb2005/

10 Twitter. "What are hashtags?" https://support.twitter.com/articles/49309-what-are-hashtags-symbols#

11 VMware. VMware API Reference Documentation, https://www.vmware.com/support/pubs/sdk_pubs.html

12 VMware. VMware vCenter Application Discovery Manager, http://www.vmware.com/products/application-discovery-manager/overview.html

13 VMware. VMware vCenter Operations Manager, http://www.vmware.com/products/datacenter-virtualization/vcenter-operations-management/overview.html

14 VMware. VMware vCloud Director, http://www.vmware.com/products/vcloud-director/overview.html

15 VMware. VMware vSphere, http://www.vmware.com/products/datacenter-virtualization/vsphere/overview.html

16 VMware. VMware View, http://www.vmware.com/products/view/overview.html

17 VMware. VMware vStorage APIs for Array Integration, http://communities.vmware.com/docs/DOC-14090

18 VMware. VMworld 2012, http://www.vmworld.com/community/conference/us

19 VMware. VMworld Europe 2012, http://www.vmworld.com/community/conference/europe/

20 Zenoss. Zenoss Virtualization Monitoring, http://www.zenoss.com/solution/virtualization-monitoring

21 Zimman, A., Roberts, C., and Van Der Welt, Mornay. VMworld 2011 Hands-on Labs: Implementation and Workflow. VMware Technical Journal, Vol 1. No. 1. April 2012. pp. 70-80.

VMware View® Planner: Measuring True Virtual Desktop Experience at Scale

Banit Agrawal
banit@vmware.com

Rishi Bidarkar
rishi@vmware.com

Sunil Satnur
ssatnur@vmware.com

Tariq Magdon-Ismail
tariq@vmware.com

Lawrence Spracklen
lspracklen@vmware.com

Uday Kurkure
ukurkure@vmware.com

Vikram Makhija
vmakhija@vmware.com

VMware, Inc.

Abstract

In fast-changing desktop environments, we are witnessing increasing use of Virtual Desktop Infrastructure (VDI) deployments due to better manageability and security. A smooth transition from physical to VDI desktops requires significant investment in the design and maintenance of hardware and software layers for adequate user base support. To understand and precisely characterize both the software and hardware layers, we present VMware View® Planner, a tool that can generate workloads that are representative of many user-initiated operations in VDI environments.

Such operations typically fall into two categories: *user* and *admin* operations. Typical user operations include typing words, launching applications, browsing web pages and PDF documents, checking email and so on. Typical admin operations include cloning and provisioning virtual machines, powering servers on and off, and so on. View Planner supports both types of operations and can be configured to allow VDI evaluators to more accurately represent their particular environment. This paper describes the challenges in building such a workload generator and the platform around it, as well as the View Planner architecture and use cases. Finally, we explain how we used View Planner to perform platform characterization and consolidation studies, and find potential performance optimizations.

I. Introduction

As Virtual Desktop Infrastructure (VDI) [6,7,8] continues to drive toward more cost-effective, secure, and manageable solutions for desktop computing, it introduces many new challenges. One challenge is to provide a local desktop experience to end users while connecting to a remote virtual machine running inside a data center. A critical factor in remote environments is the remote user experience, which is predominantly driven by the underlying hardware infrastructure and remote display protocol. As a result, it is critical to optimize the remote virtualized environment [13] to realize a better user experience. Doing so helps IT administrators to confidently and transparently consolidate and virtualize their existing physical desktops.

Detailed performance evaluations and studies of the underlying hardware infrastructure are needed to characterize the end-user experience. Very limited subjective studies and surveys on small data sets are available that analyze these factors [19,20,21], and results and techniques do not measure up to the scale of the requirements in VDI environments. For example, the required number of desktop users easily ranges from a few hundred to tens of thousands, depending on the type of deployment. Accordingly, there is a pressing need for an automated mechanism to characterize and plan huge installations of desktop virtual machines. This process should qualitatively and quantitatively measure how user experience varies with scale in VDI environments. These critical measurements enable administrators to make decisions about how to deploy for maximum return on investment (ROI) without compromising quality.

This paper presents an automated measurement tool that includes a typical office user workload, a robust technique to accurately measure end-to-end latency, and a virtual appliance to configure and manage the workload run at scale. Ideally, the workload needs to incorporate the many traditional applications that typical desktop users use in the office environment. These applications include Microsoft Office applications (Outlook, PowerPoint, Excel, Word), Adobe Reader, Windows Media player, Internet Explorer, and so on. The challenge lies in simulating the operations of these applications so they can run at scale in an automated and robust manner. Additionally, the workload should be easily extensible to allow for new applications, and be configurable to apply the representative load for a typical end user. This paper discusses these challenges and illustrates how we solved them to build a robust and automated workload, as described in Section II.

The second key component to a VDI measurement framework is precisely quantifying the end-user experience. In a remote connected session, an end-user only receives display updates when a particular workload operation completes. Hence, we need to leverage the display information on the client side to accurately measure the response time of a particular operation. Section III presents how a watermarking technique can be used in a novel way to measure remote display latency. The watermarking approach was designed to:

- Present a new encoding approach that works even under adverse network conditions
- Present smart watermarking placement to ensure the watermark does not interfere with application updates
- Make the location adaptive to work with larger screen resolutions

The final piece is the automated framework that runs the VDI user simulation at scale. We built a virtual appliance with a simple and easy-to-use web interface. Using this interface, users can specify the hardware configuration, such as storage and server infrastructure, configure the workload, and execute the workload at scale. The framework, called VMware View Planner, is designed to simulate a large-scale deployment of virtualized desktop systems and study the effects on an entire virtualized infrastructure (Section IV). The tool scales from a few virtual machines to thousands of virtual machines distributed across a cluster of hosts. With View Planner, VDI administrators can perform scalability studies that closely resemble real-world VDI deployments and gather useful information about configuration settings, hardware requirements, and consolidation ratios. Figure 1 identifies the inputs, outputs, and use cases of View Planner.

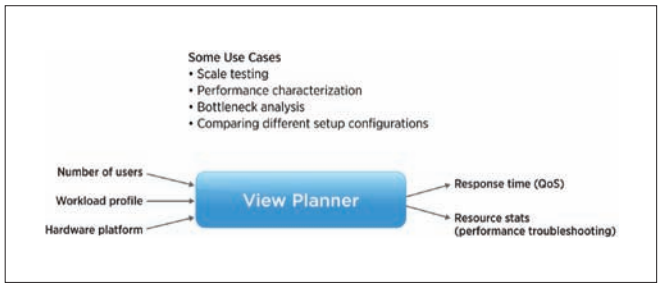

Figure 1. View Planner: Inputs, Outputs, and Use cases

Using the View Planner tool, we enable the following use cases in the VDI environment:

- **Workload generation.** By configuring View Planner to simulate the desired number of users and configure applications, we can accurately represent the load presented in a given VDI deployment. Once the workload is running, resource usage can be measured at the servers, network, and storage infrastructure to determine if bottlenecks exist. Varying the load enables required sizing information to be obtained for a particular VDI architecture (Section V).
- **Architectural comparisons.** To determine the impact of a particular component of VDI architecture, we can configure a fixed load using View Planner and measure the latencies of administrative operations (provisioning, powering on virtual machines, and so on) and user operations (steady-state workload execution). Switching the component in question and measuring latencies again provides a comparison of the different options. A note of caution here is that both administrative and user operation latencies can vary significantly based on the underlying hardware architecture, as described in sections V.C and V.E.
- **Scalability testing.** In this use case, hardware is not varied. The system load is increased gradually until a resource of interest is exhausted. Typical resources measured include CPU, memory, and storage bandwidth. Example use cases are presented in section V.B.

There are many other use cases, such as remote display protocol performance characterization (section V.D), product features performance evaluation, and identification of performance bottlenecks. Section VI provides relevant and related work in VDI benchmarking.

2. Workload Design

Designing a workload to represent a typical VDI user is a challenging task. Not only is it necessary to capture a representative set of applications, it also is important to keep the following design goals in mind when developing the workload.

Scalability. The workload should run on thousands of virtual desktops simultaneously. When run at such scale, operations that take a few milliseconds on a physical desktop could slow down to several seconds on hardware that is configured in a suboptimal manner. As a result, it is important to build high tolerances for operations that are sensitive to load.

Robustness. When running thousands of virtual desktops, operations can fail. These might be transient or irreversible errors. Operations that are transient and idempotent can be retried, and experience shows they usually succeed. An example of a transient error is the failure of a PDF document to open. The open operation can be retried if it fails initially. For operations that are not idempotent and cannot be reversed, the application simply is excluded from the run from that point onwards. This has the negative effect of altering the intended workload pattern. After extensive experimentation, this tack was decided upon so the workload could complete and upload the results for other operations that complete successfully. Our experience shows that only a few operations fail when run at scale, and the overall results are not altered appreciably.

Extensibility. Understanding that one set of applications is not representative of all virtual desktop deployments, we chose a very representative set of applications in our workload and enable View Planner users to extend the workload with additional applications.

Configurability. Users should be able to control the workload using various configurations, such as the application mix, the load level to apply (light, medium, heavy), and the number of iterations.

We overcame many challenges during the process of ensuring our workload was scalable, robust, extensible, and configurable. To realize these goals, we needed to automate various tasks in a robust manner.

Command-line-based automation. If the application supports command-line operations, it is very easy to automate by simply invoking the commands from the MS-DOS command shell or Microsoft Windows PowerShell.

GUI-based automation. This involves interacting with the application just like a real user, such as clicking window controls, typing text into boxes, following a series of interactive steps (wizard interaction), and so on. To do this, the automation script must be able to recognize and interact with the various controls on the screen, either by having a direct reference to those controls (Click “Button1”, Type into “Textbox2”) or by knowing their screen coordinates (Click <100,200>). The user interfaces of Microsoft Windows applications are written using a variety of GUI frameworks. Windows applications written by Microsoft extensively use the Win32 API to implement windows, buttons, text boxes, and other GUI elements. Applications written by third-party vendors often use alternative frameworks. Popular examples include the Java-based

SWT used by the Eclipse IDE, or the ActionScript-based Adobe Flash. Automating applications with a Win32 API-based GUI is relatively straightforward with the AutoIT scripting language [12]. Automating applications that use alternative frameworks for the GUI is not straightforward and requires other tools.

API based automation. This involves interacting with the application by invoking its APIs to perform specific actions. Microsoft's COM API is a good example of this model. All Microsoft Office applications export a COM interface. Using the COM API it is possible to do almost everything that a user can do using the GUI. API-based automation is chosen over GUI-based automation when the GUI elements are very complicated and cannot be accessed directly. For example, it is very difficult in Microsoft Outlook to click on an individual mail item using direct GUI controls, let alone obtain information about the mail item, such as the identity of the sender. On the other hand, the Microsoft Outlook COM API provides a rich interface that lets you locate and open a mail item, retrieve information about the sender, receiver, attachments, and more.

The next sections provide a description of the workload composition, discuss how to avoid the workload starting at the same time, and illustrate how to perform timing measurements.

A. Workload Composition

Instead of building a monolithic workload that executes tasks in a fixed sequence, we took a building-block approach, composing each application of its constituent operations. For example, for the Microsoft Word application we identified Open, Modify, Save, and Close as the operations. This approach gave us great flexibility in sequencing the workload in any way desired. The applications and their operations are listed in Table 1.

APPLICATION ID	APPLICATION	OPERATIONS
1	Firefox	["OPEN", "CLOSE"]
2	Excel	["OPEN", "COMPUTE", "SAVE", "CLOSE", "MINIMIZE", "MAXIMIZE", "ENTRY"]
3	Word	["OPEN", "MODIFY", "SAVE", "CLOSE", "MINIMIZE", "MAXIMIZE"]
4	AdobeReader	["OPEN", "BROWSE", "CLOSE", "MINIMIZE", "MAXIMIZE"]
5	IE_ApacheDoc	["OPEN", "BROWSE", "CLOSE"]
6	Powerpoint	["OPEN", "RUNSLIDESHOW", "MODIFYSLIDES", "APPENDSLIDES", "SAVEAS", "CLOSE", "MINIMIZE", "MAXIMIZE"]
7	Outlook	["OPEN", "READ", "RESTORE", "CLOSE", "MINIMIZE", "MAXIMIZE", "ATTACHMENT-SAVE"]
8	7zip	["COMPRESS"]
10	Video	["OPEN", "PLAY", "CLOSE"]
12	Webalbum	["OPEN", "BROWSE", "CLOSE"]

Table 1: Applications with their IDs and operations

B. Avoiding Synchronized Swimming

Operations in a desktop typically happen at discrete intervals of time, often in bursts that consume many CPU cycles and memory. We do not want all desktops to execute the same sequence of operations for two reasons. It is not representative of a typical VDI deployment, and it causes resource over commitment. To avoid synchronized swimming among desktops, the execution sequence in each desktop is randomized so the desktops perform different things at any given instant of time and the load is distributed evenly distributed (Figure 2).

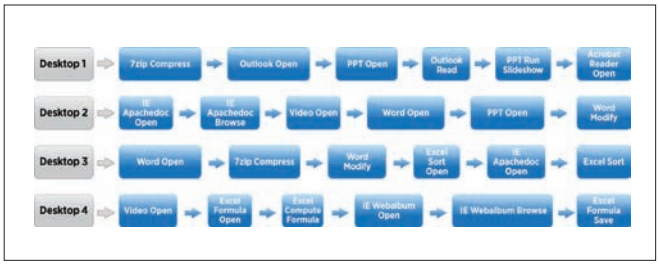

Figure 2. Shows the randomized execution of different operations across different desktop virtual machines. While the figure is not drawn to scale, the box width shows the relative timing of different operations.

C. Timing

The time taken to perform desktop operations ranges from a few milliseconds to tens of seconds. A high resolution timer is needed for operations that complete very quickly. The **Query Performance Counter (QPC)** call in Microsoft Windows cannot be relied upon because it uses the virtualized timestamp counter, which may not be accurate [3]. Consequently, the timestamp counter must be fetched from the performance counter registers on the host CPU. The virtual machine monitor in VMware software enables a virtual machine to issue an **rdpmc()** call into the host machine to fetch hardware performance counters. To measure the latency of an operation, we simply wrap the operation with these **rdpmc()** calls to obtain a much more reliable measurement. Since the **rdpmc()** call is intercepted by the hypervisor, translated, and issued to the host, it can take more cycles than desired. Our measurements indicate this call consumes approximately 150,000 cycles on a 3 GHz processor, or approximately 50 microseconds. The operations measured take at least 50 milliseconds to complete, which means the overhead of using the **rdpmc()** call is less than 0.1 percent.

D. Extending the Workload with Custom Applications

As mentioned earlier, a fixed set of applications is not representative of possible virtual desktop deployments. View Planner is designed to be extensible, enabling users to plug in their applications. Users must follow the same paradigm of identifying operations constituting their application. Since the base AutoIT workload included with View Planner is compiled into an executable, users cannot plug in their code into the main workload. To help this issue, we implemented a custom application framework that uses TCP sockets to communicate between the main workload and the custom application script. Using this feature, users can add

customized applications into their workload mix and identify the applications suite that best fits their VDI deployments. For example, a healthcare company can implement a health related application and mix it with typical VDI user workloads and characterize their platform for VDI deployments.

E. Workload Scalability Enhancements

Virtualized environments make effective use of hardware by allowing multiple operating system instances to run simultaneously on a single computer. This greatly improves utilization and enables economies of scale. While there are innumerable benefits to virtualization, poorly designed virtual environments can cause unpredictability in the way applications behave, primarily due to resource over commitment. The goal of the View Planner workload is to reliably detect and report poor designs in virtual desktop environments. Because the View Planner workload is technically another application running inside virtual desktops, it is susceptible to the same unpredictability and failures under load. To make the process of timing measurement and reporting reliable, we built mechanisms into the workload that ensure the workload runs to completion even under the most stressful conditions.

1) Idempotent Operations and Retries

To make the workload more manageable, we split the operations of each application into the smallest unit possible. We also designed most of these operations to be idempotent, so that failed operations can be retried without disturbing the flow of operations. Our experience indicates many operations fail due to transient load errors and many typically succeed if tried again. As a result, the software retries the operation (just as a normal user) three times before declaring a failure. While the retry mechanism has significantly improved the success rate of individual operations passing under high load, some operations might still fail.

Two options are available when an operation fails. The first option is to fail the workload because one of its constituent operations failed after three retries. Another option is to continue with the workload by ignoring further operations of the application that encountered a failure. We decided to leverage the second approach for two reasons:

- Our workload is composed of many applications and an even greater number of operations. Failing the entire workload for one or two failed applications discards all successful measurements and results in wasted time.
- Since our workload runs on multiple virtual desktops simultaneously, failures in a few desktops do not have a significant impact on the final result if we consider the successful operations of those desktops.

By selectively pruning failed applications from a few virtual machines, we are able to handle failures at a granular level and still count the successful measurements in the final result, resulting in improved robustness and less wasted time for users. We also flag the desktops that failed the run.

2) Progress Checker (View Planner Watchdog)

In situations where extreme reliability and robustness are needed, a watchdog mechanism is needed to ensure things progress smoothly. We use this concept by employing a progress checker process, a very simple user-level process with an extremely low chance of failure. A progress file, a simple text file, keeps track of the workload progress by storing the number of operations completed. The progress checker studies workload progress by reading the progress file.

When the workload starts, it keeps an operations count in a known Microsoft Windows registry location and tries to launch the progress checker process. The workload fails to run if it cannot launch the progress checker. When the workload starts performing regular operations, it increments the count stored in the registry. The progress checker process periodically wakes up and reads the registry. It terminates the workload if progress is not detected. The progress checker sleeps for three times the expected time taken by the longest running operation in the workload. This ensures the workload is not terminated accidentally. Finally, if the progress checker needs to terminate the workload, it does so and reports the timing measurements completed so far so they can be included in the final score.

3. End-User Measurement Framework

The second part of the View Planner framework is the precise measurement of the user experience from the client side. This section describes the novel measurement technique used to measure application response time from the client side. It presents the VDI watermarking approach as well as a brief description of our plugin (client agent) implementation.

A. View Planner Workload Watermarking

In our previous approaches [1, 2], the idea was to use the virtual channel to signal the start and end of events through the display watermarking on the screen. In these techniques, the display watermarking location overlapped with applications, resulting in a chance for the watermarking update to be overlapped by application updates. This results in the particular event being missed and the workload not progressing as expected. We needed a mechanism in which the watermarking location is disjoint to application rendering. To enable better decisions on the client side, we also required metadata encoded in the watermark. This allows us to identify the operation on the client side, as well as drive the workload from the client side to realize true VDI user simulation.

We designed our watermarking approach such that it:

- Uses a new encoding approach that works even under adverse network conditions
- Uses smart watermarking placement to ensure the watermark does not interfere with application updates
- Adapts to larger screen resolutions

To precisely determine application response time on the application side, we overlay metadata on top of the start menu button that travels with the desktop display. This metadata can be any type useful information, such as the application operation type and number of events executed. Using this metadata information timing information for the application operation can be derived on the client side. We detect the metadata and record the timestamps at the start and end of the operation. The difference between these timestamps enables the estimation of application response time. There are many challenges associated with getting accurate metadata to the client side, making it unobtrusive to the application display, and ensuring it always is visible. As a result, the location of metadata display and the codec used to encode metadata is very important.

Table 1 illustrates an example of the encoding of different application operations. The table shows the applications (Firefox, Microsoft Office applications, Adobe Reader, Web Album, and so on) that are supported with View Planner along with their operations. Each application is assigned an application ID and has a set of operations. Using this approach, we can encode a PowerPoint "Open" operation in pixel values by doing the following:

- Note that the Microsoft PowerPoint application has application ID "6"
- See that "Open" is the first operation in the list of operations
- Calculate the encoding of each operation using the following formula: (application_id * NUM_SUBOPS) + (operation_id)
- Determine the encoding of the Microsoft PowerPoint "Open" operation is (6 * 10) + 1 = 61
- The value 10 is used for NUM_SUBOPS since assume the number of operations for a particular application will not exceed 10.

After looking at one encoding example, let's see how we send this encoded data to the client side and how robustly we can infer the code from the client side. As shown in Figure 3, we display the metadata on top of the start menu button since it does not interfere with the rendering of other applications. The watermark is composed of three lines composed of white and black pixel colors, each 20x1 pixels wide. The first line is used to denote the test ID, the event code for the current running operation. The next two markers signal the start and end of a particular operation. Using the example shown at bottom of Figure 3, we can explore how these three lines are used to monitor response time. When the workload executes the Microsoft PowerPoint "Open" operation, the workload watermarks the test ID location with the event code 61, as calculated earlier. The workload puts the sequence number for the number of operations "n" that have occurred (1 in this case) in the start location. The protocol sends the watermark codes to the client when a screen change occurs. The encoded watermarks are decoded as a Microsoft PowerPoint "Open" operation on the client side. When the client observes the start rectangle update, the "start" timestamp is recorded. When the "Open" operation is complete, the workload watermarks the test ID location again with the Microsoft PowerPoint "Open" event code (61) and a code (1000-n) in the end location.

When the client observes the end rectangle update with the sum of the start ID and end ID equaling 1000, the "end" timestamp is recorded. The difference between the timestamps is recorded as the response time of "Open" operation.

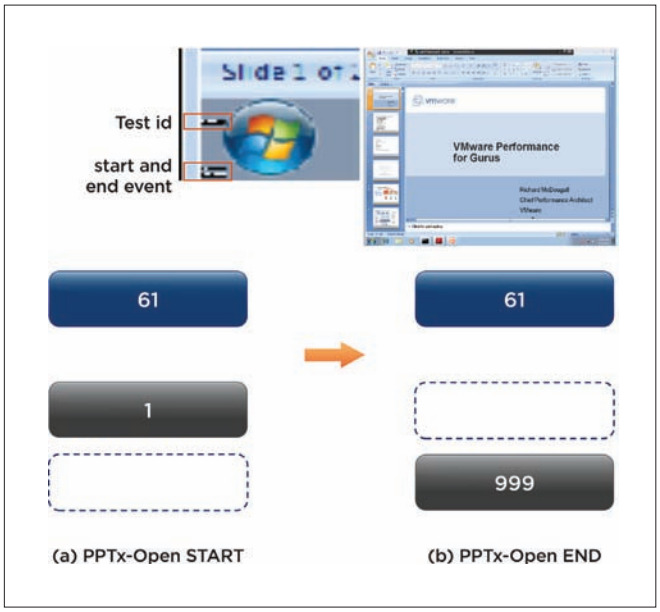

Figure 3. This figure shows the VDI desktop with the Microsoft PowerPoint application running in the background. The watermarking region is near the start menu and is shown on the left. The watermarking codes below show the PowerPoint Open operation. Figure 3a shows the start of the operation and Figure 3b shows the end of the operation.

B. Measurement Plugin Implementation

This section describes the internal details of the measurement plugin. There are common requirements, such as getting rectangle updates, finding pixel color values, sending key events, and using the timer for timestamps. With these four APIs, we can extend the measurement technique to any client device, such as Apple iPads or Android-based tablets. Any mirror driver can be used to get rectangle updates. The function of a mirror driver is to provide access to the display memory and screen updates as they happen. In our implementation, we use the SVGA DevTap interface that we built and implemented as part of the SVGA driver. The software performs approximately 40 scans per second (25 ms granularity) to process incoming display updates to look for encoded watermark events.

On the client side, the plugin runs a state machine. It changes state from sending an event for the next operation, waiting for the start event, waiting for the end event, and finally waiting for the think time. In the "sending event" state, the plugin sends a key event to signal the desktop to start the next operation. The plugin records the time when it sees the end of the event. It continues to iterate through different states of the state machine until the workload "finish" event is sent from the desktop. During video play operation, the main plugin switches to the video plugin [2] and records frame timings. After the video playback is complete, it switches back to the main measurement plugin to measure the response time of other applications. For timing, the host RDTSC is used to read the timestamp counters and divide by the processor frequency to determine the elapsed time.

4. View Planner Architecture

This section provides an overview of the third piece of VDI simulation framework—the View Planner framework to simulate VDI workloads at scale—and discuss its design and architecture. To run and manage workloads at large scale, we designed an automated controller. The central piece in the View Planner architecture, the automated controller is the harness or appliance virtual machine that controls everything, from the management of participating desktop and client virtual machines, to starting the workload and collecting results, to providing a monitoring interface through a web user interface.

The View Planner appliance is essentially a CentOS Linux-based appliance virtual machine that interacts with many VDI server components. It also runs a web server to present a user-friendly web interface. Figure 4 shows the high-level architecture of View Planner. As shown in the diagram, the appliance interacts with a VMware vCenter View connection server or Virtual Center server to control desktop virtual machines. It also communicates with client virtual machines to initiate remote protocol connections. The appliance is responsible for starting the workload simulation in desktop virtual machines. Upon completion, results are uploaded and stored in a database inside the appliance. Results can be viewed using the web interface or extracted from the database at any time.

The harness controller provides the necessary control logic. It runs as a Linux user-level service in the appliance and interacts with many external components. The control logic implements all needed functionality, such as:

- Keeping state and statistics
- Controlling the run and configurations
- Providing monitoring capabilities
- Interacting with the database and virtual machines
- Collecting and parsing results and reporting scores

The View Planner tool uses a robust and asynchronous remote procedure call (RPC) framework (Python Twisted) to communicate with desktop or client virtual machines. Testing shows successful connection handling for up to 4,000 virtual machines.

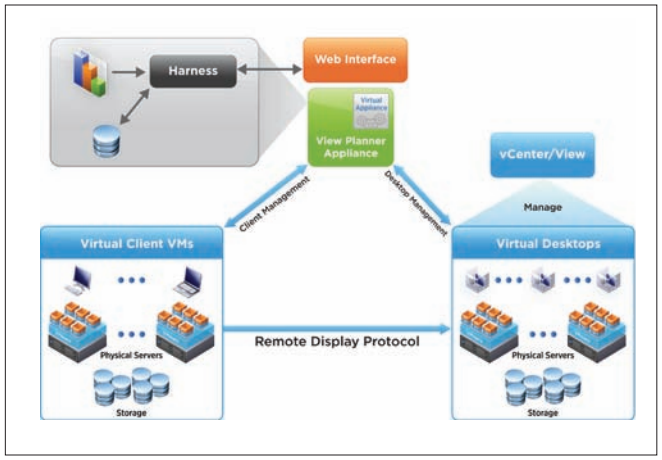

Figure 4. View Planner high-level architecture

A. View Planer Operations

This section discusses the View Planner flow chart and how View Planner operates the full run cycle. Once the harness is powered on, the service listens on a TCP port to serve requests from the web interface. Figure 5 shows the operation flow chart for View Planner. In the first phase, View Planner stores all server information and their credentials. Next, desktop virtual machines can be provisioned (an administrative operation) using the web interface. Following this step, the View Planner user defines the workload profile (applications to run) and the run profile (the number of users to simulate), and so on. Next, the run profile is executed. After the run completes, results are uploaded to the database and can be analyzed. This process can be repeated with a new workload and run profile.

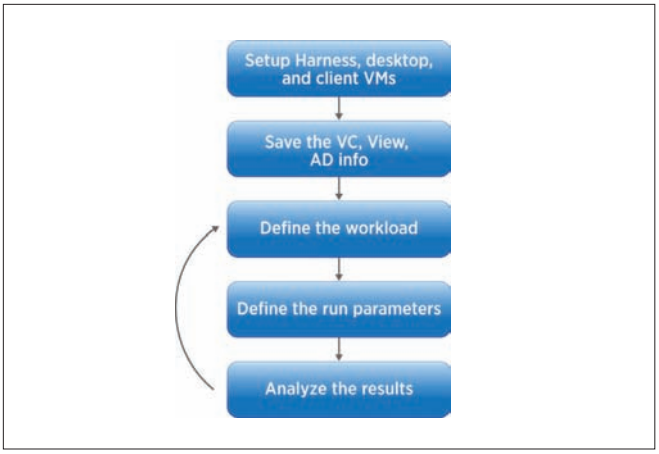

Figure 5. View Planner operational flow chart

Let's discuss in detail what happens in the background, starting when a VDI evaluator executes a particular run profile until the final results are uploaded. This is illustrated in Figure 6 for three different modes of View Planner:

- In "local" mode, the workload executes locally without clients connected.
- In "remote" mode, workload execution and measurement are performed with a remote client connected to one desktop (one-to-one).
- In "passive" mode, one client can connect to multiple desktops (one-to-many).

In these modes, View Planner first resets the test for the previous run, and powers off virtual machines. At this stage, the harness is initialized and ready to execute the new run profile. Next, a prefix match stage finds participating virtual machines based on the desktop or client prefix provided in the run profile. View Planner powers on these participating virtual machines at a staggered rate that is controlled by a configurable parameter. VDI administrators needing to investigate bootstorm issues can increase this value to a maximum, causing View Planner to issue as many power on operations as possible every minute.

Once the desktops are powered on, they register their IP addresses. Upon meeting a particular threshold, View Planner declares a certain number of desktop or client virtual machines are available for the run. At this stage, View Planner waits for the ramp up time for all virtual machines to settle. Next, it obtains the IP address for each desktop and client virtual machines and uses these IP addresses to initiate the run.

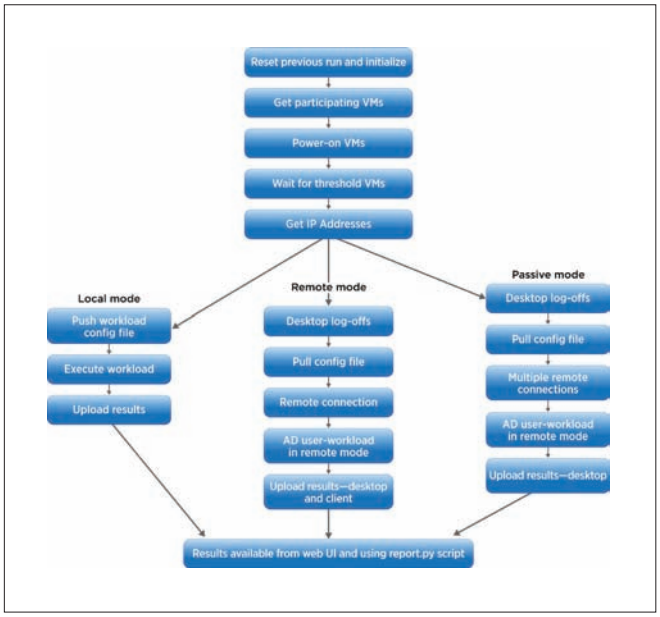

Figure 6. View Planner operations for different modes

In remote and passive mode, View Planner executes an extra phase in which it logs off desktop virtual machines (Figure 6). After the logoff storm, View Planner waits for the virtual machines to settle and CPU usage to return to normal. After this stage, the appliance sends a command to the desktops to execute the workload in local mode. For remote and passive modes, commands are sent to clients to execute logon scripts. This is logon storm is similar to what happens when employees arrive at the office in the morning and log into their desktops. The login storm is an "admin" operation and can be controlled by a configurable parameter. Once the connections are established, they update their status to the harness and View Planner records how many workload runs have started. After the run completes, the desktops upload the results. For remote mode, View Planner finds the matching clients and asks the clients to upload the results. These results are inserted into the database. The View Planner passive mode is good to use when VDI evaluators do not have sufficient hardware to host client virtual machines.

B. Handling Rogue Desktop and Client Virtual Machines

When simulating a large user run, there may be situations in which a few desktops are stuck in the customization state or are unable to obtain an IP address. In this case, a timer runs all the time. After every registration, reset the timer is reset and the software waits for 30 minutes. If more registrations are not seen after 30 minutes, and the required threshold is not met, we kick off the logic to find the bad virtual machines and reset them. After obtaining the IP address of each registered desktop, we use the virtual machine name to IP address mapping to find the rogue virtual machines. Once the rogue virtual machines are reset, they register their IP addresses and, on meeting the threshold, the run starts If the threshold is still not met, the timer is reset a few times and the run is started with the registered number of virtual machines.

C. Scoring Methodology

An invocation of the View Planner workload in a single virtual machine provides hundreds of latency events. As a result, when scaling to thousands of desktops virtual machines using View Planner, the number of latency measurements for different operations grows very large. Hence, a robust and efficient way to understand and analyze the different operational times collected from the numerous desktop virtual machines is required. To better analyze these operations, we divided the important operations into two buckets. Group A consists of interactive operations, Group B consists of I/O intensive operations. Group A and Group B operations are used to define the *quality of service* (QoS), while the remaining operations are used to generate additional load. For a benchmark run to be valid, both Group A and Group B need to meet their QoS requirements. QoS is defined in terms of a threshold time limit, T_h. For each group, 95 percent of the operations must complete within T_h. Limits are set based on extensive experimental results spanning numerous hardware and software configurations. The View Planner benchmark "score" is the number of users (virtual machines) used in a run that pass QoS.

5. Results And Case Studies

This section presents workload characterization results, describes several View Planner use cases, and presents associated results.

For most of the experiments presented in subsequent sections, all of the applications supported in View Planner ran with 20 seconds of think time. Think time is used to simulate the random pause when a VDI user is browsing a page or performing another task. For the 95 percent Group A threshold (T_h), we selected a response time of 1.5 seconds based on user response time and statistical analysis.

A. Workload Characterization

We first characterized the workload based on how many operations View Planner executed and how randomly the operations were distributed across different iterations in different desktop virtual machines.

Event Counts

Figure 7 shows the average number of times each operation is executed per virtual machine. The *-Open and *-Close operations are singleton operations and occur at low frequency, while interactive operations (AdobeReader-Browse, IE_ApacheDoc-Browse, Word-Modify, and so on) typically are executed more than 10 times. This is very close to real-world user behavior: the document is opened and closed once, with many browse and edit operations performed while the document is open.

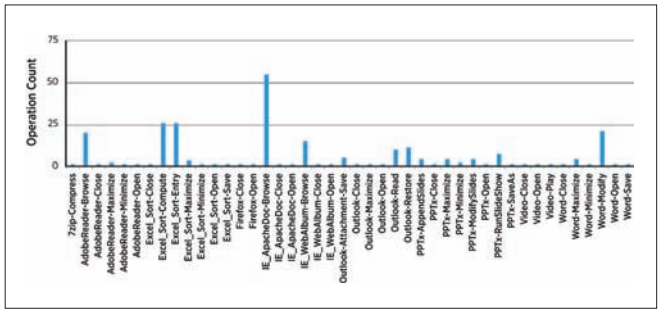

Figure 7. Average event count for each application operation

Response Time

As discussed in Section III.A, using the proposed watermarking technique the response time of applications from the client side can be measured. The response time chart in Figure 8 shows the application latency seen for different operations with the RDP protocol in LAN conditions. As seen from the graph, most of the Open and Save operations take more than two seconds, as they are CPU and I/O intensive operations, while most interactive operations, such as Browse and maximize operations, take less than a second.

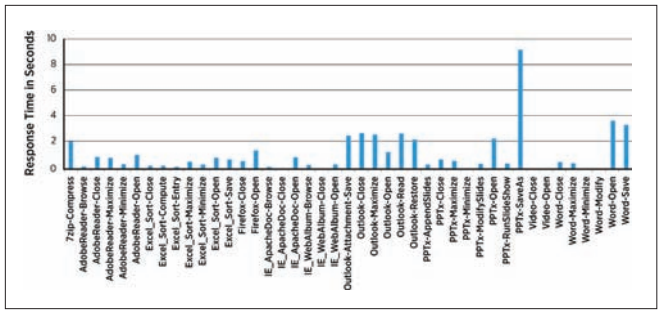

Figure 8. Average response time for each application operation measured from the client side

Iteration Overlap

Figure 9 shows virtual machine execution of operations over each of the seven iterations. The y-axis corresponds to the ID of a particular virtual machine and the x-axis is time. The colors represent different iterations. The data plotted is from a 104 virtual machine run on an 8-core machine. Due to heavy load on the system, there is a skew between the iteration start and stop times across virtual machines, resulting in iteration overlap in the system. We can see that different virtual machines start and finish at different times due to randomized load distribution on the physical host.

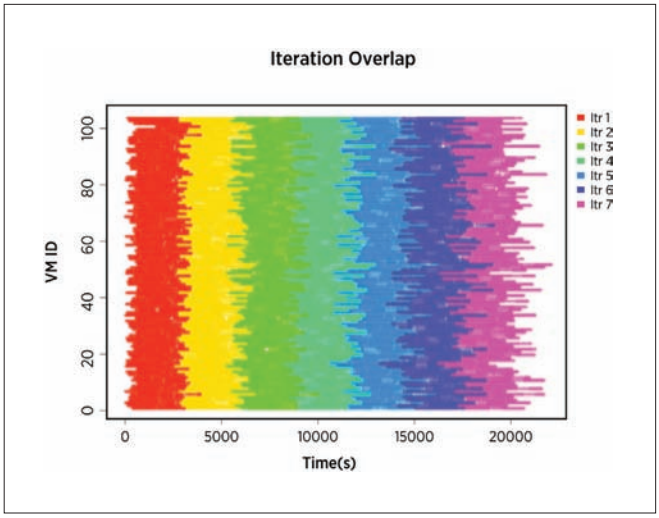

Figure 9. Shows the iteration overlap and how applications run in each virtual machine in 104 virtual machine run.

B. Finding the Score (Number of Supported Users)

One of most important use cases of View Planner is determining how many users can be supported on a particular platform. We used View Planner on a single host with the VMware® vSphere® 5 platform and Fibre Channel (FC) storage to determine the maximum number of supported users. Detailed results are shown in Table 2. The simulation started with 96 users. We observed that the Group A 95th percentile was 0.82 seconds, which was less than the threshold value of 1.5 seconds. We systematically increased the number of simulated desktops and looked at the QoS score. When the number of users was increased to 124, we could no longer satisfy the latency QoS threshold. Consequently, this particular configuration can only support approximately 120 users.

TOTAL # VMS	GROUP A 95% (SEC)	QOS STATUS
96	0.82	passed
112	0.93	passed
120	1.43	passed
124	1.54	failed
136	4.15	failed

Table 2: QoS score for different numbers of users

C. Comparing Different Hardware Configurations

View Planner can be used to compare the performance of different platforms:

- Storage protocols, such as the Network File System (NFS), Fibre Channel, and Internet SCSI (iSCSI) protocols
- Processor architectures, such as Intel Nehalem, Intel Westmere, and so on
- Hardware platform configuration settings, such as CPU frequency settings, memory settings, and so on

To demonstrate one such use case, we evaluated the memory over-commitment feature in VMware vSphere [15]. Table 3 shows the 95th percentile QoS threshold of View Planner with different percentages of memory over-commitment. The results show that even with 200 percent memory over-commitment, the system passed the QoS metric of 1.5 seconds.

VIRTUAL MACHINES/ HOST	LOGICAL MEM	PHYSICAL MEM	%MEM OVERCOMMIT	95TH PERCENTILE LATENCY
30	30GB	20GB	50%	0.59
40	40GB	20GB	100%	0.74
60	60GB	20GB	200%	0.86

Table 3: QoS with different memory over-commitment settings

D. Comparing Different Display Protocols

To compare different display protocols, we need to precisely characterize the response time of application operations. This is a capability of our watermarking technique. We simulated different network conditions—using LAN, WAN, and extreme WAN (very low bandwidth, high latency)—to see how the response time increased for different display protocols. Figure 10 shows the normalized response time chart comparing the PC-over-IP (PCoIP), PortICA, and RDP display protocols for three network conditions. These results show that PCoIP provides much better response time in all network conditions compared to other protocols. View Planner enables this kind of study and provides a platform to characterize the "true" end-user experience.

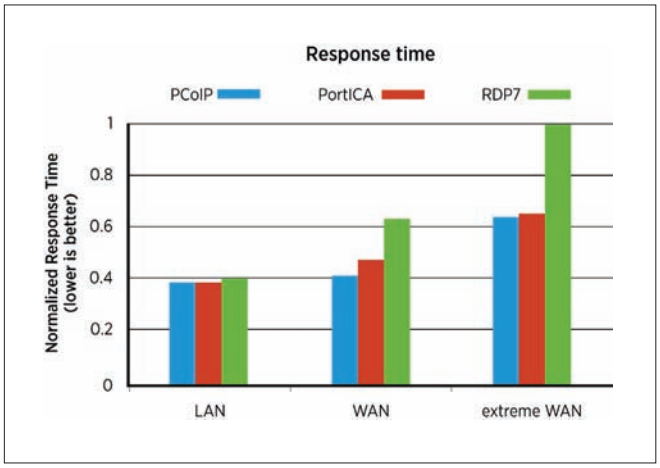

Figure 10. The normalized View Planner response time for diferent display protocols for different network conditions

E. Performance Characterization

View Planner can be used for performance characterization. To illustrate one study, we investigated the differing numbers of users a given platform could support using different versions of VMware View™. Figure 11 shows the 95th percentile response time for VMware View 4.5 and 5.0. We set threshold of 1.5 seconds and required the 95th percentile response time to fall below this threshold. For VMware View 4.5, the response time threshold crossed this threshold at 12 virtual machines (or 12 users) per physical CPU core. Hence, we can support between 11 to 12 users per core on VMware View 4.5. Looking at VMware View 5 result, we see it can easily support 14.5 Windows 7 virtual machines per core. Using View Planner, we were able to characterize the number of users that can be supported on a physical CPU core, and compare two versions of a product to analyze performance improvements (30 percent better consolidation in VMware View 5 compared to VMware View 4.5).

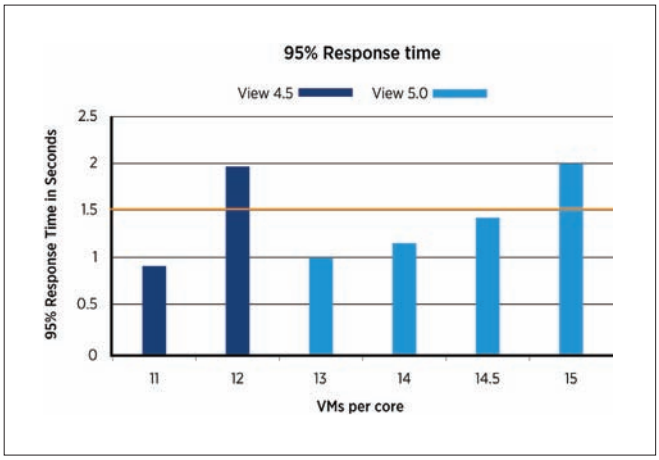

Figure 11. 95th percentile response time for VMware View 4.5 and VMware View 5 as the number of virtual machines per CPU core was increased.

F. Performance Optimizations

View Planner can help users find bottlenecks in the virtualization layer during a scalability study and apply performance optimizations. Using this tool, we can pinpoint the performance bottleneck at a particular component and try to fix the problem. For example, in a particular run, we found that application latencies (user experience) were poor. Upon further analysis, we traced the problem to the storage array, where disk latency was quite high and available I/O bandwidth was fully saturated. We also can study the performance of many protocol features and understand their impact on overall end-user experience. In addition, we identified many performance issues in the CPU and network (downlink bandwidth) usage in various applications during our protocol performance analysis, highlighting the significant potential of this workload in VDI environments.

6. Related Work

There are a number of companies providing VDI test solutions. Some, such as View Planner, focus on the entire VDI deployment [16, 17], while others offer limited scope and focus on a specific aspect of a VDI solution, such as storage [18]. At a high level, the functionality provided by these solutions might appear similar to View Planner at first glance. However, these solutions do not leverage watermarking techniques to accurately gauge the operation latency that is experienced by an end-user. Instead, they rely on "out-of-band" techniques to estimate remote response. For instance, out-out-band techniques include explicitly communicating event start and stop events to the client using separate artificially created events. In this situation, the start and stop events, unlike our watermarking technique, do not piggyback on the remote display channel and may not accurately reflect the operation latency observed by a user. Other approaches involve network layer taps to attempt to detect the initiation and completion of operations. Not only are these approaches potentially inaccurate, they introduce significant complexities that limit portability between operating systems and VDI solutions.

The out-of-band signaling exploited by other VDI test solutions can lead to significant inaccuracies in the results, which in turn c an lead to misleading conclusions about permissible consolidation ratios, protocol comparisons, and result in invalid analysis of VDI deployments. Other approaches include analyzing screen updates and attempting to automatically detect pertinent events (typically used for comparative performance analysis) [19], and inferring remote latencies by network monitoring and slow-motion benchmarking [20, 21]. While these approaches work for a single VDI user on an unloaded system, they can significantly perturb (and are perturbed by) the behavior of VDI protocols under load, making them unsuitable for the robust analysis of realistic VDI workloads at scale.

Finally, a variety of other techniques have been developed for latency analysis and to look for pauses due to events such as garbage collection [22]. These approaches assume (and depend on the fact) the user is running on a local desktop.

7. Conclusion

This paper presented View Planner, a next-generation workload generator and performance characterization tool for large-scale VDI deployments. It supports both types of operations (user and administrative operations) and is designed to be configurable to allow users to accurately represent their particular VDI deployment. A watermarking measurement technique was described that can be used in a novel manner to precisely characterize application response time from the client side. For this technique, watermarks are sent with the display screen as notifications and are piggybacked on the existing display. The detailed architecture of View Planner was described, as well as challenges in building the representative VDI workload, and scalability features. Workload characterization techniques illustrated how View Planner can be used to perform important analysis, such as finding the number of supported users on a given platform, evaluation of memory over-commitment, and identifying performance bottlenecks. Using View Planner, IT administrators can easily perform platform characterization, determine user consolidation, perform necessary capacity planning, undertake performance optimizations, and a variety of other important analyses. We believe View Planner can help VDI administrators to perform scalability studies of nearly real-world VDI deployments and gather useful information about their deployments.

Acknowledgement

We would like to thank Ken Barr for his comments and feedback on the early drafts of this paper, as well as other reviewers. Finally, we thank everyone in the VDI performance team for their direct or indirect contributions to the View Planner tool.

Any further information and details about the View Planner tool can be achieved by sending an email to viewplanner-info@vmware.com

References

1 B. Agrawal, L. Spracklen, S. Satnur, R. Bidarkar, "VMware View 5.0 Performance and Best Practices", VMware White Paper, 2011.

2 L. Spracklen, B. Agrawal, R. Bidarkar, H. Sivaraman, "Comprehensive User Experience Monitoring", in VMware Tech Journal, March 2012.

3 VMware Inc., Timekeeping in VMware Virtual Machines, http://www.vmware.com/vmtn/resources/238

4 Python Twisted Framework, http://twistedmatrix.com/trac/

5 Google web toolkit (GWT), http://code.google.com/webtoolkit/

6 Sturdevant, Cameron. VMware View 4.5 is a VDI pacesetter, eWeek Vol. 27, no. 20, pp. 16-18. 15 Nov 2010.

7 VMware View: Desktop Virtualization and Desktop Management www.vmware.com/products/view/

8 Citrix Inc., Citrix XenDesktop 5.0. http://www.citrix.com/English/ps2/products/feature.asp?contentID=2300341

9 J. Nieh, S. J. Yang, and N. Novik, "Measuring Thin-Client Performance Using Slow-Motion Benchmarking", ACM Trans. Comp. Sys., 21:87–115, Feb. 2003.

10 S. Yang, J. Nieh, M. Selsky, N. Tiwari, "The Performance of Remote Display Mechanisms for Thin-Client Computing", Proc. of the USENIX Annual Technical Conference, 2002.

11 VI SDK. https://www.vmware.com/support/developer/vc-sdk/

12 AutoIT documentation http://www.autoitscript.com/autoit3/docs/

13 Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, Joonwon Lee, Task-aware virtual machine scheduling for I/O performance., Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, March 11-13, 2009, Washington, DC, USA

14 Micah Dowty, Jeremy Sugerman, GPU virtualization on VMware's hosted I/O architecture, ACM SIGOPS Operating Systems Review, v.43 n.3, July 2009

15 VMware White Paper. "Understanding Memory Resource Management in VMware® ESX™ Server ", (2010).

16 LoginVSI—Admin Guide of Login Consultants "Virtual Session Indexer" 3.0, http://www.loginvsi.com/en/admin-guide

17 Scapa Test and Performance Platform, http://www.scapatech.com/wpcontent/uploads/2011/04/ScapaTPP_VDI_0411_web.pdf

18 VDI-IOmark, http://vdi-iomark.org/content/resources

19 N. Zeldovich and R. Chandra, "Interactive performance measurement with VNCplay", Proceedings of the USENIX Annual Technical Conference, 2005.

20 A. Lai, "On the Performance of Wide-Area Thin-Client Computing", ACM Transactions on Computer Systems, May 2006.

21 J. Nieh, S. J. Yang, N. Novik, "Measuring Thin-Client Performance Using Slow-Motion Benchmarking", ACM Transactions on Computer Systems, February 2003.

22 A. Adamoli, M. Jovic and M. Hauswirth, "LagAlyzer: A latency profile analysis and visualization tool", International Symposium on Performance Analysis of Systems and Software, 2010.

vSOM: A Framework for Virtual Machine-centric Analysis of End-to-End Storage IO Operations

Sandeep Uttamchandani
VMware, Inc.
suttamchandani@vmware.com

Wenhua Liu
VMware, Inc.
liuw@vmware.com

Samdeep Nayak
VMware, Inc.
samdeep@vmware.com

Abstract

Diagnosis of an I/O performance slow down is a complex problem. The root cause could be one among a plethora of event combinations such as VMware® ESXi misconfiguration, an overloaded fabric switch, disk failures on the storage arrays, and so on. As a virtualization administrator, diagnosing the end-to-end I/O path today requires working with discrete fabric and storage reporting tools, and manually correlating component statistics to find performance abnormalities and root-cause events. To address this pain point, especially for cloud scale deployments, we developed the VMWare SAN Operations Manager (vSOM), a framework for end-to-end monitoring, correlation, and analysis of storage I/O paths. It aggregates events, statistics, and configuration details across the ESXi server, host bus adapters (HBAs), fabric switches, and storage arrays. The correlated monitoring data is analyzed in a continuous fashion, with alerts generated for administrators. The current version invokes simple remediation steps, such as link and device resets, to potentially fix errors such as link errors, frame drops, I/O hangs, and so on. vSOM is designed to be leveraged by advanced analytical tools. One example described in this paper, VMware® vCenter™ Operations Manager™, uses vSOM data to provide end-to-end virtual machine-centric health analytics.

1. Introduction

Consider an application administrator responding to multiple problem tickets: "The enterprise e-mail service has a 40-60 percent higher response time compared to its average response time over the last month." Because the e-mail service is virtualized, the administrator starts by analyzing virtual machines, manually mapping the storage paths and associated logical and physical devices. Today, there is no single tool to assist with the complete end-to-end diagnosis of the Storage Area Network (SAN), starting with the virtual machine, through the HBAs, switches, storage array ports and disks (Figure 1). In large enterprises, the problem is aggravated further with specialized storage administrators performing isolated diagnosis at the storage layer: "The disks look fine; I/O rate for e-mail related volumes has increased, and the response time seems within normal bounds." This to-and-fro between application and storage administrators can take weeks to resolve. The real root cause could be accidental rezoning of the server ports, combined with a change in the storage array configuration that labeled those ports as "low priority traffic."

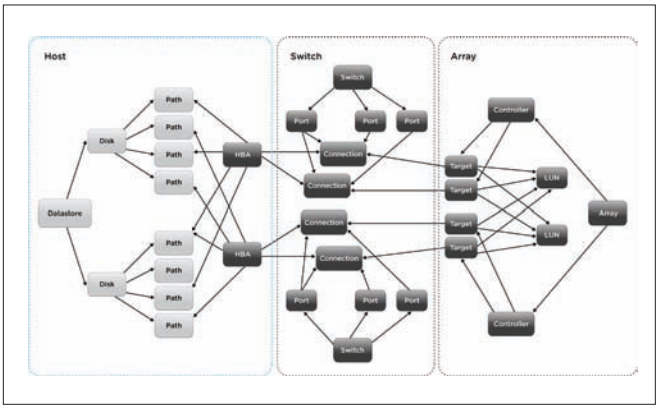

Figure 1: Illustration of the Real-world End-to-end I/O Path in a Virtualization Environment

In summary, end-to-end analysis in a SAN environment is a complex task, even in physical environments. Virtualization makes the analysis even more complex, given the multiplexing of virtual machines on the same physical resources.

This paper describes vSOM, the VMWare SAN Operations Manager, a general-purpose framework for heterogeneous, cloud-scale SAN deployments. vSOM provides a virtual machine-centric framework for end-to-end monitoring, correlation, and analysis across the I/O path including SAN components, namely HBAs, switches, and storage arrays. vSOM is designed to provide correlated monitoring statistics to radically simplify diagnosis, troubleshooting, provisioning, planning, and infrastructure optimization. In contrast, existing tools [1, 2, 3, 4, 5] are discrete in monitoring one or more components, falling short of the end-to-end stack. vSOM internally creates a correlation graph, mapping the logical and physical infrastructure elements, including virtual disks (VMDK), HBA ports, switch ports, array ports, logical disks, and even physical disk details if exposed by the controller. The elements in the correlation graph are monitored continuously, aggregating both statistical metrics and events.

Developing a cloud-scale end-to-end I/O analysis framework is nontrivial. The following are some of the design challenges that vSOM addresses:

- **Heterogeneity of fabric and storage components**: There is no single available standard that is universally supported for out-of-band management of fabric and storage components. SNIA's

Storage Management Interface Specifications (SMI-S)[7] has been adopted by a partial subset of key vendors. Leveraging the VMware vSphere® Storage APIs for Storage Awareness (VASA) [10], provides a uniform syntactic and semantic interface to query storage devices, but is not supported by fabric vendors.

- **Scalability of the monitoring framework**: The current version of vSphere supports 512 virtual machines per host, 60 VMDKs per virtual machine, and 2,000 VMDKs per host. The ratio of VMDKs to physical storage LUNs typically is quite large. vSOM needs to monitor and analyze a relatively large corpus of monitored data to notify administrators for hardware saturation, anomalous behavior, correlated failure events, and so on.
- **Continuous refinement for configuration changes**: In a virtualized environment, the end-to-end configuration is not static. It evolves with events, such as VMware vSphere vMotion®, where either a virtual machine moves to different server, associated storage relocates, or both. The analysis of monitoring data needs to take into account the temporal nature of the configuration and appropriately correlate performance anomalies with configuration changes.

vSOM employs several interesting techniques, summarized as the key contributions of this paper:

1. Discovery and monitoring of the end-to-end I/O path that consists of several heterogeneous fabric and storage components. vSOM stiches together statistical metrics and events using a mix of standards and propriety APIs.
2. Correlation of the configuration details is represented internally as a directed acyclic dependency graph. The edges in the graph have a weight, representing I/O traffic between the source and destination vertices. The graph is self-evolving and updated based on configuration events and I/O load changes.
3. Analysis of statistics and events to provide basic guidance regarding the health of the virtual machine based on health of the I/O path components. Additionally, vSOM plugs into a richer set of analytics, planning, and trending capabilities of VMWare's Operations Manager.

The rest of the paper is organized as follows: Section 2 gives a bird's eye-view of vSOM. Sections 3, 4, and 5 cover details of the monitoring, correlation, and analysis modules respectively. The conclusion and future work are summarized in Section 6.

2. A Bird's Eye View of vSOM

The objective of vSOM is to monitor, correlate, and analyze the health of the virtual machine, as a function of the SAN I/O components (HBA, switches, and storage array). This section describes the system model and a high-level overview of the vSOM architecture.

2.1 System Model

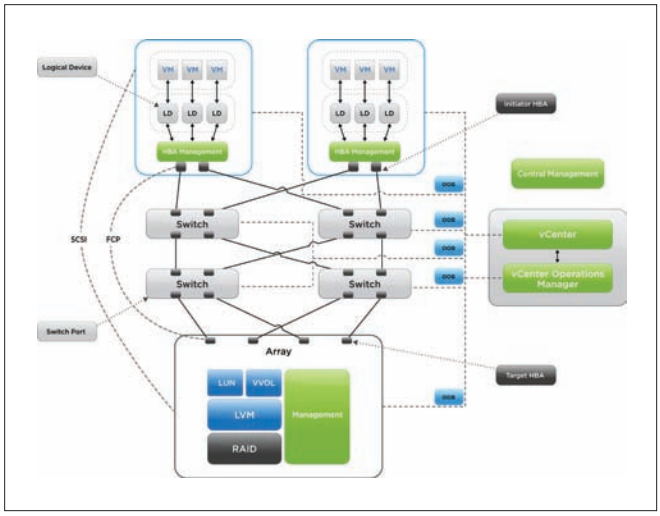

Figure 2: vSOM System Model

The overall system model is shown in Figure 2. The hypervisor abstracts physical storage LUNs (also referred to as Datastore), and exports Virtual Disks (VMDKs) to virtual machines. The hypervisor supports a broad variety of storage protocols, such as Fibre Channel (FC), Internet SCSI (iSCSI), Fibre Channel over Ethernet (FCoE), the Network File System (NFS), and so on. A VMDK is used by the Guest operating system either as a raw physical device abstraction (referred to as Raw Device Mapping or RDM), or a logical volume derived from VMFS or a NFS mount point. RDMs are not a common use-case, since they bypass the hypervisor I/O stack and key features of the hypervisor such as vMotion, resource scheduling, and so on, cannot be used. As illustrated in Figure 2, a typical end-to-end I/O path from the virtual machine to physical storage consists of Virtual machine → VMDK → HBA Port → Switch Port →Array Port → Array LUN → Physical device. The current version of vSOM supports block devices only. NFS volumes are not supported.

Multiple industry-wide efforts try to standardize the management of HBAs, switches, and storage arrays. The most popular and widely adopted standard is SNIA's Storage Management Initiative Specifications (SMI-S)[7]. The standard defines profiles, methods, functions, and interfaces for monitoring and configuring system components. This standard is widely adopted by switch and HBA vendors, with limited adoption by storage array vendors. The standard is built on the Common Information Model (CIM)[8] that defines the architecture to query and interface with system management components. In the context of CIM, a CIM Object Manager (CIMOM) implements the management interface, accessible locally or through a remote connection.

2.2 vSOM Overview

vSOM tracks the end-to-end I/O path and collects data across ESXi hosts, VMware Virtual Center™, fabrics, and storage arrays. component is monitored continuously to collect configuration details, performance statistics, and events. Data collected from the components is correlated to create a virtual machine-centric analysis including VMDKs, HBAs, switches, and storage arrays. The preciseness of the end-to-end correlation depends on the configuration. For a virtual machine with a raw mapped VMDK, there is a one-to-one mapping between VMDK and the physical LUN—the statistics gathered from the LUN and HBA paths can be attributed directly to the virtual machine. Conversely, with a virtual machine using VMDKs carved on a VMFS volume, the statistics of the storage array LUN and HBA paths reflect the status of a set of virtual machines sharing the LUN.

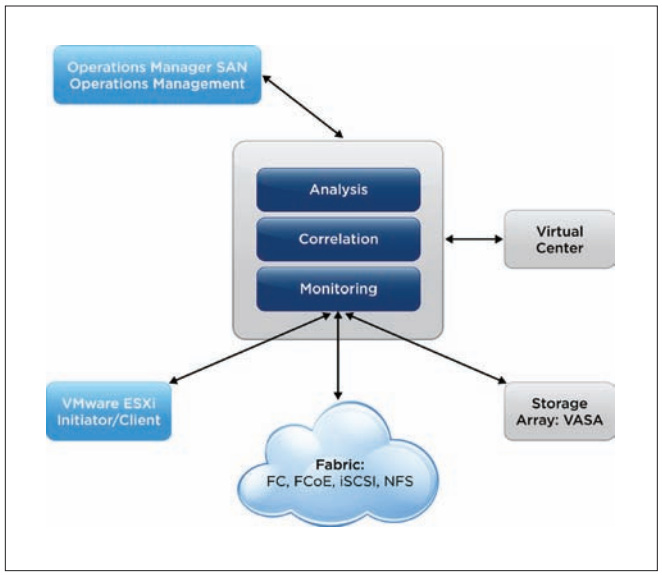

Figure 3: High-level vSOM Architectural Modules

The vSOM architecture consists of three key building blocks: Monitoring, Correlation, and Analysis Modules (Figure 3).

- **Monitoring Module**: The monitoring framework consists of Agents and a centralized Management Station. Agents collect statistics and events on the components. The initial component discovery uses a combination of vSphere configuration details from vCenter, combined with the Service Location Protocol (SLP)[11]. Agents collect monitoring data in real time, using a combination of SMI-S standards and vSphere APIs. The Management Station aggregates data collected by the individual component agents, and internally uses different protocols to connect with the host, switch, and storage array agents.
- **Correlation Module**: The configuration details collected from the component agents are used to create an end-to-end dependency graph. The graph is updated continuously for events such as vMotion, Storage vMotion, Storage DRS, HA failover, and so on. An update to the I/O path configuration is tracked with a unique Configuration ID (CID). The historical statistical data from each component is persisted by tagging with the corresponding CID. This enables statistical anomaly detection and the effect of changes to the configuration.
- **Analysis Module**: The monitored data is analyzed to determine the health of individual components. Statistical anomaly analysis of monitored data can be absolute or relative to other components. The current version of vSOM also supports rudimentary remediation actions that are triggered when an erroneous pattern is observed over a period of time, such as an increased number of loss sync or parity errors. vSOM plugs into Operations Manager, and provides data for end-to-end virtual machine-centric analysis.

3. Monitoring Module

As mentioned earlier, the Monitoring module consists of Agents and the centralized Management Station. Agents implement different mechanisms to collect data. The ESXi agent uses CIM, fabric agents uses SMI-S, storage array agents use either SMI-S or VMWare's API for Storage Awareness (VASA). Besides the agents, the Management Station also communicates with vCenter Server to collect event details.

The monitoring details collected from Agents are represented internally as software objects. The schema of these objects leverages CIM-defined profiles wherever possible. The objects are persisted by the Management Station using a circular buffer implementation. The size of the circular buffer is configurable, and corresponds to the amount of history. Component monitoring is near-real-time, with the monitoring interval typically being 30-60 seconds.

This section describes the steps involved in the monitoring bootstrapping process, as well as details of the internal software object representation. vSOM implements a specialized CIM-based agent for ESXi hosts, and this section covers the key implementation details.

3.1 Bootstrapping Process

Bootstrapping involves the discovery of entities associated with a given virtual machine. This is accomplished using a combination of vSphere configuration analysis and the Service Location Protocol (SLP). The steps involved in the discovery process are summarized as follows:

1. vSOM queries the vCenter Server to identify all hosts and datastores that host active virtual machines and VMDKs, respectively. Querying the vCenter Server acts as white-box knowledge, limiting the search space and enabling faster convergence. In contrast, black-box discovery of all devices with a vSphere cluster and SAN setup would be much slower to complete.
2. For each VMDK, the ESXi CIM provider is queried to provide the logical device details.
3. Using the logical device details, vCenter is queried to retrieve the associated storage port IDs at the host and storage array (commonly referred to as initiator and target ports). Each port is uniquely identified with a World Wide ID (WWID). At the end of this step, for each VMDK, the corresponding initiator and target port WWIDs are discovered. For VMDKs mapped on the same logical device (datastore), the initiator and target ports are the same.

4. The following schemes are adopted to discover the fabric topology, depending on whether the storage protocol is FC, FCoE, or iSCSI.

    a. FC and FCoE fabrics implement CIM. vSOM uses SLP to discover the CIMOM for each switch, followed by validation of support for the SMI-S profile. If the SMI-S profile is supported by the CIMOM, vSOM queries the switch ports associated with the initiator and target WWID and identifies any inter-wwitch links that might be connected between the host and the target.

    b. For iSCSI, vSOM uses fast traceroute to identify the fabric topology between the host and the target. vSOM queries the individual port details using the Simple Network Management Protocol (SNMP).

3.2 End-to-end I/O Monitoring

The end-to-end I/O path is represented as a combination of Host, Switch, and Array objects. Each object stores a combination of performance statistics and events.

3.2.1 Host Object

The Host object includes monitoring of the VMDKs, SCSI disks, and HBA. The host object is exported as a CIM profile, accessed by the central Management Station. Instead of defining a new data model, the Host object follows the SMI-S *Block Server Performance Subprofile* [7]. The profile defines classes and methods for managing performance information, and was originally designed for storage arrays, virtualization engines, and volume managers. In designing the data model (Figure 4),

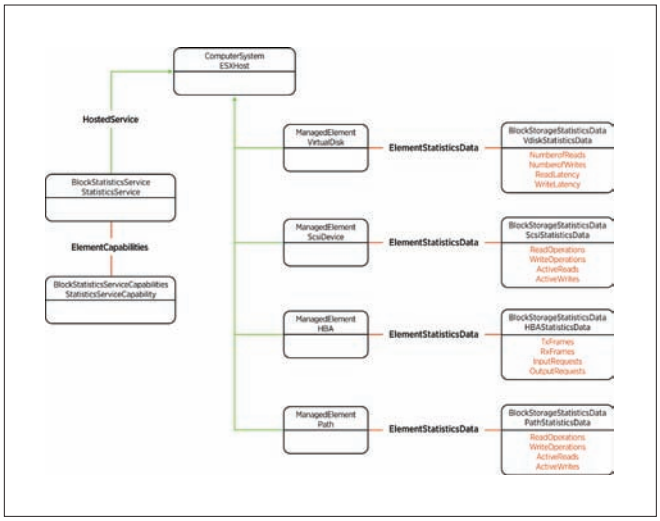

Figure 4: Data Model for ESXi Host Object Derived from the SMI-S Block Server Performance Sub-profile

we observed a one-to-one mapping between the abstractions on an ESXi server and a storage array: Virtual machines within ESXi are equivalent to hosts of a storage array. VMDKs are equivalent to LUNs exported by a storage array. SCSI devices on the host are similar to physical drives on an array, HBAs as the back-end ports, datastores as the storage pools of a storage array. In other words, a Block Server Performance Subprofile maps well with vSOM requirements. The Host CIM provider implements only a subset of the classes, associations, properties, and methods of the profile, as required for the vSOM monitoring framework. For the HBA object, the data collected includes generic SCSI performance data and transport-specific performance data.

3.2.2 Switch Object

The Switch object consists of a collection of ports across one or more switches that are in the I/O path that serves the virtual machine's storage traffic. Major SAN switch vendors have implemented SMI-S compliant CIM providers. vSOM uses these CIM providers as data collection agents. In the context of storage, switches typically use Fibre Channel or standard Ethernet. For each FC port, vSOM switch objects use the data model defined in CIM schema 2.24 (CIM_FCPortRateStatistics and CIM_FCPortStatistics). In SCSI, only Class 3 service is used on Fibre Channel. As a result, the attributes in the CIM profile containing Class 1 or Class 2 are ignored. Performance data is collected and persisted only for the ports on which ESX hosts or storage arrays are connected.

3.2.3 Storage Array Object

The Storage Array is the final destination of the I/O, and is referred to as the Target. A typical storage array consists of array ports, controllers, and logical volumes. Additionally, storage arrays can export details on physical disks and back-end storage ports. In vSOM, two mechanisms are available to collect data from the storage array: VMWare's VASA profile or the CIM storage profile. The latter provides a limited set of attributes, generalizing the differentiated capabilities of the storage arrays. The Management Station connects with the Array agents using the WSDL-based protocol[9] for VASA or a Generic Storage Adapter for CIM. Note that for the purpose of vSOM, the existing VASA specifications have been extended with certain performance attributes.

3.3 Internals for Data Collection from Host

As a part of the vSOM initiative, the ESXi host has been extended with I/O Device Management (IODM). This module provides functionality to configure, monitor, and deliver Storage object I/O details to the vSOM Management Station. This subsection describes details of the IODM implementation within ESXi (Figure 5).

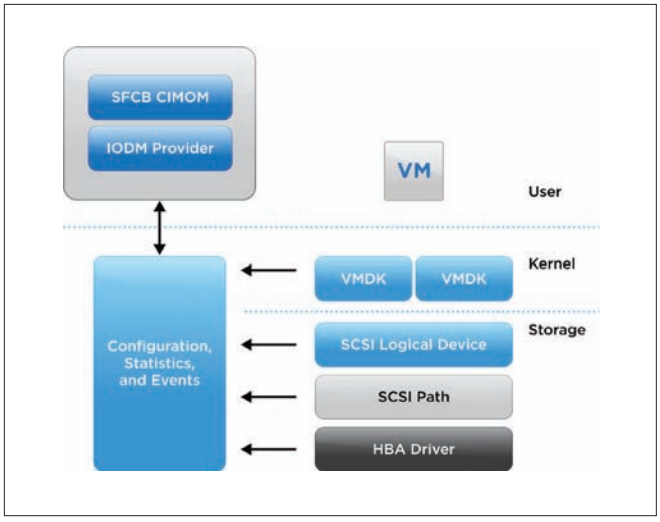

Figure 5: I/O Device Management (IODM) Implementation for ESXi

IODM implementation is split into two parts: an IODM Kernel Module and an IODM CIM Provider for the user world. The kernel module captures statistics and events about VMDKs, SCSI logical devices, SCSI paths, and HBA transport details. The kernel module also presents an asynchronous mechanism to deliver events to the user world.

The IODM CIM provider implemented consists of two parts: Upper layer and Bottom layer. The Upper layer is the standard CIM interface, with both the intrinsic and extrinsic interfaces implemented. The intrinsic interface includes **EnumInstances**, **EnumInstanceNames**, and **GetInstance**. This interface is used to get I/O statistics and error information for virtual machines, devices, and HBAs. The extrinsic interface controls the IODM behavior with functions such as start/stop data collection. CIM indication for events also is part of the Upper layer. It interacts with the IODM kernel modules to get events and alerts.

4. Correlation Module

The Correlation Module maintains the dependency between the components in the I/O path. The dependency details are persisted as an acyclic directed graph. Vertices represent the I/O path components, and edges represent the correlation weight, as illustrated in Figure 6. The steps involved in discovering correlation details between the components is similar to the bootstrapping process covered in Section 3.1.

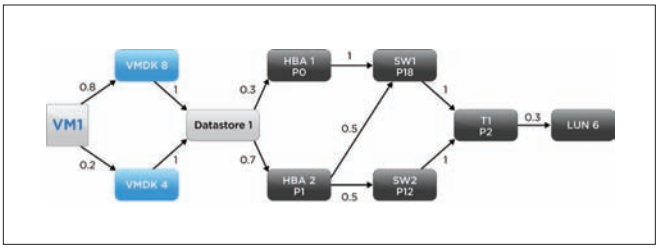

Figure 6: Representation of the Correlation between I/O Path Components

Figure 6 shows the correlation details between VM1 and LUN 6. VM1 has configured VMDK 4 and VMDK 8 as storage volumes. These VMDKs are mapped to logical SCSI device (Datastore 1). The Datastore is connected to HBA 1 and HBA 2 on ports P0 and P1 respectively. These ports are mapped to Switches 1 and 2 (SW1 and SW2) on ports 18 (P18) and 12 (P12), respectively. Finally, the switches connect to the Storage Array Target (T1 on Port 2), accessing LUN 6. With respect to correlation granularity, VMDK 4 and VMDK 8 merge into Datastore 1. The outgoing traffic from the Datastore can be a combination of other VMDKs as well (in addition to VMDK 4 and 8). The weights for an edge are normalized to a value between 0 and 1. The summation of the outgoing edges from a vertex should typically be 1. Note that this might not always be the case. For instance, the traffic from Target T1 Port 2 is mapped to other LUNs besides LUN 6. As a result, the total of the outgoing edges from T1 P2 is shown in Figure 6 as 0.3.

vSOM continuously monitors for configuration changes and updates the dependency graph. Each version of the configuration is tracked using a unique 32-byte Configuration ID. Maintaining the versions of the dependency graph help in the time travel analysis of configuration changes, and their corresponding impact on configuration changes. As mentioned earlier, the performance statistics are tagged with the Configuration ID.

The dependency graph is updated in response to either configuration change events or updates to the edge weights in response to workload variations. Configuration change events such as vMotion and HA, among others, are tracked from vCenter, while CIM indications from individual components indicate the creation, deletion, and other operational status change of switches, FC ports, and other components. The edge weights in the dependency graph are maintained as a moving average and are updated over longer time windows (3-6 hours).

5. Analysis Module

The goal of the Analysis module is to use monitoring and correlation details to provide an intuitive representation of virtual machine health as a function of the heath of the individual components (Host machine, HBAs, Fabric, and Storage Arrays). The Analysis module categorizes the health of each component into green, yellow, orange, or red:

- Green indicates normal, with the component behaving within expected thresholds
- Yellow means attention is needed, primarily based on reported error events
- Orange indicates an increasingly degrading condition, based on a combination of statistical analysis and error events
- Red means I/O can no longer flow and virtual machines cannot operate

Based on the dependency graph, the health of individual components is rolled up at the virtual machine level (Figure 7).

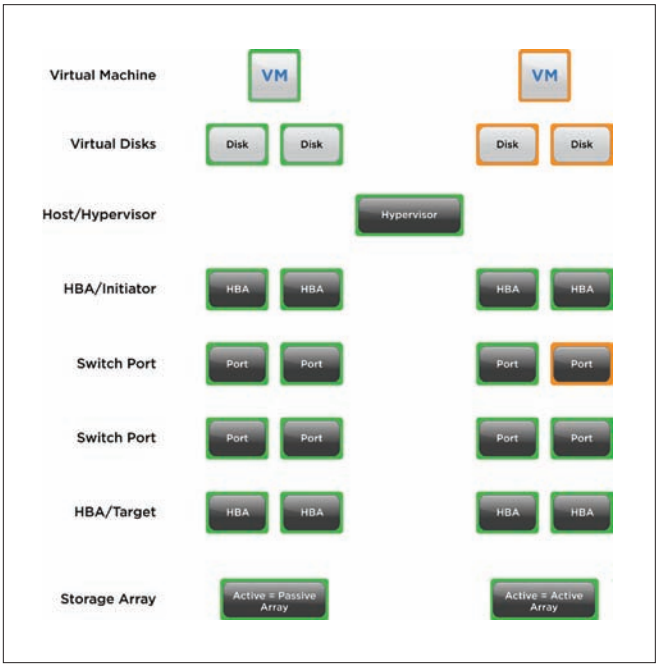

Figure 7: Virtual Machine Health is a Cumulative Roll Up of Individual Components in the End-to-end Path

As shown, degradation in the Switch Port affects the virtual machine, based on its high correlation weight for host to storage array connectivity.

The health of a component can be deduced using different approaches. Traditionally, administrators are expected to define alert thresholds. For example, when capacity reaches 70 percent, the health of the disk is marked yellow or orange. Defining these thresholds typically are nontrivial and arbitrary. Further, given the scale of cloud systems, it is unrealistic for administrators to define these thresholds.

vSOM employs two different techniques to determine component health. The first approach is referred to as Absolute Anomaly Analysis. The history of monitored data is analyzed to determine whether current component performance is anomalous. There are standard data mining techniques for anomaly detection. vSOM uses the K-means Clustering approach that detects an anomaly and associates a weight to help categorize the anomaly as yellow, orange, or red. The second approach is based on Relative Analysis. In this approach, peer components (such as ports on the same switch, or events on different ports of the same HBA) are analyzed to determine if the observed behavior anomaly is similar to other components. In large-scale deployments, Relative Analysis is an effective approach, especially if the available history of monitored data is not sufficient for Absolute Anomaly Analysis.

Analysis can help pinpoint the root cause of the problem and be used to trigger automated remediation. Automated root-cause analysis is nontrivial, especially in large-scale, real-world deployments where the cause and effect might not always be on the same component, or the problem might be a result of multiple correlated events. vSOM implements a limited version of auto remediation, using link or device resets. Complex remediation actions, such as changing the I/O path or vMotion, is beyond the scope of the current version of vSOM.

Reset is an effective correction action for a common set of link-level and device-level erroneous patterns. For errors observed over a period of time, such as an increased number of loss sync or parity errors, failure of protocol handshaking, and so on, are commonly fixed with a link reset. A link reset can reinitialize the link and put I/O back on track. If several link resets do not fix the problem, the path can be disabled to trigger a path failover to a backup path if multiple paths are available.

vSOM correlates events from components in the end-to-end path. This helps in determining the root cause of events such as link down, frame drops, I/O or virtual machine hangs, and similar events. For instance, link down events are collected from the ESX host by subscribing to CIM indications, helping to isolate the root cause on the ESX host versus a specific HBA or switch port.

The vCenter Operations Manager is an existing VMware product that provides rich analytical capabilities for managing performance and capacity for virtual and physical infrastructures. It provides analytics for performance troubleshooting, diagnosis, capacity planning, trending, and so on. Using advanced statistics analysis, Operations Manager currently associates a health score to resources such as compute, and continuously tracks the health to raise alerts for abnormal behavior, sometimes even before a problem exhibits any symptoms. vSOM plugs into Operations Manager using its standard adapter interface. vSOM complements the existing analysis of Operations Manager for virtual machine and datastore level, with details of the SAN components (HBAs, Fabric, and Storage Array). Operations Manager stores historic statistical data in a specialized database and implements anomaly detection algorithms for historic data analysis. In addition to end-to-end monitoring and troubleshooting, Operations Manager can help with planning and optimization use cases, such as balancing workloads across all controllers and switch ports.

6. Conclusion and Future Work

This paper describes an end-to-end SAN management framework implemented for vSphere. It addresses the pain points associated with monitoring I/O components from the viewpoint of virtual machine-centric performance. While problem diagnosis is the most intuitive use case, vSOM is applicable to other use cases, such as planning, single-pane of glass monitoring, load balancing, and more.

We plan to extend this work in several dimensions. Automated remediation has significant value for administrators. We plan to extend beyond the current reset action to support complex multistep actions. For root-cause analysis, we plan to combine our current black-box anomaly analysis with rule-based techniques, particularly to correlate error events across different components in the I/O path. Finally, we are exploring monitoring module extensions to include application-level statistics in the end-to-end I/O path.

References

1 HP Systems Insight Manager Overview, http://h18013.www1.hp.com/products/servers/management/hpsim/index.html?jumpid=go/hpsim

2 Dell OpenManage Systems Management, http://www.dell.com/content/topics/global.aspx/sitelets/solutions/management/en/openmanage?c=us&l=en&cs=555

3 IBM Tivoli Storage Management Solutions, http://www-01.ibm.com/software/tivoli/solutions/storage/

4 Shen, K., Zhong, M., and Li, C. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies (FAST) (2005), pp. 23–23.

5 Pollack, K. T., and Uttamchandani, S. Genesis: A Scalable Self-Evolving Performance Management Framework for Storage Systems. In IEEE International Conference on Distributed Computing Systems (ICDCS) (2006), p. 33.

6 VMWare vCenter Operations Manager, http://www.vmware.com/support/pubs/vcops-pubs.html

7 Storage Management Initiative Specification (SMI-S), http://www.snia.org/tech_activities/standards/curr_standards/smi

8 Common Information Model, http://en.wikipedia.org/wiki/Common_Information_Model_(computing)

9 Web Services Description Language (WSDL), http://www.w3.org/TR/wsdl

10 vSphere Storage API for Storage Awareness (VASA), http://blogs.vmware.com/vsphere/2011/08/vsphere-50-storage-features-part-10-vasa-vsphere-storage-apis-storage-awareness.html

11 Service Location Protocol, http://www.ietf.org/rfc/rfc2608.txt

It's my pleasure to introduce the second volume of the VMware Technical Journal. As VMware CEO and throughout my career I have emphasized the importance of engagement between industry and academia as an integral part of the "golden triangle" of innovation—organic innovation within dynamic companies alongside innovative research developments with universities and the passion of start-up environments. This volume of the Journal highlights how VMware taps into this approach, with many voices coming together to demonstrate the relationship between our academic research collaboration, internship programs and organic R&D efforts.

The journal begins with perspectives shared by collaborators at Georgia Institute of Technology, Carnegie Mellon University and Ohio State University. Their contributions highlight research projects supported by VMware through a combination of financial sponsorship, software donations and participation by our engineers. VMware-sponsored projects such as these span multiple focus areas, such as cluster scheduling, performance monitoring and security in cloud and virtualized environments. We continue to identify new research opportunities, and our next RFP scheduled for Spring 2013 will provide a great opportunity for a number of researchers to begin new collaborative projects.

We then feature three papers written by past interns, describing the breadth of projects undertaken by these students while at VMware. Our internship program is a highly structured opportunity for outstanding students to work on a specially defined project, which often leads to significant output such as a research publication or product enhancement. The program fosters long-term relationships, with many interns participating in multiple internships and frequently joining VMware upon graduation. Our PhD interns often integrate their summer projects into their PhD relationships, which in turn contribute to our ongoing academic partnerships—an example of the golden triangle at work!

We close this edition with contributions from a number of VMware engineers. These perspectives share how innovation occurs in R&D, whether providing a historical perspective on a key component of VMware's development infrastructure - FrobOS - or describing a new idea for leveraging social networking to efficiently manage large datacenters. Many of our engineers act as mentors to interns and also participate in research partnerships, thus reinforcing the relationship between every aspect of innovation.

We hope you enjoy this edition as much as we've enjoyed the collaboration and impact highlighted in the stories we share with you here. We welcome your comments and ideas for future articles—keep the input and innovation coming!

Pat Gelsinger
CEO, VMware

VMLab: Infrastructure to Support Desktop Virtualization Experiments for Research and Education

Prasad Calyam
The Ohio State University
pcalyam@oar.net

Alex Berryman
The Ohio State University
berryman@oar.net

Albert Lai
The Ohio State University
albert.lai@osumc.edu

Matthew Honigford
VMware, Inc.
mhonigford@vmware.com

Abstract

In terms of convenience and cost-savings, user communities have benefited from transitioning to virtual desktop clouds (VDCs) that are accessible via thin-clients, moving away from dedicated hardware and software in "traditional desktops". Allocating and managing VDC resources in a scalable and cost-effective manner poses unique challenges to cloud service providers. User workload profiles in VDCs are bursty, such as in daily desktop startup, or when a user switches between text and graphics-intensive applications. Also, the user quality of experience (QoE) of thin-clients is highly sensitive to network health variations within the Internet.

To address the challenges associated with developing scalable VDCs with satisfactory thin-client user QoE, we developed a "VMLab" infrastructure for supporting desktop virtualization experiments in research and educational user communities. This paper describes our efforts in using VMLab infrastructure to support the following:

- Desktop virtualization sandboxes for system administrators and educators
- Research and development activities relating to VDC resource allocation and thin-client performance benchmarking
- Virtual desktops for classroom lab user trials involving faculty and students
- Evaluation of the feasibility to deploy computationally intensive interactive applications in virtual desktops, such as remote volume visualization
- Educational laboratory course curriculum development involving desktop virtualization exercises

I. Motivation and Significance

Today, common user applications such as email, photos, videos, and file storage are supported at Internet-scale by cloud platforms, including HP Cloud Assure, Google Mail, and Amazon S3. Even academia increasingly is adopting cloud infrastructures and related research themes to support scientific research and education communities, such as the National Science Foundation Cluster Exploratory (NSF CluE) and the Department of Energy's (DOE) Magellan project. The next frontier for these user communities is to transition traditional distributed desktops with dedicated hardware and software installations into virtual desktop clouds (VDCs) that are accessible via thin-clients.

Moreover, in the not so distant future, we can envisage home users signing up for virtual desktops (VDs) with a VDC service provider providing Desktop-as-a-Service (DaaS) as a utility. With such a utility service, a thin-client such as a settop box can be shipped to a residential user to access a VD in a manner similar to what we have today for other common computing and communication needs, such as VoIP and IPTV. The settop box can be connected to television monitors or computer monitors, and multiple residential users can have their own unique login through this box to personalized VDs.

The drivers for transitioning traditional desktops to VDCs are obvious in terms of user convenience and cost-savings:

- Easier management of desktop support in terms of operating system, application and security upgrades
- Reduction in the number of underutilized distributed desktops unnecessarily consuming power
- Wider access to applications and data by mobile users

Allocating and managing VDC resources in a scalable and cost-effective manner poses unique challenges for service providers. User workload profiles in VDCs are bursty, such as in daily desktop startup, or when a user switches between text and graphics-intensive applications. Also, the user quality of experience (QoE) of thin-clients is highly sensitive to network health variations within the Internet. Unfortunately, existing solutions focus mainly on managing server-side resources based on utility functions of CPU and memory loads [1–4] and do not consider network health and thin-client user QoE. There is surprisingly little work being done [5–6] on resource adaptation coupled with measurement of network health and user QoE. Investigations such as [6] and [7] highlight the need to incorporate network health and user QoE factors into VDC resource allocation decisions.

It is self-evident that any cloud platform's capability to support large user workloads is a function of both server-side desktop performance as well as remote user-perceived QoE. In other words, *a CSP can provision adequate CPU and memory resources to a VD in the cloud, but if the thin-client protocol configuration does not account for network health degradations and application context, the VD is unusable for the user.* Another real-world scenario that motivates intelligent resource allocation is the fact that: *CSPs today do not have frameworks and tools that can estimate how many concurrent VD requests can be handled on a given set of system and network resources within a data center such that resource utilization is maximized, and at worst, the minimum user QoE is guaranteed as negotiated in service-level agreements (SLAs).* Resource allocations without combined utility-directed information of system loads, network health, and thin-client user experience in VDC platforms inevitably results in costly guesswork and over-provisioning, even for as few as tens of users. Also, due to lack of tools to measure the user experience from the server-side of VDCs, management functions in VDCs, such as configuring thin-client protocol parameters, often are performed using guesswork, which in turn impacts user QoE.

2. VMlab Infrastructure

A. Resources and Setup

To address the research and development challenges in developing scalable VDCs with satisfactory thin-client user QoE, we developed a "VMLab" infrastructure [8] that supports desktop virtualization experiments for research and education user communities. Initially funded by the Ohio Board of Regents, VMLab now is supported by the VMware End-user Computing Group, VMware Academic Program, Dell Education Cloud Services, and the National Science Foundation (under award numbers NSF CNS-1050225 and NSF CNS-1205658).

In the current VMLab infrastructure, an IBM® BladeCenter® S Chassis acts as a VDC data center that can concurrently support up to approximately 50 VDs. The BladeCenter has two IBM HS22 Intel® blade servers each with two quad-core CPUs, 32 GB of RAM, and four network interface cards (NICs). The storage resource is approximately 9 TB of shared SAS storage. The client-side uses a mix of several physical thin-clients from IBM, HP, and Wyse.

A netem network emulator [18] on a Linux Kernel is used for laboratory experiments to introduce network latency and loss and constrain end-to-end available bandwidth between the client and server sides. The VMware View™ desktop virtualization solution is used primarily to provision resources and broker virtual desktops, and has prerequisites such as VMware vSphere®. A web portal (http://vmlab.oar.net) enables information sharing about VMLab resources, capabilities, and salient project results. The web portal also provides information for VMLab users to gain hands-on access to run desktop virtualization experiments in their own sandboxes.

Each sandbox in VMLab has a dedicated virtual network with a separate blade allocation, as well as storage and firewall resources. Resource provisioning is performed based on user requirements and experiment plans. VPNs are set up using the OpenVPN™ server [19] for WAN connections to avoid using public IP address for VDs, and to accept VPN connections from external IP addresses. A virtual pfSense® server is used as a firewall appliance to handle all traffic between VDs and the Internet. Firewall rules are set or modified to restrict access to certain ports and addresses based on user sandbox requirements. The hypervisor, Active Directory, web portal and other supporting infrastructure are hosted in individual virtual machines within the VMLab infrastructure.

For experimentation involving distributed, multi-data center VDCs with realistic settings, VMLab resources are augmented with additional data center and thin-client resources from the NSF-supported Global Environment for Network Innovations (GENI) [17] infrastructure. The GENI infrastructure is a federated cloud of system (Emulab [20], PlanetLab [21]) and network resources (Internet2® and National LamdaRail (NLR)) for controlled as well as real-world experiments. It also provides a sliceable Internet infrastructure with wide area network (WAN) programmability using OpenFlow technologies that enable the dynamic allocation and migration of virtual machines in experiment slices. A multi-domain test bed with extended VLAN connectivity is set up between two data centers: VMLab at The Ohio State University and Emulab at the University of Utah. Distributed GENI nodes located at several university campuses (Stanford University, Georgia Institute of Technology, University of Wisconsin, Rutgers University) are used as thin-client sites.

B. Users and Activities

Over the last three years, the VMLab infrastructure has supported:

- Desktop virtualization sandboxes for system administrators and educators [8]
- Research and development activities relating to VDC resource allocation and thin-client performance benchmarking [9]–[13]
- Virtual desktops for classroom lab user trials involving faculty and students [14] [15]
- Evaluation of the feasibility to deploy computationally-intensive interactive applications, such as remote volume visualization, in virtual desktops [14] [16]
- Educational laboratory course curriculum development involving desktop virtualization exercises [13]

The desktop virtualization sandboxes were set up for several campus system administrators in Ohio, Michigan, and Texas for a variety of experiments involving VMware Virtual Desktop Infrastructure (VDI) technologies, web portal and electronic lab notebook staging, and thin-client video streaming performance testing. Educators in three departments (Dept. of Chemistry, Dept. of Industrial and Systems Engineering, Small Animal Imaging Shared Resource (SAISR)) at The Ohio State University (OSU), as well as in Polymer Ohio have experimented with VMLab resources to set up VDs for classroom labs. The classroom lab applications within VDs ranged from common applications such as Microsoft Word and Microsoft Windows Media Player to remote volume visualization applications, such as surgical simulation and polymer injection-flow modeling, that are computationally-intensive, have massive datasets, and are highly interactive in nature.

In addition, the Ohio Board of Regents CIO Advisory Board Members recently sponsored a VDPilot project, a feasibility study of a VDC for classroom labs. This VMLab related study leverages universities' pre-existing high-speed access to the OARnet network and to national networks such as Internet2 and NLR in order to assess the user QoE of accessing desktops remotely compared to physically going to a computing lab, as well as analyzing the challenges and cost savings due to shared resources amongst collaborating institutions.

Development of novel, dynamic VDC resource allocation schemes and an OpenFlow controller are ongoing, integrating them into a VDC-Sim simulator. The VDC-Sim can act as a *cloud resource broker* to "control and manage routing flows", as well as "measure and monitor user QoE delivery" in a "Run Simulation" mode, or can actually interact with VDC components in a GENI slice in a "Run Experiment" mode. VDC-Sim is being adapted to develop a graduate course curriculum involving desktop virtualization exercises in VMLab-GENI infrastructures through collaboration with the Department of Computer Science at Purdue University. These exercises include a study of resource allocation schemes, and comparing and optimizing thin-client protocol performance.

Lastly, a project involving underserved communities is being led by researchers in the Department of Computer Science and Engineering at Ohio State University. Here, researchers are working to equip the Linden community around Columbus, Ohio with VDCs as part of their emerging STEM education and other critical community support programs that can benefit from virtual desktop access.

3. Exemplar Use Cases, Experiments, and Results

A. Thin-client Performance Benchmarking Toolkit Development

Clearly, instrumentation and measurement are needed on the server and thin-client sides to gather performance data for making the best resource adaptations in VDC platforms around system loads, network health, and thin-client user QoE. VMLab resources are being used to develop a novel virtual desktop benchmarking toolkit, called VDBench, [9] to create application and user group profiles based on CPU, memory, and network bandwidth measurements.

The current VDBench prototype can measure user QoE of atomic and aggregate application tasks in terms of interactive response times or timeliness metrics such as application launch time, web page download time, "Save As..." task time. Tasks are executed via different thin-client protocol configurations, such as RDP, RGS, and PCoIP, under synthetic system loads and network health impairments. It uses the concept of "marker packets" to correlate thin-client user events with server-side resource performance events in packet captures. It also leverages measurements from built-in memory management techniques in VMware® ESX®, such as ballooning under heavy loads [22], and earlier research on slow-motion benchmarking of thin-client performance under varying network health conditions [23]. Figure 1 shows the VDBench Java client prototype. The software can run on Windows and Linux platforms, and has capabilities for NIC selection for test initiation as well as interactions with the benchmarking engine for reporting test results. Enhancements to the VDBench Java client prototype are ongoing, including the ability to install and configure the software to run on physical desktops or commercial Windows or Linux operating system embedded thin-clients.

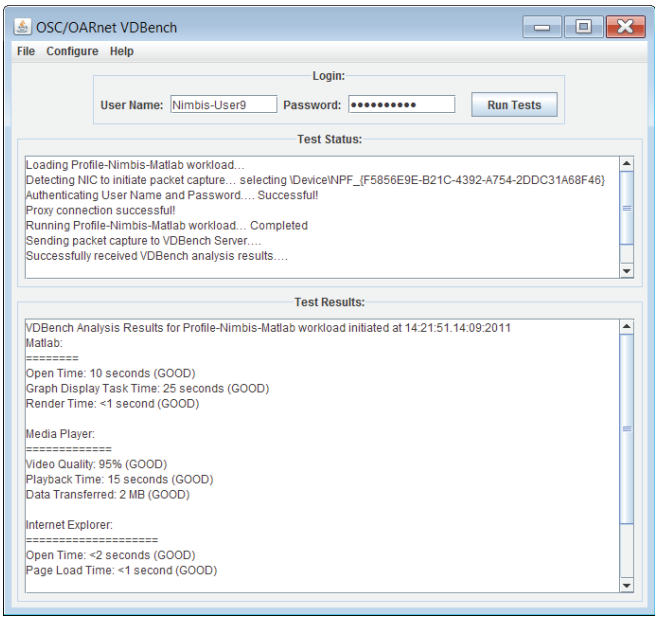

Figure 1. VDBench Java Client Prototype User Interface

B. Utility-directed Resource Allocation Scheme Development

In another set of salient research activities, application and user group profiles obtained through VDBench in VMLab are being used to develop utility-directed resource allocation schemes for VDCs at Internet scale. More specifically, we developed a utility-directed resource allocation model (U-RAM) [10] that uses offline benchmarking-based utility functions of system, network, and human components to dynamically (online) create and place VDs in resource pools at distributed data centers, while optimizing resource allocations along timeliness and coding efficiency quality dimensions. We showed how this resource allocation problem approximates to a binary integer problem whose solution is NP-hard.

To solve this problem, we proposed an iterative algorithm with fast convergence that uses combined utility-directed decision schemes based on Kuhn-Tucker optimality conditions [24]. The ultimate optimization objective was to allocate resources (CPU, memory, network bandwidth) to all VDs such that the global utility is maximized under the constraint that each VD at least meets it minimum quality requirement along timeliness and coding efficiency dimensions.

To assess the VDC scalability that can be achieved by U-RAM provisioning, simulations were conducted to compare U-RAM performance with other resource allocation models:

- Fixed RAM (F-RAM), where each VD is over provisioned, something that is common in today's cloud platforms due to a lack of system and network awareness
- Network-aware RAM (N-RAM), where allocation is aware of required network resources yet over provisions system resources (RAM and CPU) due to a lack of system awareness information
- System-aware RAM (S-RAM), where allocation is the opposite of N-RAM
- Greedy RAM (G-RAM), where allocation is aware of system and network resource requirements based purely on conservative rule-of-thumb information rather than the objective profiling used by U-RAM

Several data center sites were considered, assuming each site had 64 GB of RAM, a 100 Mbps duplex network bandwidth interface, and a scalable number of 2 GHz CPU cores. Several factors were varied during the simulation runs, including the number of data center sites, the number of CPU cores at each site, and the type of desktop pools to which incoming VD requests belonged. The simulation results clearly showed that U-RAM outperforms other schemes by supporting more VDs per core and allowing a greater number of user connections to the VDC with satisfactory user QoE.

In addition, U-RAM and F-RAM are implemented in the VMLab-GENI test bed, as illustrated in Figure 2 [11]. Using Matlab-based animation of a horse point-cloud as the thin-client application, we demonstrated that U-RAM provides improved performance and increased scalability in comparison to F-RAM under realistic settings.

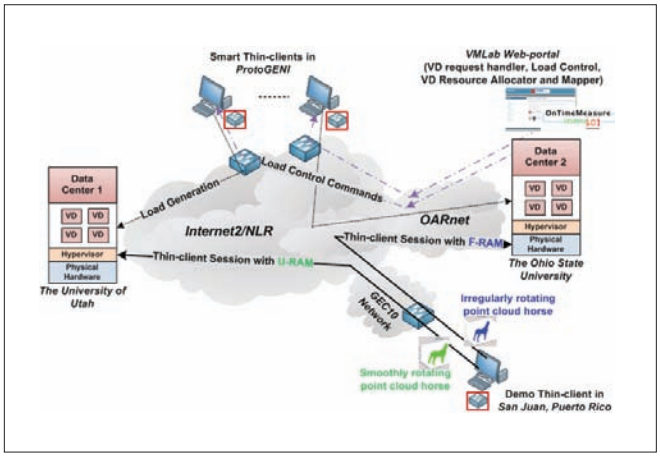

Figure 2. VMLab-GENI provisioning experiment to compare U-RAM and F-RAM schemes

The U-RAM work is being extended for provisioning VDs by investigating salient problems with subsequent placement of VDs across distributed data centers [12]. Placement decisions are influenced by session latency, load balancing, and operation cost constraints. In addition, placement decisions need to be changed over time for proactive defragmentation of resources for improved performance and scalability, as well as reactive VD migrations for increased resilience and sustained availability. Proactive defragmentation of resources is performed using global optimization schemes to overcome the resource fragmentation problem in VDCs that results from placements being done opportunistically to reduce user wait times for initial VD access. We refer to opportunistic placements as those that are performed using local schemes that use high-level information about resource status in data centers.

Over time, resource fragmentation due to careless packing of VDs on resources and changing application workloads leads to the "tetris effect" that decreases scalability (VDs per core) and performance (user QoE), thereby affecting the VDC Net-Utility. In contrast, reactive VD migrations are triggered by cyber attack or planned maintenance events, and should be performed in a manner that does not drastically affect the VDC Net-Utility. Not all VD migrations suggested by proactive or reactive schemes generate positive benefit in VDC Net-utility, since VD migration is an expensive and disruptive process. Therefore, we model the cost of migration and normalize it to utility of VDs, and migrate only the VDs (positive pairs) that generate positive Net-benefit in the VDC.

C. VDPilot: Virtual Classroom Lab User Trials

Providing access to expensive, computational software such as Matlab® and SPSS has always been a logistical and licensing challenge for professors who want to train their students with industry-standard software. Although universities have labs with pre-licensed versions of the software available, lab access for some students is inconvenient. Furthermore, many students need pervasive access to the software and have trouble obtaining a license and installing the software correctly on their home computers. Professors who want to manage lab exercises, assignments, and exams use e-mail to send and receive large files, and are limited in their ability to access and assist in the work-in-progress of students.

To address these problems, the Ohio Board of Regents CIO Advisory Board Members recently commissioned a VDPilot feasibility study for hosting virtual desktops and shared storage for classroom labs within the Ohio-based university system. The study was initiated to investigate the use of federated shared infrastructure resources that would simplify classroom lab computing for faculty and students, and reduce costs for universities.

As part of the VDPilot study, VMLab was reconfigured to support subjective testing for approximately 50 faculty and students with secure remote access to lab software using thin-clients over the Internet [15]. User trials were conducted with professors and students, as well as some IT administrators, who were asked to compare going to a physical lab versus using the remote thin-clients while performing tasks in the virtual desktops using applications such as Microsoft Excel®, Matlab, SPSS, Windows Media Player, and Internet Explorer®.

Figure 3 shows the VDPilot survey (screenshot) that participants completed after following subjective testing instructions provided to them through the VDPilot web portal.

Figure 3. VDPilot survey participants completed after subjective testing

The survey results showed that 50 percent of participants found the virtual desktop user experience to be comparable to their home computer's user experience, while 17 percent could not decide which user experience was better. Interestingly, 8 percent found the virtual desktop user experience to be better than their home computer user experience, particularly in the case of resource-intensive applications such as SPSS. Quotes recorded from faculty and students indicated they liked the virtual computer lab access in the pilot project in its current form. Two of the professors who participated in the study were eager to have their students use the pilot project test bed immediately as part their ongoing course offering. This confirms there is a real and current need for hosting virtual desktops and shared storage for classroom labs at universities.

D. Remote Interactive Volume Visualization for Researchers

VMLab resources have been used in experiments for the evaluation and support of a Remote Interactive Volume Visualization Infrastructure for Researchers (RIVVIR) [14] [16] to serve an increasing user base in the Small Animal Imaging Shared Resource (SAISR) at the Ohio State University and Polymer Ohio communities. RIVVIR provides an environment in which users can access VDs that host computationally-intensive interactive applications and their related massive data sets.

Given the growing trend of users of data-intensive remote volume visualization applications that deal with gigabyte to petabyte sized data sets, it is impractical to carry or download these data sets and run computational analyses. Users of such applications in communities such as the ones supported by SAISR and Polymer Ohio inevitably must use VDs that have high-performance computing (HPC) capabilities at the data location and high-speed intermediate networks. Moreover, VDs allow a visual interpretation of large data sets, a powerful medium that can foster science and engineering innovations. Further, RIVVIR allows users to access their computationally-intensive interactive applications using handheld devices. They can jointly collaborate on the application steering with researchers at other institutions for visualization and analytics tasks related to their research and development efforts.

In our RIVVIR development efforts, we are interested in exploration that is beyond the classical thin-client model. We are exploring hybrid computing models and advanced multimedia stream content processing schemes, where execution is apportioned between thin-clients and the back-end server. For high-motion session output or computationally-intensive rendering such as 3D, we are investigating thin-client protocol optimizations that leverage increased server processing power to improve frame rate. The session switches to a general thin-client protocol configuration for low-motion video and other routine rendering.

In addition, we are evaluating the potential of caching repetitive video blocks, such as desktop backgrounds and menu items, to reduce server-side processing, bandwidth consumption, and interaction delays. Furthermore, there has been a trend lately in the use of thin-clients with high-resolution displays and significant computing power, especially with the latest generation of Apple iPads and similar products. To support these emerging thin-client platform applications, we are exploring related hybrid computing issues, where computing is distributed between the client and server ends, depending upon application context.

Figure 4 shows the RIVVIR configurations within VMLab for SAISR and Polymer Ohio community users for high-end demonstrations [14] [16]. A remote SAISR or Polymer Ohio user accesses a VD application similar to other users of typical applications, such as Microsoft Word or Internet Explorer, while their computationally-intensive interactive applications in the back-end rely on HPC infrastructure at the Ohio Supercomputer Center (OSC).

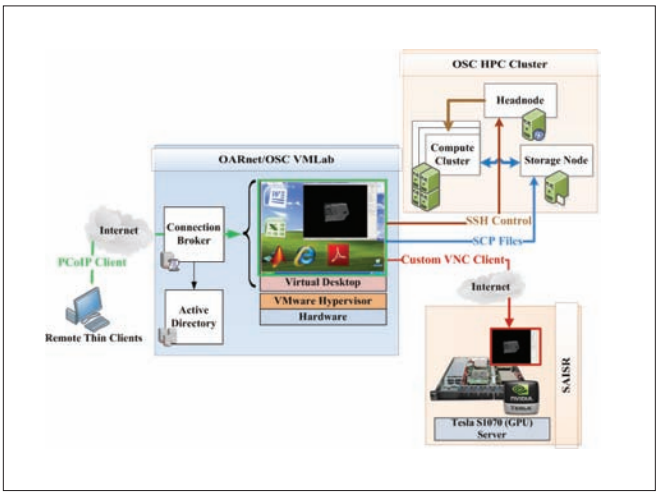

Figure 4. RIVVIR configurations for SAISR and Polymer Ohio community users

There are several challenges in configuring VDCs to support such applications due to their computational, storage, and network resource requirements for optimal user QoE. As in the case of SAISR users, custom applications need to be deployed that may rely on legacy thin-client protocols such as VNC, which are unsuitable for large delay or lossy networks that are common on the commercial Internet, as shown from our previous studies [25] [26]. We plan to

experiment with various reconfigurations of the enhanced VMLab infrastructure to study optimizations in thin-client protocols, such as tunneling and network-awareness, that can deliver satisfactory remote volume visualization user QoE. In the case of tunneling experiments, working through campus firewalls is a challenge. Additionally, we plan to address challenges in tunneling legacy protocols within the latest thin-client protocols, such as PCoIP to support remote users with large delay or lossy networks between thin-clients and servers.

4. Conclusion

Our VMLab deployment efforts have provided valuable operations experience to support a wide variety of research and education use cases relating to desktop virtualization experimentation. These experiences are helping significantly the engineering and operations of production services for desktop virtualization at the Ohio Supercomputer Center and OARnet, and new service models for research and education are being developed for user communities. Developers of the GENI community within infrastructure groups and instrumentation and measurement services groups have developed new capabilities by supporting unique experiment requirements of our VDC Future Internet application. We expect future desktop virtualization experimenters to benefit from these advancements.

The number of researchers and educators desiring to use VMLab resources is increasing rapidly, and several new initiatives are looking to use VMLab to scale to a large number of actual users in classroom labs and underserved communities. Our ongoing research and development projects on VDC resource allocation and thin-client performance benchmarking are ramping up for more extensive experiments. As a result, the VMLab infrastructure is being enhanced to support the concurrent provisioning up to 150 VDs. In addition, more than 20 physical thin-clients are being deployed. These units can be shipped to end-user sites for VDC experiments with a diverse group of geographically distributed users. This will enable us to validate successful DaaS offerings, where service providers can observe performance and control the end-to-end components to consistently meet SLAs and deploy an economically viable service delivery model.

We believe research and development outcomes from VMLab enable the realization of the next frontier—one that transforms end-user computing capabilities in the future Internet and enables society to derive benefits from computer and network virtualization. As a result, VMLab infrastructure enhancements are focused on collecting valuable real-world data sets to gain a better understanding of workload profiles for diverse applications and user groups in VDCs. This is the kind of understanding, missing in today's research literature, that could, in turn, fuel the development of new frameworks and tools to allocate and manage resources for improved performance, increased scalability, and cost-effective VDCs.

Acknowledgment

This material is based upon work supported by the VMware Academic Program and the National Science Foundation under award numbers CNS-1050225 and CNS-1205658. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of VMware or the National Science Foundation.

The following students and colleagues at The Ohio State University have contributed towards the various VMLab projects described in this paper: Rohit Patali, Aishwarya Venkataraman, Mukundan Sridharan, Yingxiao Xu, David Welling, Saravanan Mohan, Arunprasath Selvadhurai, Sudharsan Rajagopalan, Rajiv Ramnath, and Jayshree Ramanathan.

References

1 D. Gmach, S. Krompass, A. Scholz, M. Wimmer, A. Kemper, "Adaptive Quality of Service Management for Enterprise Services", *ACM Transactions on the Web,* Vol. 2, No. 8, Pages 1-46, 2008.

2 P. Padala, K. Shin, et. al., "Adaptive Control of Virtualized Resources in Utility Computing Environments", *Proc. of the 2nd ACM SIGOPS/EuroSys, 2007.*

3 B. Urgaonkar, P. Shenoy, et. al., "Agile Dynamic Provisioning of Multi-Tier Internet Applications", *ACM Transactions on Autonomous and Adaptive Systems*, Vol. 3, No. 1, Pages 1-39, 2008.

4 H. Van, F. Tran, J. Menaud, "Autonomic Virtual Resource Management for Service Hosting Platforms", *Proc. of ICSE Workshop on Software Engineering Challenges of Cloud Computing,* 2009.

5 K. Beaty, A. Kochut, H. Shaikh, "Desktop to Cloud Transformation Planning", *Proc. of IEEE IPDPS,* 2009.

6 N. Zeldovich, R. Chandra, "Interactive Performance Measurement with *VNCplay*", *Proc. of USENIX Annual Technical Conference,* 2005.

7 J. Rhee, A. Kochut, K. Beaty, "DeskBench: Flexible Virtual Desktop Benchmarking Toolkit", *Proc. of Integrated Management (IM),* 2009.

8 P. Calyam, A. Berryman, A. Lai, R. Ramnath, "VMLab Testbed for Desktop Virtualization to Support Research and Education", *MERIT Desktop Virtualization Summit,* 2010, http://vmlab.oar.net.

9 A. Berryman, P. Calyam, A. Lai, M. Honigford, "VDBench: A Benchmarking Toolkit for Thin-client based Virtual Desktop Environments", *IEEE Conference on Cloud Computing Technology and Science (CloudCom),* 2010.

10 P. Calyam, R. Patali, A. Berryman, A. Lai, R. Ramnath, "Utility-directed Resource Allocation in Virtual Desktop Clouds", *Elsevier Computer Networks Journal (COMNET),* 2011.

11 P. Calyam, M. Sridharan, Y. Xiao, K. Zhu, A. Berryman, R. Patali, "Enabling Performance Intelligence for Application Adaptation in the Future Internet", *Journal of Communications and Networks (JCN)*, 2011.

12 M. Sridharan, P. Calyam, A. Venkataraman, A. Berryman, "Defragmentation of Resources in Virtual Desktop Clouds for Cost-Aware Utility-Optimal Allocation", *IEEE Conference on Utility and Cloud Computing (UCC)*, 2011.

13 P. Calyam, A. Venkataraman, A. Berryman, M. Faerman, "Experiences from Virtual Desktop Cloud Experiments in GENI", *GENI Research & Educational Experiment Workshop (GREE)*, 2012.

14 P. Calyam, D. Stredney, A. Lai, A. Berryman, K. Powell, "Using Desktop Virtualization to access advanced Educational Software", *Internet2/ESCC Joint Techs*, Columbus, OH, 2010.

15 P. Calyam, A. Berryman, D. Welling, S. Mohan, R. Ramnath, J. Ramnathan, "VDPilot: Feasibility Study of Hosting Virtual Desktops for Classroom Labs within a Federated University System", *International Journal of Cloud Computing*, 2012.

16 A. Lai, D. Stredney, P. Calyam, B. Hittle, T. Kerwin, D. Reed, K. Powell, "Remote Interactive Volume Visualization Infrastructure for Researchers", *AMIA Annual Symposium*, 2011.

17 Global Environment for Network Innovations, http://www.geni.net

18 The Linux Foundation Netem, http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

19 OpenVPN, http://openvpn.net/

20 Emulab/ProtoGENI Infrastructure, http://www.emulab.net

21 PlanetLab- GENI, http://groups.geni.net/geni/wiki/PlanetLab

22 C. Waldspurger, "Memory Resource Management in VMware ESX Server", *ACM SIGOPS Operating Systems Review*, Vol. 36, Pages 181 - 194, 2002.

23 A. Lai, J. Nieh, "On The Performance Of Wide-Area Thin-Client Computing", *ACM Transactions on Computer Systems*, Vol. 24, No. 2, Pages 175-209, 2006.

24 R. Rajkumar, C. Lee, J. Lehoczky, D. Slewlorek, "A Resource Allocation Model for QoS Management", *Proc. of IEEE RTSS*, 1997.

25 P. Calyam, A. Kalash, A. Krishnamurthy, G. Renkes, "A Human-and-Network Aware Encoding Adaptation Scheme for Remote Desktop Access", *Proc. of IEEE MMSP*, 2009.

26 P. Calyam, A. Kalash, R. Gopalan, S. Gopalan, A. Krishnamurty, "RICE: A Reliable and Efficient Remote Instrumentation Collaboration Environment", *Journal of Advances in Multimedia's special issue on Multimedia Immersive Technologies and Networking*, 2008.

vQuery: A Platform for Connecting Configuration and Performance

Ilari Shafer
Carnegie Mellon University

Snorri Gylfason
VMware, Inc.

Gregory R. Ganger
Carnegie Mellon University

Abstract

Discovering the causes of performance problems in virtualized systems is often more difficult than without virtualization, because they can be caused by changes in infrastructure configuration rather than the user's application. vQuery is a system that collects, archives, and exposes configuration changes alongside fine-grained performance data, so the two can be correlated. It gathers configuration change data without modifying the systems it collects from and copes with platform-specific details within a general, graph-based model of Infrastructure-as-a-Service (IaaS) infrastructures. Configuration data collected from two VMware® vSphere™ environments reveals that configuration changes are frequent and involved, opening interesting new directions for configuration-aware performance diagnosis techniques.

1. Introduction

Consolidating computing activities onto shared infrastructures, such as in cloud computing and other virtualized data centers, offers substantial efficiency benefits for providers and consumers alike. But, it also introduces complexities when trying to understand the performance behavior of any given activity, since it can depend on many factors not present when using dedicated infrastructure. For example, the VMs used for the activity can migrate or be resized, or new VMs for other activities can be instantiated on shared hardware. As on-demand resource allocation (as in cloud computing) and automated configuration optimization (e.g., via VMware DRS[1]) grow more common, such factors increasingly create potentially confusing performance effects.

Traditionally, to understand application performance and diagnose performance problems, administrators and application engineers rely on resource usage instrumentation data from infrastructure runtime systems, such as time-sampled CPU utilization, memory allocated, and network packets sent/received. In VM-based infrastructures, the same data types can be captured for each VM as well. But, while such data exposes how resource usage changed at a given point in time, it offers little insight into why. Deducing why, so that one can decide what (if any) reactive steps to take, often is left entirely to the intuitions and experience of those involved in the diagnosis.

We believe an invaluable additional source of information should be captured and explicitly correlated with resource utilization data: the configuration history. Of course, any runtime infrastructure maintains its current configuration, and many log at least some configuration changes. Since configuration changes often cause performance changes, purposefully or otherwise, correlating the two should make it possible to highlight root causes of many problems automatically. In addition, the combination of the two offers the ability to expose powerful insights for system management and automation, such as which configuration changes usually improve performance and how particular problems were overcome in the past.

This paper describes our prototype system (called vQuery) for configuration change tracking and mining, together with initial experiences. vQuery collects time-evolving configuration state alongside fine-grained resource usage data from a VMware vCloud™-based infrastructure, stores it, and allows it to be queried. Configuration changes are captured by listening to vSphere's API and vCloud's internal update notifications. They are stored as a time-evolving graph of entities (e.g., VMs and physical hosts) as vertices and relationships as edges. This general approach avoids changes to the infrastructure software, accommodates a range of IaaS systems, and allows a range of configuration history queries.

We deployed vQuery on a local VMware software based private cloud (referred to as Carnegie Mellon's vCloud) as well as a VMware testbed, with positive initial results. We illustrate some of the power of configuration change history with interesting anecdotes and data from these deployments, and discuss challenges still ahead on this line of research.

The remainder of this paper is organized as follows. Section 2 explains what we mean by "configuration data" and configuration changes in more detail, including examples from VMware systems. Section 3 describes the design and current implementation of vQuery, focusing on how configuration data is captured, stored, and queried. Section 4 presents some data, early experiences, and anecdotes from vQuery deployments. Section 5 discusses our ongoing research on vQuery and exploiting configuration change data. Section 6 discusses related work.

2. Configuration Data and Changes

In a distributed computing infrastructure, various types of configuration are spread across files, databases, and within software. The word "configuration" often is used for concepts that include command-line flags to programs, OS-level settings, and the layout of virtual machines across computing resources. In this work, we focus on the last type: infrastructure-level properties that affect how virtualized environments, such as vSphere and vCloud, function and that reflect their current state.

Even this type of configuration is very heterogeneous. Some data are as simple as key-value pairs, but other data encodes lists, objects, and hierarchies. Some is controlled by end users (e.g., the choice of guest operating system for a VM), some is primarily automated by the computing infrastructure (e.g., which IP address a VM is assigned), and some can be managed by both (e.g., the choice of physical resources that back a VM). More concretely, Table 1 shows a selection of configuration properties in a VMware-based environment, ranging from simple descriptive properties to relationships with other entities.

ENTITY TYPE	CONFIGURATION PROPERTIES
VM	**vSphere**: *host system*, *networks*, *datastore*, name, annotation, memory, vCPUs, CPU allocation (reservation/limit/shares), memory allocation (reservation/limit/shares), virtual disk layout (chain length), power state, guest OS type, guest OS state, guest OS screen dimensions, guest NIC (IPs, network, state), guest disk (capacity, free space, path), IP address, VMware tools state + version **vCloud**: *vApp*, *networks*, name, vCPUs, memory, guest OS, status, storage
vApp	**vCloud**: *owner*, *vDC*, *networks*, status
User	**vCloud**: name, VM quota
Host	**vSphere**: *network*, CPU (frequency, number of cores and packages), memory size, power state
Network	**vSphere**: name **vCloud**: fence mode, parent, DNS (addresses, suffix), netmask, IP ranges
Datastore	**vSphere**: name, capacity, free space, type, url

Table 1: selected configuration properties in vSphere and vCloud. The properties that represent other entities are shown in italic.

Modeling configuration consistently is one focus of vQuery. In addition to the different types and meanings of configuration within vSphere and vCloud, different virtualized infrastructures expose different configuration properties. For example, where vSphere has a platform-independent datastore abstraction, the OpenStack infrastructure platform separates storage into block storage, local storage, and a separate VM image service. We would like to represent configuration in a sufficiently general way to model such different environments.

A key aspect of configuration on which we focus is that much of it changes over time. Some configuration properties may change very slowly (e.g., the amount of RAM on a physical host is seldom adjusted), while others are increasingly dynamic (e.g., the placement of a VM on a physical host is adapted by DRS). In tracking configuration as it relates to performance, we focus on recording changes in order to ask questions such as "was there a relevant configuration change around the same time as a given performance change?" and "what were all the configuration changes associated with a given VM?" Beyond diagnosis, maintaining a change history can also help us understand how and why systems evolve[2].

Additionally, infrastructure configuration is not a collection of unrelated facts. Configuration properties are associated with entities, whether physical (hosts and physical networks) or virtual (VMs and users), and these entities are meaningfully related. For example, VMs are placed on physical hosts, and users own vApps, which contain VMs. Examples of these "relational" properties are shown in italic in Table 1. We believe maintaining information about the relational structure of configuration—and how it changes—is important for diagnosis. It is intuitively important to be able to ask questions, such as "which VMs were on a given host when there was a performance problem?" Additionally, a variety of diagnosis approaches have taken advantage of the fact that the effects of changes often propagate through causal dependencies among the components of a distributed system[3 4 5], many of which are directed along these relations.

3. vQuery: Design

vQuery is designed to track fine-grained configuration data in a way that maintains the features described above. At a high level, the problems we need to solve are the same as those for performance monitoring: how to collect, store, and access configuration data. A simplified overview of our approach is shown in Figure 1, and this section describes each component in turn.

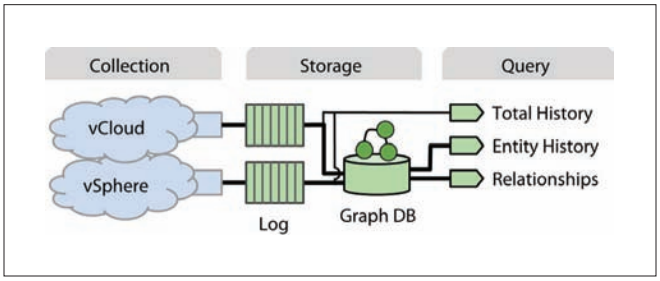

Figure 1: high-level overview of the vQuery configuration tracking system. It collects data from vSphere and vCloud, stores them in separate logs, and ingests them into a graph database to be queried.

3.1 Configuration Collection

Changes to configuration occur from both human and automated sources, and they clearly do not happen only at a fixed interval. It is insufficient for just the current configuration to be exposed by infrastructure APIs. For the collection process to be more efficient

and accurate than polling, there must be some way of obtaining updates. Ideally, the mechanism should require minimal, if any, modification of the infrastructure. We built our prototype without modifying code in vSphere or vCloud.

For vSphere, we build on the existing interface for subscribing to update notifications. Specifically, we use a PropertyCollector and its WaitForUpdates method to receive changes to a set of configuration properties of interest.

Although vCloud does not currently offer such an interface, we collect configuration from vCloud by listening to internal messages as a signal for when to query its configuration API. As an example, vCloud sends a message to start an action (e.g, start a VM, (1) in Figure 2), which results in a message sent to an AMQP message bus (1) and actions in vSphere (2). When the task is finished, a completion message is posted to the message bus. Our configuration collector listens to the same AMQP message bus (3), filters to listen to only task completion messages, and queries an appropriate API to find details about configuration change after a task completes (4).

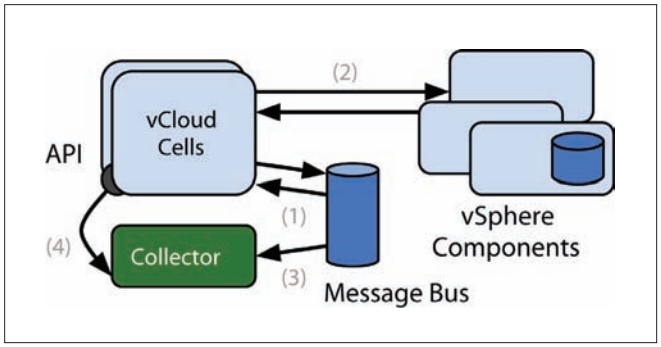

Figure 2: Configuration collection strategy for vCloud and OpenStack. The configuration collector listens to messages on the vCloud message bus and polls an appropriate API upon intercepting a task completion message.

This design pattern is not restricted to vCloud. The recently popular OpenStack IaaS also uses an AMQP message bus for inter-node communication[6]. We built enough of a collector for OpenStack to confirm that messages can be intercepted for configuration event notification. The same technique is not limited to message bus transports. For example, systems based on bare Remote Procedure Calls (RPC) could be instrumented similarly, albeit with a lower-level interceptor. In addition to vSphere, we have started collecting limited configuration data from an instance of the Tashi cluster management system[7].

In an ideal world, we would capture changes to all types of configuration that might affect performance, including those from the application layer. For example, recent research describes mechanisms for capturing changes to configuration files within guest VMs without modifying guest software[8]. Integrating such changes with those accessible from vCloud and vSphere is a direction for future work.

3.2 Configuration Storage

A primary challenge in storing configuration is how to represent it. Here, we describe a time-evolving representation of configuration information that is designed to support historical and relational queries using a general model. The representation is a graph with a loose schema—formally a typed, directed, attributed multigraph that also tracks time.

Infrastructure entities (VMs, physical hosts, storage nodes, networks, and so on) are vertices of this graph. Each vertex is associated with three mandatory fields: a unique identifier (id), a type (VM, host, and so on), and a valid-time interval[9]. Infrastructure entities can be created and removed over time (as in Figure 3, VMs can be allocated and deleted), but their historical presence must be remembered to support retrospective analytics. Each vertex also has a map of attribute names to a list of time-changing values ordered by time. The intuition behind this format is that each infrastructure attribute can change over time (e.g, a user changing the allocation of a VM, as shown as vRAM in Figure 3). The after-image of each value is appended to the list. Our implementation currently supports primitive types (strings, integers, floating-point numbers) and arrays thereof.

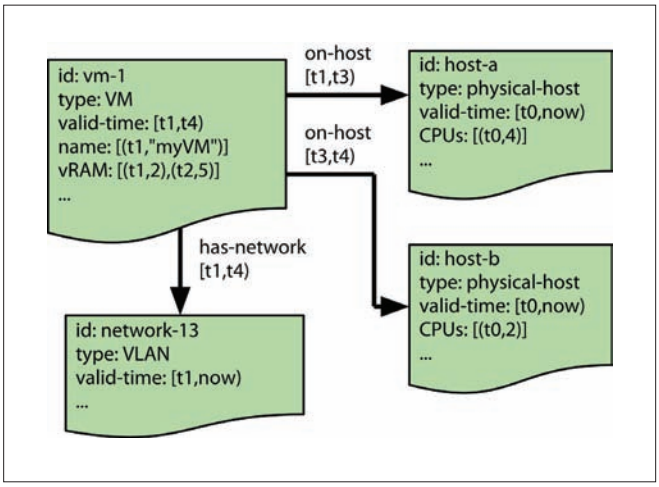

Figure 3: property graph of changing configuration. Each entity (VM, host, etc.) is associated with a number of time-evolving properties, in addition to time-evolving relationships with other entities. A unique identifier (id) and type of entity (type) are the only two required properties. The properties that track relationships (e.g. host) are specified by a user.

Edges between entities have two mandatory fields: a type of relationship and a valid-time interval. Similar to entities, such a relationship often exists only for a given time interval. For instance, in Figure 3, **vm-1** moved from **host-a** to **host-b** at time **t3** and was removed at time **t4**. In this way, the graph captures events such as VM migration not simply as events but as changes that relate entities. These edges—akin to foreign keys—are the schema of the configuration graph. Administrators must specify which configuration attributes have semantic meaning as dependencies (and must contain identifiers as values).

Maintaining configuration in this format also allows for the use of existing graph databases as an underlying persistence layer—in particular, those that store property graphs and have the ability to build indexes on properties.

Updating the graph, while relatively straightforward in principle, requires care in practice. The principle of the algorithm is reminiscent of that used to update a transaction-time state table in a temporal database[10], as applied to a property graph. Unfortunately, when receiving configuration updates from different layers of infrastructure, dependencies can be reported before the entities to which they refer. Consider the following ordered sequence of observations, similar to events observed in practice:

1. The VLAN **network-1** is created
2. vCloud reports that **vm-1** is connected to **network-1**
3. vSphere reports the existence of **network-1**

The final desired graph should contain a **has-network** edge from **vm-1** to **network-1**. If updates are applied in the given order, the graph will contain an invalid edge after step #2, since the existence of **network-1** is not yet known. We maintain a set of these "pending edges," which are scanned as new updates arrive. If one matches a newly-created entity the dependency is added with the original valid-time. As a beneficial side effect, this technique allows the update algorithm to operate with insertion batches atop the transactional graph database used (Neo4j[11]).

One drawback of storing configuration so generally is that we push the problem of forming meaningful queries to the querier. For example, retrieving a list of VMs requires selecting entities with the VM type rather than scanning a table named "VMs." Also, we assume loosely synchronized timestamps across different reporters of entity information, a property provided by the underlying VMware infrastructure.

3.3 Configuration Query

To ask questions about configuration history, we build a few abstractions on top of the graph database to supplement its query language[12]. Here, we focus on a few that align with our primary goals of historical queries that provide the history of an entity or the system, and relational queries that discover entities that likely depend on or influence each other.

- Historical: get-backlog(t_{start}, t_{end}): obtain all configuration changes to any entity between times t_{start} and t_{end}.
- Historical: get-property-names(*E*): get a list of properties associated with entity *E*, followed by get-property (*E*, *name*) to get a time-ordered list of changes to the property with name *n*.
- Relational: get-subgraph (*E*, *d*): do a breadth-first traversal of entities connected to entity *E*, up to a maximum depth *d* (or, with get-subgraph(*E*, *n*), up to a maximum number of entities *n*).

3.4 Performance Collection

In addition to the technique for storing configuration data described above, a source of performance data is necessary to connect configuration with performance. The performance data we consider consists of time series streams of metrics reported by the hypervisor and aggregated by management software. In contrast to configuration data, many mature systems exist for collecting and archiving this data at the infrastructure level[13 14 15].

For performance data collection, we use the StatsFeeder prototype described in more detail in the first issue of the VMware technical journal[16]. We collect nine metrics from each virtual machine and 15 metrics from each physical host every 20 seconds. These performance metrics are described briefly in Table 2.

VIRTUAL MACHINE	
CPU	**usage**: time used by this VM **system**: time spent in the VMkernel **wait**: time spent waiting for hardware/kernel locks **ready**: time spent waiting for a CPU (e.g., on an oversubscribed host) **guaranteed**: time used of the total guaranteed to the VM **extra**: time used beyond what the VM was originally assigned
Memory	**swapped**: amount of VM memory swapped out to disk **swaptarget**: amount of memory the VMkernel is aiming to swap **vmmemctl**: size of the memory balloon
HOST	
CPU	**usage**: aggregated time the CPU was used **idle**: time the CPU was idle
Disk	**usage**: average disk throughput **read**: average read throughput **write**: average write throughput **commands**: disk commands issued **commandsAborted**: disk commands aborted **busResets**: SCSI bus reset commands **numberRead**: number of disk reads **numberWrite**: number of disk writes
Network	**packetsRx**: packets received **packetsTx**: packets transmitted **usage**: average transmit + receive KB/s **received**: average receive KB/s **transmit**: average transmit KB/s

Table 2: collected performance data. All metrics are times, averages, or sums over a sample period (20s).

4. Early experiences with vQuery

A full evaluation of the vQuery framework would assess whether it can answer real diagnosis and monitoring queries. Although the project is still in the preliminary stage, this section provides some early experiences with configuration data collection and synthetic relational queries.

4.1 Historical

A functional configuration monitor collects and stores configuration changes over extended periods. This section describes some of the output from the two vSphere instances to which vQuery has been connected. The Virtual SE Lab (vSEL)[17] is an environment at VMware that is used for events at VMworld, training, and demos. We collected configuration changes from it a month prior to VMworld 2011. The vCloud at Carnegie Mellon (CMU) is a cloud we deployed to support academic workloads from courses, individual researchers, and groups with large research computing demands submitted via batch schedulers. Table 3 lists a few basic metrics of configuration change for each environment.

The last row of Table 3 highlights the diversity of configuration—and the need to be somewhat selective in what is collected and retained. One of the configuration properties exposed by vSphere and collected in the CMU dataset was datastore free space, a frequently updated property that accounted for over 63% of the configuration changes we observed. Although free space changes can be important to monitor, either collecting them infrequently or treating them as time series metrics (rather than as configuration changes) is more appropriate.

	vSEL	CMU
Collection period	12 days, starting 21 July 2011	75 days, starting 12 July 2012
Number of physical hosts	100	15
Number of changes	27888	63820
Number of configuration properties gathered	11	36
Number of changes, less free space changes	27888	23466

Table 3: Basic metrics of configuration change from two vSphere instances.

To better understand configuration changes that have occurred, visualization is crucial. As one example view, Figure 4 shows the configuration changes that occurred in the 75-day CMU dataset. Since there are so many types of configuration changes, we only show the top 10 types of change in the legend (by number of changes). A number of noteworthy events are visible from temporal and spatial groups on the chart. (See the caption for detail.)

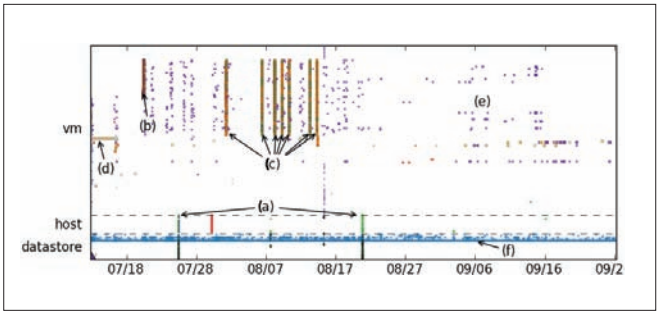

Figure 4: Configuration changes to a small datacenter. Dots represent configuration events to VMs, hosts, and datastores (spread on the vertical axis) across the horizontal time axis. The labeled periods are:

a) changes to many hosts and datastores around the time of a switch outage (first event) and switch replacement (second event) in another virtual datacenter

b) a user adding 30 VMs to an existing set of VMs to run experiments

c) the same user restarting the entire set of VMs when they became unresponsive

d) a user setting up a Windows VM, including many restarts

e) many points in this region (and between (b) and (c)) are VM migrations

f) this row of changes is primarily changes to datastore free space. (The VM disk free space changes shown in Table 3 are filtered out of this image.)

An additional observation we make about the snapshot of configuration in Figure 4 is that many configuration changes co-occur. For example, when VMs are restarted (e.g., the events marked as (c)), their power state changes along with the status of VMware tools in the guest OS and the status of its connection to a virtual network. Together, these changes represent the event "VM restart". Attributing its performance effects to a single one of these changes (particularly a change such as the state of VMware tools) would be misleading. Together with the observation in Table 3 that some configuration events are less meaningful than others, distilling semantically meaningful changes from the noise in configuration will be an important step forward.

4.2 Relational

One important aspect of vQuery is providing query access to related entities, which builds on database support for rapid neighborhood queries in the spatio-temporal configuration graph. We use a graph database (Neo4j); these databases are typically optimized for fast constant-time adjacency lookup[18]. This feature is one key way to manage queries across large graphs: the entities that are closer through dependency traversal are those that are more likely related.

For example, when performing a diagnosis query involving the performance of VM v, likely culprits include configuration changes to its resources (e.g., compute, networking, storage), which are within a traversal distance of 1. Furthermore, other VMs contending for those resources are also of interest, and are within a distance of 2. Although infrequent, relationships with a distance of 3 also arise: VMs in vCloud are modeled as abstract entities that are backed by VMs in vSphere.

Correlating configuration changes from a vCloud VM to a colocated vSphere VM needs 3 hops. If one needs to connect configuration changes to another vCloud VM, the distance would be 4 hops. Most cases involve just 1-2 hops.

To demonstrate that queries in common cases are relatively fast, Figure 5 shows the time required to run a query starting over the largest portion of the vSEL configuration graph. We run queries starting from a random entity in the 1821-entity graph up to a given depth. One can observe that querying for entities separated by a distance of 1 is fast (typically less than two milliseconds), and queries to distance 2 are typically under 10ms.

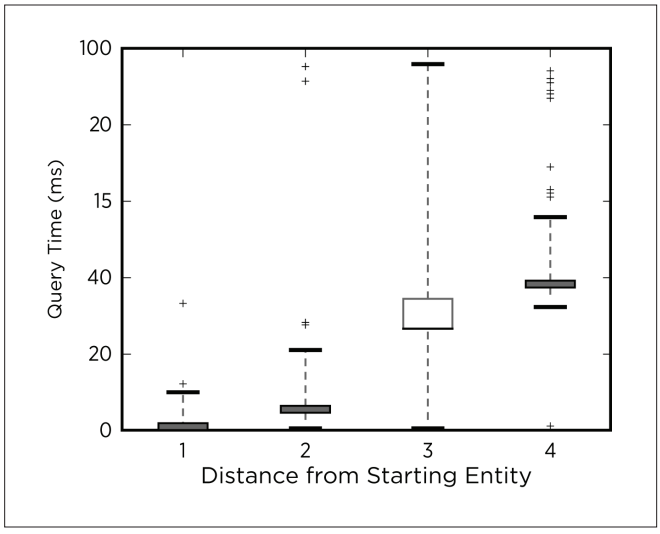

Figure 5: time taken to retrieve related entities to a random starting entity. 1,000 queries were performed at each distance, and boxes extend to the interquartile range.

5. Next Steps

As described above, vQuery forms an infrastructure for collecting, storing, and querying fine-grained configuration and performance data. Moving forward, we plan to use these augmented sources of monitoring data to perform more accurate diagnoses than with traditional black-box performance data alone. In particular, our next aim is to find configuration changes that are the root cause of performance problems. Three concrete examples are:

- **Short-term changes**. VM migration performed by DRS and virtual disk migration performed by storage DRS require network bandwidth and physical host resources. By monitoring performance, we hope to observe the short-term impact of these mechanisms and attribute it to the host and storage configuration changes we observe. We believe the fine-grained performance information we collect will be important to distinguish these performance variations, in addition to recent historical configuration
- **Contention**. virtualized workloads contend for resources, and perfect isolation is not yet a reality across resources, such as caches and disks. Migrating or starting workloads that use a host, datastore, or network can be a source of performance variation for VMs sharing that resource. The configuration changes we measure include migrations and power state changes, which we hope to correlate with performance monitoring data of contending entities. We believe relational queries will be necessary to identify configuration changes that occur to "neighbors," which are potential sources of contention.
- **Explaining parameters**. simply understanding which performance metrics are influenced by a configuration change can be a valuable source of guidance when identifying configuration-related problems, since the impact of configuration parameters often is unclear from name or documentation alone. Identifying performance changes related to configuration could allow us to annotate configuration parameters with the metrics they affect, providing guidance towards how they behave.

6. Related Work

6.1 Configuration Management

Recognition of the complexity of deploying and managing applications across clusters has spurred many configuration management efforts. Tools that have received recent attention include Chef[19] and Puppet[20], which focus on automated application deployment and configuration. CFengine[21] was among the first such tools, designed to reduce the burden of manually scripting policies and configurations across Unix workstations. It has since added support for deploying policies across the cloud computing environments we consider.

These tools primarily facilitate the creation of configuration rather than monitoring changes over time. That is, most focus on actuating configuration rather than monitoring what exists. CFengine is notable among the examples above for also incorporating a familiar-sounding notion of "knowledge management," which is a collection of facts about infrastructure and the relationships between them.

6.2 Correlating Configuration with Performance

Much work on understanding the connection between configuration and performance is focused on tuning configuration to optimize application performance. At least a few techniques, though, focus on our primary motivating use case: finding configuration changes that are the root cause of performance changes.

Many of these techniques have emerged from work on diagnosis in large-scale networks. MERCURY[22] considers an instance of the problem in ISP networks, and identifies the impact of upgrades and routing configuration changes on time series performance indicators, such as CPU utilization and packet loss. Whereas MERCURY considers mostly long-term changes in performance, PRISM[23] operates in the same setting and focuses instead on shorter time-scale changes, such as "spikes." WISE[24] also operates on ISP configuration and performance, but uses it to answer questions of the form "what would be the performance impact of making a configuration change?"

In the context of distributed applications, although NetMedic[25] uses two known snapshots in time as "good" and "problematic" points for diagnosing application-level errors, it uses some of the same concepts discussed here—notably, inference based on system performance data and an (automatically generated) dependency graph. ASDF[26] also correlates multiple time evolving measurements, similar to the black-box monitoring data described here, to perform root-cause diagnosis of performance problems.

6.3 Problem Diagnosis

Our work shares high-level goals with efforts to diagnose problems in distributed systems using widely available black-box performance metrics, such as CPU time and network throughput. For instance, Kasick et al. use statistical comparison across multiple machines to perform root-cause diagnosis in parallel file systems[27]. At the application level, work focused on multi-tier distributed systems has used time series CPU performance metrics to localize faults to individual machines[28], and domain-specific counters in IP networks[29].

By taking advantage of deeply instrumented "white-box" systems, a broader range of distributed system diagnosis techniques have been used for finding the sources of performance problems. For example, end-to-end traces , which track activity as it moves across system components, can be a rich source of insight[30]. Spectroscope[31] is one such tool that leverages these traces for root-cause performance problem diagnosis.

7. Summary

In virtualized environments, such as VMware vSphere, the additional indirection between workloads and the resources they use can lead to additional challenges when finding the source of performance problems. Infrastructure configuration changes can be a hidden source of performance variation. Identifying such effects requires configuration change capture and analysis. vQuery is a system for tracking configuration changes so that we can correlate them with traditional performance data, and early experiences with it are promising. Moving forward, we plan to integrate the data we collect to automatically produce insight about configuration-related performance problems in virtualized infrastructures.

References

1 A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu. "VMware distributed resource management: design, implementation, and lessons learned." VMware Technical Journal, vol. 1, no. 1, pp. 47–64, 2012.

2 H. Kim, T. Benson, and A. Akella. "The Evolution of Network Configuration: A Tale of Two Campuses," IMC 2011.

3 A. A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and J. Emmons, "Detecting the performance impact of upgrades in large operational networks." ACM SIGCOMM Computer Communication Review, vol. 40, no. 4, pp. 303–314, 2010.

4 P. Bahl, R. Chandra, and A. Greenberg, "Towards highly reliable enterprise network services via inference of multi-level dependencies," ACM SIGCOMM 2007.

5 M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," NSDI 2004.

6 OpenStack, http://www.openstack.org.

7 M. A. Kozuch, M. P. Ryan, R. Gass, S. W. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. López, M. Stroucken, and G. R. Ganger, "Tashi: location-aware cluster management," ACDC 2009.

8 W. Richter, M. Satyanarayanan, J. Harkes, and B. Gilbert, "Near-Real-Time Inference of File-Level Mutations from Virtual Disk Writes," Technical Report CMU-CS-12-103, 2012.

9 C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass, "A Glossary of Temporal Database Concepts," ACM SIGMOD Record, vol. 21, no. 3, pp. 35–43, Sep. 1992.

10 R. T. Snodgrass, Developing time-oriented database applications in SQL. Morgan Kaufmann Publishers Inc., 1999.

11 Neo4j, http://neo4j.org/

12 Neo4j cypher query language, http://docs.neo4j.org/chunked/stable/cypher-query-lang.html

13 Nagios, http://www.nagios.org

14 Zenoss, http://www.zenoss.com/

15 IBM Tivoli, http://www.ibm.com/developerworks/tivoli/

16 V. Soundararajan, B. Parimi, and J. Cook, "StatsFeeder: An Extensible Statistics Collection Framework for Virtualized Environments," VMware Technical Journal, vol. 1, no. 1, pp. 32–44, 2012.

17 F. Donald. "cim1436 - Virtual SE Lab (vSEL): Building the VMware Hybrid Cloud." VMworld 2011.

18 M. Rodriguez. "MySQL vs. Neo4j on a Large-Scale Graph Traversal," http://java.dzone.com/articles/mysql-vs-neo4j-large-scale

19 Opscode Chef, http://www.opscode.com/

20 Puppet, http://puppetlabs.com/

21 M. Burgess, "A site configuration engine," Computing systems, vol. 8, no. 2, pp. 309–337, 1995.

22 A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, and J. Wang, "Detecting the Performance Impact of Upgrades in Large Operational Networks," SIGCOMM 2010.

23 A. Mahimkar, Z. Ge, J. Wang, and J. Yates, "Rapid detection of maintenance induced changes in service performance," CoNEXT 2011.

24 M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with WISE," in ACM SIGCOMM Computer Communication Review, 2008, vol. 38, no. 4, pp. 99–110.

25 S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," ACM SIGCOMM Computer Communication Review, vol. 39, no. 4, p. 243, Aug. 2009.

26 K. Bare, S. Kavulya, J. Tan, X. Pan, E. Marinelli, M. Kasick, R. Gandhi, and P. Narasimhan, "ASDF: An Automated, Online Framework for Diagnosing Performance Problems," in Architecting Dependable Systems VII, A. Casimiro, R. de Lemos, and C. Gacek, Eds. Springer Berlin / Heidelberg, 2010, pp. 201–226

27 M. Kasick, J. Tan, R. Gandhi, and P. Narasimhan, "Black-box problem diagnosis in parallel file systems," FAST 2010.

28 K. A. Bare, S. Kavulya, and P. Narasimhan, "Hardware performance counter-based problem diagnosis for e-commerce systems," NOMS 2010.

29 S. P. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan, "Draco : Statistical Diagnosis of Chronic Problems in Large Distributed Systems," DSN 2012.

30 E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger, "Stardust: Tracking activity in a distributed storage system," SIGMETRICS 2006.

31 R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," NSDI 2011.

Elastic Resource Allocation in Datacenters: Gremlins in the Management Plane

Mukil Kesavan
Center for Experimental
Research in Computer
Systems (CERCS)
Georgia Institute of Technology
mukil@cc.gatech.edu

Ada Gavrilovska
Center for Experimental
Research in Computer
Systems (CERCS)
Georgia Institute of Technology
ada@cc.gatech.edu

Karsten Schwan
Center for Experimental
Research in Computer
Systems (CERCS)
Georgia Institute of Technology
schwan@cc.gatech.edu

Abstract

Virtualization has simplified the management of datacenter infrastructures and enabled new services that can benefit both customers and providers. From a provider perspective, one of the key services in a virtualized datacenter is elastic allocation of resources to work-loads, using a combination of virtual machine migration and per-server work-conserving scheduling. Known challenges to elastic resource allocation include scalability, hardware heterogeneity, hard and soft virtual machine placement constraints, resource partitions, and others. This paper describes an additional challenge, which is the need for IT management to consider two design constraints that are systemic to large-scale deployments: failures in management operations and high variability in cost. The paper first illustrates these challenges, using data collected from a 700-server datacenter running a hierarchical resource management system built on the VMware vSphere platform. Next, it articulates and demonstrates methods for dealing with cost variability and failures, with a goal of improving management effectiveness. The methods make dynamic tradeoffs between management accuracy compared to overheads, within constraints imposed by observed failure behavior and cost variability.

1. Introduction

Virtualization of physical datacenter resources enables a fluid mapping in which resource allocations can be varied elastically in response to changes in workload and resource availability. This is critical to realizing the benefits of utility computing environments like cloud computing systems, which can dynamically grow and shrink the resources allocated to customer workloads based on actual and current demands. Such elasticity of resources results in operational efficiency for cloud providers and in potential cost savings for customers.

IT managers face a number of challenges when implementing elastic resource allocation in current-generation virtualized datacenters that are often populated with tens of thousands of machines [10]. These challenges include scalability, hard-ware heterogeneity, hard and soft virtual machine (VM) placement constraints, resource partitions, and others, to which the research community has responded with novel techniques and associated system support [5, 8, 14, 13, 11].

This paper highlights the importance of two additional factors posing challenges to elastic resource management for large-scale datacenter and cloud computing systems. First, management operations may fail because the majority of these higher-level services are implemented in a best effort management plane. Second, there can be large variations in the costs of these management operations. For example, consider these three elastic resource allocation scenarios: 1) dynamic virtual machine placement to address long-term virtual machine demands, 2) live virtual machine migration or offline placement during power-on, and 3) using per-server resource schedulers for finer-grained allocation [2]. Prior work has shown that these management plane operations, including live virtual machine migration, can fail and that they exhibit varying resource costs [12]. These failures decrease the effectiveness of elastic resource allocation and variable costs complicate dealing with management overhead, relative to the benefits derived from elastic resource management. These facts, then, contribute to a 'glass ceiling' in the management plane that limits the improvements achievable by elastic resource allocation services [7, 9].

This paper illustrates the effects and importance of understanding management plane operations and their behavior, including empirical evidence of the operation failure rates and cost variability, observed in a 700-server datacenter running VMware vSphere. In this system, the base functionality of the vSphere platform has been extended with Cloud Capacity Manager (CCM), a scalable, hierarchical, elastic resource allocation system that is built on top of VMware's DRS. CCM consists of three hierarchical levels: (i) clusters (small groups of hosts as defined by DRS), (ii) superclusters (groups of clusters), and (iii) cloud (a group of superclusters). In addition to the load balancing and re-source allocation performed for virtual machines of a single cluster by DRS, CCM dynamically shuffles *capacity* be-tween clusters and superclusters in response to aggregate changes in demand.

This paper quantifies the impact of management operation failures and cost variability on CCM, and presents simple methods for coping with these issues. It concludes with a brief discussion of the broader implications this poses when designing and constructing large-scale datacenter infrastructure services. Future research points out that design for management operation failures and cost variability, explored in the context of elastic resource allocation, is more

broadly applicable to higher-level services pertaining to high availability, power management, virtual machine backups/disaster recovery, virtual machine environment replication, and more generally, to adaptive systems and control.

2. CCM Overview

The overall architecture of CCM is shown in Figure 1. Demand-aware load balancing is periodically per-formed by capacity managers at the cluster and supercluster levels, and at the overall cloud level. Capacity managers operate independently, but share with the level above (if present) the combined resource demand information of all virtual machines on the hosts they manage. Sharing, as well as load balancing, operates at progressively larger time scales when moving up the hierarchy. Based on the combined virtual machine resource demand information (including some additional headroom), an *imbalance* metric is computed at each manager, as the standard deviation of the normalized demand of sub-entities. Load balancing is triggered when this imbalance is above an administrator-specified threshold during an invocation of the algorithm, and capacity is moved from entities with low-normalized demand to those with high-normalized demand.

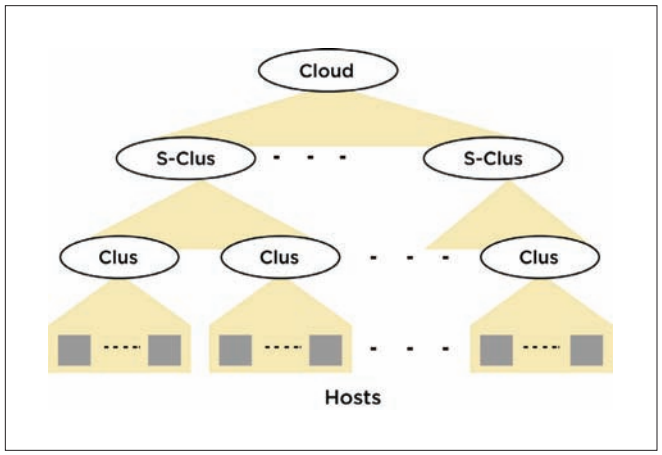

Figure 1. Hierarchical resource management architecture.

At the cluster level, VMware DRS is used to make independent and localized decisions to balance loads by migrating virtual machines using per-virtual machine demand estimates. At the supercluster and cloud levels, coarse-grained allocation changes are carried out by *logically re-associating capacity*. This process migrates individual evacuated hosts, rather than individual virtual machines, across clusters and superclusters. All virtual machines running on a host to be re-associated are migrated to other hosts that are part of the same cluster to which the host currently belongs. This is done to seamlessly integrate with DRS and to minimize the amount of state that must be moved between capacity managers during each migration. DRS automatically adapts to increased and decreased capacity in a cluster without requiring any changes.

CCM is implemented in Java, using the vSphere Java API [3] to collect metrics and enforce management actions in the vSphere provisioning layer. DRS is used in its standard vSphere server form. Both the cloud manager and supercluster managers are implemented as part of a single, multithreaded application running on a single host to make it simple to prototype and evaluate.

The remainder of this paper treats CCM as a black-box system that ingests monitoring information and emits management actions. The paper studies the management actions carried out in the management plane, or management enactment, and focus on enactment failures and variation in enactment cost.

Host-move action: A *host-move* is one of the basic actions performed by CCM at the supercluster and cloud levels, the purpose being to elastically allocate resources in the datacenter. There are two significant types of moves: host-move between clusters, and host-move between super-clusters. Each host-move is composed of a series of *macro* operations in the management plane that must be executed in order for an inter-cluster move, as shown in Figure 2. Each macro operation may be implemented by one or more lower level *micro* management plane operations.

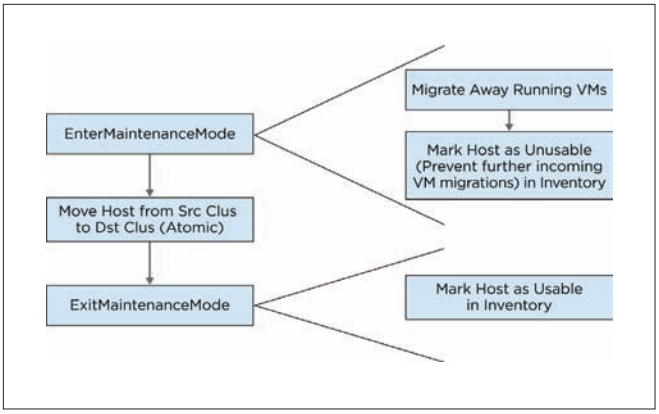

Figure 2. Inter-cluster host-move.

The "EnterMaintenanceMode" operation, for instance, places a host into a *maintenance* state, in which no virtual machines currently use it, so that this host can be moved from one cluster to another. This is potentially the most resource-intensive of these management plane operations because all virtual machines currently running on the host must be evicted. The time to complete this operation depends on factors like virtual machine size and active virtual machine memory footprints [4]. DRS selects other hosts in the source cluster and moves evicted virtual machines to those hosts.

The important point to note about these composite host-move operations is that the failure of a macro operation always results in complete failure of the whole host-move, whereas a failure of a micro-operation may or may not result in total failure depending on whether it is a pre-requisite for future operations (e.g., "Getting un-collected stats" need not result in total failure). As discussed further below, this classification of management plane operations helps when devising ways to cope with failures.

3. Failure and Cost Variability Analysis

This section outlines the experimental setup and presents data on virtual machine migration and host-move operation failures for representative large runs across 256 hosts of the 700-host datacenter. It also shows how these failures impact the performance of CCM, by defining a 'goodput' metric termed *effective management action throughput (emat)*.

Testbed: Each datacenter host has two dual-core AMD Opteron 270 processors, 4 GB of memory, and runs the VMware vSphere Hypervisor (ESXi) v4.1. The hosts are all connected to each other and four shared storage arrays (4.2 TB total capacity) via a Force 10 E1200 switch over a flat IP space. The common shared storage is exported as NFS stores to make it easy to migrate virtual machines across the datacenter machines. The open source Cobbler installation server runs on a dedicated host for DNS, DHCP, and PXE booting needs. The VMware vSphere server and client are used to provision, monitor, and partially manage the cloud.

Workload and setup: Realistic datacenter-wide load patterns are generated by replaying the resource usage traces released by Google Inc. [1]. The first four hours of the resource usage pattern of the jobs in the trace are replayed on 1024 virtual machines, 256 per job. There are a total of 16 clusters covering the 256 hosts, with each cluster initially containing 16 hosts and 64 virtual machines. A supercluster is composed of a set of 4 clusters, separate from other superclusters, with a total of 64 hosts and 256 virtual machines. This results in a total of four superclusters in the cloud. A given job's virtual machines fit within a single supercluster. A more detailed explanation of the load generation framework and trace replay appears in [6].

The load-balancing algorithm at the cluster level runs once every 5 minutes, at the supercluster level once every 20 minutes, and at the cloud level once every hour. Results for four different configurations of CCM are shown. The first configuration attempts to carry out *all* of the host-move actions recommended by the balancing algorithms during their respective invocations. Further, all of the moves are also carried out in parallel, with the intent to reduce the amount of time the system is in a state of flux. This sort of a configuration is not uncommon in practice where system developers assume a low and stable cost for enforcing management. This configuration is named CCM_nr, short for CCM with "no restrictions". In CCM_nr and the rest of the configurations, DRS is set to carry out priority 1, 2, and 3 virtual machine migration recommendations, with at most eight virtual machines per host in parallel.

Figure 3 shows the number of successful host-move operations. All of the attempted host-moves are inter-cluster moves, which are determined by the nature of the work-load. The results showed a 38% host-move failure rate and low number of successful host-moves that result in little to no effect on work-load performance (data not shown here). In addition to outright failures, there were also a large number of extremely slow operations that did not complete during the course of the experiment. In the figure, these are also counted as failures. Figure 4 presents the proportion of failures due to a failure of each of the three macro management plane operations in the inter-cluster move. It can be seen that more than 90% of the failures are due to a failure of the "EnterMaintenanceMode" operation, or in other words, a failure to evict (by migrating them away) all of the running virtual ma-chines on the hosts in question.

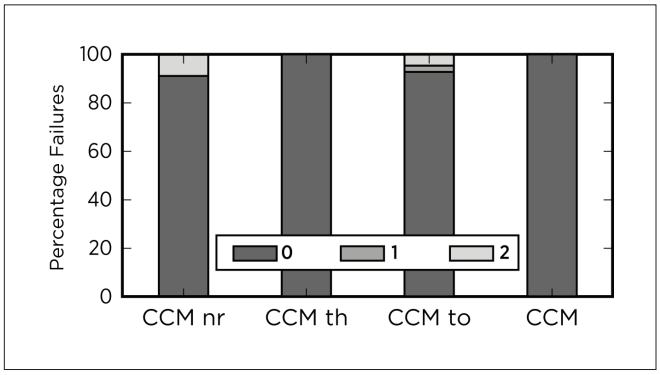

Figure 4. Fraction of inter-cluster host-move failure due to failure of each macro operation. Key: 0=EnterMaintenance-Mode, 1=Move-Host, 2=ExitMaintenanceMode.

In Figure 5, CCM_nr shows a noticeable 29% virtual machine migration failure over the course of the experiment. Note that the migration failures reported here include those due to host-moves and those performed by DRS during its load balancing. However, it still gives a general link between high migration failure rates and host-move failures given that a failure to evacuate a single running virtual machine would result in the failure of the entire operation.

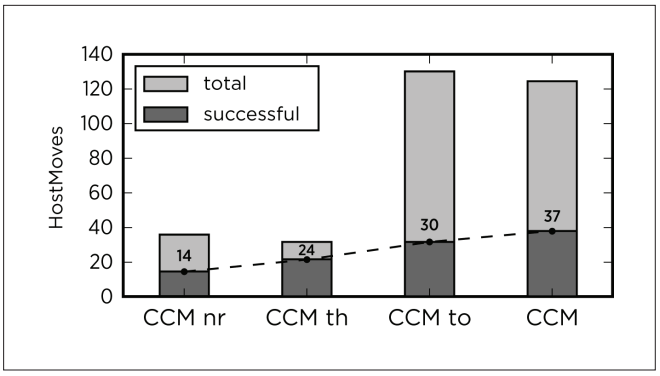

Figure 3. Number of successful host-moves.

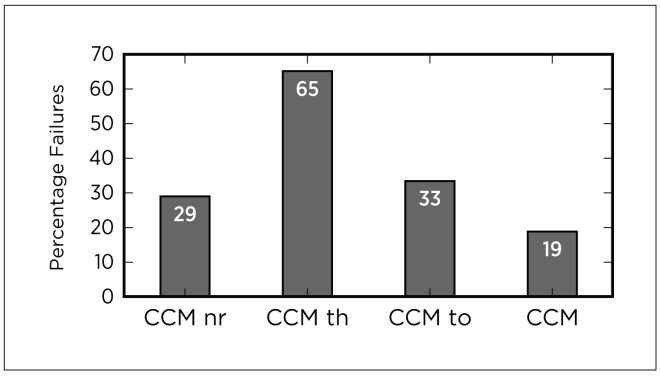

Figure 5. Migration failures.

Table 1 shows some of the major causes of VM migration failures collected from the vSphere layer and their percentage contribution to the overall number of migration failures. Some of the causes appear to be random or transient in nature, possibly due to software bugs or configuration issues, whereas others point to the fact that migrations may have failed directly or indirectly due to high resource pressure given the way CCM_nr enforces actions (e.g., "Operation timed out").

ABBV. CAUSE	nr	th	to	CCM
General system error	31	22	10	21
Failed to create journal file	45	7	63	54
Operation timed out	4	49	18	13
Operation not allowed in cur state	12	22	4	12
Insuf. host resources for VM reserv.	0	0	3	0
Changing mem greater than net BW	0	0	1	0
Data copy fail: already disconnected	8	0	0	1
Error comm. w/ dest host	0	0	1	0

Table 1: VM migration failure causes breakdown for each con-figuration. Values denote percentages.

Given this intuition, three different configurations of CCM are presented in which the degree of resource pressure imposed by management actions is controlled. Pressure is changed by: (1) explicit throttling of management enactions, i.e., the number and parallelism of host-moves (CCM_th), (2) automatically aborting long-running actions using timeouts (CCM_to) and, (3) a combination of both action throttling and timeouts (CCM). Configurations differ in their use of values for throttling host-moves, i.e., by limiting the maximum number of host-moves per balancing period to eight and reducing the parallelism in moves to no more than four at a time for each supercluster. In addition, a static timeout setting of 16 minutes is used for a single host-move operation for CCM_to and CCM. Experimental results with these configurations test the hypothesis that the resource pressure induced by management causes the observed failures.

The CCM_th, CCM_to, and CCM configurations exhibit higher success rates (41%, 53%, and 62%, respectively, compared to CCM_nr) in the host-move operations performed, as shown in Figure 3. This success points to the fact that there is a quantifiable benefit in managing the resource pressure of management action enforcement. Note that the monitoring and load-balancing algorithms, and the workload, remain the same for all configurations. In the case of CCM_to and CCM, both configurations achieve a much higher success rate while also attempting almost three times as many host moves as CCM_nr and CCM_th.

This is because a timeout allows stopping long running host-moves where the cost-to-workload benefit ratio is unfavorable. If the load imbalance in the workload continues to persist, the balancing algorithms recommend a fresh set of moves during the next round that may be more cost effective. In this fashion, the higher-level service could explore more efficient host-move alternatives than those available at a prior point in time.

To better quantify the behavior exhibited in the experiments discussed above required a new to more objectively compare the different configurations: *effective management action throughput (emat)*—the ratio of the number of successful host-moves to the total amount of time spent in making all moves (successful and failed). Table 2 shows the total minutes spent performing the host-moves and the emat metric for each of the four configurations. The total time is summed over all host-moves attempted by each configuration, counting parallel moves in serial order. Even though CCM and CCM_to perform nearly three times as many moves (see Figure 3) as the other two configurations, the total time spent enforcing the moves was significantly lower. This, combined with the fact that these configurations also delivered a higher host-move success rate, results in a two-to-six fold advantage in terms of the *emat* metric.

METRIC	CCM_nr	CCM_th	CCM_to	CCM
Total Host-move Mins	4577	703	426	355
emat (moves/hr)	0.18	2.05	4.23	6.25

Table 2: Management Metrics

Note that for the CCM_nr and CCM_th configurations, the balancing algorithms cannot start recommending and enforcing new host-moves while the previous moves are still in progress and the system is in an unstable state. This is the reason for the rather smaller number of host-move attempts in both of these cases. The large proportion of host-move failures for CCM_to and CCM are due to a combination of explicit aborts and the fact that the configurations are still afflicted by a non-negligible fraction of virtual machine migration failures as shown in Figure 5.

In addition to the host-move failures, it is also important to consider the cost-to-benefit ratio of the enforced management actions. Figure 6 depicts the average, minimum, and maximum values of successful host-move action times for all four configurations. Given that the resource cost of an action is directly proportional to

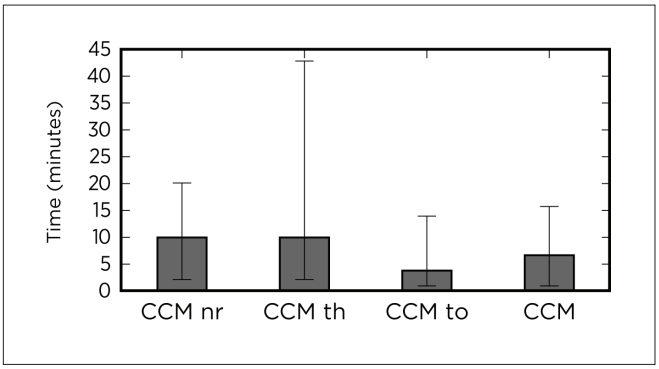

Figure 6. Host-move times.

the amount of time the action takes, it is important to abort overly long-running host-moves, which is illustrated with the more stable host-move times observed with CCM_to and CCM, each of which

are more likely to deliver a favorable cost-to-benefit ratio. Further, since the most resource-intensive stage of a host-move operation is the "EnterMaintenanceMode" operation, with its need to evict all of the current virtual ma-chines running on a host, this operation is particularly affected by high variations in virtual machine migration times. Variability in these times is evident from the fact that the 90th percentile values of virtual machine migration times computed for CCM_nr, CCM_th, CCM_to, and CCM show high values of 377s, 320s, 294s, and 335s, respectively. These variations have two causes: those attributable to management pressure and natural causes due to differences in virtual behavior, leading to runtime variations in the active memory footprints. The latter are beyond the control of the management layer, but demonstrate that host-move times can vary widely, and as a result, the abort/search method of exploring more efficient moves is still relevant. Alternatively, the system can also explicitly track and predict management costs and make moves when the moves will produce the most favorable cost-to-benefit ratio.

4. Conclusions

This paper draws attention to the importance of designing for failures, not only for the applications running in large-scale datacenter systems, but also and perhaps even more importantly, for the management actions that are in-tended to improve application performance. Intuitively, this is because management failures can have a substantial effect on the efficiency of datacenter operation. First, because failed actions consume resources without contributing to the desired improvements and second, because the resource pressure induced by such actions can lead to action failures or undue delays. Therefore, it is important to design a data-center's management plane that considers management failures as well as variability in the costs of enforcing actions in the management plane.

The experimental results shown in this paper illustrate the efficacy of simple methods for improving otherwise cost-variable and/or failure-intolerant management action. Methods include explicit action throttling to reduce the resource pressure imposed by such actions, and aborts that prevent undue resource consumption by actions experiencing delays. Results shown in the paper use static settings to test the usefulness of the action throttling and early aborts. Future work will develop techniques to dynamically derive these parameters. Additional experiments are in process with alternative strategies to hedge against failures that cannot be controlled, in order to minimize overall management cost, reduce failure rates, and maximize the bene-fits to application workloads.

References

1 googleclusterdata: traces of google tasks running in a production cluster, http://code.google.com/p/googleclusterdata/

2 VMware DRS, http://www.vmware.com/products/DRS

3 VMware VI (vSphere) Java API, http://vijava.sourceforge.net/

4 C. Clark et al. Live migration of virtual machines. In NSDI'05.

5 D. Gmach et al. Workload analysis and demand prediction of enterprise datacenter applications. In IISWC '07.

6 M. Kesavan et al. Xerxes: Distributed load generator for cloud-scale experimentation. Open Cirrus Summit, 2012.

7 S. Kumar et al. vManage: loosely coupled platform and virtualization management in datacenters. ICAC '09.

8 X. Meng et al. Efficient resource provisioning in compute clouds via virtual machine multiplexing. In ICAC '10.

9 R. Moreno-Vozmediano et al. Elastic management of cluster-based services in the cloud. ACDC '09.

10 D. Schneider et al. Under the hood at Google and Facebook. Spectrum, IEEE, 48(6):63 –67, June 2011.

11 Z. Shen et al. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In SoCC '11.

12 V. Soundararajan et al. The impact of management operations on the virtualized datacenter. In ISCA '10.

13 T. Wood et al. Black-box and gray-box strategies for virtual machine migration. In NSDI'07.

14 X. Zhu et al. 1000 islands: Integrated capacity and workload management for the next generation datacenter. In ICAC '08.

Toward a Paravirtual vRDMA Device for VMware ESXi Guests

Adit Ranadive[1]
Georgia Institute of Technology
adit.ranadive@gatech.edu

Bhavesh Davda
VMware, Inc.
bhavesh@vmware.com

Abstract

Paravirtual devices are common in virtualized environments, providing improved virtual device performance compared to emulated physical devices. For virtualization to make inroads in High Performance Computing and other areas that require high bandwidth and low latency, high-performance transports such as InfiniBand, the Internet Wide Area RDMA Protocol (iWARP), and RDMA over Converged Ethernet (RoCE) must be virtualized.

We developed a paravirtual interface called Virtual RDMA (vRDMA) that provides an RDMA-like interface for VMware ESXi guests. vRDMA uses the Virtual Machine Communication Interface (VMCI) virtual device to interact with ESXi. The vRDMA interface is designed to support snapshots and VMware vMotion® so the state of the virtual machine can be easily isolated and transferred. This paper describes our vRDMA design and its components, and outlines the current state of work and challenges faced while developing this device.

Categories and Subject Descriptors
C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks- *High-speed*; C.4 [**Performance of Systems**]: Modeling techniques; D.4.4 [**Operating Systems**]: Communication Management- *Network Communication*;
General Terms
Algorithms, Design, High Performance Computing, Management
Keywords
InfiniBand, Linux, RDMA, Subnet Management, Virtualization, virtual machine

1 Adit was an intern at VMware when working on this project.

1. Introduction

Paravirtualized devices are common in virtualized environments [1-3] because they provide better performance than emulated devices. With the increased importance of newer high-performance fabrics such as InfiniBand, iWARP, and RoCE for Big Data, High Performance Computing, financial trading systems, and so on, there is a need [4-6] to support such technologies in a virtualized environment. These devices support zero-copy, operating system-bypass and CPU offload [7-9] for data transfer, providing low latency and high throughput to applications. It is also true, however, that applications running in virtualized environments benefit from features such as vMotion (virtual machine live migration), resource management, and virtual machine fault tolerance. For applications to continue to benefit from the full value of virtualization while also making use of RDMA, the paravirtual interface must be designed to support these virtualization features.

Currently there are several ways to provide RDMA support in virtual machines. The first option, called passthrough (or VM DirectPath I/O on ESXi), allows virtual machines to directly control RDMA devices. Passthrough also can be used in conjunction with single root I/O virtualization (SR-IOV) [9] to support the sharing of a single hardware device between multiple virtual machines by passing through a Virtual Function (VF) to each virtual machine. This method, however, restricts the ability to use virtual machine live migration or to perform any resource management. A second option is to use a software-level driver, called SoftRoCE [10], to convert RDMA Verbs operations into socket operations across an Ethernet device. This technique, however, suffers from performance penalties and may not be a viable option for some applications.

With that in mind, we developed a paravirtual device driver for RDMA-capable fabrics, called Virtual RDMA (vRDMA). It allows multiple guests to access the RDMA device using a *Verbs API*, an industry-standard interface. A set of these *Verbs* was implemented to expose an RDMA-capable guest device (vRDMA) to applications. The applications can use the vRDMA guest driver to communicate with the underlying physical device. This paper describes our design and implementation of the vRDMA guest driver using the VMCI virtual device. It also discusses the various components of vRDMA and how they work in different levels of the virtualization stack. The remainder of the paper describes how RDMA works, the vRDMA architecture and interaction with VMCI, and vRDMA components. Finally, the current status of vRDMA and future work are described.

2. RDMA

The Remote Direct Memory Access (RDMA) technique allows devices to read/write directly to an application's memory without interacting with the CPU or operating system, enabling higher throughput and lower latencies. As shown on the right in Figure 1, the application can directly program the network device to perform DMA to and from application memory. Essentially, network processing is pushed onto the device, which is responsible for performing all protocol operations. As a result, RDMA devices historically have been extremely popular for High Performance Computing (HPC) applications [6, 7]. More recently, many clustered enterprise applications, such as databases, file systems and emerging Big Data application frameworks such as Apache Hadoop, have demonstrated performance benefits using RDMA[6, 11, 12].

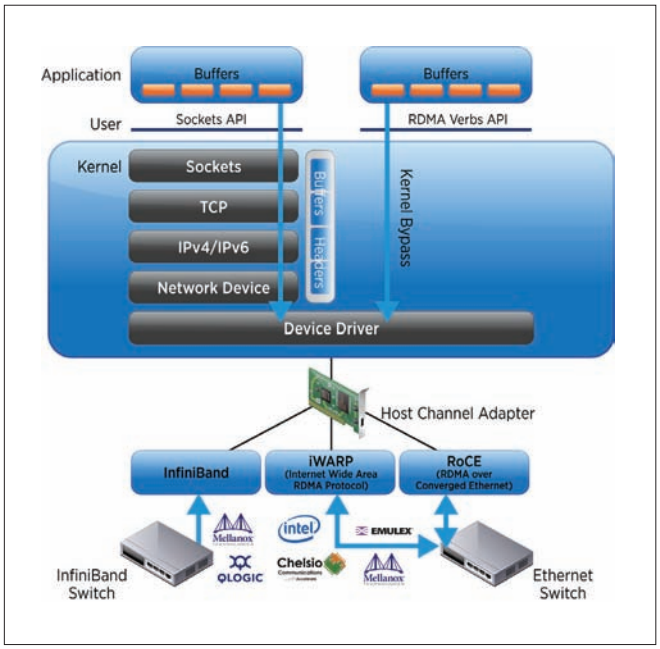

Figure 1. Comparing RDMA and Sockets

While data transfer operations can be performed directly by the application as described above, control operations such as allocation of network resources on the device need to be executed by the device driver in the operating system for each application. This allows the device to multiplex between various applications using these resources. After the control path is established, the application can directly interact with the device, programming it to perform DMA operations to other hosts, a capability often called *OS-bypass*. RDMA also is said to support *zero-copy* since the device directly reads/writes from/to application memory and there is no buffering of data in the operating system. This offloading of capabilities onto the device, coupled with direct user-level access to the hardware, largely explains why such devices offer superior performance. The next section describes our paravirtualized RDMA device, called Virtual RDMA (vRDMA).

3. vRDMA over VMCI Architecture

Figure 2 illustrates the vRDMA prototype. The architecture is similar to any virtual device, with a driver component at the guest level and another at the hypervisor level that is responsible for communicating with the physical device. In the case of our new device, we include a modified version of the OpenFabrics RDMA stack within the hypervisor that implements the core Verbs required for RDMA devices. Using this stack allows us to be agnostic with respect to RDMA transport, enabling the vRDMA device to support InfiniBand (IB), iWARP, and RoCE.

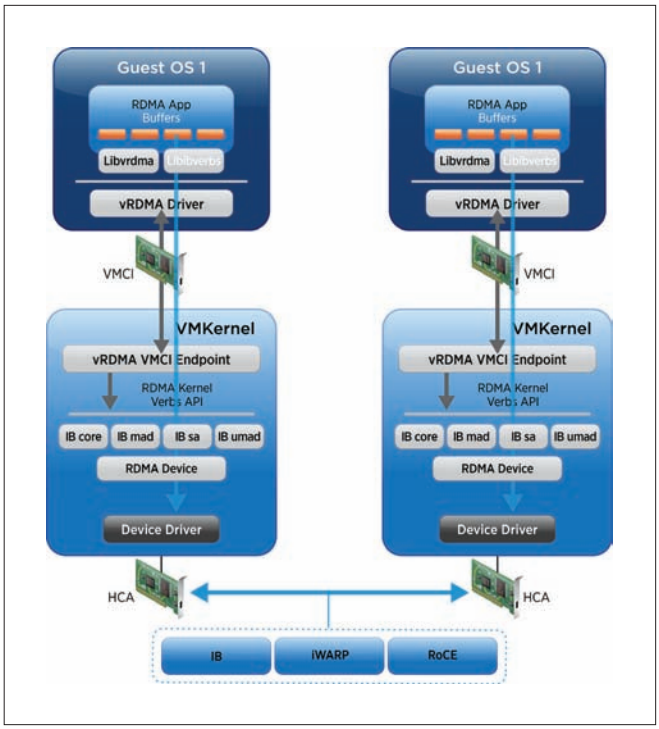

Figure 2. vRDMA over VMCI Architecture[2].

We expose RDMA capabilities to the guest using VMCI [13], a virtual PCI device that supports a point-to-point bidirectional transport based on a pair of memory-mapped queues and datagrams serving as asynchronous notifications. Using VMCI, we construct communication endpoints between each guest and an endpoint in the hypervisor called the *vRDMA VMCI endpoint*, as shown in Figure 2. All guests connect with the hypervisor endpoint when the vRDMA driver is loaded in the guests.

To use RDMA in a virtual environment, guest operating systems use the standard OpenFabrics Enterprise Distribution (OFED) RDMA stack, along with our guest kernel vRDMA driver and a user-level library (**libvrdma**). These additional components could be distributed using VMware Tools, which already contain the VMCI guest virtual device driver. The OFED stack, the device driver, and library provide an implementation of the industry-standard Verbs API for each device.

2 The "RDMA Device" shown here and in other figures refers to a software abstraction in the I/O stack in the VMKernel.

Our guest kernel driver communicates with the vRDMA VMCI endpoint using VMCI datagrams that encapsulate Verbs and their associated data structures. For example, a 'Register Memory' Verb datagram contains the memory region size and guest physical addresses associated with the application.

When the VMKernel VMCI Endpoint receives the datagram from the guest, it handles the Verb command in one of two ways, depending on whether the destination virtual machine is on the same physical machine (intra-host) or a different machine (inter-host).

The vRDMA endpoint can determine if two virtual machines are on the same host by examining the QP numbers and LIDs[14] assigned to the virtual machines. Once it has determined the virtual machines are on the same host, it emulates the actual RDMA operation. For example, when a virtual machine issues an RDMA Write operation, it specifies the source address, destination address, and data size. When the endpoint receives the RDMA Write operation in the Post_Send Verb, it performs a memory copy into the address associated with the destination virtual machine. We can further extend the emulation to the creation of queue pairs (QPs), CQs, MRs (see [15] for a glossary of terms) and other resources such that we can handle all Verbs calls in the endpoint. This method is described in more detail in Section 4.4.

In the inter-host case, the vRDMA endpoint interacts with the ESXi RDMA stack as shown in Figure 2. When a datagram is received from a virtual machine, it checks to see if the Verb corresponds to a creation request for communication resources, such as Queue Pairs. These requests are forwarded to the ESXi RDMA stack, which returns values after interacting with the device. The endpoint returns results to the virtual machine using a VMCI datagram. When the virtual machine sends an RDMA operation command, such as RDMA Read, RDMA Write, RDMA Send, or RDMA Receive, the endpoint directly forwards the Verbs call to the RDMA stack since it already knows it is an inter-host operation.

4. Components of vRDMA

This section describes the main components of the vRDMA paravirtual device and their interactions.

4.1 libvrdma

The **libvrdma** component is a user-space library that applications use indirectly when linking to the **libibverbs** library. Applications (or middleware such as MPI) that use RDMA link to the device-agnostic **libibverbs** library, which implements the industry-standard Verbs API. The **libibverbs** library in turn links to **libvrdma**, which enables the application to use the vRDMA device.

Verbs are forwarded by **libibverbs** to **libvrdma**, which in turn forwards the Verbs to the main RDMA module present in the guest kernel using the Linux /**dev** file system. This functionality is required to be compatible with the OFED stack, which communicates with all underlying device drivers in this way. Figure 3 illustrates the RDMA application executing a '*Query_Device*' Verb.

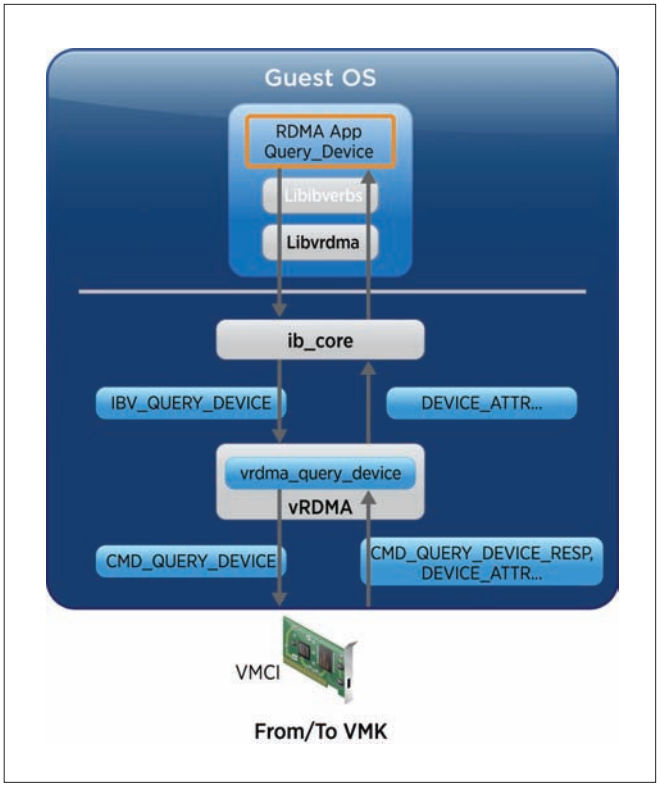

Figure 3. Guest vRDMA driver and libvrdma

4.2 Guest Kernel vRDMA Driver

The OFED stack provides an implementation of the kernel-level Verbs API called the **ib_core** framework. The framework allows device drivers to register themselves using callbacks for each Verb. Each device driver must provide implementations of a mandatory list of Verbs (Table 1). Therefore, our vRDMA guest kernel driver must implement this list of Verbs to register successfully with the OFED stack. Underneath these Verbs calls we communicate with the vRDMA endpoint to ensure valid responses for each Verb. Next, using the **Query_Device** Verb as an example, we describe how the Guest kernel driver handles Verbs.

Query_Device	Modify_QP
Query_Port	Query_QP
Query_Gid	Destroy_QP
Query_Pkey	Create_CQ
Create_AH	Modify_CQ
Destroy_AH	Destroy_CQ
Alloc_Ucontext	Poll_CQ
Dealloc_Ucontext	Post_Send
Alloc_PD	Post_Recv
Dealloc_PD	Reg_User_MR
Create_QP	Dereg_MR

Table 1: Verbs supported in the vRDMA prototype.

As shown in Figure 3, the **Query_Device** Verb is executed by the application. It is passed to the **ib_core** framework, which calls the **Query_Device** function in our vRDMA kernel module using the registered callback. The vRDMA guest kernel module packetizes the Verbs call into a buffer to be sent using a VMCI datagram. This datagram is sent to the vRDMA VMKernel VMCI Endpoint that handles all requests from virtual machines and issues responses.

4.3 ESXi RDMA stack

The ESXi RDMA stack is an implementation of the OFED stack that resides in VMKernel and contains device drivers for RDMA devices. In our prototype, the ESXi RDMA stack mediates access to RDMA hardware on behalf of our vRDMA device in the guest. Other hypervisor services, such as vMotion and Fault Tolerance, could use this stack to access the RDMA device.

4.4 VMKernel vRDMA VMCI Endpoint

The main role of the VMKernel vRDMA VMCI endpoint is to receive requests from virtual machines and send appropriate responses back by interacting with the ESXi RDMA stack. Most requests from virtual machines are in the form of Verbs calls. Responses depend on the location of the destination virtual machine. As mentioned in Section 3, the vRDMA endpoint handles the Verb command in two ways depending on whether the destination virtual machine is on the same host. To decide whether the virtual machine is on the same host, the endpoint consults a list of communication resources used by the virtual machine: QP numbers, CQ number, MR entries, and LIDs. These identify the communication taking place and, therefore, the virtual machines.

For example, the endpoint can identify the destination virtual machine when the source virtual machine issues a Modify QP Verb, which includes the destination QP number and LID[14]. It matches these with its list of virtual machine communication resources, connecting the two virtual machines in the endpoint when it finds a match. Once connected, the Modify QP/CQ Verb is not forwarded to the RDMA stack. Instead, the endpoint returns values to the virtual machine, stating the Verb completed successfully. When the source virtual machine subsequently executes an RDMA data transfer operation, the vRDMA endpoint performs a memory copy based on the type of operation. For example, when a virtual machine issues an RDMA Write, it specifies the source, destination address, and data size. The endpoint performs the memory copy when it receives the Verb. Figure 4 shows the vRDMA architecture when virtual machines reside on the same host.

When virtual machines are determined to be on different hosts based on a previous Modify QP Verb call, the vRDMA endpoint forwards any Verbs received to the RDMA stack in the VMKernel. After this check, the endpoint forwards all Verbs calls to the RDMA stack. The RDMA stack and the physical device are responsible for completing the Verb call. Once the Verb call completes, the endpoint accepts the return values from the RDMA stack and sends them back to the source virtual machine using VMCI datagrams.

Figure 5 shows the **Query_Device** Verb being received by the vRDMA endpoint. As an optimization, the values for such Verbs can be cached in the endpoint. Therefore, the first Verb call is forwarded to the RDMA stack, which sends a Management

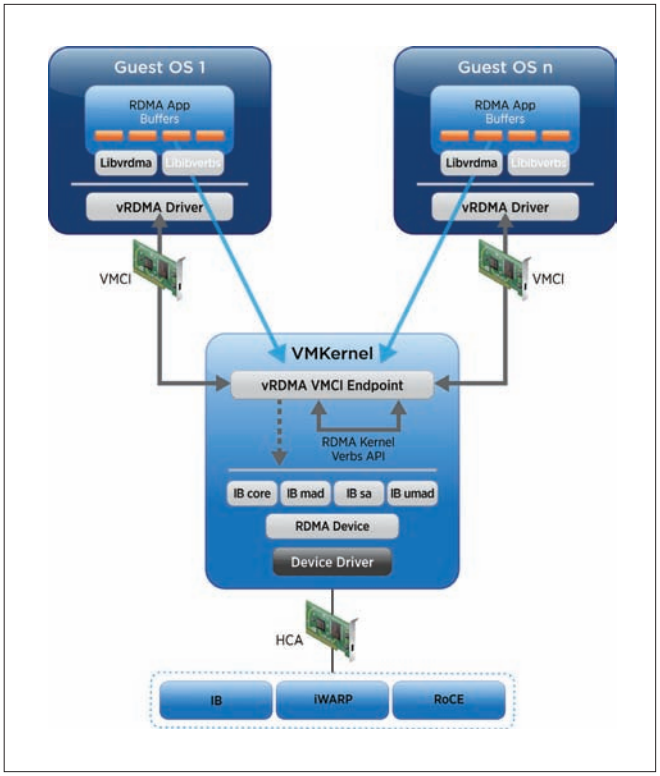

Figure 4. vRDMA architecture for virtual machines on the same host

Datagram (MAD) to the device to retrieve the device attributes. Additional **Query_Device** Verb calls from the virtual machine can be returned by the endpoint using the cached values, reducing the number of MADs sent to the device. This can be extended to other Verbs calls. The advantage of this optimization: mimicking or emulating the Verbs calls enables RDMA device attributes to be provided without a physical RDMA device being present.

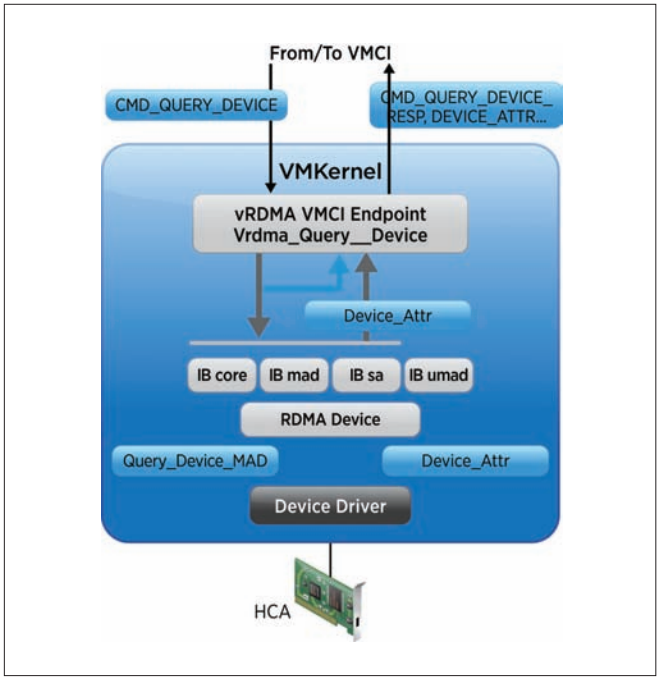

Figure 5. VMKernel vRDMA Endpoint and ESXi RDMA stack

Emulating these Verbs calls enables us to store the actual device state, containing the QP, CQ, and MR structures within the vRDMA endpoint, which is extremely useful in allowing RDMA-based applications to run on machines without RDMA devices and without modifying the applications to use another transport. This is an important attribute of our vRDMA solution.

5. Current Status and Future Directions

The vRDMA prototype is feature complete, with some testing and debugging remaining. We expect half-roundtrip vRDMA latencies to be about 5µs, lower than the SoftRoCE over vmxnet3[19] option, but higher than what one can achieve in the bare-metal, passthrough, or SR-IOV VF cases. We intend to measure and report latencies and bandwidths for our prototype when testing is completed.

5.1 Supporting RDMA without RDMA Devices

The Verbs API is an abstraction for the actual functionality of the RDMA device, and device drivers provide their own implementation of these verbs to register with **ib_core**. It is possible for the device driver to emulate Verbs by returning the expected response to the **ib_core** framework without interacting with the device. In our prototype, the guest vRDMA driver acts as the RDMA device driver and the vRDMA endpoint acts as the RDMA device, emulating Verbs calls. This layering and abstraction enables the vRDMA endpoint to use any network device and support Verbs-based applications without a physical RDMA device being present.

5.2 Support for Checkpoints and vMotion

One of the main advantages of a paravirtualized interface is the ability to support snapshots and vMotion. Because the state of the vRDMA device is fully contained in guest physical memory and VMCI device state, features such as checkpoints, suspend/resume, and vMotion can be enabled. Additional work will be required to tear down and rebuild underlying RDMA resources (QPs and MRs) during vMotion operations. This is work we are interested in exploring.

5.3 Subnet Management

One of the bigger challenges is to integrate paravirtual RDMA interfaces with subnet management. Consider the InfiniBand case in which the Subnet Manager (SM) assigns Local IDs (LIDs) [14] to IB ports and Global IDs (GIDs) to HCAs. One way to maintain addressability is to let ESXi query the IB SM for a list of unique LIDs and GIDs assignable to the virtual machines. In a large cluster with multiple virtual machines per host, the 16-bit range limits the number of virtual machines with unique LIDs. We might need to modify the SM to provide more LIDs in the subnet with virtual machines. Another alternative is to extend VMware® vCenter™ to be the "subnet manager" for virtual RDMA devices, and assign unique LIDs and GIDs within the vCenter cluster.

6. Related Work

While virtualization is very popular in enterprises, it has not made significant inroads with the HPC community. This can be attributed to the lack of support for high-performance interconnects and perceived performance overhead due to virtualization. There has been progress toward providing access to high-performance networks such as InfiniBand [4, 16] to virtual machines. With our prototype, we do not expect to meet the latencies as shown in [4]. We can, however, offer all the virtualization benefits at significantly lower latencies than alternative approaches based on traditional Ethernet network interface cards (NICs).

While there has been work to provide the features of virtualization [17, 18] to virtual machines, these approaches have not been widely adopted. Therefore, the disadvantage of this virtual machine monitor (VMM)-bypass approach is the loss of some of the more powerful features of virtualization, such as snapshots, live migration, and resource management.

7. Conclusion

This paper describes our prototype of a paravirtual RDMA device, which provides guests with the ability to use an RDMA device while benefiting from virtualization features such as checkpointing and vMotion. vRDMA consists of three components:

- **libvrdma**, a user-space library that is compatible with the Verbs API
- Guest kernel driver, a Linux-based module to support the kernel-space Verbs API
- A VMkernel vRDMA Endpoint that communicates with the Guest kernel driver using VMCI Datagrams

A modified RDMA Stack in the VMKernel is used so the vRDMA endpoint can interact with the physical device to execute the Verbs calls sent by the guest. An optimized implementation of the vRDMA device was explained, in which the data between virtual machines on the same host is copied without device involvement. With this prototype, we expect half round trip latencies of approximately 5µs since our datapath passes through the vRDMA endpoint and is longer than that of the bare-metal case.

Acknowledgements

We would like to thank Andy King for his insight into our prototype and for lots of help in improving our understanding of VMCI and vmware-tools. We owe a deep debt of gratitude to Josh Simons, who has been a vital help in this project and for his invaluable comments on the paper.

References

1. *The VMWare ESX Server*. Available from: http://www.vmware.com/products/esx/
2. Barham, P., et al. *Xen and the Art of Virtualization*. In Proceedings of SOSP, 2003.
3. *Microsoft Hyper-V Architecture*. Available from: http://msdn.microsoft.com/en-us/library/cc768520.aspx
4. Liu, J., et al. *High Performance VMM-Bypass I/O in Virtual Machines*. In Proceedings of *USENIX Annual Technical Conference*, 2006.
5. Ranadive, A., et al. *Performance Implications of Virtualizing Multicore Cluster Machines*. In Proceedings of *HPCVirtualization Workshop*, 2008.
6. Simons, J. and J. Buell, *Virtualizing High Performance Computing*. SIGOPS Oper. Syst. Rev., 2010.
7. Liu, J., J. Wu, and D.K. Panda, *High Performance RDMA-Based MPI Implementation over InfiniBand*. International Journal of Parallel Programming, 2004.
8. Liu, J. *Evaluating Standard-Based Self-Virtualizing Devices: A Performance Study on 10 GbE NICs with SR-IOV Support*. In Proceedings of *International Parallel and Distributed Processing Symposium*, 2010.
9. Dong, Y., Z. Yu, and G. Rose. *SR-IOV Networking in Xen: Architecture, Design and Implementation*. In Proceedings of *Worksop on I/O Virtualization*, 2008.
10. SystemFabricWorks. *SoftRoCE*. Available from: http://www.systemfabricworks.com/downloads/roce
11. Sayantan Sur, H.W., Jian Huang, Xiangyong Ouyang and Dhabaleswar K. Panda. *Can High-Performance Interconnects Benefit Hadoop Distributed File System?* In Proceedings of *Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds*, 2010.
12. Mellanox Technologies. *Mellanox Unstructured Data Accelerator (UDA)*. 2011; Available from: http://www.mellanox.com/pdf/applications/SB_Hadoop.pdf
13. VMware, Inc. VMCI API; Available from: http://pubs.vmware.com/vmci-sdk/
14. Wickus Nienaber, X.Y., Zhenhai Duan, *LID Assignment In InfiniBand Networks*. IEEE Transactions on Parallel and Distributed Systems, 2009.
15. InfiniBand Trade Association. *InfiniBand Architecture Specification, Release 1.2.*
16. Huang, W., J. Liu, and D.K. Panda. *A Case for High Performance Computing with Virtual Machines*. In Proceedings of *International Conference on Supercomputing*, 2006.
17. Huang, W., et al. *High Performance Virtual Machine Migration with RDMA over Modern Interconnects*. In Proceedings of *IEEE Cluster*, 2007.
18. Huang, W., et al. *Virtual Machine Aware Communication Libraries for High Performance Computing*. In Proceedings of *Supercomputing*, 2007.
19. VMware KB 1001805, "Choosing a network adapter for your virtual machine": http://kb.vmware.com/kb/1001805

Intrusion Detection Using VProbes

Alex Dehnert
VMware, Inc./Massachusetts Institute of Technology
adehnert@mit.edu

Abstract

Many current intrusion detection systems (IDS) are vulnerable to intruders because they run under the same operating system as a potential attacker. Since an attacker often attempts to co-opt the operating system, the IDS is vulnerable to subversion. While some systems escape this flaw, they generally do so by modifying the hypervisor. VMware® VProbes technology allows administrators to look inside a running virtual machine, set breakpoints, and inspect memory from a virtual machine host. We aim to leverage VProbes to build an IDS for Linux guests that is significantly harder for an attacker to subvert, while also allowing the use of a common off-the-shelf hypervisor.

1. Introduction

A common mechanism for defending computers against malicious attackers uses intrusion detection systems (IDSes). Network IDSes monitor network traffic to detect intrusions, while host-based IDSes monitor activity on specific machines. A common variety of host-based IDSes watches the kernel-application interface, monitoring the system calls that are used [2][4][7][8]. Based on the sequences of system calls used and their arguments, these IDSes aim to determine whether or not an attack is underway.

While intrusion detection systems are not fully effective, they have proven to be useful tools for catching some attacks. Since a host-based IDS runs on the host it is protecting, it is vulnerable to a virus or other attacker that seeks to disable it. An attacker might block network connectivity the IDS requires to report results, disable hooks it uses to gather information, or entirely kill the detection process. This is not a theoretical risk. Viruses in the wild, such as SpamThru, Beast, Win32.Glieder.AF, or Winevar[6] directly counter anti-virus software installed on their hosts.

The robustness of a host-based IDS can be improved by running it on the outside of a virtual machine, using capabilities exposed by the hypervisor to monitor the virtual machine, and gather information the agent in the guest would ordinarily use.

2. Design

VMware ESXi™ supports VProbes, a mechanism for examining the state of an ESXi host or virtual machine, similar to the Oracle Solaris Dynamic Tracing (DTrace) facility in the Oracle Solaris operating system. VProbes allows users to place user-defined probes in the ESXi kernel (VMkernel), the monitor, or within the guest. Probes are written in a C-like language called Emmett, and perform computation, store data, and output results to a log on the ESXi host. While primarily used to diagnose performance or correctness issues, VProbes also can be used to supply data to the IDS.

Our IDS is architected as two components:

- The **gatherer** uses VProbes to retrieve system call information from the guest virtual machine. This allows it to run outside of the guest while still gathering information from within the guest.
- The **analyzer** uses the data gathered to decide whether or not a system call sequence is suspicious. The analysis component is similar to components in other intrusion detection systems, and can use the same types of algorithms to identify attacks.

One advantage of splitting the gatherer from the analyzer is modularity. Two major variants of the gatherer exist currently: one for Linux and one for Microsoft Windows, with specialized code for 32-bit versus 64-bit Linux and the different operating system versions. All of these variants produce the same output format, enabling attack recognition strategies to be implemented independently in the analysis component. The gatherer does not need to change based on the attack recognition strategy in use. The analysis component can be oblivious to the operating system, architecture, or version. In addition, it is possible to run several analyzers in parallel and combine results. Running the analyzers with saved data rather than live data could be useful for regression testing of analyzers or detecting additional past exploits as improved analyzers are developed.

The division is as strict as it is for a different reason: language. VProbes provides quite limited support for string manipulations and data structures. Additionally, the interpreter has relatively low limits on how much code probes can include. While these limitations likely are solvable, separating them required significantly less work and allows the analyzer to use the functionality of Python or other modern languages without reimplementation.

The gatherer essentially outputs data that looks like the output of the Linux **strace** utility, with names and arguments of some system calls decoded. Depending on what seems most useful to the analysis component, this may eventually involve more or less symbolic decoding of names and arguments.

The gatherer also is responsible for outputting the thread, process, and parent process IDs corresponding to each system call, as well as the program name (**comm** value or Microsoft Windows equivalent, **ImageFile**Name, and binary path). Analysis scripts use this data to

build a model of normal behavior and search for deviations. Generally, these scripts build separate models for each program, as different programs have different normal behavior.

3. Implementation

3.1 Gatherer

The data gathering component uses VProbes to gather syscall traces from the kernel. Gatherers exist for 32-bit and 64-bit Linux (across a wide range of versions), and 64-bit Microsoft Windows 7. The Linux gatherers share the bulk of their code. The Microsoft Windows gatherer is very similar in structure and gathers comparable data, but does not share any code.

To run the gatherer, VProbes sets a breakpoint on the syscall entry point in the kernel code. When a syscall starts to execute the breakpoint activates, transferring execution to our callback. The callback extracts the system call number and arguments from the registers or stack where it is stored. In addition, the probe retrieves data about the currently running process that the analysis components need to segregate different threads to properly sequence the system calls being used and associate the syscalls with the correct per-program profile.

Optionally, the gatherer can decode system call names and arguments. The Linux callback has the name and argument format for several system calls hardcoded. For numeric arguments, the probe simply prints the argument value. For system calls that take strings or other types of pointers as arguments, it prints the referenced data. It also prints the name of the system call in use. Since the current analysis scripts do not examine syscall arguments, this capability was not implemented for the Microsoft Windows gatherer.

Another optional feature reports the return values from system calls. As with argument values, current analysis scripts do not utilize this data. Consequently, while support is available for Linux, it was not implemented for Microsoft Windows.

Writing a gatherer requires two key steps. First, relevant kernel data structures storing the required information must be identified. Second, the offsets of specific fields must be made available to the Emmett script. While the general layout changes little from one version of the kernel to another, the precise offsets do vary. As a result, an automated mechanism to find and make available these offsets is desirable.

3.1.1 Relevant Kernel Data Structures

The first step in implementing the gatherer is to find where the Linux or Microsoft Windows kernel stores the pertinent data. While a userspace IDS could use relatively well-defined, clearly documented, and stable interfaces such as system calls, or read /**proc** to gather the required information, we are unable to run code from the target system. As a result, we must directly access kernel memory. Determining the relevant structures is a process that involves reading the source, disassembling system call implementations, or looking at debugging symbols.

In Linux, the key data structure is the **struct task_struct**. This contains pid (the thread ID), tgid (the process ID), and pointers to the parent process's **task_struct**. We output the thread and process IDs, as well as the parent process ID, to aid in tracking fork calls. On Microsoft Windows, broadly equivalent structures exist (Table 1).

	LINUX	MICROSOFT WINDOWS
Key structure	**task_struct**	**ETHREAD** **EPROCESS**
Breakpoint at	**syscall_call, sysenter_do_call (32-bit)**; **system_call (64-bit)**	**nt!KiSystemServiceStart**
Thread ID	**syscall**	**Cid.UniqueThread**
Process ID	**tgid**	**Cid.UniqueProcess**
Parent PID	**parent->tgid**	**InheritedFromUniqueProcessId**
Program name	**comm**	**ImageFileName**
Program binary	**mm->exe_file**	**SeAuditProcessCreationInfo**

Table 1: Key fields in the Linux and Microsoft Windows process structures

Identifying the running program is surprisingly difficult. The simplest piece of information to retrieve is the **comm** field within the Linux **task_struct**. This field identifies the first 16 characters of the process name, without path information. Unfortunately, this makes it difficult to distinguish an **init** script (which tends to use large numbers of **execve** and **fork** calls) from the program it starts (which may never use **execve** or **fork**). Hence, full paths are desirable.

Path data is available through the **struct mm_struct**, referenced by the **mm** field of the **task_struct**. By recursively traversing the **mount** and **dentry** structures referenced through the **mm_struct exe_file** field, the full path of the binary being executed can be retrieved. Since **exe_file** is the executed binary, the entry for shell scripts tends to be **/bin/bash**, while Python scripts typically have a **/usr/bin/python2.7** entry, and so on. Therefore, it is important to identify the current program based on both the **comm** and **exe_file** fields—simply using **comm** is insufficient because of **init** scripts, while **exe_file** cannot distinguish between different interpreted programs.

In Microsoft Windows, finding this data poses different challenges. The program name (without a path) is easy to find. It is stored in the **ImageFileName** field of the **EPROCESS** structure. As with Linux, the full path is harder to find. On Windows, the **EPROCESS** structure **SeAuditProcessCreationInfo.ImageFileName->Name** field is a **UNICODE_STRING** containing the path to the process binary. Unlike Linux, recursive structure traversal is not required to read the path from the field. However, Microsoft Windows stores this path as a UTF-16 string. As a result, ASCII file names have alternating null bytes, which means Emmett's usual string copy functions do not work. Instead, we individually copy alternating bytes of the Unicode string into an Emmett variable. This converts non-ASCII

Unicode characters into essentially arbitrary ASCII characters. We believe these will be rare in program paths. Additionally, since the current analysis scripts treat the path as an opaque token, a consistent, lossy conversion is acceptable.

3.1.2 Accessing Fields from Emmett

Even after correct structures and fields are found a challenge remains. Although instrumentation has the structure address, it needs to know the offset of each field of interest within the structure so it can access appropriate memory and read data. Unfortunately, kernels are not designed to make the format of internal data structures easily accessible to third-party software. Emmett has two main features that make this feasible: the **offat*** family of built-in functions and **sparse** structure definitions.

3.1.2.1 Finding Offsets at Runtime

The **offat*** family of functions allows finding these offsets at runtime. Each function scans memory from a specified address, checking for instructions that use offsets in particular ways. Frequently, symbol files are used to find the appropriate address.

Emmett supplies three functions in this family. The first is **offatret**, which finds the ret instruction and returns the offset loaded into **rax** to be returned. By passing the address of a simple **accessor** function such as the Windows **nt!PsGetProcessImageFileName** to **offatret**, we can find the offset of a structure field such as the **ETHREAD ImageFileName**. The second is **offatseg**, which finds the first offset applied to the **FS** or **GS** segment registers. These registers are commonly used for thread-local state, making them helpful for finding thread-specific structures such as the **task_struct** in Linux or the Microsoft Windows **ETHREAD**. With a Microsoft Windows guest, **offatseg(&nt!PsGetCurrentProcess)** finds the offset of the **CurrentThread** pointer within the **KPCR** structure. Finally, **offatstrcpy** searches for calls to a specific function address and returns the offset used to load **RSI** for the call. This could be used to find the offset of a string element in a structure, but is not currently used by any gatherers.

The **offat*** functions offer the advantage of allowing a single Emmett script to be used against kernel binaries with different offset values. As a result, the VProbes distribution includes library code that uses **offat*** to find the current PID and certain other fields, which was used for the Microsoft Windows gatherer. However, **offat*** requires finding an appropriate accessor in which to search for offsets and is dependent on the precise assembly code generated by the compiler. Consequently, another mechanism was desirable for new gatherer code.

3.1.2.2 Encoding Offsets in Scripts

Emmett also supports sparse structure definitions, allowing required offsets to be conveniently encoded within the script. A sparse structure is defined similar to a normal C structure, except that some fields can be omitted. Prefixing a field definition with an @ character and an offset enables Emmett to use the specified offset instead of computing an offset based on the preceding fields in the structure and their size. Given a way to find offsets, this allows only relevant fields to be specified, ignoring those that are not needed.

For the Linux gatherer, a Linux module assembles the necessary offsets. When loaded, it exposes a file in the **/proc/** directory. The file contains offsets of a number of important fields in the kernel, formatted as **#define** statements. The file can be copied to the host and **#included** in a script that uses those offsets directly or to define sparse structures. Currently, users must compile the module, copy it to a machine running the correct Linux version, and load it into the kernel. In the future, we plan to provide a script to extract the relevant offsets directly from debug symbols in the module binary, instead of needing to load the module.

In the Microsoft Windows gatherer, the requisite offsets are extracted directly from the debugging symbols. A script uses the **pdbparse** Python library[5] to convert **ntkrnlmp.pdb**, which Microsoft makes available for use with debuggers, into Emmett structure definitions containing the fields of interest. The output file can be **#included** from the gatherer, and the structures can be traversed just as they would be in kernel code.

While this technique requires updating the script for different kernel versions, we find it more convenient. One advantage is that files with offsets can be generated automatically, and the Emmett preprocessor used to incorporate the current offsets into the rest of the code. Using sparse structures allows the Emmett compiler to perform type checking. The process of writing scripts is less prone to error when the compiler distinguishes between pointer and non-pointer members, or an offset in the **ETHREAD** and **EPROCESS** structures. In addition, code is much clearer when the same names and syntax can be used as is present in the source or debugger output. Therefore, while the Microsoft Windows gatherer uses both **offat*** and sparse structures, the Linux gatherer uses only the latter.

3.2 Analyzer

While our work focused on gatherers, we wrote two simple analyzers to validate the technique was sound. Both analyzers build profiles on a per-program basis. Programs are identified by their **comm** value and binary path (or Microsoft Windows equivalent). Recall that using only the former causes an **init** script and the daemon it runs to share a profile, while using only the latter combines all programs in a given interpreted language into one profile.

Both analyzers are passed logs of a normally executing system, as well as a potentially compromised system. They use this information to build a profile of normal behavior. If the potentially compromised system deviates from the profile, they report a potential attack.

3.2.1 Syscall whitelist

The simplest form of a profile is a simple whitelist of allowed system calls. As the normal log is read, the analyzer notes which system calls are used, such as **open**, **read**, **write**, and so on. When the potentially compromised log is read, the analyzer sees if any new system calls are used. If any are, it reports a possible intrusion.

While extremely simple, this analyzer can detect some attacks. We installed a **proftpd** server that was vulnerable to **CVE-2010-4221** and attacked it using Metasploit Project's exploit[3]. Under normal operation, an FTP server has a very simple system call pattern: it mostly just opens, reads, writes, and closes files. An attack, however,

often uses the **execve** function. Since the daemon does not normally use **execve**, it does not appear in the whitelist and the analyzer immediately triggers.

One advantage of this technique is that it requires little tuning and has few false positives. The profile is simple enough that building a complete "normal" profile is quite manageable. A disadvantage, of course, is that it detects relatively few intrusions. For example, an attack on a web server running CGI scripts is quite hard to detect, since a web server uses a much wider variety of system calls.

3.2.2 stide

A mechanism commonly used in the academic intrusion detection literature is sequence time-delay embedding[8], or *stide*. In this technique, the profile consists of all n-tuples of consecutive system calls. For example, with n=3 and a system call sequence of **open**, **read**, **write**, **read**, **write**, **close** the 3-tuples would be **(open, read, write)**, **(read, write, read)**, **(write, read, write)**, and **(read, write, close)**. To scan a trace for an intrusion, the analyzer checks for tuples that are not found in the training data. If enough such tuples are found in a sliding window an intrusion is likely. The length of the tuples, size of the sliding window, and threshold of how many tuples are required to trigger the system can be tuned to achieve an acceptable level of false positives and negatives.

We had difficulty getting this technique to produce reasonable error rates. One refinement that may help is to build tuples on a per-thread basis, so that multi-threaded programs or programs with multiple instances running at once do not interleave system calls. The runs were performed with only a handful of system calls reported to the analyzer. Using a more complete set might be more successful. Finally, we could try larger sets of training data and further tweak the parameters.

4. Performance

One concern for security software such as this is the performance overhead it entails. Since VProbes primarily causes a slowdown when the probe triggers, we expect performance to depend largely on how often system calls occur. In a workload involving a large number of system calls, we expect performance to suffer. In a largely computational workload, however, we expect performance to be roughly the same as without instrumentation.

To measure the performance overhead of instrumentation, we ran an Apache **httpd** instance in a virtual machine and used **ab**[1] to measure how long it took to make a large number of requests to the server. Several different file sizes were tested, as well as static and dynamic content to get a sense of how different system call patterns could affect performance. For this test, a version of the Linux instrumentation was used that printed approximately a dozen system calls and decoded their arguments. We found that for small static files (a couple hundred or thousand bytes), performance was prohibitive. Larger files were more reasonable: a 13 KB file had roughly 5 percent overhead compared to an uninstrumented server, and 20 KB or larger files had overhead well under 1 percent. We also considered a common web application, Mediawiki, to see how dynamic sites compared. An 11 KB page had approximately 13 percent overhead, while a 62 KB page saw 3 percent overhead.

Our current instrumentation is not optimized. To get a better sense of what portions of the instrumentation are slow, we enabled printing of all system calls and experimented with removing various parts of the instrumentation. With a 10 KB file, we found downloading it 10,000 times took 4.7 seconds on average. With full instrumentation, downloads took approximately 14.4 seconds. Of that 9.7 second increase, it appears that setting the breakpoint accounts for approximately 28 percent and computing the path to the binary is approximately 42 percent. With 70 percent of the runtime cost apparently due to these two components, optimizing either could have a significant impact.

For the breakpoint, VProbes supplies two ways to trigger probes. Users can set a breakpoint at an arbitrary guest address (as we currently do) or trigger when a certain limited set of events occurs, such as taking a page fault, changing CR3, or exiting hardware virtualization. This latter type of probe is significantly faster. If one of these probe points can be used instead of the current breakpoint, it could nearly eliminate the 28 percent of cost of the breakpoint.

For computing the path to the binary, more data is gathered then necessary. Simply retrieving the base name of the binary (**bash, python27**, and so on) is likely to be sufficient in combination with the **comm** value, and should be substantially faster. Alternatively, the binary can be cached and only recomputed when the CR3 or another property changes.

5. Conclusion

VProbes provides a viable replacement for an in-guest agent for a system call-based intrusion detection system. Our system is divided into two components: a gatherer that uses VProbes and an analyzer that determines whether a sequence of system calls is suspicious. The unoptimized performance of the gatherer likely is acceptable, depending on the workload, and several opportunities exist for further optimization.

While we focused on data gathering rather than analysis, we have an end-to-end proof of concept that uses whitelisted system calls to successfully detect certain attacks on a simple FTP server. More elaborate analysis techniques have been amply studied in the literature and could be combined with our instrumentation.

References

1 Apache. ApacheBench, http://httpd.apache.org/docs/2.2/programs/ab.html

2 Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In Proc. Network and Distributed Systems Security Symposium, pages 163–176, 2003.

3 jduck. ProFTPD 1.3.2rc3 - 1.3.3b Telnet IAC Buffer Overflow (Linux), http://www.metasploit.com/modules/exploit/linux/ftp/proftp_telnet_iac

4 Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. IEEE Softw., 14(5):35–42, September 1997.

5 pdbparse, http://code.google.com/p/pdbparse/

6 Raghunathan Srinivasan. Protecting anti-virus software under viral attacks. Master's thesis, Arizona State University, 2007.

7 Kymie M. C. Tan, Kevin S. Killourhy, and Roy A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In RAID, pages 54–73. Springer-Verlag, 2002.

8 Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In IEEE Symposium on Security and Privacy, pages 133–145. IEEE Computer Society, 1999.

Design and Implementation of a Cloud Tenant UI

Louis Weitzman
VMware, Inc.
weitzman@vmware.com

Alister Lewis-Bowen
VMware, Inc.
alister@vmware.com

Elizabeth Li
Carnegie Mellon University
etli@andrew.cmu.edu

Jason Fedor
Brown University
jfedor@cs.brown.edu

Abstract

This paper documents the design and implementation of SilverLining, a student intern project to create a simplified user experience to the cloud. vCloud Director® (vCD) provides a full-featured interface for system administrators and others to configure and control their cloud computing resources. The SilverLining project streamlines user workflows and interactions, making it easier to find virtualized applications and add them into a personal workspace. To support this effort, we created a JavaScript SDK to communicate with vCD through its API.

1. Introduction

The current vCloud Director interface allows a user to work with VMs in a very sophisticated way, enabling the management of storage, networks and compute resources. However, this can be very challenging for end users doing straightforward tasks. SilverLining is an implementation of an interface designed for an end user with minimal requirements who just wants to create and start predefined virtualized applications.As a summer project, we had very specific objectives to create a simple interface in the short period of an internship. To accomplish these goals, we needed to make some basic assumptions that could be relaxed going forward.

Objectives

Typically, a user would search a library of application templates and add them to their workspace. This process of instantiating templates creates working virtual applications (vApps). vCloud Director places these workloads into virtual datacenters (VDCs) where the appropriate resources, for example, CPU, storage and networks, are available to run the vApp. Unfortunately, this process can be complex for the casual user. SilverLining provides the basic functionality to find, instantiate and manage these cloud workloads in a simple and effective manner appropriate for a consumer.

The main objectives of SilverLining are two-fold; 1) demonstrate how easy it could be for a consumer to use the cloud, and 2) provide a JavaScript SDK that helps web developers build customized browser applications that communicate with VMware's cloud implementation using familiar technologies.

The interface should not only be easy to use, but also should demonstrate a responsive design that automatically adapts to different display devices and output formats. This makes it possible to show the same information at different resolutions and in different layouts as dictated by the device on which it is rendered.

The JavaScript SDK enables developers to exercise standard web technologies at a lower cost of entry. This allows them to easily create their own branded interface to the cloud. Hopefully, through this process, we can demonstrate concepts and guidance for the future development of VMware consumer UIs.

Assumptions

We made a few assumptions to simplify the end-user interaction. To scope the amount of work for this summer project and provide simplified workflows, we assumed reasonable defaults for user interactions. These defaults, including which virtual datacenter and networks to use in the organization, are saved as user preferences in local storage on the client-side. This allowed us to create the option of a *1-click* instantiation instead of forcing the user through a more complicated wizard workflow. Also, we added a setting to automatically power on templates when they are added to the workspace. A long-term solution might be to save these user preferences in server storage so that these become available to the user across workstations.

We started the project by assuming one VM per vApp but later relaxed this restriction to allow multiple VMs, adding a bit more flexibility. We wanted to create interactions that make the abstractions of the vApp layer simpler to the end-user. To further simplify the user interface, we hid the internal states of vApps and VMs that could confuse the user.

vCloud Director's User Interface

The current vCloud Director's interface presents three primary areas to end users: Home, My Cloud and Catalogs. Home shows a 'card view' of the vApps available to the user and appropriate vApp actions. My Cloud, also referred to as the workspace, does the same using a 'list view' adding navigation to all VMs within these vApps and logs of tasks performed by the user. The Catalog area provides a way to navigate through libraries of available templates that can be instantiated into the user's workspace. Figure 1 shows the current My Cloud page listing all of the user's vApps.

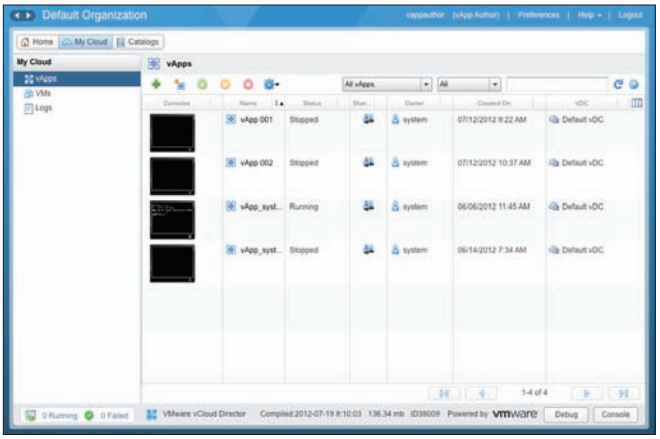

Figure 1. Current vCloud Director showing the My Cloud view, a workspace where users manage vApps and VMs.

2. User Workflows

For our summer project, we focused on a reduced set of workflows that provide the end user with the basic functionality to find, instantiate and manage vApps in their Cloud workspace. These include:

- Authenticate with vCD to gain access to an account within a predefined organization on a VMware Cloud installation.
- Browse a library, choose a template, instantiate it, and power it on. Do this by one-click with a limited number of customized user settings.
- Navigate through the user interface revealing different representations of vApps and Library templates.
- Perform power operations on vApps and VMs.
- Sort/Filter/Search vApps and templates by attributes in a faceted search.
- Show VM console thumbnails and provide console access.
- Use extremely simplified workflows for network configuration and VDC selection.
- Utilize tagging with metadata.
- Handle long running tasks and display notifications to the user.

3. User Experience Design

A few design principles guided us through this project. As we iterated over design solutions, we focused on the end-user workflows. Starting with sketches and wireframe mockups, we produced working examples of our design and refined these implementations, iterating to continually improve our solution.

Design Principles

Some basic design principles informed our decision making throughout the project, including the following:

- **Keep it simple** – Keep it as simple as possible but no simpler.
- **Use direct manipulation** – Objects are represented in the interface and the user should take actions directly on those objects.
- **Use simple hierarchies** – Reduce complexity, but allow future implementations the capability to drill into more detail as needed. For example, use search/filtering to find appropriate templates rather than digging deep into libraries. Also allow implementations on different devices to take advantage of similar navigation.
- **Simplify the workflows** – Limit the use of confirmation dialogs, use undo where appropriate, don't allow the user to get into dead-end or error conditions, and use strong defaults to avoid unnecessary user input.
- **Provide a 'sense of place'** – For both the cloud workspace and library, create a unique look to distinguish the space and the tasks to be done there.

Design Iterations

Initially, our designs were highly influenced by the existing interface. We began with sketches and refined them in low-resolution wireframes.

In our first implementation, we recycled the notion of representing vApps using cards, but added a "card flip" interaction as a means of showing more details. We also planned other interactions such as sorting and drag-and-drop. Each vApp was represented as a distinct object that could be directly manipulated. Any information not needed right away could be accessed in a single click (Figures 2 and 3).

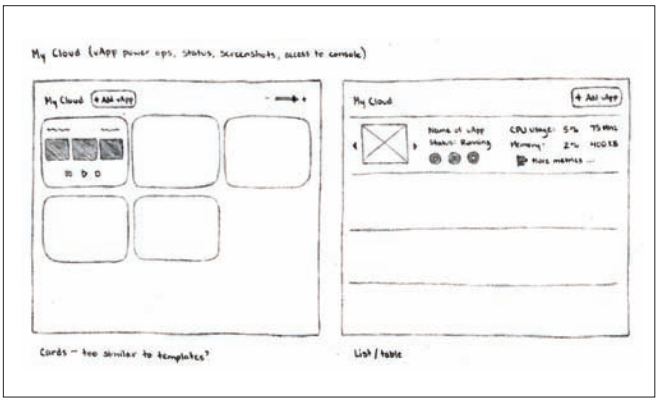

Figure 2. Initial sketches exploring card and list views of vApps and templates similar to what exists in the current UI.

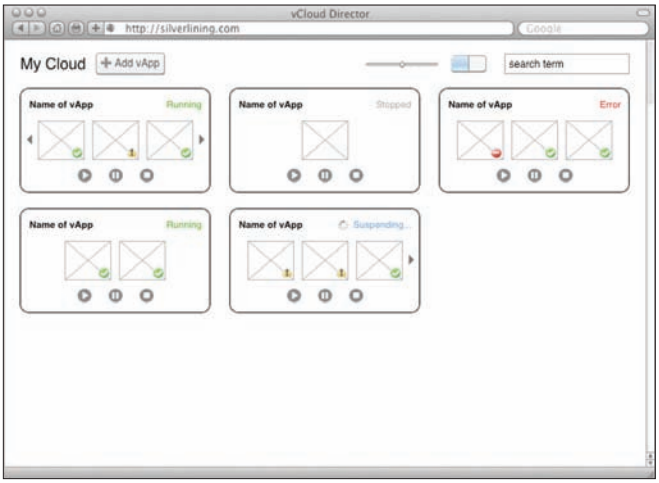

Figure 3. Wireframe mockup of the My Cloud as cards.

We designed templates to be displayed in a simple table that can be filtered using a powerful, faceted search syntax (Figure 4). For example, to find vApps with a name containing the string "windows", the user could type "name:windows" and immediately filter the view. An early implementation of this design allowed the user to specify searchable values for several object attributes. We expanded this capability in the final implementation to include non-string attributes, such as memory size, as well. This search is very quick and effective, and after using it to narrow down options, templates could quickly be scanned down the list for comparison.

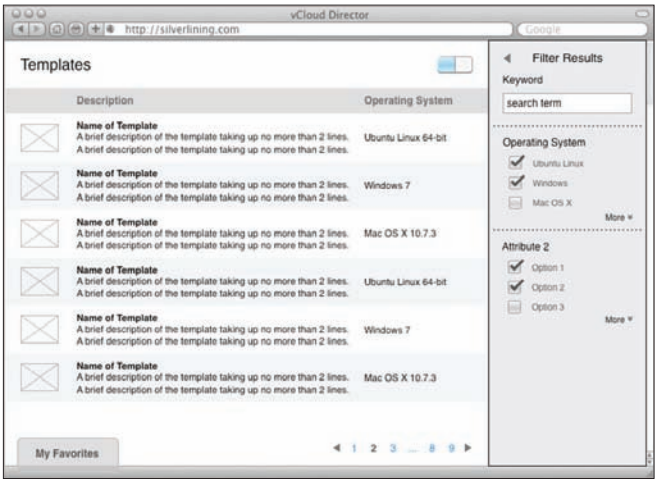

Figure 4. Wireframe mockup of the Library displayed as a list with faceted search.

4. User Feedback

To get feedback on the initial designs, we conducted a small focus group and demonstrated SilverLining to the vCD team in Cambridge. From these sessions we gained two key insights. First, there needed to be better representation and navigation of the object hierarchy. The vApp cards did not suggest an intuitive way to drill into details about vApps and VMs and their associated resources.

Second, choosing library templates needed to be more of a *shopping experience*; template descriptions, ratings, number of instantiations and cost information should be shown in a way that helps users make informed decisions about what to choose.

Also, it became obvious that our original designs did not provide for different layouts while maintaining a common interaction pattern. So we went back to the drawing board to focus on how the UI would behave in various situations (Figures 5 and 6).

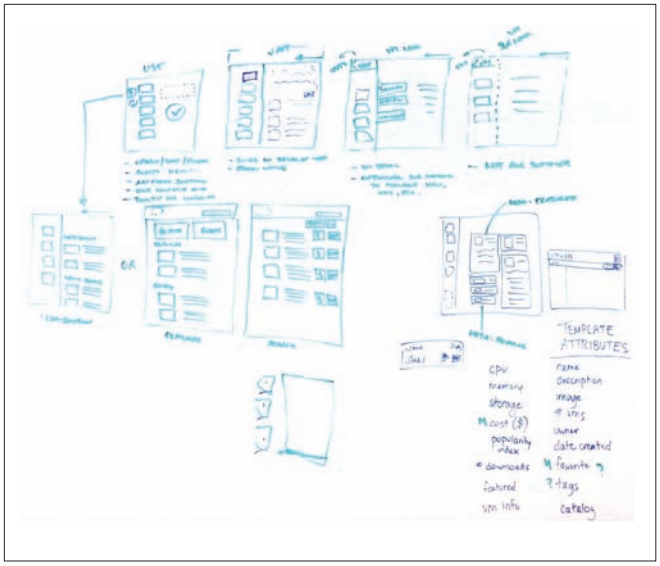

Figure 5. Back to the drawing board (literally) with a focus on the dynamic interaction between outline and detail views.

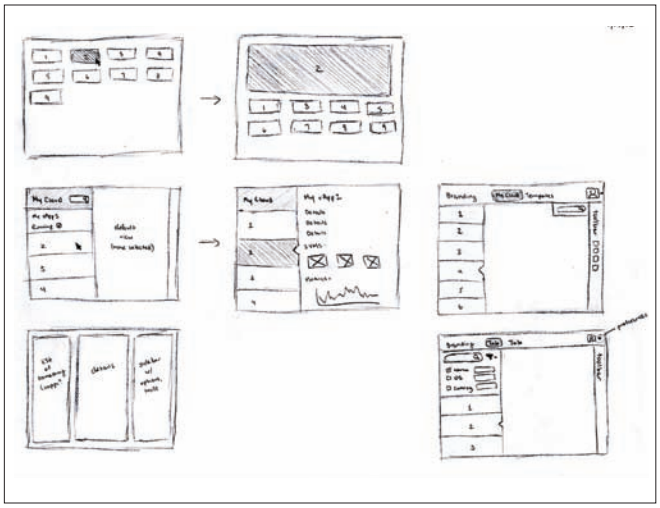

Figure 6. More sketches with expanding cards and multi-view layouts using a left navigation pane.

5. Design Implementation

After exploring many alternatives, we converged on an implementation utilizing sliding panels. This design maintains context, enables drag and drop between the library and the cloud, and supports layouts in other devices while maintaining the application's interaction pattern. The panels are divided into two columns, with a list of vApps on the left and details on the right. The panels slide left and right to navigate linearly through the hierarchy and can be extended to drill deeper through any number of levels (Figures 7 and 8).

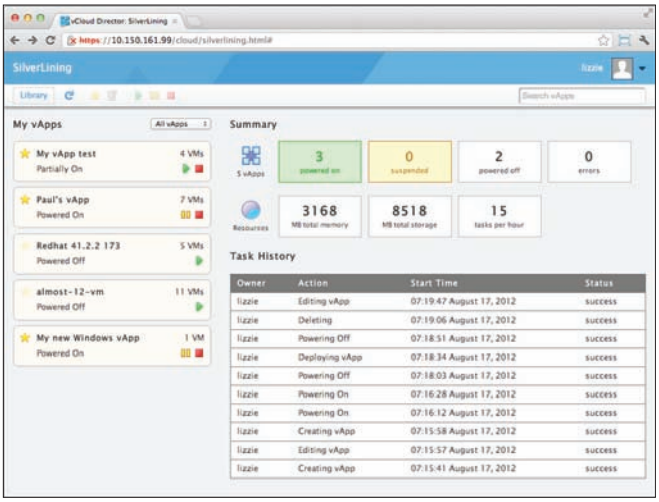

Figure 7. SilverLining's home page showing vApps, general status and recent tasks.

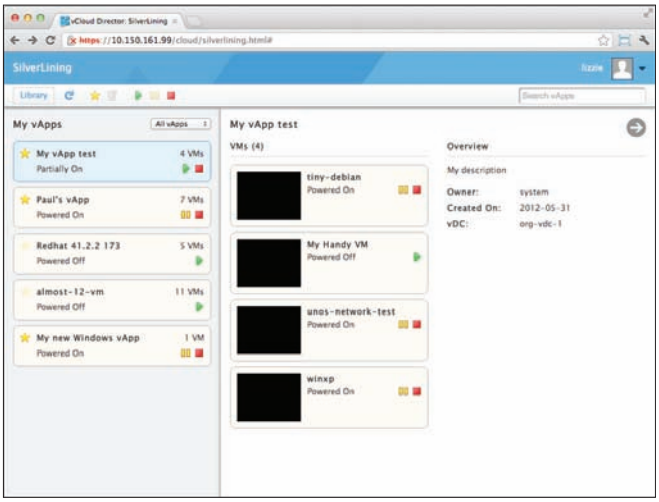

Figure 8. Navigating into the details of a vApp displays all its VMs and other associated metadata.

The template library was designed as a separate, distinctly branded layout for browsing with a shopping catalog feel, showing featured templates and information about popularity (Figure 9). The library slides open next to the user's list of vApps so that the user knows what context they are in and how to go back. These sliding panels provide an affordance that worked well with the 'swipe' gesture on touch devices. Figure 10 shows an interaction in the library filtering the content with a faceted search. Figure 11 illustrates an instantiation workflow allowing the user to customize a few parameters. In comparison, the instantiation workflow in the current vCloud Director presents a multi-step wizard requiring the user to choose many options, some of which they may not understand or even care about.

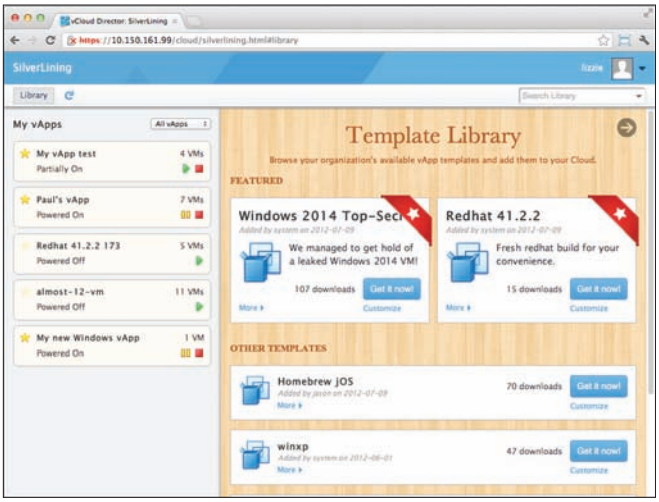

Figure 9. With vApps still visible, the Library displays all templates in a distinctive layout, promoting featured ones.

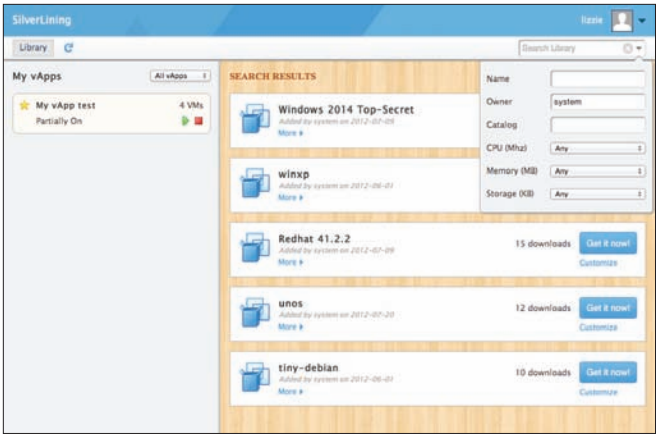

Figure 10. Exploring the Library with faceted search filtering templates to view those owned by "system".

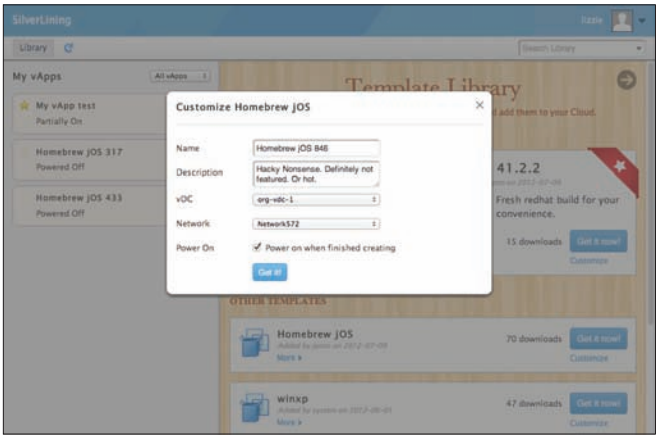

Figure 11. One-click instantiation immediately adds the template to your workspace while the customize option, shown here, provides the ability to override the default values.

Testing our implementation using the panel layouts on the different standard device sizes validated this last design iteration. As the screen real estate was reduced, the layout degraded gracefully but still maintained the same interaction pattern. The layout for a mobile phone can be seen in Figure 12. In the end we had a fully functional application written in HTML5, CSS3 and JavaScript.

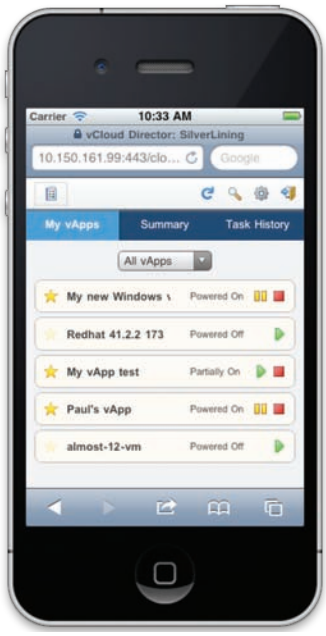

Figure 12. Alternative layout of the same content for smartphones illustrates responsive design organizing information in tabs when limited space is available.

6. Implementation and Technology

Our versatile JavaScript SDK powers the SilverLining project. This library communicates with VMware vCloud Director (vCD) v5.1 through its API. It assumes that a vCD Organization and a user account already exist. This latest version of vCloud Director includes enhanced metadata support and access to server side support for an HTML5 console to VMs. We limited the scope of the SDK development to provide just the functionality that our defined end user would require. This scoping effort helped encapsulate the requirements to produce a demonstrable deliverable for our summer project.

The SDK functions can fetch information on a certain object from vCD, or parse information already stored in the local cache. After authentication, the SDK fetches basic information about the user's workloads—the list of vApps and their corresponding VMs, basic network and VDC info, catalogs of vApp templates, and user information. Then, based on these starting objects, the web developer can pull specific details based on what they want to display. For example, to learn about metadata for a certain vApp, you call a function, 'cloud.metadata.get(vAppObject)', and the SDK will perform the sequence of calls necessary to learn about the metadata for that vApp. Prefetching provides some speed benefit and allows the rendering logic to operate independently from the SDK.

Using the credentials after logging in, the JavaScript SDK has full access to vCloud Director's REST API. It performs a number of AJAX calls in order to navigate the object hierarchy. If the SDK ran on the user's local host or some other website, it could trigger browser security measures designed to prevent cross-site scripting (XSS) when talking to a remote server [9]. Initially, we remedied the situation by running each request through a local proxy server, which then relayed the calls through HTTPS to the vCD instance and back. However, this additional proxy code added to the size of the SDK and limits the ability for this library to be dropped into existing web applications. We decided to install this SDK directly into the vCD cell thus increasing the likelihood that the API's IP address and the resulting web application's IP address are the same, thus thereby eliminating any cross-site scripting limitations.

Metadata

We used the powerful metadata feature in vCD 5.1 to extend the functionality in useful ways. Metadata provides typed, key-value pairs for various vCD objects including vApps, VMs and templates. Once defined, objects can be filtered and sorted to identify important aspects of these objects.

This metadata feature created many new possibilities for our design, several of which we implemented. For example, for the notion of a *shopping experience*, we labeled some templates as *featured* to allow the rendering logic to display those templates more prominently in the Library section. In an actual use-case, we could imagine the vCD Organization Admin role promoting some particular templates for a specific project. We also provide live data about the number of downloads with a counter that is incremented on every instantiation. *Favorites* is another metric stored as metadata for an individual user. Once tagged, the user can then filter for these metadata enhanced vApps or templates. Socializing the popularity of shared objects, in this case templates, provides users with ways to gauge which templates might be best to add to their workspace.

Local Storage

Network latency can occasionally cause the vCD cell to take a very long time to respond to requests. To make our system more responsive, we decided to use a caching system so we could display data immediately at startup. Since the rendering logic is decoupled from the SDKs logic to fetch the data, the UI can display cached information while the live data loads. Then, once the SDK has live data, it can present this to the UI seamlessly. As a result, the user has immediate access to their data as soon as they log in, even though it might be a little old. They can navigate around the hierarchy, even queue up instantiation requests or power something on before any information has even transferred to the client. In effect, this is an implicit *Offline Mode*, creating the illusion of connectivity before it has been established. To store the cache, we are compressing the cloud's data, and storing it as a JSON object in local storage, an HTML5 feature. This approach is appropriate for the type of user we have in mind for this design. But we are aware that storing data this way could become un-scalable when managing one or several vCD organizations or an entire cloud.

Notifications

To support long running operations, the SDK takes advantage of the Task system within the vCloud API. Whenever an action is submitted, a task object is returned. This object can be polled to get the current status until completion. This ensures that the SDK only pulls information as needed. If any task results in an error while the user is logged in, a notification is issued and noted in the task log. Also, the data model maintains the status of all objects, including any errors, thereby allowing the interface to render object status as it sees fit.

Separation of Form and Content

As mentioned earlier, we decoupled fetching data and the presentation of that data when rendering the UI. To do this we used a Model-View-View Model (MVVM) architecture through a popular JavaScript library plugin called Knockout.js [7]. With this separation, we can use the same responsive CSS3 design to provide a polished presentation and adaptive layout depending on the device used.

7. Intern Coordination

Having two summer interns created some challenges as well as opportunities for SilverLining. We needed to plan the project and provide some groundwork before they arrived, and, once onboard, bring them up to speed on the problem and current system. Also, we needed to clearly separate responsibilities and deliverables. Luckily, a separation around JavaScript SDK development, UI design, and implementation was straightforward. The design helped create the requirements for the SDK, and the UI implementation provided a testing harness for SDK development.

Recognizing that communication is a critical part of any successful project, we made it a priority. Throughout the summer we met regularly to discover any roadblocks and to map current progress. Also, we made an effort to communicate our results to other interested parties, including the key product drivers within the organization. The timely feedback received helped move the design and implementation forward. Specifically, we created and maintained a wiki with sections for requirements, schedules, processes, and results for all to view, including video screen-casts showing our progress. Weekly meetings reviewed past work and planned future work. Additional meetings were held to resolve on-going issues and critique the latest design iterations.

8. Conclusion

With two interns providing critical mass, we were able to make significant progress over a very short time. As a result, SilverLining was a great success and met all of our expectations. We were able to demonstrate a UI to vCloud Director that was both simple and effective for the end user we defined. The JavaScript SDK we created now provides a building block for future efforts using standard web technologies. Our responsive design provides a glimpse at how future efforts can adapt to the ever-increasing number of display devices.

We plan to solicit more user feedback beyond our immediate group to help guide our implementation. Once completed, we plan to distribute SilverLining as a Fling [3] to allow developers both inside and outside VMware to exercise the JavaScript SDK and provide us with feedback. The goal is to allow third parties to easily create custom interfaces uniquely branded for their own organizations, and scoped to the needs of specific classes of users wanting to use vCloud Director.

Acknowledgments

We would like to acknowledge and thank those who helped us along the way, including our managers: Brad Meiseles, Vinod Johnson; our colleagues: Stephen Evanchik, Jeff Moroski, Zach Shepherd, Simon Threasher; user research: Amy Grude, Peter Shepherd; our other interns: Paul Furtado, Dan Trujillo and Shivam Tiwari; our product managers: Catherine Fan, Maire Howard, and Dmitri Zimine and our reviewers: Eric Hulteen and Steve Strassmann.

References

1. CSS, http://www.w3.org/TR/CSS/
2. Krug, S. “Don’t make me think, A Common Sense Approach to Web Usability,” Macmillan, 2000.
3. Fling, http://labs.vmware.com/flings
4. HTML5, http://www.w3.org/TR/html5/
5. “HTML5 Leads a Web Revolution,” Anthes, G., Communication of the ACM, Vol 55, No. 7, July 2012.
6. jQuery, http://docs.jquery.com/Main_Page
7. Knockout, http://knockoutjs.com/
8. VMware vCloud® API., http://communities.vmware.com/community/vmtn/developer/forums/vcloudapi
9. XSS, http://en.wikipedia.org/wiki/Cross-site_scripting

FrobOS is turning 10:
What can you learn from a 10 year old?

Stuart Easson
VMware, Inc
stuart@vmware.com

Abstract

The Virtual Machine kernel, and the virtual machine monitor (VMM) in particular, have a difficult but critical task. They need to present a set of virtual CPUs and devices with enough fidelity that the myriad supported guest operating systems (GOS) run flawlessly on the virtual platform. Producing a very-high-quality VMM and kernel requires all low-level CPU and device features to be characterized and tested, both on native hardware and in a virtual machine.

One way to work toward this goal is to write tests using a production operating system, such as Linux. Because an operating system wants to manage CPU and device resources, it ends up hiding or denying direct access to CPU and low-level device interfaces. What is desired is a special-purpose operating system that enables easy manipulation of machine data structures and grants direct access to normally protected features. This article presents the main features of FrobOS, a test kernel and associated tools that directly address this development goal. Written in C, FrobOS provides access to all hardware features, both directly and through sets of targeted APIs. Using a built-in boot loader, a FrobOS test is run by being booted directly on hardware or in a virtual machine provisioned within a hypervisor. In either case, FrobOS scales very well. Tests are simple to run on systems ranging from a uniprocessor 32-bit with 128MB of memory to a 96 thread, 64-bit, 1TB Enterprise server.

Despite FrobOS maturity,[1] information about FrobOS has not been widely disseminated. Today, only a handful of teams actively use the system, with the greatest use occurring within the virtual machine monitor team. This article describes FrobOS for the broader development community to encourage its use.

1. Introduction

The virtualization of a physical computer poses a number of difficult problems, but the Virtual Machine kernel (VMX) and virtual machine monitor (VMM) in particular have both a difficult and functionally critical task: present a set of virtual CPUs with their associated devices as a coherent virtual computer. This pretense must be executed with enough fidelity that the myriad supported (GOS) run flawlessly on the virtual platform. The task is made more difficult by the requirement that the virtual CPU must be able to change some of its behaviors to conform to a different reference CPU, sometimes based on the host system, sometimes defined by the cluster of which it is a part, or based on the users preferences and the GOS expectations. Despite these conflicting requirements it must do its work with little or no overhead.

Great software engineering is an absolute requirement to implement a high-quality virtual machine monitor. The best engineering is ultimately only able to rise to greatness through the rigors of thorough testing. Some testing challenges can be mitigated through focused testing with reference to a high-quality model. Unfortunately, these represent two major difficulties: both the reference model and focused testing are hard to find.

For many purposes, the behaviors one wants in the virtualized CPU are described “in the manual”. Yet there are problems:

- Vendor documentation often contains omissions, ambiguities, and inaccuracies.
- Occasionally CPUs do not do what manuals suggest they should.
- Some well-documented instructions have outputs that are ‘undefined’, a potential headache for a ‘soft’ CPU that is required to match the underlying hardware.
- Different generations of x86 hardware do not match for undefined cases, even those from the same vendor.

1 Perforce spelunking suggests the first check-in was more than 10 years ago.

For the cases where reference documents fail to provide a complete answer, the only recourse is to execute the instructions of interest and study what happens in minute detail. These details can be used to make sure the behavior is modeled correctly in the virtual environment. This brings us to the second difficulty: how do you execute the instructions of interest in an appropriate way? Commodity operating systems (COS) are not helpful in this regard. Despite offering development tools, they use CPU protection mechanisms to stop programs from running many interesting instructions.

This article describes FrobOS, a testing tool designed specifically to address the issues described above. The Monitor team uses FrobOS to build their unit tests, performing checks on the virtual hardware that would be difficult or impossible using a COS.

The FrobOS test development process is hosted on Linux and uses familiar tools (Perl, gcc, gas, make, and so on) provided by the VMware® tool chain. The result of building a FrobOS test is a bootable image that can be started from floppy, PXE, CD, or other disk-like device. FrobOS startup is quite efficient, and is limited essentially by the host BIOS and boot device. As a bootable GOS, FrobOS can be run easily in a virtual machine, and virtual machine-specific customizations to the run-time environment can be made at build or test execution time.

This article presents the use of FrobOS on VMware hosted products such as VMware Workstation™. However, it also runs on VMware® ESX®. See the end of this article for links to more information about this important tool.

2. The Problem

Two types of problems need to be addressed to make a high-quality VMM: the generation of reference models, and testing.

1. **Generating reference models.** What do you expect your virtual CPU and devices to do? When vendor manuals fail to provide an adequate answer, all that is left is to execute code and see what the CPU does under the conditions of interest.
2. **Testing.** In general, testing has a number of issues:
   - • Development efficiency: How hard is it to create a test?
   - • Run-time efficiency: How much overhead does running a test incur?
   - • Environmental accessibility: What can one touch and test before crashing the world?
   - • Determinism: If you do the same steps again, do you get the same results?
   - • Transparency: Can a test result be interpreted easily?

(COS) such as Linux and Microsoft Windows offer tools, documentation, and support for making end-user programs. They seem like a good vehicle for testing. Unfortunately, running a test requires a very different environment than the one used for creating it. When considering an operating system such as Linux or Microsoft Windows for running tests, a number of problems emerge.

- Typical COS work hard to stop programs from accessing anything that would interfere with fairness, stability, or process protection models.
- While it is possible to use an operating system-specific extension to gain supervisor privileges, the environment is fragile and very unpredictable.
- Many desired tests are 'destructive' to their run-time environment, requiring the machine to be rebooted before testing can continue.
- The long boot time for a typical COS makes for very low run-time efficiency.
- Inevitably, the boot of a COS grossly pollutes the lowest level machine state.
- Scheduling in a GOS makes the exact reproduction of machine state difficult or impossible.

Given these problems what is needed is a special-purpose operating system that does not protect or serve. It should be largely be idle unless kicked into action. It needs to boot quickly, and ideally does not do anything outside the test author's notice.

3. The Idea

Our approach is to create an operating system with a kernel specifically designed for efficient generation and completely unfettered execution of low-level CPU and device tests. Ideally, development for this kernel would use familiar tools in a stable run-time environment. While all levels of x86 hardware and devices need to be accessible, users should not be required to know every detail of the hardware to use features in a normal way. Where possible, our operating system provides high-level programmatic access to CPU features, a set of C language APIs that encapsulate the complexities of the x86 architecture and its long list of eccentricities. Done well, it would perhaps mitigate the inherent pitfalls in ad-hoc coding. Properly realized, the benefits of FrobOS are many, including:

- **Ease of use.** FrobOS tests are cross-compiled from Linux, using normal development tools and processes. Running the output in VMware Workstation is automatic, while doing so in VMware ESX requires only a few more command-line parameters. Running natively requires the boot image to be DD'ed onto a floppy disk, or better yet, setting up an image for PXE booting.
- **Efficiency.** The early development of FrobOS was driven by the recognition that unit tests being built by Monitor engineers necessarily shared a lot of code—code that was tricky to write—so why not do it once, and do it right? Over time, this library of routines has grown to include the previously mentioned, as well as PCI device enumeration and access, ACPI via Intel's ACPICA, a quite complete C RTL, an interface to Virtual Performance counters, a complete set of page management routines, and more.

- **Quick results.** Shortest runtime has always been a philosophy for FrobOS. Monitor engineers are an impatient bunch, and the net effect is a very fast boot time. In a virtual machine, a FrobOS test can boot, run, and finish in less than 10 seconds. In general, the boot process is so fast it is limited by the BIOS and the boot device, virtual or native.
- **Flexibility.** By default, the output of a test build is a floppy disk image that can be booted natively (floppy, CD, PXE...), in VMware Workstation (default), or VMware ESX (in a shell or via remote virtual machine invocation). If needed, the build output can be a hard disk image. This is the default for a UEFI boot, and might be needed if the test generates large volumes of core files.
- **Footprint.** A typical FrobOS test run-time footprint is small. The complete test and attendant run-time system fit on a single 1.44MB floppy disk, with room for a core file if the test crashes. FrobOS hardware requirements are small. Currently, it boots in ~128Mb, but a change of compile time constants can reduce it to the size of an L2 cache.
- **Restrictions.** FrobOS imposes very few restrictions. All CPLs and CPU features are trivially available. There is no operating system agenda (fairness, safety, and so on) to interfere with test operation.
- **Testing.** FrobOS addresses the need for low-level CPU and device testing in a way not seen in a COS. Since FrobOS is cross-compiled from Linux, the programming environment for FrobOS has a familiar feel. Engineers have their normal development tools at hand. The FrobOS kernel, libraries, and tests are built with GCC and GAS. While the majority of code is written in C, the initial bootstrap code and interrupt handlers are assembly coded.
- **Scalability.** FrobOS is eminently scalable. It runs on a Pentium III system with less 128Mb of memory, yet offers complete functionality on systems with 96+ CPUs and more than 1TB of RAM. A recent check-in to improve the efficiency of the SMP boot required statistics to be gathered. The results highlighted just how quick this process is—all 80 threads in a server with an Intel® Westmere processor can be booted and shutdown by a FrobOS SMP test in approximately 1 second.
- **Booting.** For characterization purposes, FrobOS tests can be booted directly on hardware from floppy, CD-ROM, and PXE, and there has been success with USB sticks. There are several suites populated with tests that are expected to function usefully in a non-virtual machine (native environment).

4. Welcome to the Machine

What is FrobOS? Depending on the specific interest one has, FrobOS can be seen in a number of different ways: For a test developer, FrobOS is perhaps most accurately described as a GOS construction kit. For someone performing a smoke test of a new build of VMware Workstation, FrobOS is a catalog of unit tests. While for an engineer working on enabling x86 Instruction set extensions in the Monitor FrobOS is an instruction level characterization tool, providing many convenient interfaces to low level details.

Looking around the FrobOS tree within the bora directory, one finds a set of scripts, sources, and libraries. The purpose of the pieces is to build a bootable image designed to execute the test(s) in a very efficient way. The tests are stored in the **frobos/test** directory. Each test is stored in an eponymously named directory and represents a unit test or regression test for a particular bug or area of the monitor.

At present, FrobOS has three teams developing new tests: the Monitor, SVGA, and Device teams have all generated great results with the platform. Most recently, an intern in the Security team made a USB device fuzzer with great results. He filed several bugs as a result of his work, and added basic USB functionality to FrobOS. Later sections of this article present a real example of a device test, specifically demonstrating testing proper disablement of the SVGA device.

FrobOS tests are defined in the **suite.def** file. This file can be found, along with the rest of the FrobOS infrastructure, under the bora directory **vmcore/frobos**. The **doc** subdirectory contains documentation about FrobOS. The test subdirectory contains the tests and **suite.def**. The **runtime/scripts** subdirectory contains scripts, such as **frobos-run**, for running FrobOS.

Most FrobOS operational functions are controlled with an executive script called **frobos-run**. Written in Perl, the script uses the catalog of tests defined in **suite.def** to build each requested test's bootable image(s) and then, by default, starts the execution of the images in VMware Workstation as virtual machines. As each test is built and run, **frobos-run** provides several levels of test-specific parameterization. At build time, a handful of options control the compiler's debug settings (**-debug**), whether to use a flat memory model (**-offset**), and which BIOS to use when booting (**-efi**), and so on.

At run time, **frobos-run** creates a unique configuration file to control VMM-specific parameters, such as the number of virtual CPUs (VCPUs), memory size, mounted disks, and so on. Additionally, other test or VMM-specific options can be applied via the command line. These options are applied to the current run, and can override settings in **suite.def**. FrobOS uses GRUB as its boot loader and supports reading parameters from the GRUB command line, so options can be passed to a test to control specific behaviors. Ordinarily, a FrobOS test runs to some level of completion. In normal cases a test can Pass, Fail, or Skip. The final states of Pass and Fail are easily understood. Skip is slightly unusual—it means either **frobos-run** or the test discovered an environmental issue that would make running the test meaningless. An example might be running a test for an Intel CPU feature on a VIA CPU, or running an SMP test with only one CPU. In the event something happens to terminate the test prematurely, **frobos-run** inspects the logs generated and notices the absence of Pass messages and Fails the test.

So what do you need to make a FrobOS test? The minimal 32-bit FrobOS test can be assembled from the following four items:

1. Two additional lines in **suite.def**, describing how **frobos-run** should find, build, and run the test. Example: **legacymode**
2. A directory in **../frobos/test/** whose name matches the entry in **suite.def**.

3. A file named **frobostest.mk** that describes the source files required to build the test.

```
CFILES = main.c
```

4. The source file (main.c) containing the test code. Example:

```
#define ALLOW_FROBOS32
#include <frobos.h>

TESTID(0, "Journal example test");

void
Frobos_Main(void)

{
EXPECT_TRUE(1 == 1);
}
```

Assuming a VMware Workstation development tree and tool chain are already established, the test is built and invoked using frobos-run as follows:

```
frobos-run -mm bt legacymode:example
```

This invocation produces the following output:

```
Found 1 matching test...

Building tests....
Launching: legacymode:example (BT) (PID 27523), using 1 VCPU
PowerOn

Random_Init: Using random seed: 0x34a37b99379cfef2

TEST: 0000: Journal example test CHANGE: 1709956
PASS: Test 0000: Journal example Test (1 cases)
Frobos: Powering off VM.
      PASS: legacymode:example (BT) (PID 27523) after 5s.

Hostname:       shanghai
Command Line:   legacymode:example -mm bt
Environment:    /vmc/bora:ws:obj
Client:         vmc, synced on 2012/02/07, change number 1709956
Suite spec:     legacymode:example
Monitor modes:  BT
Start time:     Tue Feb  7 12:38:00 2012
End time:       Tue Feb  7 12:38:07 2012
Duration:       0h:00m:07s

Tests run:      1
Passes          1
Skipped Tests:  0
Test Failures:  0
Log file: .../build/frobos/results/shanghai-2012-02-07.5/frobos-runlog
------------------------------
```

While there is a lot of bookkeeping information, the lines starting "PASS:..." show the test booted and ran successfully. The total time of 7 seconds includes starting VMware Workstation, booting FrobOS, and running the test. If the test were run natively, the lines from "**Random_Init:...**" to "**PASS: legacymode:...**" would be identical. Such log lines are copied to **com1** as well.

Code reuse is critical for productivity and reliability. FrobOS makes testing across three major CPU modes relatively trivial. The test example is part of the legacymode suite and runs in 'normal' 32-bit protected mode. It can be made into a 64bit test with the addition of one line (#define ALLOW_FROBOS64), seen here in situ:

```
#define ALLOW_FROBOS32
#define ALLOW_FROBOS64
#include <frobos.h>

TESTID(0, "Journal example test");
...
```

One line in **suite.def** also is needed:

```
...
example:
        legacymode,
        longmode
...
```

The 64-bit (longmode) version of the test is invoked with the following command:

```
frobos-run -mm bt longmode:example
```

While not shown here, the output looks very similar, and as expected the test passes again. To run the test in compatibility mode, use **#define ALLOW_FROBOS48**.

By default, **frobos-run** launches a test three times, once for each of the monitor's major execution modes: Binary Translation (BT), Hardware Execution/Software MMU (HV), and Hardware Execution/Hardware MMU (HWMMU). I used the **-mm bt** switch to override this since I did not want all that output. Note that **-mm** is the short form of the **–monitorMode** option. After your test is ready, **frobos-run** allows all instances for a particular test to be **run** using the all pseudo suite:

```
frobos-run all:example
```

This runs all the entries in **suite.def** for the test named **example** using all three monitor modes. In this case, it runs six tests, the product of the CPU modes and monitor execution modes: (32, 64) x (BT, HV, HWMMU). Because the number of tests can explode quickly, **frobos-run** is SMP-aware. It knows how many CPUs each test needs (from **suite.def**) and determines how many are available on the host. Using the **-j nn** command line option, it schedules multiple tests to run in parallel. As a result, the six tests can be run much more quickly on a 4-way host:

```
frobos-run all:example -j 4
```

This results in the following output:

Found 6 matching tests...

```
Building tests....

Launching: legacymode:example (BT) (PID 16853), using 1 VCPU
Launching: longmode:example (HV) (PID 16855), using 1 VCPU
Launching: legacymode:example (HWMMU) (PID 16856), using 1 VCPU
Launching: legacymode:example (HV) (PID 16858), using 1 VCPU
        PASS: legacymode:example (HWMMU) (PID 16856) after 4s.
Launching: longmode:example (HWMMU) (PID 17009), using 1 VCPU
        PASS: legacymode:example (BT) (PID 16853) after 5s.
Launching: longmode:example (BT) (PID 17048), using 1 VCPU
        PASS: longmode:example (HV) (PID 16855) after 5s.
        PASS: legacymode:example (HV) (PID 16858) after 5s.
        PASS: longmode:example (HWMMU) (PID 17009) after 3s.
        PASS: longmode:example (BT) (PID 17048) after 4s.

Duration:       0h:00m:12s

Tests run:      6
Passes:         6
Skipped Tests:  0
Test Failures:  0
Log file: .../build/frobos/results/shanghai-2012-02-07.9/frobos-runlog

-----------------------------------
```

As you can see, **frobos-run** starts four tests and waits. As each test finishes, **frobos-run** starts another test. The total run time is about twice as long as individually run tests, for a net speed increase of approximately 300 percent. Additional host CPUs allow more tests to execute in parallel. Since some tests require more than one CPU, **frobos-run** keeps track and schedules accordingly. In addition, the **frobos-run** scheduler is quite sophisticated. By default, the scheduler attempts to keep the host fully committed—but not over committed—so tests are scheduled according to CPU and memory requirements.

The following shows how to create a test that would be tricky, perhaps impossible, in an operating system such as Linux or Microsoft Windows. It first writes code to touch an unmapped page, generating a page fault. Once the fault is observed, it checks a few important things that should have occurred or been recorded by the CPU as a result of the page fault exception.

1. A page fault occurs. (This is possible in Linux and Microsoft Windows!)
2. The error code reported for the page fault is correct.
3. The address reported for the page fault is correct.

```
#define ALLOW_FROBOS32
#include <frobos.h>

TESTID(1, "Journal example Test 1");

void
Frobos_Main(void)
{
        // MM_GetPhysPage() returns the address of an unmapped physical
page
        PA physAddr = MM_GetPhysPage();

        // There is no mapping for physAddr, this must #PF...

        EXPECT_ERR_CODE(EXC_PF, PF_RW, *(uint8 *)physAddr = 0);
        EXPECT_INT(CPU_GetCR2(), physAddr, "%x");
}
```

Running the test results in the following:

```
TEST: 0001: Journal example Test 1 CHANGE: 1709956
PASS: Test 0001: Journal example Test 1 (1 cases)
Frobos: Powering off VM.
        PASS: legacymode:example1 (BT) (PID 22715) after 4s.
```

With a few extra lines in the C source, and two additional entries in **suite.def**, we can test for the same conditions in both compatibility and 64-bit modes.

```
/* C source */
#define ALLOW_FROBOS32
#define ALLOW_FROBOS48
#define ALLOW_FROBOS64

/* suite.def */
...
example:
        legacymode,
        compatmode,
        longmode
...
```

The **frobos-run** tool automatically runs the test in the BT, HV, and HWMMU monitor modes.

```
...
        PASS: legacymode:example1 (HWMMU) (PID 23171) after 4s.
        PASS: compatmode:example1 (HWMMU) (PID 23175) after 4s.
        PASS: longmode:example1 (HWMMU) (PID 23192) after 4s.
        PASS: legacymode:example1 (BT) (PID 23165) after 5s.
        PASS: compatmode:example1 (HV) (PID 23167) after 5s.
        PASS: legacymode:example1 (HV) (PID 23172) after 5s.
        PASS: longmode:example1 (HV) (PID 23168) after 5s.
        PASS: longmode:example1 (BT) (PID 23195) after 5s.
        PASS: compatmode:example1 (BT) (PID 23177) after 6s.
...
Duration:       0h:00m:10s
```

That is a lot of testing for 5 (or 3?) real lines of 'new' code.

This test uses a couple of **EXPECT** macros, wrappers for code to check for a certain expected value or behavior, and a lot more

code to catch all sorts of unexpected behaviors. In the event the values sought are not presented, that fact is logged and the test is flagged as a failure. Execution continues unless overridden, since there may be other tests of value yet to run. The **EXPECT** macros can hide a lot of very complex code. If they cannot perform the task needed, the underlying implementation is available as a try/fail macro set for more generality.

5. So just how much do I have to do?

We saw earlier that making test code start in 32-bit, compatibility, or 64-bit mode is easy. To enable cross-mode testing, the FrobOS runtime library is as processor mode agnostic as possible. Calling the **MM_MapPage()** function achieves the same result in 32-bit/PAE or 64-bit modes. The following, slightly more elaborate, code fragment retrieves an unused page, identity maps, zeros it, and announces it did so—and it works as expected in all processor and paging modes.

```
...
PA physAddr = MM_GetPhysPage();
ASSERT(physAddr != NULL);
MM_MapPage(physAddr, physAddr, PTE_P | PTE_RW | PTE_US | PTE_A);
memset(PA_TO_PTR(physAddr), 0, PAGE_SIZE);
Log("zeroed page at %p\n", PA_TO_PTR(physAddr));
...
```

I hesitate to claim that every RTL routine achieves complete register size and mode agnosticism, but it is a very good percentage. If the library-provided types, accessors, and the like are used, tests tend to be easily moved to all modes with only a little extra effort.

6. Let's Get Real

While it is easy for me to say that writing tests in FrobOS is simple, perhaps the point is better amplified with a real example: a FrobOS test written in response to a bug. Let's dive into device testing. The purpose of the test is to make sure the VMware SVGA device is disabled and invisible when turned off in the virtual machine configuration file, and remains off across a suspend and resume operation.

As discussed earlier, there are three pieces:

1. The entry in **suite.def** that includes the configuration option that turns the SVGA device off, specifies which hardware version to use, and so on:

```
...
835729-svga-not-present: # Basic testing when SVGA device is removed
        all  (-passthru "svga.present=FALSE"
                -passthru "virtualhw.version=8"
                -bits 32 -cpus 1),
        svga
        device
```

2. The **frobostest.mk file**, which is the same as shown previously
3. The test source

```
/*****************************************************
 * Copyright 2012 VMware, Inc.  All rights reserved. -- VMware Confidential
 *
*****************************************************/

/*
 * main.c --
 *
 *      Test basic functionality with the svga device removed.
 */

#define ALLOW_FROBOS32
#include "frobos.h"
#include "vm_device_version.h"

TESTID(835729, "SVGA-not-present");
SKIP_DECL(SK_NATIVE);

/*
 *-----------------------------------------------------
 *
 * Frobos_Main --
 *
 *      This is the main entry point for the test.
 *
 * Results:
 *      None
 *
 *-----------------------------------------------------
 */
void
Frobos_Main(void)
{
        PCI_Device *pci;

        PCI_Init();

        Test_SetCase("Verify SVGA device not present");

        pci = PCI_Search(PCI_VENDOR_ID_VMWARE,
PCI_DEVICE_ID_VMWARE_SVGA2, NULL);
        EXPECT_TRUE(pci == NULL);

        pci = PCI_Search(PCI_VENDOR_ID_VMWARE, PCI_DEVICE_ID_
VMWARE_SVGA, NULL);
  EXPECT_TRUE(pci == NULL);

        pci = PCI_Search(PCI_VENDOR_ID_VMWARE, PCI_DEVICE_ID_
VMWARE_VGA, NULL);
        EXPECT_TRUE(pci == NULL);

        if (Test_IsDevelMonitor()) {
                Test_SetCase("Suspend resume with no SVGA device");
        Mon_SuspendResume();
        }
}
```

A couple of new features are used here. The (optional) **SKIP_DECL** makes the test refuse to run (SKIP) if it is booted directly on hardware. The PCI library is used to gain access to the device. The source calls **PCI_Init()** and searches for the various device IDs that have been used by our SVGA device, and tests to make sure it is not found. The special backdoor call to force a suspend/resume

sequence only is supported on a developer build, so we protect the call appropriately. This test reproduces the bug on an unfixed tree, and runs in less than 10 seconds.

The author of the test tells me this test took him 30 minutes to write, including a coffee break. A similar test, if possible at all, would take much longer in any other operating system. Plus, programmers can convert this test to run in compatibility mode and 64-bit mode with the addition of two lines of source and two lines in **suite.def**. These additional lines are seen at the start of the next and prior examples.

7. What about SMP?

By convention, the x86 architecture distinguishes the first CPU to boot from those CPUs that boot later. The first is called the Boot Strap Processor (BSP), and those that follow are called auxiliary processors (AP). Assuming you want to test CPU features in an SMP environment, FrobOS offers APIs to bring APs into the action. Additional CPUs configured in a virtual machine, or those present natively, are booted and then 'parked' looping on a shared variable.*

So what do we need to make an SMP FrobOS test? Following in our minimalist vein, here are the source and differences from the simplest possible test:

• The suite.def file adds an option to enable more CPUs:

```
...
smpexample:
        smp (-cpus 0)      # use all available CPUs
...
```

• The source file main.c is as follows:

```
#define ALLOW_FROBOS32
#define ALLOW_FROBOS48
#define ALLOW_FROBOS64
#include <frobos.h>

SKIP_DECL(SK_SMP_2)

TESTID(2, "Journal SMP example test");

static void
SayHello(void * unused)
{
        Log("Hello from CPU %u\n", SMP_MyCPUNum());
}

void
Frobos_Main(void)
{
        int i;

        SayHello(NULL);
        for (i = 0; i < SMPNumCPUs(); i++) {
                SMP_RemoteCPUExecute(i, SayHello, NULL);
        }

        SMP_WaitAllIdle();
}
```

I made the test able to run in all three basic modes, and introduced a couple of new features. This particular **SKIP_DECL(SK_SMP_2)** makes the test skip when there is only one CPU present. In addition, I used **SMP_RemoteCPUExecute()** to kick each AP into action, while **SMP_WaitAllIdle()**, as its name suggests, waits for all APs to become idle.

The test simply iterates through all available CPUs, asking each CPU to execute the **SayHello()** function. Each AP polls for work and executes the function as soon as it can, resulting in the following output:

```
...
TEST: 0002: Journal SMP example Test CHANGE: 1709956
Hello from CPU 0
Hello from CPU 6
Hello from CPU 5
Hello from CPU 4
Hello from CPU 2
Hello from CPU 3
Hello from CPU 7
Hello from CPU 1
PASS: Test 0002: Journal SMP example Test
Frobos: Powering off VM.
...
```

This was performed on an 8-way virtual machine. As one might hope, the code functions without modification in environments with any number of CPUs. Plus, the same code runs in 32-bit, compatibility, and 64-bit modes.

* Using a shared variable might seem like a strange choice, after all there are other mechanisms such as Inter-Processor Interrupts (IPI) that are specifically designed for one CPU to send messages to another CPU. Using an IPI requires both the sender and receiver to have their APIC enabled, and the receiver must be ready and able to receive interrupts. While this is certainly a valid set of test conditions, it is not reasonable to impose them as a limitation on all SMP tests.

The SMP environment provided is not overly complex. Once booted into a GCC compatible runtime environment, APs do nothing unless asked. Data is either SHARED or PRIVATE (default). The run-time library provides basic locks, simple CPU coordination routines, and reusable barriers. For almost all runtime functions, the AP can do whatever the BSP can do.

8. There's More

When **frobos-run** runs a test in a virtual machine, it has the means to set a number of important options in the configuration file, including the Virtual Hardware version. This allows (or denies, depending on your perspective) guest visibility for certain hardware features, including instruction set extensions, maximum memory, maximum number of CPUs, and so on. If needed, a particular FrobOS test can inspect a variable to determine the specific hardware value and thereby tailor error checking or feature analysis as needed.

9. Conclusion

FrobOS is a great tool for writing low-level x86 and hardware tests on the PC platform, both for virtual machines and native operation. For test authors, it offers a rich run-time environment with none of the blinkers and constraints so typical of COS running on this platform. For product testing, FrobOS offers an ever-growing catalog of directed tests, with extensive logging and failure reporting capabilities. For most test running purposes, **frobos-run** hides the differences between VMware ESX and VMware Workstation, allowing the test author to check the behavior of either environment.

By its nature, FrobOS is a great characterization tool. The simple run-time model and very low requirements mean almost any x86-compatible machine that can boot from a floppy disk, PXE, USB disk or CD-ROM can be tested. It is a particularly easy fit for developers running Linux that wish to make or run tests, FrobOS uses the normal tools (make, gcc, gas, and so on) to build the kernel and a large catalog of tests. It has a very capable executive script that hides most of the complexities of building and running the tests: individually, as specific collections, or as a whole.

We are extending the use of FrobOS in several areas, including:

- Building fuzzers, with three already built (CPU, USB device, and VGA FIFO)
- Dynamic assemblers, because sometimes you just cannot make up your mind
- RTPG, a CPU fuzzer
- UEFI, a new BIOS
- Coverage, examining who executes what and why

Now, it is your turn. In addition to the documentation and sources in the **bora/vmcore/frobos** directory, I encourage you to explore the following resources:

- https://wiki.eng.vmware.com/Frobos
- The output of the **frobos-run** –help command
- Existing test and **suite.def** examples

Join my team in helping better test and understand our hypervisor.

Acknowledgements:

Thanks to present and past authors of the tests, libraries, and tools that have become what we call FrobOS. The list is long, but includes the past and present Monitor team, FrobOS maintainers, and Monitor reliability: Rakesh Agarwal, Mark Alexander, Kelvin Fong, Paul Leisy, Ankur Pai, Vu Tran, and Ying Yu.

Many thanks to Alex Garthwaite and Rakesh Argarwal for their guidance in writing this article, the reviewers without whom there would be many more mistakes of all sorts, and Mark Sheldon for the real SVGA enablement test.

Storage DRS: Automated Management of Storage Devices In a Virtualized Datacenter

Sachin Manpathak
VMware, Inc.
smanpathak@vmware.com

Ajay Gulati
VMware, Inc.
agulati@vmware.com

Mustafa Uysal
VMware, Inc.
muysal@vmware.com

Abstract

Virtualized datacenters contain a wide variety of storage devices with different performance characteristics and feature sets. In addition, a single storage device is shared among different virtual machines (*VMs*) due to ease of VM mobility, better consolidation, higher utilization and to support other features such as VMware Fault Tolerance (*FT*) [19] and VMware High Availability (*HA*) [20], that rely on shared storage. According to some estimates, the cost of managing storage over its lifetime is much more expensive as compared to initial procurement costs. It is highly desirable to automate the provisioning and runtime management operations for storage devices in such environments.

In this paper, we present Storage DRS as our solution for doing automated storage management in a virtualized datacenter. Specifically, Storage DRS handles initial placement of virtual disks, runtime migration of disks among storage devices to balance space utilization and IO load in a unified manner, and respect business constraints while doing so. We also present how Storage DRS handles various advanced features from storage arrays and different virtual disk types. Many of these advanced features make the management more difficult by hiding details across different mapping layers in the storage stack. Finally, we present various best practices to use Storage DRS and some lessons learned from initial customer deployments and feedback.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques.
C.4 [**Performance of Systems**]: Measurement techniques.
C.4 [**Performance of Systems**]: Performance Attributes.
D.4.8 [**Operating Systems**]: Performance—*Modeling and Prediction*
D.4.8 [**Operating Systems**]: Performance—*Measurements*
D.4.8 [**Operating Systems**]: Performance—*Operational analysis*

General Terms

Algorithms, Management, Performance, Design, Experimentation.

Keywords

VM, Virtualization, Resource Management, Scheduling, Storage, Hosts, Load Balancing

1. Introduction

Virtualized infrastructures provide higher utilization of physical infrastructure (servers and storage), agile IT operations, thereby reducing both capital and operating expenses. Virtualization offers unprecedented control and extensibility over consumption of compute and storage resources, allowing both VMs and their associated virtual disks to be placed dynamically based on current load and migrated seamlessly around the physical infrastructure, when needed. Unfortunately, all this sharing and consolidation comes at the cost of extra complexity. A diverse set of workloads that are typically deployed on isolated physical silos of infrastructure now share a bunch of heterogeneous storage devices with a variety of capabilities and advanced features. Such environments are also dynamic—as new devices, hardware upgrades, and other configuration changes are rolled out to expand capacity or to replace aging infrastructure.

In large datacenters, the cost of storage management captures the lion's share of the overall management overhead. Studies indicate that over its lifetime, managing storage is four times more expensive than its initial procurement [9]. The annualized total cost of storage for virtualized systems is often three times more than server hardware and seven times more than networking-related assets [13]. Due to inherent complexity and stateful nature of storage devices, storage administrators make most provisioning and deployment decisions in an ad-hoc manner trying to balance the space utilization and IO performance. Administrators typically rely on rules of thumb, or risky and time-consuming trial-and-error placements to perform workload admission, resource balancing, and congestion management.

There are several desirable features that a storage management solution needs to provide, in order to help administrators with the automated management of storage devices:

- Initial placement: Find the right place for a new workload being provisioned while ensuring that all resources (storage space and IO) are utilized efficiently and in a balanced manner.
- Load balancing: monitor the storage devices continuously to detect if free space is getting low on a device or if IO load is imbalanced across devices. In such cases, the solution should take remediation actions or recommend resolution to the administrators.

- Constraint handling: Handle a myriad of hardware configuration details and enforce business constraints defined by the administrator, such as anti-affinity rules for high availability.
- Online data gathering: All of the above needs to be done in an online manner by collecting runtime stats and without requiring offline modeling of devices or workloads. The solution should automatically estimate available performance in a dynamic environment.

Toward automated storage management, *VMware introduced Storage Distributed Resource Scheduler (Storage DRS)*, the first practical, automated storage management solution that provides all of the functionality mentioned above. Storage DRS consists of three key components: 1) a continuously updated storage performance and usage model to estimate performance and capacity usage growth; 2) a decision engine that uses these models to place and migrate storage workloads; and 3) a congestion management system that automatically detects overload conditions and reacts by throttling storage workloads.

In this paper, we describe the design and implementation of Storage DRS as part of a commercial product in VMware's vSphere management software [17]. We start with some background on various storage concepts and common storage architectures that are used in VMware based deployments in Section 2. Then we present a deep-dive in to the algorithms for initial placement and load balancing that are used for generating storage recommendations while balancing multiple objectives such as space and IO utilization (Section 3). We provide various use case scenarios and how Storage DRS would handle them along with the description of these features.

In practice, arrays and virtual disks come with a wide variety of options that lead to different behavior in terms of their space and IO consumption. Handling of advanced array or virtual disk features and various business constraints is explained in Sections 4 and 5. Given any solution of such complexity as Storage DRS there are always some caveats and recommended ways to use them. We highlight some of the best practices in deploying storage devices for Storage DRS and some of the key configuration settings in Section 6.

Finally, we present several lessons learned from real world deployments and outline future work in order to evolve Storage DRS to handle the next generation of storage devices and workloads (Section 7).

We envision Storage DRS as the beginning of the journey to provide software-defined storage, where one management layer is able to handle a diverse set of underlying storage devices with different capabilities and match workloads to the desired devices, while doing runtime remediation.

2. Background

Storage arrays form the backbone for any enterprise storage infrastructure. These arrays are connected to servers using either fiber channel based SAN or Ethernet based LANs.

These arrays provide a set of data management features in addition to providing a regular device interface to do IOs.

In virtualized environments, shared access to the storage devices is desirable because a bunch of features such as live migration of VMs (*vMotion* [8]), VMware HA and VMware FT depend on the shared storage to function. Shared storage also helps in arbitrating locks and granting access of files to one of the servers only, when several servers may be contending. Some features such as *Storage IO Control (SIOC)* also use shared storage to communicate information between servers in order to control IO requests from different hosts [16].

2.1 Storage Devices

Two main interfaces are used to connect storage devices are with VMware ESX hypervisor: block-based interface and file-based interface. In the first case, the storage array exposes a storage device (also called as LUN) as a set of blocks on which one can do regular IO operations using SCSI commands. ESX hypervisor installs a clustered file system called VMFS [14] on that LUN. VMFS allows every ESX host to see the same file system and all changes, in a consistent manner. In the second case, a storage device is exported by as a mount point by NFS server. ESX hypervisor accesses the device using NFS protocol. Currently ESX supports and implements NFSv3 protocol.

In both cases, ESX creates a concept of a datastore and exposes that to the administrator as a management and provisioning entity. Typically, a datastore is backed by a single LUN or NFS mount point, but we also allow a VMFS file system to extend across two devices. Such configuration is not supported by some of the features and is uncommon in customer deployments. We use the term datastore to denote a LUN or NFS mount point in this paper.

At the storage array, the storage controllers pool a set of underlying physical devices to create a volume or a RAID-group on them. Different vendors use different terms but the overall concept of creating a pool of underlying physical resources is pervasive. This pool is governed typically by similar properties in terms of reliability, fault handling, and performance sharing across the underlying devices. On top of this volume or a RAID-group, one can create a LUN, which is striped across the underlying devices. Finally these LUNs are exposed as a block device or a mount point over NFS.

This virtualization of underlying devices is hidden from the hypervisor and the exact performance and device level characteristics are not known outside the array. Since there are multiple layers of mappings across the storage stack from virtual disks to all the way down to physical disks, it is often quite difficult to discern where exactly a given block is stored.

In order to manage these devices, a solution such as Storage DRS needs to infer the performance characteristics in an online manner.

Storage controllers leverage the mapping of LUN address space to physical disk pool in many different ways to provide space savings and performance enhancing functionality. One of the widely used features is thin provisioning, where a LUN with a fixed reported capacity is backed by a much smaller address space from among the physical drives. For example, a 2 TB datastore can be backed up by a total of 500 GB physical drive pool. Storage controllers only map the used (i.e., written) blocks in the datastore to the allocated space in the backing pool.

The capacity of the backing pool can be adjusted dynamically by adding more drives when the space demand on the datastore increases over time. The available free capacity in the backing pool is managed at the controller, and controllers provide dynamic events to the storage management fabric when certain threshold in the usage of the backing pool is reached. These thresholds can be set individually at storage controllers. The dynamic block allocation in thin provisioned datastores allow storage controllers to manage its physical drive pool in a much more flexible manner by shifting the available free capacity to where it is needed.

In addition, storage controllers also offer features like compression and de-duplication to compress and store the identical blocks only once. Common content, such as OS images, copies of read-only databases, and data that does not change frequently can be compressed transparently to generate space savings. A de-duplicated datastore can hold more logical data than its reported capacity due to the transparent removal of identical blocks.

Storage controllers also offer integration with hypervisors to offload common hypervisor functions for higher performance. VM cloning, for example, can be performed efficiently using copy-on-write techniques by storage controllers: a VM clone can continue to use the blocks of the base disk until it writes new data, and new blocks are dynamically allocated during write operations. Storage controller keeps track of references to each individual block so that the common blocks are kept around until the last VM using them is removed from the system. Clones created natively at the storage controllers are more efficiently stored as long as they remain under the same controller. Since the independent storage arrays can't share reference counts, all the data of a clone needs to be copied in case a native clone is moved from one controller to another.

Storage controllers also take advantage of the mapping from logical to physical blocks to offer dynamic performance optimizations by managing the usage of fast storage devices.

Nonvolatile storage technologies such as solid-state disks (SSDs), flash memory, and battery backed RAM can be used to transparently absorb a large fraction of the IO load to reduce latency and increase throughput. Controllers remap blocks dynamically across multiple performance tiers for this purpose. This form of persistent caching or tiering allows fast devices to be used efficiently depending on the changing workload characteristics.

2.2 Virtual Disks

The datastores available to the hypervisor are used to store virtual disks belonging to a VM and other configuration files (e.g., snapshots, log files, swap files, etc.). Hypervisor controls the access to physical storage by mapping IO commands issued by VMs to file read and write IOs on the underlying datastores. This extra mapping layer allows hypervisors to provide different kinds of virtual disks for increased flexibility.

In the simplest form, a virtual disk is represented as a file in the underlying datastore and all of its blocks are allocated. This is called a thick-provisioned disk. Since the hypervisor controls the block mapping, not all blocks in a virtual disk have to be allocated at once. For example, VMs can start using the virtual disks while the blocks are being allocated and initialized in the background. These are called lazy-zeroed disks and allow a large virtual disk to be immediately usable without waiting for all of its blocks to be allocated at the datastore.

VMFS also implements space saving features that enable virtual disk data to be stored only when there is actual data written to the disk. This is similar to thin provisioning of LUNs at the array level. Since no physical space is allocated before a VM writes to its virtual disk, the space usage with thin-provisioned disks starts small and increases over time as the new data is written to the virtual disk. One can overcommit the datastore space by provisioning substantially larger number of virtual disks on a datastore, as there is no need to allocate unwritten portions of each virtual disk. This is also known as space over commitment. Figure 1 depicts the virtual disk types. ESX also provides support for VM cloning where clones are created without duplicating the entire virtual disk of the base VM. Instead of a full replica, a delta disk is used to store the modified blocks by the clone VM. Any block that is not stored in the delta disk is retrieved from the base disk. These VMs are called linked clones as they share base disks for the data that is not modified in the clone. This is commonly used to create a single base OS disk storing the OS image and share that across many VMs. Each VM only keeps the unique data blocks as delta disk that contains instance specific customizations.

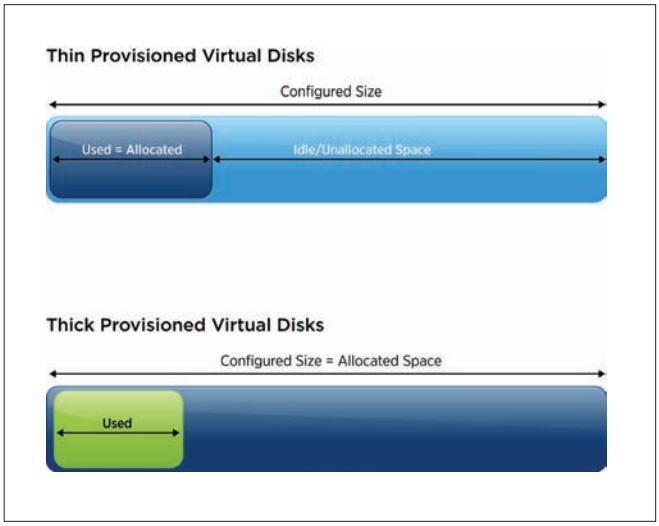

Figure 1. Virtual Disk Types

Finally, we use the key primitive of live storage migration (also called *Storage vMotion* [8]) provided by ESX that allows an administrator to migrate a virtual disk from one datastore to another without any downtime of the VM. We rely heavily on this primitive to do runtime IO and space management.

2.3 Storage Management Challenges

Despite the numerous benefits of virtualization in terms of extensible and dynamic allocation of storage resources, the complexity and large number of virtual disks and datastores calls for an automated solution. It is hard for an administrator to keep track of the mappings and sharing at various levels, in order to make simple decisions such as finding the best datastore for an incoming VM. The design may need to consider several metrics as explained below:

- **Space requirements**: One might think that using a datastore with the most available space is the best option. As we have described, determining the datastore with the most available space is harder in the presence of thin provisioning, de-duplication, linked clones, and unequal data growth from thin provisioned virtual disks. For example, it is often more space efficient to utilize an existing base disk for a linked clone than to create a full clone on another datastore - the former will use a fraction of the space as compared to the full clone.
- **Performance requirements**: Using a datastore with the most available performance headroom (IOPS or latency) seems to be a good option, but it is hard to determine the available headroom in the presence of rampant sharing of underlying physical resources behind many layers of mapping. Estimating the performance of a storage system and dynamically detecting the sharing among multiple datastores are hard problems that need to be solved to efficiently manage performance.
- **Multiple-dimensions**: It is quite possible that the best datastore for available space is not the same as the best datastore for available performance. Determining the relative goodness of a placement when there are multiple optimization criteria is needed in any solution.
- **Constraints**: Administrators can also provide constraints such as anti-affinity rules (e.g., keeping multiple VMs on different datastores), datastore compatibility (e.g., using a native datastore feature), connectivity constraints, or datastore preference (e.g., requiring a certain RAID level) that need to be taken into account for placement. In these cases, a provisioning operation might have to move other virtual disks to be successful.

Beyond just the initial placement, storage resources must be continuously monitored to detect changes in capacity consumption and IO loads. It is common practice to relocate workloads in the server infrastructure using vMotion or Storage vMotion to carry out various remediation actions.

This level of flexibility requires that the management layer be sufficiently intelligent to automatically detect resource imbalance, formulate actions that will fix the issue before it can develop into a service disruption, and carry out these actions in automated fashion.

3. Solution: Storage DRS

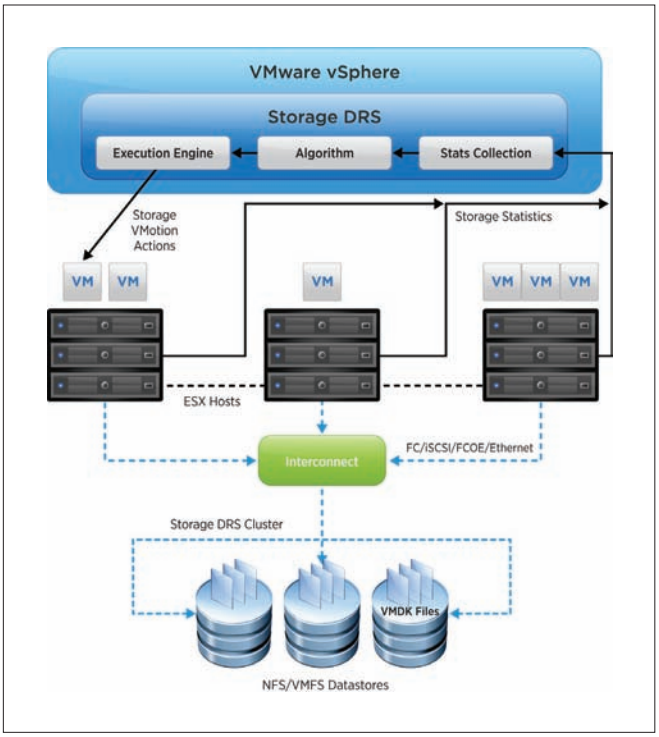

Figure 2. Storage DRS as Part of VMware vSphere

Figure 2 describes vSphere architecture with Storage DRS. As illustrated in the figure, Storage DRS runs as part of vCenter Server Management Software [18]. Storage DRS provides a new abstraction to the administrator, which is called datastore cluster. This allows administrators to put together a set of datastores into a single cluster and use that as the unit for several management operations. Within each datastore cluster, VM storage is managed in the form of Virtual Disks (*VMDKs*). The configuration files associated with a VM are also pooled together as a virtual disk object with space requirements only. When a datastore cluster is created, the user can specify three key configuration parameters:

- **Space Threshold**: This threshold is specified in percentage and Storage DRS tries to keep the space utilization below this value for all datastores in a datastore cluster. By default this is set to 80%.
- **Space-Difference Threshold**: This is an advanced setting that is used when all datastores have space utilization higher than the space threshold. In that case a migration is recommended from a source to a destination datastore only if the difference in space utilization is higher than this space-difference value. By default, this is set to 5%.
- **IO Latency threshold**: A datastore is considered overloaded in terms of IO load only if its latency is higher than this threshold value. If all datastores have latency smaller than this value, no IO load balancing moves are recommended. This is set to 10 ms by default. We compute 90th percentile stats in terms of datastore IO latency to compare with this threshold.

- **Automation level**: Storage DRS can be configured to operate in fully automated or manual modes. In fully automated mode, it will not only make recommendations but execute them without any administrator intervention. In manual mode, administrator intervention is needed to approve the recommendations.

All these parameters are configurable at any time. Based on these parameters, the following key operations are supported on the datastore cluster:

1. Initial placement API for VMDKs of a VM can be called on a datastore cluster instead of a specific datastore. Storage DRS implements that API and provides a ranked list of possible candidate datastores based on space and IO stats.
2. Out-of-space situations are avoided in the datastore cluster by monitoring the space usage and comparing that to the user-set threshold. If there is any risk that a datastore may cross the space utilization threshold, Storage DRS recommends Storage vMotion to handle that case.
3. Monitor and balance IO load across datastore, if the 90th percentile latency is higher than the user-set threshold for any datastore, Storage DRS tries to migrate some VMDKs out of that datastore to lightly loaded ones.
4. Users can put a datastore in the maintenance mode. Storage DRS tries to evacuate all VMs off of the datastore to suitable destinations within the cluster. The user can then perform maintenance operations on the datastore without impacting any running VMs.

All resource management operations performed by Storage DRS respect user set constraints (or rules). Storage DRS allows specification of anti-affinity and affinity rules between VMDKs in order to fulfill business, performance and reliability considerations.

As depicted in Figure 2, Storage DRS architecture consists of three major components - (1) Stats Collection: Storage DRS collects space and IO statistics for datastores as well as VMs. VM level stats are collected from the corresponding ESX host running the VM and datastore level stats are collected from SIOC, which is a feature that runs on each ESX host and computes the aggregated datastore level stats across all hosts. For example, SIOC can provide per datastore IOPS and latency measures across all hosts accessing the datastore. (2) Algorithm: This core component does computation of resource entitlements and recommends feasible moves. Multiple choices could be possible, and recommendations carry a rating indicating their relative importance/benefit. (3) Execution Engine: This component monitors and executes the Storage vMotion recommendations generated by the algorithm.

In this paper, we focus on handling of space, user constraints and virtual disk features such as linked clones. To handle IO, Storage DRS performs online performance modeling of storage devices. In addition, it continuously monitors IO stats of datastores as well as VMDKs. Details of IO modeling and corresponding algorithms are described in earlier papers [4][5]. We do not discuss those topics here.

In the remaining section, we first describe automated initial placement of VMs in a Storage DRS cluster. Then we go into details of load balancing, which form the core of storage resource management followed by the discussion on constraint handling.

3.1 Initial Placement

Initial placement of VMs on datastores is one of the most commonly used features of Storage DRS. Manually selecting appropriate storage for a VM often leads to problems such as poor IO performance and out of space scenarios. Storage DRS automatically selects most fitting datastore to place a VM based on space growth modeling and IO load history of the datastores. During initial placement, Storage DRS does not have any history of VMs IO profile or its future space demand. So, it uses conservative estimates for space and IO: space is assumed to be full disk size for thick provisioned disks and a fixed small size for thin provisioned ones. In terms of IO load we use the average load from other existing VMDKs on the datastore.

In addition, Storage DRS gives preference to datastores connected to as many hosts as possible. This allows solutions like Distributed Resource Scheduler (DRS) [1] to do load balancing of VMs across hosts more effectively. Storage DRS supports all virtual disk formats for placement—thin, thick eager zeroed as well as thick lazy zeroed. Thin disks in particular, start off with a small size; but can consume all the space up to their configured size over time. This poses a risk of running out of space on a datastore. Such placements are done with future space growth considerations.

Many aspects of Storage DRS such as constraints, prerequisite moves, pending recommendations are common to initial placement and load balancing. We discuss them in detail when we describe load balancing. Figure 3 below describes the initial placement used by Storage DRS.

Data: A set of datastores D, on which VM *v* can be placed.
Result: A set of recommendations R
For *datastore d ∈ D*; **Do**
  If *d has capacity for v and affinity rules allow the placement*, then
    Prop = [place v on d];
    Goodness(prop) = Imbalance(before) – Imbalance(after) placement;
    Make sure prop does not conflict with pending recommendations
  Add Prop to R
Sort R in descending order of goodness

Figure 3. Initial Placement Algorithm

Not all available datastores can be used for initial placement. (1) A datastore may not have sufficient resources to admit the VM (2) Inter-VM anti-affinity rules may be violated due to particular placement or (3) VM may not be compatible with some datastores

in the datastore cluster. Storage DRS applies these filters before evaluating VM placement on a datastore. For each datastore that passes the filters, the placement is checked for conflicts with pending recommendations. Then Storage DRS computes datastore cluster imbalance as if the placement was done on that datastore. Goodness is measured in terms of change in imbalance as a result of the placement operation. After evaluating VM placements, they are sorted based on their goodness values. Datastore with highest goodness value is overall the best datastore available for VM placement.

3.2 Load Balancing

Load balancing ensures that datastores in a datastore cluster do not exceed their configured thresholds as space consumption and IO loads change during runtime. Unlike DRS, which minimizes the resource usage deviation across hosts in a cluster, Storage DRS is driven by threshold trigger. Its load balancing mechanism moves VMs out of only those datastores, which exceed their configured threshold values. Figure 4 gives the outline of load balancing as used by Storage DRS. For each pass of Storage DRS, the algorithm is invoked first for datastores that exceeded their space threshold and later for those violating IO threshold, effectively fixing space violations followed by IO violations in the datastore cluster.

```
Data: Inventory snapshot S representing datastores,
      VMs in Storage DRS cluster
Data: Set of datastores D from S which exceed given threshold
Result: A set of recommendations R
R ← {}"
while D ≠ empty do
    Sort D by descending order of excess usage;
    bestMove ← None;
    for datastore d ∈ D do
        V ← Set of VMs running on d;
        for VM v in Set V do
            move ← [Move v from d to d1 where d1 ∈ D and d1 ≠ d];
            if CostBenefit(move) < 0 then
                continue
            if Goodness(Move) > Goodness(bestMove)
            then
                bestMove ←move;
    if bestMove = None then
        break;
    R ←RU bestMove;
    Update S;
    D ← Set of datastores from S which exceed threshold;
```

Figure 4. Storage DRS Load Balancing

Storage DRS is able to factor in multiple resources (space, IO and connected compute capacity) for each datastore when generating a move proposition. It uses weighted resource utilization vector to compare utilizations. Resources that are closer to their peak utilization get higher weights compared to others. Following example best illustrates the effectiveness of Storage DRS in multi-resource management.

Consider a simple setup of two (DS1 and DS2) datastores with 50GB space each and same IO performance characteristics. Both datastores are part of same Storage DRS cluster.

Two types of VMs are to be placed on these datastores. (1)

High IO VMs, which run a workload of 5-15 outstanding IOs for the duration of the test. (2) Low IO VMs with IO workload of 1-3 outstanding IOs for the duration of test.

The VMs have pre-allocated thick disk of 5GB each. During the experiment, high IO VMs are placed on DS1 and low IO VMs on DS2. The initial setup is as described in table 1 below:

DATASTORE	VM TYPE	NUMBER OF VMs	SPACE USED	LATENCY
DS1	High IO	4	20GB	28ms
DS2	Low IO	5	25GB	<10ms

Table 1: Initial Placement—Space and IO

The latency numbers are computed using a device model, which dictates the performance of the device as a function of workload. Storage DRS is invoked to recommend placement for a **new** High IO VM, making the total number of VMs to ten. Placement on DS1 had rating of 0, while placement on DS2 received rating of 3, even though DS1 has more free space than DS2. This is because DS1 is closer to exhausting its IO capacity. By placing the VM on DS2, Storage DRS ensures that IO bottleneck can be avoided. Furthermore, another Storage DRS pass over the datastore cluster recommended moving out one of the High IO VM from DS1 to DS2. The balanced cluster, at the end of evaluation was as in table 2 below:

DATASTORE	VM TYPE	NUMBER OF VMs	SPACE USED	LATENCY
DS1	High IO	3	15GB	<10ms
DS2	Low IO High IO	5 2	35GB	<10ms

Table 2: Final Datastore Cluster State

Note that the space threshold for datastores is set to 80% of their capacity (40GB). So from space perspective, the final configuration is still balanced.

Storage DRS is equally effective in balancing multiple resources while load balancing VMs in a cluster. Consider the two datastores (DS1, DS2) setup as before. This example uses 9 VMs with high IO workload with 1.2GB disk and 8 VMs with low IO workload and 5GB disk. All VMs were segregated on datastores based on their workload and space profiles, so the initial setup looks as described in Table 3.

DATASTORE	VM TYPE	NUMBER OF VMs	SPACE USED	LATENCY
DS1	High IO, Low Space	9	10.8GB	40ms
DS2	Low IO, High Space	8	40.0GB	<10ms

Table 3: Imbalanced Initial Configuration

The space threshold was configured at 75% datastore capacity (37.5GB). As is evident from the description, the setup is imbalanced from both space and IO perspective. Storage DRS load balancing was run multiple times on this cluster, during which 5 moves were proposed. The final configuration looked as in Table 4.

DATASTORE	VM TYPE	NUMBER OF VMs	SPACE USED	LATENCY
DS1	High IO, Low Space	6	17.2GB	31ms
	Low IO, High Space	2		
DS2	High IO, Low Space	3	33.6GB	19ms
	Low IO, High Space	6		

Table 4: Balanced Final Configuration

Note that final configuration does not perfectly balance space as well as IO, but performs only the moves sufficient to bring the space consumption below the configured threshold and balance IO load more evenly.

3.2.1 Filters

Load balancing is done in two phases. First, moves balancing space usage are generated followed by IO balancing. Space moves are evaluated for future space growth. A move which violates space or IO threshold or that may cause destination datastore to run out of space in future is rejected. Similarly, IO load balancing move of a VM to datastore with higher latency than source is filtered out.

3.2.2 Cost Benefit Analysis

Although a move is useful in balancing resource usage of datastore cluster, Storage vMotion is lengthy and costly operation. After a VM is moved to its destination, it should make cluster resource usage fair for a long time. Otherwise, changes in VM workload/growth rate can cause ping-pong moves. So, in addition to goodness, each move is evaluated per its cost and resulting benefit. For Storage, the benefit is computed as net reduction in normalized resource usage for space and IO on source with respect to increase on destination. The cost is computed in terms of total storage transferred as part of Storage vMotion and the number of IOs, which experience increased latency for that period. Linked clones indirectly affect cost benefit analysis. A move to a datastore, which has larger portion of clone disk chain is preferred, because it results in greater space savings as well as less costly Storage vMotion.

3.3 Pending Recommendations

Provisioning of new VMs and Storage vMotion of existing VMs can take several minutes to complete in a typical case. New invocation of Storage DRS under such transient conditions may perceive available free space and IO incorrectly and generate recommendations, based on stale information. Execution Engine tracks lifecycle of recommendations and corresponding actions. Prior to load balancing run, an accurate snapshot is generated in order to produce valid recommendations. This ability is important for group deployments such as vApp and test/dev environments. Similarly, during the algorithm run Storage DRS maintains a snapshot of resource state of datastores and VMs in the datastore cluster. After generating a valid recommendation, its impact is applied to the snapshot and resource values are updated before searching for next load balancing move. That way, each subsequent recommendation does not conflict with prior recommendations.

4. Constraint Handling

Storage capacity and IO performance are the primary resources Storage DRS considers in its initial placement and load balancing decisions. In this section, we describe several additional constraints that are taken into account when a virtual disk is placed on a datastore. There are two types of constraints considered by Storage DRS:

- **Platform constraints**: Some features that are required by a virtual disk may not be available in certain storage hardware or operations may be restricted due to firmware revision, connectivity, or configuration requirements.
- **User specified constraints**: Storage DRS provides a rule engine that allows users to specify affinity rules that restrict VM placement. In addition, Storage DRS behavior is controlled by a number of configuration options that can be used to relax restrictions or specify user preferences on a variety of placement choices. For example, the degree of space overcommit on a datastore with thin provisioned virtual disks can be adjusted dynamically using a configuration setting.

Storage DRS considers all constraints together when it is evaluating virtual disk placement decisions. Constraint enforcement can be strict or relaxed for a subset of constraints.

In strict enforcement, no placement or load balancing decision that violates a constraint is possible, even though they may be desirable from performance point of view. In contrast, relaxed constraint enforcement is a two-step process. In the first step, Storage DRS attempts to satisfy all constraints using strict enforcement. If this step is successful, no further action is necessary. In the second step, relaxed constraints can be violated when Storage DRS considers placement decisions. The set of constraints that can be relaxed and under what conditions is controlled by user configuration settings.

4.1 Platform Constraints

In this section, we describe platform constraints enforced by Storage DRS. The first group of platform constraints are defined as part of VMware APIs for Storage Awareness [15] (VASA) that enable storage devices to communicate their configuration constraints to Storage DRS. As we have described earlier, storage controllers determine the amount of actual capacity available for thin provisioned datastores. When the available backing space for a thin provisioned datastore runs low, storage controllers send events to the vCenter management server using the VASA API so that Storage DRS can adjust its placement decisions accordingly.

In these cases, even though the actual available space is running low, the datastore may appear to have a much larger free space. In response, Storage DRS does not place new virtual disks or move

other virtual disks to datastores that are identified as running low on capacity. These restrictions are lifted when the storage administrator provisions new physical storage to back a thin provisioned datastore. This is an example of a dynamic constraint declared entirely by the storage controllers.

VASA APIs are also used to control whether Storage DRS can move virtual disks from one datastore to another in response to imbalanced performance. Since relocating a virtual disk to a different datastore also shifts the IO workload to that datastore, a performance improvement is possible only when independent physical resources back two datastores. In a complex storage array, two different datastores may be sharing a controller or physical disks along the IO path. As a result, their performance may be coupled together. Storage DRS considers these relationships and attempts to pick other, non-correlated datastores to correct performance problems.

If virtual disks are created using storage-specific APIs (native cloning or native snapshots), Storage DRS restricts the relocation of such virtual disks, as native API must be used for their movement. Alternatively, if new provisioning operations require storage-specific APIs to be used, Storage DRS constrains the candidate pool of datastores that has the necessary capabilities. For example, thin-provisioned virtual disks are only available in recent versions of VMFS. In these cases, the capabilities are treated as strict constraints on actions generated by Storage DRS.

Platform specific constraints can also be specified through VMware Storage Policy Based Management (*SPBM*) framework, by using storage profiles. Storage profiles associate a set of capabilities generated by users, system administrators, or infrastructure providers. For example, a high performance, highly available datastores can be tagged as "*Gold storage*", whereas other datastores with lower RAID protection can be tagged as "*Silver storage*". Storage DRS clusters consist of datastores with identical storage profiles. That way, load balancing and initial placement operations satisfy SPBM constraints. In the future, storage clusters can be more flexible and allow for datastores with different storage profiles as we discuss later.

Storage DRS is compatible with VMware HA. HA adds a constraint that the VM's configuration files can only be moved to a datastore visible to the host running HA master. Storage DRS checks for such conditions before it proposes moves for HA protected VMs.

4.2 User-specified Constraints

Storage DRS provides a rich rule enforcement engine that allows users to specify affinity rules that restricts VM placement. Storage DRS supports the following rules:

- **Virtual Machine Anti-Affinity**: If one or more VMs are part of an anti-affinity rule, Storage DRS ensures that they are placed on different datastores. This is useful to identify a set of VMs that should not fail together so that services supported by those VMs are always available. In enforcing anti-affinity rules, Storage DRS also considers physical resource sharing as reported through VASA APIs so that anti-affine VMs are not placed on datastores where storage controllers identified as sharing resources.
- **Virtual Disk Anti-Affinity**: If a single VM has more than one virtual disk, Storage DRS supports placing them on different datastores. Anti-affinity is useful for managing storage of I/O intensive applications such as databases. For example, log disks and data disks can be automatically placed at different datastores, enabling better performance and availability.
- **Virtual Disk Affinity**: Using this rule, all virtual disks of a virtual machine can be kept together on the same datastore. This is useful for majority of the small servers and user VMs as it is easier for administrators when all the files comprising a VM are kept together. Furthermore, this simplifies VM restart after a failure.

Storage DRS also supports relaxation of affinity-rule enforcement during rare maintenance events, such as when VMs are evacuated from a datastore. Since affinity-rule enforcement might constrain the allocation of available resources, it is useful to temporarily relax these constraints when available resources will be reduced intentionally for maintenance.

Storage DRS respects constraints for all of its regular operations such as initial placement and load balancing. User specified constraints could be added or removed at any given time. If there is any constraint violation after the rule set is modified, Storage DRS immediately generates actions to fix the rule violations as a priority. User-specified constraints can also be added using certain advanced configuration settings. For example, the degree of space over provisioning due to thin provisioning can be explicitly controlled by instructing Storage DRS to keep some reserve space for virtual disk growth. This is done by using a certain fraction of unallocated space for thin disks as consumed. This fraction is controlled by an advanced option.

5. Estimating Space Usage

IIn this section, we describe how Storage DRS estimates the space requirements of virtual disks for scenarios where space consumption is highly dynamic. Thin provisioned virtual disk grows over time as the new data is written for the first time, up to the provisioned capacity of the virtual disk. The rate of growth of a virtual disk is variable depending on the applications running inside the VM. In addition, frequent creation and deletion of virtual machine snapshots also contribute to the variable growth rates. This is because there is no separate placement step for snapshot creation; VM's current datastore is used for the newly created snapshots. Finally, linked clones themselves use delta disks that contain only the different content from the base disk using which the clone is created. Since the delta disk grows over time similar to a thin provisioned disk, different datastores experience varying space usage growth rates. It is important to keep track of datastore space usage automatically and take actions with the unequal growth rates taken into account. This prevents a datastore from running out of space and causing a service disruption to the running VMs.

Storage DRS avoids out of space scenarios and extraneous moves by modeling space growth. It maintains a running average of datastore space usage over a period of time and predicts the space growth based on this running average. Using the model,

Storage DRS avoids placing VMs on datastores where space will be running out faster than other datastores for a fixed time in the future. Following example illustrates space modeling. The initial setup is as described in table 5 below:

DATASTORE	FREE SPACE	NUMBER OF VMs	TIME TO FULL SIZE
DS1	50GB	23	80Hrs
DS2	50GB	20	30Hrs

Table 5: Initial Placement with Growing Disk

Each of the VMs starts with 100 MB and eventually grows to 2 GB in size. This is analogous to typical lifecycle of a VM with thin provisioned virtual disks. VMs on DS1 grow slowly, and attain a size of 2 GB in 80 hours, while VMs on DS2 grow to 2 GB in 30 hours. After 80 hours, DS1 will have used 46 GB of space, while in 30 hours; DS2 will hit 40 GB space usage. Next time Storage DRS places a VM; it chooses DS1 even though it has less free space at the time of placement. Since growth rate of VMs on DS1 is slower, over time the space usage across these datastores will be balanced.

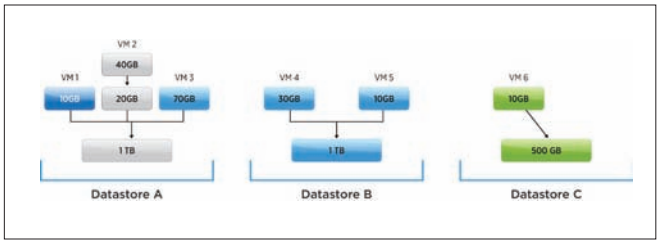

Figure 5. Space Usage with Linked Clones

Linked clones have a different type of complexity for space usage estimation: relocation of a linked clone will consume different amounts of space depending on the availability of base disks. Consider the setup as outlined in Figure 5. Datastore A and Datastore B have both an identical base disk that are used by VM1, VM2, VM3, VM4, and VM5. VM1 currently uses 10 GB in its delta disk plus the 1 TB base disk at Datastore A. Similarly, VM4 is using 30 GB of delta disk and the 1 TB base disk at Datastore B. If VM1 is to be relocated to Datastore B, only 10 GB of the delta disk will have to be moved, since VM1 can start using the identical base disk at Datastore B. However, if VM1 is to relocate to Datastore C, a copy of a 1 TB base disk has to be made as well, and as a result, both the base disk and the delta disk have to be moved. Note that relocating VM1 to Datastore B not only results in a smaller total space being used but also will complete faster since a much smaller amount of data needs to be moved.

The base disk of linked clones is retained so long as there is at least one clone using the base disk. For example, unless both of Vm4 and Vm5 are relocated to a different datastore, the 1 TB base disk will continue to occupy space at Datastore B. Moving either of Vm4 or Vm5 alone will result in space savings of only 30 GB and 10 GB respectively, leaving the 1 TB base disk intact.

6. Best Practices

Given the complexity and diversity of storage devices available today, it is often hard to design solutions that work for a large variety of devices and configurations. Although we have tried to set the default knobs based on our experimentation with a couple of storage arrays, it is not possible to procure and test many of the devices out there. In order to help system administrators to get the best out of Storage DRS, we suggest the following practices in real deployments:

(1) Use devices with similar data management features: Storage devices have two types of properties: data management related and performance related. The data management related properties include RAID level, back up capabilities, disaster recovery capabilities, de-duplication etc. We expect all datastores in a storage DRS cluster to have similar such data management properties so that the virtual disks can be migrated among them without violating any business rules. Storage DRS handles the performance variation among devices but assumes that virtual disks are compatible with all devices based on the data management features. This is something that we plan to relax in future, by using storage profiles and placing or migrating virtual disks only on the datastores with compatible profile.

(2) Use full or similar connectivity to hosts: We suggest keeping full connectivity among datastores to hosts. Consider a datastore DS1 that is only connected to one host as an extreme case. The VM whose virtual disk is placed on DS1 can not be migrated to other hosts in case that host has high CPU and memory utilization. In order to migrate the virtual machine, we will also have to migrate the disks from that datastore. Basically, poor connectivity constraints the movement of VMs needed for CPU and memory balancing to a few set of hosts.

(3) Correlated datastores: Different datastores exposed via a single storage array may share the same set of underlying physical disks and other resources. For instance, in case of EMC ClaRiiON array, one can create RAID groups using a set of disks, with a certain RAID level and carve out multiple LUNs from a single RAID group. These LUNs are essentially sharing the same set of underlying disks for RAID and it doesn't make sense to move a VMDK from one to another for IO load balancing. In storage DRS, we try to find such correlation and also allow storage arrays to tell us about such performance correlation using VASA APIs. In future, we are also considering exposing an API for admins to provide this information directly, if the array doesn't support VASA API to provide correlation information.

(4) Ignoring stats during management operations: Storage DRS collects IO stats for datastores and virtual disks continuously and computes online percentiles during a day. These stats are reset once a day and a seven-day history is kept although only one-day stats are used right now. In many cases, there are nightly background tasks such as back-up and virus scanners that lead to a very different stats profile during the night as compared to actual workday. This can also happen for tasks with a different periodicity, such as a full backup on a weekend. These tasks can distort the view of load on a datastore for Storage DRS. We have provided API support to declare such time periods during which the stats should not be collected and integrated in to the daily profile. We suggest storage administrators

to use this API, to avoid any spurious recommendations. Furthermore, it is a good idea to ignore recommendations after a day of the heavy management operation such as RAID rebuild or something, unless it is actually desirable to move out of the datastore that went through the rebuild process and provided high latencies, to protect against future faults.

(5) Use the affinity and anti-affinity rules sparingly: Using too many rules can constrain the overall placement and load-balancing moves. By default we keep all the VMDKs of a VM together. This is done to keep the same failure domain for all VMDKs of a VM. If this is not critical, consider changing this default, so that storage DRS can move individual disks if needed. In some cases using rules is a good idea, since only the user is aware of the actual purpose of the VMDK for an application. One can use VMDK-to-VMDK anti-affinity rules to isolate data and log disks for a database on two different datastore. This will not only improve performance by isolating a random IO stream from a sequential write stream, but also provide better fault isolation. To get high availability for a multi-tier application, different VMs running the same tier can be placed on separate datastores using VM-to-VM anti-affinity rules. We expect customers to use these rules only when needed and keeping in mind that in some cases, the cluster may look less balanced due to the limitations placed by such rules on our load balancing operation.

(6) Use multiple datastore instead of using extents: Extents allow the admin to extend a single datastore to multiple LUNs. These are typically used to avoid creating a separate datastore for management. Extent based datastores are hard to model and reason about. For instance, the performance of that datastore is a function of two separate backing LUNs. IO stats are also a combination of the backing LUNs and are hard to use in a meaningful way. Features like SIOC are also not supported on datastores with multiple extents. With Storage DRS the management problem with multiple datastores is already handled. So we suggest storage administrators to use separate datastores per LUN and use storage DRS to manage them, instead of using extents to increase the capacity of a single datastore.

(7) Storage DRS and SIOC threshold IO latencies: Users specify threshold IO latency of datastore while enabling Storage DRS on a datastore cluster. Storage DRS uses threshold latency value as a trigger. If datastore latency exceeds this threshold, VMs are moved out of such datastore(s) in order to restore latency value below threshold.

When Storage DRS is enabled on a datastore cluster, SIOC is also enabled on individual datastores. SIOC operation is controlled by its own threshold latency value. By default SIOC threshold latency is higher than that of Storage DRS. SIOC operates at much higher time frequency than Storage DRS. Without SIOC, there could be situations where IO workloads kick in for short durations and IO latency can shoot up for those time intervals. Until Storage DRS can remedy the situation, SIOC acts as a guard and keeps IO latency in check. It also ensures that other VMs on that datastore do not suffer during such intervals and get their proportional IO share. It is important to make sure that SIOC threshold latency is higher than that of Storage DRS. Otherwise, datastore latency will always appear lower than the threshold value to Storage DRS and it will not generate moves to balance IO load in the datastore cluster.

7. Discussion and Future Work

7.1 Storage DRS in the Field

VMware DRS technology influenced Storage DRS to a large extent. They both use similar concepts such as cluster, load balancing domain, recommendations, rules, and faults.

Over time, as Storage DRS deployments increased in the field, a few key differences have emerged.

Unlike DRS, it is critical for Storage DRS to choose the right storage for virtual disks. While placing a VM, users want simplicity and policy driven placement. As discussed previously, we plan to expand Storage DRS to make VM placements work seamlessly across all storage in a vSphere environment. Another key difference is the scale. Ideally, Storage DRS users want aggregation of all storage in a single pool, and let Storage DRS automatically manage VM placements, IO performance, space usage, and policy compliance over the entire pool. In this regard, we are working towards improvements in scale and performance.

Storage architectures have evolved significantly since the initial design of Storage DRS. As storage controllers become more intelligent, the existing black box performance models [5] are less capable of covering all salient aspects of device performance. We are exploring new interfaces for performance modeling so that storage devices can compactly report their performance capabilities that can be used by Storage DRS. In addition; we are exploring combinations of active [5] and passive [4] performance modeling approaches.

Storage DRS best practices recommend datastores with similar data management features and even similar disk types in a datastore cluster. Many customers want the capability to include different storage types in terms of capacity, performance, protocols, protection level, etc. in a single pool. These not only require improvements in automation, but also more fine grain controls when managing datastores in a cluster.

7.2 Future Directions

Storage technologies and storage system designs are evolving at a very rapid pace. With the introduction of multi-core CPUs, high-bandwidth interconnects and solid-state disks, many new storage architectures are coming to market. Some of the common new designs include: multi-tiered storage, scale-out storage and compute-storage converged architectures.

In case of multi-tiered storage, SSDs are being used as a primary storage media either as a first tier or as a caching layer. Both designs make it harder to model the datastore from outside as a black box. This is because an outside observer cannot know the hit rate of IOs in the SSD tier.

The scale-out storage paradigm is very useful for cloud service providers. They can buy a unit of storage and scale out as demand grows. In this case, the servers may see a single connection point but the IOs can get served from one of many backend storage controllers via internal routing. Some examples of this architecture include EMC Isilon [3], NetApp ONTAP GSX [2], IBM SoNAS [7] etc. These architectures make it hard to determine the amount of available performance left on the storage device.

The server-storage converged architectures such as Nutanix [11], Simplivity [12], HP Lefthand [6] etc. take the scale-out to the next level by coupling together local storage across servers to form a shared datastore. These solutions provide high-speed local access for most of the IOs and do remote IOs only when needed or for replication. In all these cases, it is quite challenging to get a sense of performance available in a datastore. We think a good way to manage these emerging storage devices is by creating a common API that brings together the virtual device placement and management with that of the internal intelligence of the storage devices. We are working towards such a common language and incorporating it as part of VASA APIs.

So far we have talked about modeling IO performance, but similar problems arise for space management as well. Thin provisioning, compression and de-duplication for primary storage are becoming common features for space efficiency in arrays. This makes it harder to estimate the amount of space that will get allocated to store a virtual disk on a datastore. Currently Storage DRS uses the actual provisioned capacity as reported by the datastore. In reality the space consumed may be different: this has the effect of space usage estimations being slightly inaccurate. We are working on modifying storage DRS to handle such cases and also planning to add additional APIs for better reporting of space allocations.

Overall, building a single storage management solution that can do policy based provisioning across all types of storage devices and perform runtime remediation when things go out of compliance is the main goal for Storage DRS.

8. Conclusion

In this paper, we presented the design and implementation of a storage management feature called Storage DRS from VMware. The goal of Storage DRS is to make several management tasks such as initial placement of virtual disks, out of space avoidance and runtime load balancing of space consumption and IO load on datastores. Storage DRS also provides a simple interface to specify business constraints using affinity and anti-affinity rules, and it enforces them while making provisioning decisions. We also highlight some of the complex array and virtual disk features that make the accounting of resources more complex and how Storage DRS handles that. Based on our initial deployment in the field, we have gotten a very positive response and feedback from customers. We consider Storage DRS as the beginning of the journey to software managed storage in virtualized datacenters and we are planning to accommodate the newer storage architectures and storage disk types in future to make Storage DRS even more widely applicable.

References

1 Resource Management with VMware DRS, 2006, http://vmware.com/pdf/vmware_drs_wp.pdf

2 M. Eisler, P. Corbett, M. Kazar and D. S. Nydick. Data Ontap GX: A scalable storage cluster. In *5th USENIX Conference on File and Storage Technologies*, pages 139-152, 2007.

3 EMC, Inc. EMC Isilon OneFS File System. 2012, http://simple.isilon.com/doc-viewer/1449/emc-isilon-onefs-operating-system.pdf

4 A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated IO Load Balancing across Storage Devices. In *USENIX FAST, Feb 2010*.

5 A. Gulati, G. Shanmuganathan, I Ahmed, M. Uysal and C. Waltspurger. Pesto: Online Storage Performance Management in Virtualized Datacenters. In *Proc. Of the 2nd ACM Symposium on Cloud Computing*, Oct 2011.

6 Hewlett Packard, Inc. HP LeftHand P4000 Storage. 2012, http://www.hp.com/go/storage

7 IBM Systems, Inc. IBM Scale Out Network Attached Storage, 2012. http://www-03.ibm.com/systems/storage/network/sonas/index.html

8 A. Mashtizadeh, E. Celebi, T. Garnkel, and M. Cai. The Design and Evolution of Live Storage Migration in VMware ESX. In Proc. USENIX Annual Technical Conference (ATC'11), June 2011 (to appear).

9 D. R. Merrill. Storage Economics: Four Principles for Reducing Total Cost of Ownership. May 2009, http://www.hds.com/assets/pdf/four-principles-for-reducing-total-cost-of-ownership.pdf

10 M. Nelson, B. H. Lim, and G. Hutchins. Fast Transparent Migration of Virtual Machines. In Proc. USENIX, April 2005.

11 Nutanix, Inc. The SAN free datacenter. 2012, http://www.nutanix.com

12 Simplivity, Inc. The Simplivity Omnicube Global Federation. 2012, http://www.simplivity.com

13 N. Simpson. Building a data center cost model. Jan 2010, http://www.burtongroup.com/Research/DocumentList.aspx?cid=49

14 S. Vagnani. Virtual machine le system. In ACM SIGOPS Operating Systems Review 44.4, pages 57-70, 2010.

15 VMware, Inc. VMware APIs for Storage Awareness. 2010, http://www.vmware.com/technical-resources/virtualization-topics/virtual-storage/storage-apis.html

16 VMware, Inc. VMware vSphere Storage IO Control. 2010, http://www.vmware.com/files/pdf/techpaper/VMW-vSphere41-SIOC.pdf

17 VMware, Inc. VMware vSphere. 2011, http://www.vmware.com/products/vsphere/overview.html

18 VMware, Inc. VMware vCenter Server. 2012, http://www.vmware.com/products/vcenter-server/overview.html

19 VMware, Inc. VMware vSphere Fault Tolerance. 2012, http://www.vmware.com/products/datacenter-virtualization/vsphere/fault-tolerance.html

20 VMware, Inc. VMware vSphere High Availability. 2012, http://www.vmware.com/solutions/datacenter/business-continuity/high-availability.html

A Social Media Approach to Virtualization Management

Ravi Soundararajan ravi@vmware.com
Emre Celebi ecelebi@vmware.com
Lawrence Spracklen lspracklen@vmware.com
Harish Muppalla harishm@vmware.com
Vikram Makhija vmakhija@vmware.com

Performance Group and Product Management, VMware, Inc.

Abstract

The average virtualization administrator finds it difficult to manage hundreds of hosts and thousands of virtual machines. At the same time, almost anyone on earth with a smartphone is adept at using social media sites like Facebook for managing hundreds of friends. Social media succeeds because a) the interfaces are intuitive, b) the updates are configurable and relevant, and c) the user can choose arbitrary groupings for friends. Why not apply the same techniques to virtualized datacenter management?

In this paper, we propose combining the tenets of social media with the VMware® vSphere® management platform to provide an intuitive technique for virtualized datacenter management. An administrator joins a social network and "follows" a VMware vCenter™ server or another monitoring server to receive timely updates about the status of an infrastructure. The vCenter server runs a small social media client that allows it to "follow" hosts and their status updates. This client can use the messaging capabilities of social media (posting messages to lists, deleting messages from lists, sending replies in response to messages etc.) to apprise the administrator of useful events. Similarly, hosts contain a small client and can "follow" virtual machines and be organized into communities (clusters), and virtual machines can be organized into "communities" based on application type (for example, all virtual machines running Microsoft Exchange) or owner (for example, all virtual machines that belong to user XYZ). By creating a hierarchy from an administrator to the host to the virtual machine, and allowing each to post status updates to relevant communities, an administrator can easily stay informed about the status of a datacenter. By utilizing message capabilities, administrators can even send commands to hosts or virtual machines. Finally, by configuring the types of status that are sent and even the data source for status updates, and by using social media metaphors like hash tags and 'likes', an administrator can do first-level triaging of issues in a large virtualized environment.

Categories and Subject Descriptors

D.m [**Miscellaneous**]: virtual machines, system management, cloud computing.

General Terms

Performance, Management, Design.

Keywords

virtual machine management, cloud computing, datacenter management tools.

1. Introduction

One of the most challenging problems in virtualized deployments is keeping track of the basic health of the infrastructure. Operators would like to know quickly when problems occur and would also like to have guidance about how to solve issues when they arise. These problems are exacerbated at scale: it is already difficult to visualize problems when there are 100 hosts and 1000 virtual machines (virtual machines), but what about in setups with 1000 hosts and 10,000 virtual machines? Conventional means for monitoring these large environments focus on reducing the amount of data to manageable quantities. Reducing data is difficult, requiring two 2 distinct skill sets: first, knowledge of virtualization, in order to determine what issues are serious enough to be alerted and how to solve such issues; second, the ability to create intelligent visualizations that reduce data into manageable chunks.

Automated techniques for monitoring the health of an infrastructure [13] have become increasingly prevalent and helpful. Such approaches leverage the collection and analysis of a large number of metrics across an environment in order to provide a concise, simplified view of the status of the entire environment. However, despite the success of such tools, significant training is often required in order to obtain proficiency at understanding and using the output of such tools.

In this paper, we approach the problem of virtualization monitoring from a different perspective. We observe that while the average administrator might find it difficult to monitor 100 hosts and 1000 virtual machines, the same administrator might find it relatively easy to keep abreast of his hundreds of Facebook[1] friends. The reason for this is that social networking sites allow many knobs to limit the information flow to a given user. Moreover, these knobs are extremely intuitive to use. For example, users of Google+ can organize friends into circles and limit status updates to various circles, or might choose to propagate status updates only to select listeners. The knobs are also designed with a keen understanding of the problem domain: for example, in a social network, birthdays are important events, so social networks create special notifications based solely on birthdays.

We propose organizing a virtualized environment into a social network of its own, including not just humans (system administrators), but also non-human entities (hosts, virtual machines, and vCenter servers). Each member of the community is able to contribute

status updates, whether manually (in the case of humans) or programmatically (through automated scripts running on virtual machines and hosts). We organize this social network according to our understanding of the hierarchy in a virtualized environment, and we limit the information flow so that only the most important updates reach an administrator. Moreover, the administrator is capable of performing commands within the social media client. By combining the reduction of information with the ability to perform basic virtualization management operations in response to such information, and by wrapping these features into the familiar UI of a social media application, we create an intuitive, platform-independent method for basic monitoring and management of a virtualized environment.

The outline of this paper is follows. In section 2, we explain how we map the constructs of a social network to a virtual hierarchy. In section 3, we describe a prototype design for such a monitoring scheme, leveraging one set of social networking APIs, the Socialcast Developer APIs. In section 4, we describe our initial experiences deploying such a system in a real-world environment. In section 5, we discuss related work. We provide conclusions and future directions in section 6.

2. Comparing Virtual Inventories and Social Media Networks

Figure 1 depicts a sample social network. A two-way arrow suggests a friend relationship. For example, A is friends with B and B is friends with G, but G is not friends with A. In addition, B, F, and G might choose to create a separate, private group, as indicated by the dotted rectangle in the figure. There is a distinction between physical entities, namely the members of the groups like A through G, and the logical entities, like the group consisting of B, F, and G.

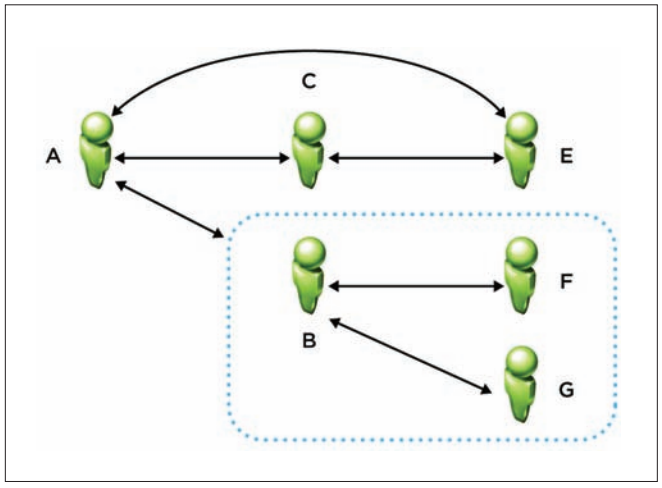

Figure 1: A sample social network, illustrating friend relationships, groups, and hierarchies.

Similar to a social network, a virtualized infrastructure also consists of 'physical' members and logical groups. For example, consider the sample VMware vSphere[15] inventory shown in Figure 2. As the figure indicates, vCenter server W1-PEVCLOUD-001 is composed of a datacenter named vCloud, which in turn consists of a cluster AppCloudCluster and a group of hosts. The cluster contains resource pools and virtual machines. This hierarchy can be mapped to a 'social' network of its own. An administrator can be a 'friend' of a vCenter server. A vCenter server can have hosts as friends, and hosts can have virtual machines as friends. Hierarchy is important because it is one method of limiting information flow. In a social network, a person like A might choose instead to only be friends with B, knowing that if anything interesting happens to F and G, that B will likely collect such information and share it with A. In a similar manner, the vCenter server need not choose to be friends with all virtual machines, but just with all hosts. If a host receives enough status updates from the virtual machines running on it, it might choose to signal a status change to vCenter. In a similar way, an administrator might choose to be friends only with vCenter, knowing that vCenter can accumulate status updates and propagate them to the administrator.

While hosts and VMs are entities in our virtualization social network, we currently do not add datacenters, clusters, and resource pools as entities. At present, this is because of a practical issue. Datacenters, clusters, and resource pools cannot be added as friends because they do not have a 'physical' manifestation. In other words, while an administrator can send and receive network packets to/from virtual machines and hosts, an administrator cannot send a message to a datacenter. Instead, datacenters, clusters, resource pools, and host/virtual machine folders are more similar to a 'group' in a social network. We extend the notion of a group to include not just virtualization hierarchy, but also allow arbitrary user-defined collections of entities. For example, it might be helpful to put all virtual machines that run a SpecJBB[9] in a group labeled SpecJBB, or it might be helpful to put all virtual machines under a given resource pool in a given group.

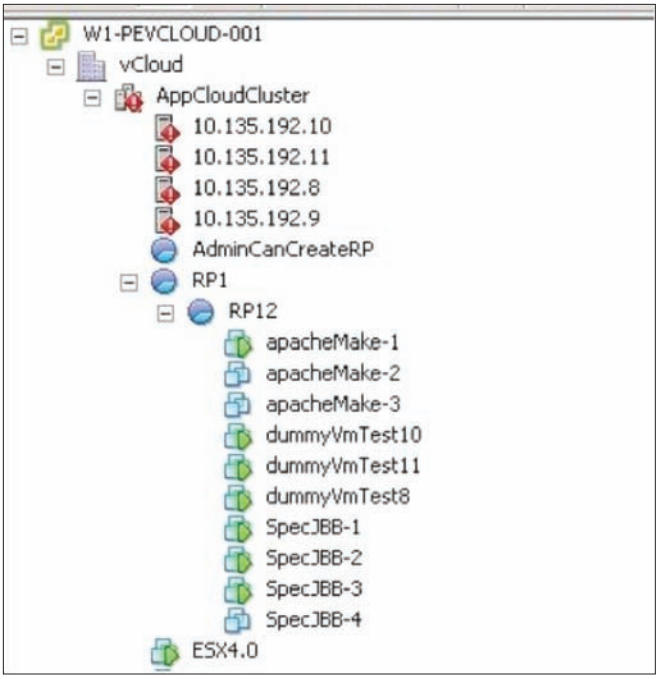

Figure 2: A Sample VC Inventory. Hosts, virtual machines and the vCenter server itself can be mapped to members of a social network, while datacenters, clusters, and resource pools are like groups.

3. Prototype Design

In this paper, we propose that virtual machines, hosts, and vCenter servers become nodes in a social network. Each one runs agents that publish various pieces of information to the social network. An administrator can then examine the web site (or her mobile device) in order to get interesting information about the virtualized infrastructure in a more intuitive manner than a standard management interface.

In order to validate this design approach, we discuss a proof-of-concept design based on using the Socialcast Developer API [7]. The verbs of social media messaging, as encapsulated in the Socialcast Developer API, map very well to the verbs required to build an efficient notification system for virtualized infrastructure. In the next two sections, we describe the Socialcast API and then indicate how it might be used to build a management infrastructure. Using a special on-premise Socialcast virtual appliance, we have been able to prototype and validate most of the proposals described in this section.

3.1 The Socialcast API

Before discussing our prototype design, we first give a brief description of social media messaging in Socialcast. There are several kinds of messages in Socialcast. There are *community streams*, in which a group of users can essentially subscribe to a given topic and see messages related to that topic. There are also *private messages*, which are messages directed to a particular user and not viewable by anyone else. There are *comments*, in which users can essentially respond to existing stream messages, and there are *private message replies*, which are similar to comments, but are responses to private messages. Messages and comments can be *liked* (in which other users express approval) or *un-liked* (in which a previously-posted 'like' is removed). Messages can be tagged with categories or filtered by content. Finally, users can be *followed*: if user A is followed by user B, then when user A makes comments, user B is notified of them. This allows user B to keep abreast of the events in A's life.

Based on this description of the message types in Socialcast, we can now take a brief look at the relevant portions of the API.

1. **Messages API**: The messages API allow users to read a single stream message or a group of stream messages, create new messages, update new messages, destroy messages, and search messages. A user can also specify the retrieval of messages since a certain date, the retrieval of messages that fit certain criteria, etc.
2. **Likes API**: The likes API allows a user to like a message or un-like a message.
3. **Comments API**: The comments API allows a user to retrieve comments, create comments, update comments, or delete comments. There is also a 'comment likes' API where a user can like or un-like a comment.
4. **Flagging API:** With flagging, a user tags a message that she has posted as being important to her, as a reminder to her to view it later.
5. **Private Messages API**: The private messages API allows users to perform all of the same actions as in the standard messages API, but for private messages.
6. **Users API**: The users API allows users to retrieve information about other users, search for users, deactivate users, and retrieve messages from a specific user.
7. **Follow/Unfollow API**: The follow API allows users to 'follow' other users (i.e., see comments or notifications by the other users).
8. **Groups API**: The groups API allows users to list groups, the members of groups, and group memberships of a given user.
9. **Attachments API**: The attachments API allows a user to create attachments (either separately or as part of a message).

These commands can be simple HTTP GET or POST requests. Simply by installing a library like libcurl[2] in virtual machines, we are able to have virtual machines programmatically send status and receive status. Adding this library to an ESX host further enables an ESX host to send/receive status. In essence, because of the ability to programmatically interact with messages, groups, etc., the hosts and virtual machines are able to be users in the virtualization social network in the same way that human beings are members of the virtualization social network.

3.2 Mapping the Socialcast API to virtualization monitoring

To understand how the Socialcast model fits in with virtualization monitoring, consider how virtualized environments are monitored. If important events happen, then notifications are sent to an administrator. These notifications are acknowledged and then cleared. Multiple similar issues might happen among a group of hosts or virtual machines, suggesting a common root cause. Messages can be flagged according to severity, and messages with common headers can be additional categorized.

Consider our canonical design in which an administrator follows the vCenter server. The vCenter server, in turn, follows hosts, as shown in Figure 3. The hosts follow virtual machines. Because hosts are often in clusters, it might be useful to organize a set of hosts into a group named after the parent cluster. Virtual machines might reside in folders or resource pools, so virtual machines might be placed in groups based on parent folder or resource pool. Moreover, other interesting groupings are possible. For example, perhaps every physical host that belongs in rack X can go into a group named X, or every virtual machine running Microsoft Exchange can go into a group named "Microsoft Exchange." An administrator might also decide to join such a group of virtual machines to view notifications related to these 'Microsoft Exchange' virtual machines.

Figure 3: Mapping virtualization relationships to social relationships. In this case, a vCenter server is managing a total of 16 hosts. From a social media perspective, vCenter is 'following' those hosts.

This simple "social network" of persons, hosts, and virtual machines forms a powerful monitoring service. For example, when a virtual machine encounters an issue like a virtual hard drive running out of space, the virtual machine can do a simple http POST request to indicate its status ("ERROR: hard drive out of space") using the messages API, as shown in Figure 4. In this case, a custom stream for Exchange virtual machines has been created, so the virtual machine sends the message to Socialcast and specifically to this custom group. If an administrator is periodically watching updates to this stream, he might notice a flurry of activity and choose to investigate the Exchange virtual machines in his infrastructure. Alternatively, a host can have an agent running that automatically reads messages to a given stream, parses them, and performs certain actions as a result. Finally, by creating a graph connecting vCenter to its end users and to administrators, it becomes easier to notify the relevant parties when an event of interest has occurred: for example, if a virtual machine is affected, then we can limit notifications to the followers of that virtual machine (presumably the users of that virtual machine).

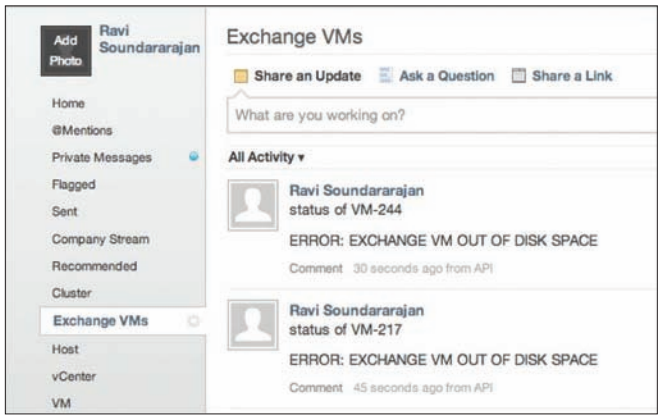

Figure 4: A community of Exchange virtual machines. The administrator has a stream for messages from Exchange virtual machines. The Exchange virtual machines send messages to the stream when they are running out of disk space.

Blindly sending messages to a stream can result in an overload of messages to a human. To avoid such an overload, we can take advantage of a helpful feature of the Socialcast API: the ability to read a stream before publishing to it. If several virtual machines are exhibiting the same issues (for example, hard drive failures), rather than each posting to the same stream and inundating an administrator with messages, the Socialcast agent on each virtual machine can programmatically read the public stream and find out if such a message already exists. If so, the virtual machine can 'like' the message instead of adding a new message to the stream. In this manner, an administrator that is subscribed to this group will not be overwhelmed with messages: instead, the administrator will see a single error message with a large number of 'like' messages. This might suggest to the administrator that something is seriously wrong with some shared resource associated with these virtual machines, like the datastore backing the virtual machines. An illustration of this is shown in Figure 5, in which a number of hosts lose connection to an NFS server. The first server that is affected posts a message, and the others 'like' that message, providing an at-a-glance view of the severity of the problem.

Figure 5: Using 'Like' as a technique for aggregating data. Each host has the same error (in this case, failure to connect to an NFS storage device), but rather than having each one post a separate message, the first affected host posts a message, and subsequent hosts 'like' that message, providing an at-a-glance view of the severity of the problem.

Along the same lines, consider a host that is following each of its virtual machines. The host can use a simple loop to poll for status updates by its virtual machines. When enough such ERROR messages are detected, the host might decide to post an aggregated "ERROR: VM disk failures" to its status. The host can also query Socialcast to find out the number of likes of a given message and thereby determine how many entities are affected by that error. The vCenter server that is following this host might then choose to update its status accordingly ("ERROR: HOST X shows VM disk failures"). The administrator, who is following this vCenter server, will then see the appropriate status notification and might decide to investigate the host. Notice that by utilizing the hierarchical propagation of messages, an administrator sees a greatly reduced set of error messages. The administrator might further decide to create a special group called a 'cluster', and put all hosts in that cluster in a group. The administrator might choose to occasionally monitor the messages in the cluster group. By seeing all messages related to the cluster in one place, the administrator might notice patterns that would not otherwise be obvious. For example, if the cluster group shows a single host disconnect message and a number of 'likes' for that single 'host disconnect' message from the other hosts, it might be the case that a power supply to a rack containing these hosts has failed, and all hosts are subsequently disconnected. Note here that the ability to read the group messages before publishing is crucial to reduce the number of messages: Rather then sending discrete messages for each error, the 'like' attribute is used. Depending on the type of power supply (managed or not), the power supply itself might be able to join a given social network of hosts and virtual machines and emit status updates.

As yet another technique for reducing information, a host or vCenter might utilize flags or comments. For example, depending on the content of the messages (for example, ERROR vs. WARNING), a host that is following its virtual machines might examine a message stream, choose the messages with ERRORS, and flag them or comment on them, indicating that they are of particular importance. The host can later programmatically examine flagged/commented messages and send a single update to the vCenter server. The vCenter server, in turn, can notify the administrator with a single message.

3.2.1 Flexibility and Extensibility

As noted earlier, because nearly all of these messages rely on simple HTTP GETs and POSTs, any virtual machine or ESX host or vCenter server can utilize the entire breadth of the Socialcast API. In each case, it is simple matter of writing a shell script that does rudimentary monitoring and invokes appropriate GET and POST requests. Moreover, more complex workflows and messaging are possible. For example, a simple script in a Linux virtual machine that monitors vmstat[5] might notice that the free memory has dipped below a predefined threshold. The virtual machine might decide to post a status update and use the attachment API to include as an attachment the output of vmstat. Alternatively, the virtual machine might decide to generate a graph and attach the graph to its status message. For Windows virtual machines running MS-SQL, perhaps an agent can periodically monitor disk activity using perfmon[6] and send the results of perfmon as an attachment to the appropriate administrator. Moreover, the architecture can be extended to include any data source that can generate POST/GET requests, ranging from physical devices like core network switches to change management software or procurement software, providing a single pane of glass for any activity related to an individual entity.

3.2.2 Sending and receiving commands via Socialcast

Messages do not have to be limited to static read-only content. For example, perhaps an administrator can send a private message to a virtual machine that includes the body of a script. When the virtual machine reads the message, it can execute the script. Similar such commands can be sent to hosts. For example, a primitive heartbeat mechanism can also be implemented: if each virtual machine and host is configured to send a message once a day, and if a host periodically checks to see if each virtual machine has issued an update, the host can potentially detect if a virtual machine has gone offline. The host could then send itself a command to power on the virtual machine, and if no response is detected from the virtual machine, a message can be posted by the host to the administrator's group. To prevent security issues with malicious users sending arbitrary commands to hosts and virtual machines, it is important to leverage the in-built features of a Socialcast community: namely, that only authorized community members and members of a given group (for example, an administrators group for system administrators) are allowed to send messages to other members.

3.2.3 Message archival and search

The preceding sections have demonstrated various advantages to using a social-media API for virtualization monitoring, including techniques for information reduction and simple interfaces for generating arbitrary types of status information in the form of attachments. An additional advantage of using an online community for virtualization monitoring is that these communities can be hosted in a private cloud, avoiding storage space concerns on any of the entities themselves. Moreover, the Administrator can periodically flush old messages or messages that have been acknowledged and acted upon. Socialcast stores messages in its database and allows full-text search as well as searching using database indexes. Thus, messages can easily be searched, providing a helpful audit trail.

3.3 Implementation Details

3.3.1 System Architecture

Based on the discussion above, one possible implementation involves installing agents in all ESX hosts, virtual machines, and vCenter instances, and having them directly post updates to a Socialcast server. Currently, this might be quite difficult, mainly because IT administrators are understandably risk-averse, so any changes to infrastructure applications (such as vCenter and the software running on ESX hosts) require significant levels of approval. As the approach matures and such an agent is more hardened and tested in the field, however, this is certainly a viable approach. To accommodate existing environments, we chose a slightly less disruptive approach that leverages only publicly available, secure interfaces.

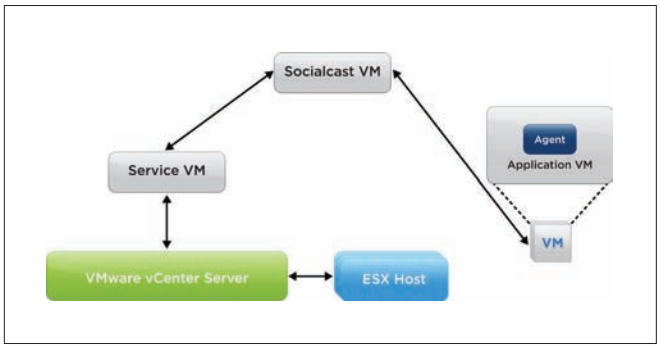

Figure 6: Initial Implementation. A Service virtual machines monitors vCenter and coordinates with vCenter to monitor ESX hosts, posting status to Socialcast on their behalf. Application virtual machines do not use the Serivce VM, and instead run monitoring agents that post status directly to Socialcast.

A logical block diagram of our initial implementation is shown in Figure 6. We use a Socialcast Virtual Machine to implement the social network. We install monitoring agents in each application virtual machine. In application virtual machines running Windows, the agent monitors WMI counters or perfmon counters and application-specific log files like Apache http logs, and it sends HTTP POST requests directly to the Socialcast Virtual Machine. For Linux virtual machines, the monitoring agent examines log files or the output of tools like iostat. For vCenter and for the ESX hosts, in our initial prototype, we chose not to install agents:* instead, we use a service virtual machine to bridge between the Socialcast Virtual Machine and vCenter. The service virtual machine communicates with vCenter over the vSphere API in order to read log data, event data, or statistics. In turn, vCenter is also able to retrieve similar data from each ESX host. In this manner, the service virtual machine only needs to authenticate to one server (vCenter) in order to retrieve information for any host in the infrastructure. The Service Virtual Machine, in turn, performs simple aggregations before posting the data to Socialcast on behalf of the appropriate ESX host or vCenter server. The Service Virtual Machine can also choose to like or comment on the data instead, on behalf of the appropriate ESX host or vCenter. Socialcast is then responsible for its own aggregations (for example, showing how many hosts are affected by a problem by displaying the number of 'likes', as shown in Figure 5). The service virtual machine monitors

* "In some early prototypes, we used agents on vCenter and ESX. However, in order to allow easier deployment in the VMware Hands-on-Lab (see section 4), we opted for this approach."

ESX hosts and vCenter and posts updates on their behalf to the Socialcast VM. While we could have used a Service Virtual Machine to probe application virtual machines in addition to vCenter and ESX, we made the tradeoff to install agents in application virtual machines for two main reasons. First, we were trying to locate application-specific behaviors, and the data we needed could not be retrieved via an API. For example, we initially wanted to probe virtual machines running VMware View [16] to detect latency issues, and such information is kept in certain log files rather than exposed publicly. Second, many application virtual machines already have existing monitoring agents, or export standard interfaces like SNMP and VMI, so there is a precedent for an administrator to admit new agents into a virtual machine. Finally, many administrators create virtual machines from templates or catalogs, so upon deployment, so it is relatively easy to automate the process of installing an agent.

3.3.2 Coupling Virtualization Management and Social Media

To effectively couple virtualization management to social media, our system must perform three functions:

1. Discover the relationships between these entities.
2. Create the appropriate mapping of these virtualization entities to entities in a social network.
3. Monitor each of these entities and post interesting status to Socialcast.

To discover the relationships between these entities, the Service Virtual Machine uses the VMware Web Services SDK to retrieve topology information about the virtualization infrastructure from the vCenter server. This topology information includes the hosts being managed by the vCenter server, the virtual machines running on each host, and the virtual datacenters and clusters. Once this topology information has been gathered, the Service Virtual Machine maps these entities to members of the social network by making calls to the Socialcast Virtual Machine to create users for the vCenter server, the virtual machines, and the hosts. The Service Virtual Machine also makes calls to the Socialcast Virtual Machine to create groups for the datacenters and clusters. To create the appropriate mapping of the relationships, we have nodes in a hierarchy 'follow' their descendants. For example, vCenter follows the hosts it is managing. The ESX hosts 'follow' the virtual machines that they are running. Virtual machines are joined to the datacenters or clusters they belong to, as are hosts. As a final step, we link users to their virtual machines, although this is currently a mostly manual step, unless the user has annotated virtual machines with user information in a structured manner amenable to auto discovery.* At this point, the graph database of the social network has a complete map of the virtualization infrastructure.

The off-the-shelf Socialcast Virtual Machine is architected primarily for human-to-human interaction and collaboration. Thus, user creation requires an administrator logging into a Socialcast instance in order to create user information and to send an email invitation, or it requires an import from another identity source like LDAP. Moreover, the joining of a group is also typically a manual operation performed by a human. As a result, the Socialcast API in its present form does not support creation of users and joining of groups. However, we have modified the API and the Socialcast Virtual Machine to support both of these operations in an automated way, allowing us to completely script the procedure of adding virtualization entities to the social network.

For monitoring these entities, we choose a variety of metrics. For vCenter itself, several factors are important. We monitor the performance metrics of the vCenter server itself like task latencies by using the vSphere API[11]. We also gather usage metrics like CPU, disk, memory and network: these can be gathered via standard interfaces like SNMP. We also examine the log files of the vCenter server itself: these log files are available via the API, given the user has appropriate permissions.

For ESX hosts, we examine performance statistics and kernel logs that are accessible via the vSphere API. The vSphere API allows administrator users with appropriate roles and permissions to login to the vCenter server and access the kernel logs of the ESX hosts. To reduce spew on Socialcast posts, we filter the kernel messages from the hosts and only post warnings and errors. To gain more insight into high vCenter task latencies, it is sometimes helpful to examine the communication logs between vCenter and the ESX hosts: we use the vSphere API to retrieve these logs. Finally, the vCenter server also has an API for retrieving performance statistics per host.

For virtual machine monitoring, we use a two-pronged approach: we collect resource usage statistics for the virtual machine (CPU, Disk, Memory, and network) using the vSphere API. In addition, our agents collect in-guest resource statistics and also examine log files. For example, certain applications like virtual desktops emit log statements when the frame rate of the desktop is low enough to cause user-perceived latencies. Our agent examines such log files and posts relevant message to Socialcast. For resource usage, we do preliminary trending analysis to see if a problem like high memory usage has occurred and then is resolved, and we post resolution of the message to Socialcast. Because these statistics are being gathered within the guest, we gain some visibility that might not be available by the virtual machine-level metrics collected using the vSphere API.

4. Monitoring Case Study: VMworld Hands-On-Labs

To validate our design and gain real-world feedback on our approach, we installed our monitoring service at the hands-on labs at VMworld 2012.[18][19][21]. The hands-on-lab allows VMworld attendees to experiment with various VMware products by following a scripted series of steps. There are over 20 different types of labs to showcase various VMware products, and there are nearly 500 users at a time. As a user enters the hands-on-lab area and indicates a preference for a lab topic, a provisioning portal determines whether a version of the requested lab is available. If not, the lab is provisioned. A lab consists of a number of virtual machines (between 10-17 virtual machines in most cases), and encompasses a mini virtual datacenter that the user can control.

* An alternate approach here could be to use commercially available application discovery tools and then provide an API to link the virtual machines to applications and applications to users.

The hands-on-lab posed unique challenges for our monitoring solution, and required modifications to our original design. The main issue is that the hands-on-lab represents a *high churn environment*, in which virtual machines are constantly being created and destroyed, existing for an hour on average. We use the vSphere API to track the creation and deletion of virtual machines and appropriately modify the relationships within Socialcast, and this environment stresses such code severely. Moreover, because the load on the ESX hosts is highly variable, virtual machines are migrated quite frequently. Our code uses the vSphere APIs to track the motion of virtual machines and update the relationships appropriately, but the frequency of updating such relationships is much higher than might be expected from a typical social network. For example, 2000 virtual machines might be created, destroyed, or moved every hour for 8 hours. In contrast, a company like VMware, with approximately 17,000 employees, might create a dozen new Socialcast users per day.

In light of these challenges and to provide adequate performance, our ultimate deployment architecture utilizes multiple Socialcast Virtual Machines organized using Socialcast clustering. We also employ multiple Service Virtual Machines and divide them among the multiple vCenter servers in the Hands-on-Lab. We also have a Service Virtual Machine for doing preliminary monitoring of the cloud management stack (VMware vCloud Director[14]) that is controlling the vCenter servers.

For our initial monitoring, we focused on a few key areas:

1. **Resource utilization of management components**. We monitored the resource usage of the management software so that we could feed the information back into our core development teams. We show an example in Figure 7, which helped us isolate a given management server that showed much higher CPU usage than others and therefore merited further investigation.

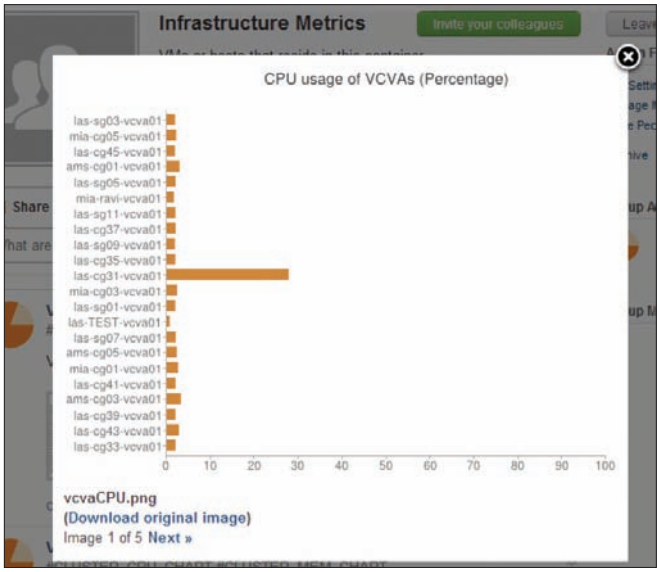

Figure 7: CPU Usage of vCenter servers across infrastructure. The chart is pushed periodically to the administrator's Socialcast stream. In this post, one of the vCenters is showing much higher CPU utilization than others and might need further investigation.

2. **Operational workload on management servers.** The hands-on lab represents an extreme of a cloud-like self-service portal. Creating a user's lab from the self-service portal ultimately results in provisioning operations on the vCenter server, including the cloning of virtual machines from templates, reconfiguring those virtual machines with the proper networking, and then destroying the virtual machines when they are no longer in use. Understanding the breakdown of operations helps developers determine which operations to optimize in order to improve infrastructure performance. We see this in Figure 8, in which we show the breakdown of tasks across all vCenter instances and notice a pattern in the workflow. Specifically, vCenter appears to perform multiple reconfigure operations per VM power operation. We can thus investigate reducing the number of reconfiguration operations as a possible orchestration optimization.

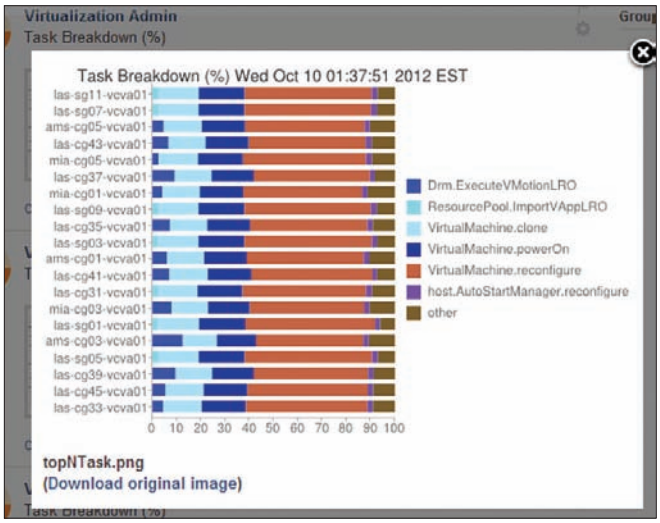

Figure 8: Operational Workload on Management Servers. The standard self-service workflow requires multiple virtual machine reconfigure operations before powering on the VM. This represents a potential optimization opportunity.

3. **Alarms/Errors on virtual machines, ESX hosts and vCenter servers.** We monitor kernel-level error events on ESX hosts and warning/error events on vCenter servers. We also monitor alarm conditions on virtual machines. An example of a kernel-level ESX host error event is loss of connectivity to shared storage. Alarm conditions on virtual machines include high CPU utilization and high disk utilization. For the alarms that are already built into the vCenter server (for example, high virtual machine CPU usage), we leverage vCenter's alarm mechanism, while for others (like high disk usage within a virtual machine), we utilize agents with the virtual machines to monitor and proactively alert Socialcast appropriately.

There were several areas in which our approach had notable advantages over conventional approaches. For example, while the resource utilization of the various management components is available via the API, it can be complex to retrieve this data across multiple installations. By collecting such data and putting it in a single pane of glass, we provided an at-a-glance view of the health of the infrastructure.

By dissecting the operational workload of the datacenter, we were able to determine some possible areas of optimization for our management stack. For example, certain operations require reconfiguration of virtual machines multiple times before the virtual machine is ultimately deployed. By tracking operational metrics and posting them for a group of management components, we were able to see the severity of this problem immediately.

By maintaining relationships between virtual machines and their hosts in an intuitive way, we were able to reason about certain operations more easily. For example, we had deployed our monitoring virtual machines across the infrastructure. At some point, we wanted to move some virtual machines from one host to another. We had created a special group consisting of just our monitoring virtual machines, and by listing the members of this group, we could easily find all of our monitoring virtual machines, and then by 'mousing over' those virtual machines, we could find the hosts on which they resided. Note that this is possible in a standard virtualized infrastructure, but the social network metaphor provides a natural way to perform such a search.

A final example illustrates an unintentional synergy between social media metaphors and virtualization management. One of the administrators wanted to change a cluster to allow automated virtual machine migrations and wanted to see how many migrations would result. Because we had tagged each migration with a hashtag[10] (#Success_drm_executevmotionlro in this case), we were instantly able to search for the number of instances of that hash tag within a given cluster and find out how many migrations resulted from changing the setting in two different clusters. This case is shown in Figure 9.

Figure 9: Synergy between social media and virtualization management with hashtags. By tagging successful tasks with a hashtag (#Success_drm_executevmotionlro), we were able to instantly determine the number of such tasks performed by 2 different vCenter servers (las-cg39, 909 times, and las-cg41, 534 times), without adding any customized aggregation code.

5. Related Work

Many corporations have used the Socialcast developer API to create real-time communication and collaboration tool. One example is an integration of Socialcast and Microsoft Sharepoint[7], in which a Socialcast community can be embedded into a SharePoint site. Communication within the site can be viewed outside of SharePoint, and newcomers to the group can view the archives of previous conversations. In this paper, we have extended the notion of a community to include not just humans but also entities like virtual machines and hosts. Virtual machines and hosts can use automated monitoring in order to 'communicate' with each other and with humans. We have also used the metaphors of social media to assist in virtual management. While adding non-humans to social networks is not a new idea [3], tying together these social media metaphors to virtualization management is novel.

As indicated in section 2, in some sense, virtualization management is already a form of an online community. Virtual machines might generate alarms that can be viewed by vCenter, and vCenter can email administrators in turn with the alarm information. What is missing is an easy-to-use aggregation system based on arbitrary tags. VMware vSphere already contains tagging capabilities, so an administrator can aggregate virtual machines and perhaps utilize analytics tools like vCenter Operations Manager™ [13] for generating alerts based on arbitrary aggregations.

Compared to standard virtualization management tools, our monitoring is more flexible: the monitoring can easily be customized for a particular type of virtual machine. Rather than trying to define a custom alarm for each type of application, a virtual machine owner can simply collect various application-level metrics within the virtual machine and then update status as required: an Exchange virtual machine owner might select messages processed per second; the owner of a virtual machine that does compilation jobs might choose to collect the build times and trigger a status change if the build times suddenly get worse; the owner of a virtual machine that is acting as an NFS datastore might trigger a notice if disk space within the virtual machine gets low. In each case, a simple shell-based script and GET/POST requests are all that are required to provide powerful notification capabilities.

6. Conclusions and Future Work

In this paper, we propose a social-media approach to monitoring virtualized environments. We draw an analogy between a social network and the network created by virtual machines, hosts, and vCenter servers. In a social network, users can follow each other's updates, send each other messages, and create closed groups within the social network for selective communication; in a similar way, we propose taking a virtualized environment and creating a 'community' that includes the various entities of a virtualized environment along with the system administrator. Each entity (whether human or not) is capable of using a simple API to communicate status updates. By judicious creation of hierarchies (for example, having administrators 'follow' vCenter servers, vCenter servers follow hosts, and hosts follow virtual machines) and by using information reduction techniques (for example, 'liking' various types of messages instead of posting the same messages repeatedly), we can avoid excess information flow to the administrator, while still allowing the administrator to view important status updates in the environment. Moreover, the simple API also allows administrators the ability to send commands to any entity, providing a technique for platform-independent remote management via any mobile device.

The approach described in this paper might be quite disruptive for an existing environment. For the user that is reluctant to turn an entire inventory into a social network, there are intermediate steps to validate the approach. The first step is to simply incorporate the

data streams from vCenter into a feed that is posted to a Socialcast site. This is the traditional use of a collaboration tool like Socialcast: administrators use a central repository for data storage, data sharing, and communication, and incorporate external data sources. The next step might be to add hosts but not virtual machines to the social network. The final stage would be to fully embrace the social networking model by adding all simple Socialcast clients to each virtual machine and host and allow virtual machines and hosts to follow and be followed.

There are many avenues for future work. First and foremost, we have learned that an integrated view of the relationships in a virtualization hierarchy is important, so we intend to continue adding more and more entities to the social network. For example, we can add virtual and physical networks, network switches, and intelligent storage devices[17]. We can also add more application awareness and associate virtual machines with each other based on whether they communicate with each other. We can also refine our algorithms for associating users with virtual machines and applications to make that process more automated. As we increase the number of entities and possible churn, we must continue to tune the performance of our service virtual machines and the Socialcast VM.

Another promising avenue for future work is integration with other data sources like vCenter Operations Manager or Zenoss[20]. Assuming modifications to the Operations Manager server to provide notifications on anomalies, the administrator can follow the Operations Manager and be apprised of anomalies or alarms. We can also envision even richer use of the data from this social network of virtualized entities. For example, the various community streams can be uploaded to off-line analytics engines to provide interesting statistics on a given environment, like the most commonly misbehaving virtual machines or hosts or the most common virtual machine error messages, or even the most common scripts that are run on a host. Socialcast itself has some basic analytics which are extremely valuable: we can perhaps imagine extending the architecture to allow plugins that provide virtualization-specific analytics.

While the use cases presented in this paper have focused on monitoring, we can also envision allowing simple commands to be sent via this interface. One simple example, as mentioned earlier, might be allowing a user to send private messages to an individual virtual machine to reboot or provide diagnostic information, although this would require strict security controls (limiting access to administrators or virtual machine owners, for example), or possibly enriching the interface to allow right-click actions on a given entity. This might require leveraging existing access control systems and coupling them to Socialcast's mechanisms for creating users and assigning permissions. Finally, we can also potentially embed the administrator's social media web page directly into the vSphere web-based client, providing a complete one-stop shop for virtualization management and monitoring, combining standard paradigms with an intuitive yet novel social media twist.

Acknowledgments

We thank Rajat Goel for providing an on-premise Socialcast virtual appliance that we could deploy for testing purposes. We thank Sean Cashin for numerous hints for using the Socialcast API effectively. We thank Steve Herrod and the office of the CTO for helpful comments on our work. Finally, we thank Conrad Albrecht-Buehler, for extremely helpful comments on our paper.

References

1 Facebook, www.facebook.com

2 Haxx. 'libcurl: the multiprotocol file transfer library,' http://curl.haxx.se/libcurl/

3 Holland, S.W. Social Networking with Autonomous Agents. United States Patent US 2012/0066301, http://www.google.com/patents/US20120066301

4 Linux iostat utility, http://www.unix.com/apropos-man/all/0/iostat/

5 Linux vmstat utility, http://nixdoc.net/man-pages/Linux/man8/vmstat.8.html

6 Microsoft. Perfmon, http://technet.microsoft.com/en-us/library-bb490957.aspx

7 Socialcast. Making Sharepoint Social: Integrating Socialcast and SharePoint Using Reach and API, http://blog.socialcast.com/making-sharepoint-social-integrating-socialcast-and-sharepoint-using-reach-and-api

8 Socialcast. Socialcast Developer API, http://www.socialcast.com/resources/api.html

9 SPEC. SpecJBB2005, http://www.spec.org/jbb2005/

10 Twitter. "What are hashtags?" https://support.twitter.com/articles/49309-what-are-hashtags-symbols#

11 VMware. VMware API Reference Documentation, https://www.vmware.com/support/pubs/sdk_pubs.html

12 VMware. VMware vCenter Application Discovery Manager, http://www.vmware.com/products/application-discovery-manager/overview.html

13 VMware. VMware vCenter Operations Manager, http://www.vmware.com/products/datacenter-virtualization/vcenter-operations-management/overview.html

14 VMware. VMware vCloud Director, http://www.vmware.com/products/vcloud-director/overview.html

15 VMware. VMware vSphere, http://www.vmware.com/products/datacenter-virtualization/vsphere/overview.html

16 VMware. VMware View, http://www.vmware.com/products/view/overview.html

17 VMware. VMware vStorage APIs for Array Integration, http://communities.vmware.com/docs/DOC-14090

18 VMware. VMworld 2012, http://www.vmworld.com/community/conference/us

19 VMware. VMworld Europe 2012, http://www.vmworld.com/community/conference/europe/

20 Zenoss. Zenoss Virtualization Monitoring, http://www.zenoss.com/solution/virtualization-monitoring

21 Zimman, A., Roberts, C., and Van Der Welt, Mornay. VMworld 2011 Hands-on Labs: Implementation and Workflow. VMware Technical Journal, Vol 1. No. 1. April 2012. pp. 70-80.

VMware View® Planner: Measuring True Virtual Desktop Experience at Scale

Banit Agrawal
banit@vmware.com

Rishi Bidarkar
rishi@vmware.com

Sunil Satnur
ssatnur@vmware.com

Tariq Magdon-Ismail
tariq@vmware.com

Lawrence Spracklen
lspracklen@vmware.com

Uday Kurkure
ukurkure@vmware.com

Vikram Makhija
vmakhija@vmware.com

VMware, Inc.

Abstract

In fast-changing desktop environments, we are witnessing increasing use of Virtual Desktop Infrastructure (VDI) deployments due to better manageability and security. A smooth transition from physical to VDI desktops requires significant investment in the design and maintenance of hardware and software layers for adequate user base support. To understand and precisely characterize both the software and hardware layers, we present VMware View® Planner, a tool that can generate workloads that are representative of many user-initiated operations in VDI environments.

Such operations typically fall into two categories: *user* and *admin* operations. Typical user operations include typing words, launching applications, browsing web pages and PDF documents, checking email and so on. Typical admin operations include cloning and provisioning virtual machines, powering servers on and off, and so on. View Planner supports both types of operations and can be configured to allow VDI evaluators to more accurately represent their particular environment. This paper describes the challenges in building such a workload generator and the platform around it, as well as the View Planner architecture and use cases. Finally, we explain how we used View Planner to perform platform characterization and consolidation studies, and find potential performance optimizations.

I. Introduction

As Virtual Desktop Infrastructure (VDI) [6,7,8] continues to drive toward more cost-effective, secure, and manageable solutions for desktop computing, it introduces many new challenges. One challenge is to provide a local desktop experience to end users while connecting to a remote virtual machine running inside a data center. A critical factor in remote environments is the remote user experience, which is predominantly driven by the underlying hardware infrastructure and remote display protocol. As a result, it is critical to optimize the remote virtualized environment [13] to realize a better user experience. Doing so helps IT administrators to confidently and transparently consolidate and virtualize their existing physical desktops.

Detailed performance evaluations and studies of the underlying hardware infrastructure are needed to characterize the end-user experience. Very limited subjective studies and surveys on small data sets are available that analyze these factors [19,20,21], and results and techniques do not measure up to the scale of the requirements in VDI environments. For example, the required number of desktop users easily ranges from a few hundred to tens of thousands, depending on the type of deployment. Accordingly, there is a pressing need for an automated mechanism to characterize and plan huge installations of desktop virtual machines. This process should qualitatively and quantitatively measure how user experience varies with scale in VDI environments. These critical measurements enable administrators to make decisions about how to deploy for maximum return on investment (ROI) without compromising quality.

This paper presents an automated measurement tool that includes a typical office user workload, a robust technique to accurately measure end-to-end latency, and a virtual appliance to configure and manage the workload run at scale. Ideally, the workload needs to incorporate the many traditional applications that typical desktop users use in the office environment. These applications include Microsoft Office applications (Outlook, PowerPoint, Excel, Word), Adobe Reader, Windows Media player, Internet Explorer, and so on. The challenge lies in simulating the operations of these applications so they can run at scale in an automated and robust manner. Additionally, the workload should be easily extensible to allow for new applications, and be configurable to apply the representative load for a typical end user. This paper discusses these challenges and illustrates how we solved them to build a robust and automated workload, as described in Section II.

The second key component to a VDI measurement framework is precisely quantifying the end-user experience. In a remote connected session, an end-user only receives display updates when a particular workload operation completes. Hence, we need to leverage the display information on the client side to accurately measure the response time of a particular operation. Section III presents how a watermarking technique can be used in a novel way to measure remote display latency. The watermarking approach was designed to:

- Present a new encoding approach that works even under adverse network conditions
- Present smart watermarking placement to ensure the watermark does not interfere with application updates
- Make the location adaptive to work with larger screen resolutions

The final piece is the automated framework that runs the VDI user simulation at scale. We built a virtual appliance with a simple and easy-to-use web interface. Using this interface, users can specify the hardware configuration, such as storage and server infrastructure, configure the workload, and execute the workload at scale. The framework, called VMware View Planner, is designed to simulate a large-scale deployment of virtualized desktop systems and study the effects on an entire virtualized infrastructure (Section IV). The tool scales from a few virtual machines to thousands of virtual machines distributed across a cluster of hosts. With View Planner, VDI administrators can perform scalability studies that closely resemble real-world VDI deployments and gather useful information about configuration settings, hardware requirements, and consolidation ratios. Figure 1 identifies the inputs, outputs, and use cases of View Planner.

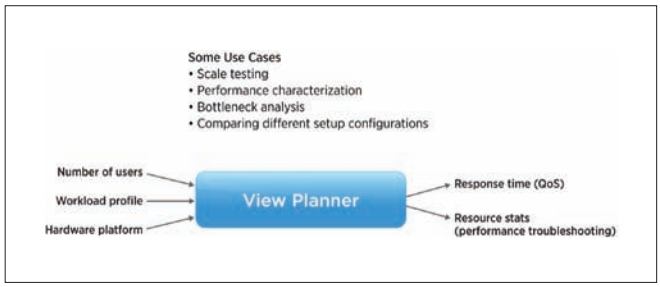

Figure 1. View Planner: Inputs, Outputs, and Use cases

Using the View Planner tool, we enable the following use cases in the VDI environment:

- **Workload generation.** By configuring View Planner to simulate the desired number of users and configure applications, we can accurately represent the load presented in a given VDI deployment. Once the workload is running, resource usage can be measured at the servers, network, and storage infrastructure to determine if bottlenecks exist. Varying the load enables required sizing information to be obtained for a particular VDI architecture (Section V).
- **Architectural comparisons.** To determine the impact of a particular component of VDI architecture, we can configure a fixed load using View Planner and measure the latencies of administrative operations (provisioning, powering on virtual machines, and so on) and user operations (steady-state workload execution). Switching the component in question and measuring latencies again provides a comparison of the different options. A note of caution here is that both administrative and user operation latencies can vary significantly based on the underlying hardware architecture, as described in sections V.C and V.E.
- **Scalability testing.** In this use case, hardware is not varied. The system load is increased gradually until a resource of interest is exhausted. Typical resources measured include CPU, memory, and storage bandwidth. Example use cases are presented in section V.B.

There are many other use cases, such as remote display protocol performance characterization (section V.D), product features performance evaluation, and identification of performance bottlenecks. Section VI provides relevant and related work in VDI benchmarking.

2. Workload Design

Designing a workload to represent a typical VDI user is a challenging task. Not only is it necessary to capture a representative set of applications, it also is important to keep the following design goals in mind when developing the workload.

Scalability. The workload should run on thousands of virtual desktops simultaneously. When run at such scale, operations that take a few milliseconds on a physical desktop could slow down to several seconds on hardware that is configured in a suboptimal manner. As a result, it is important to build high tolerances for operations that are sensitive to load.

Robustness. When running thousands of virtual desktops, operations can fail. These might be transient or irreversible errors. Operations that are transient and idempotent can be retried, and experience shows they usually succeed. An example of a transient error is the failure of a PDF document to open. The open operation can be retried if it fails initially. For operations that are not idempotent and cannot be reversed, the application simply is excluded from the run from that point onwards. This has the negative effect of altering the intended workload pattern. After extensive experimentation, this tack was decided upon so the workload could complete and upload the results for other operations that complete successfully. Our experience shows that only a few operations fail when run at scale, and the overall results are not altered appreciably.

Extensibility. Understanding that one set of applications is not representative of all virtual desktop deployments, we chose a very representative set of applications in our workload and enable View Planner users to extend the workload with additional applications.

Configurability. Users should be able to control the workload using various configurations, such as the application mix, the load level to apply (light, medium, heavy), and the number of iterations.

We overcame many challenges during the process of ensuring our workload was scalable, robust, extensible, and configurable. To realize these goals, we needed to automate various tasks in a robust manner.

Command-line-based automation. If the application supports command-line operations, it is very easy to automate by simply invoking the commands from the MS-DOS command shell or Microsoft Windows PowerShell.

GUI-based automation. This involves interacting with the application just like a real user, such as clicking window controls, typing text into boxes, following a series of interactive steps (wizard interaction), and so on. To do this, the automation script must be able to recognize and interact with the various controls on the screen, either by having a direct reference to those controls (Click “Button1”, Type into “Textbox2”) or by knowing their screen coordinates (Click <100,200>). The user interfaces of Microsoft Windows applications are written using a variety of GUI frameworks. Windows applications written by Microsoft extensively use the Win32 API to implement windows, buttons, text boxes, and other GUI elements. Applications written by third-party vendors often use alternative frameworks. Popular examples include the Java-based

SWT used by the Eclipse IDE, or the ActionScript-based Adobe Flash. Automating applications with a Win32 API-based GUI is relatively straightforward with the AutoIT scripting language [12]. Automating applications that use alternative frameworks for the GUI is not straightforward and requires other tools.

API based automation. This involves interacting with the application by invoking its APIs to perform specific actions. Microsoft's COM API is a good example of this model. All Microsoft Office applications export a COM interface. Using the COM API it is possible to do almost everything that a user can do using the GUI. API-based automation is chosen over GUI-based automation when the GUI elements are very complicated and cannot be accessed directly. For example, it is very difficult in Microsoft Outlook to click on an individual mail item using direct GUI controls, let alone obtain information about the mail item, such as the identity of the sender. On the other hand, the Microsoft Outlook COM API provides a rich interface that lets you locate and open a mail item, retrieve information about the sender, receiver, attachments, and more.

The next sections provide a description of the workload composition, discuss how to avoid the workload starting at the same time, and illustrate how to perform timing measurements.

A. Workload Composition

Instead of building a monolithic workload that executes tasks in a fixed sequence, we took a building-block approach, composing each application of its constituent operations. For example, for the Microsoft Word application we identified Open, Modify, Save, and Close as the operations. This approach gave us great flexibility in sequencing the workload in any way desired. The applications and their operations are listed in Table 1.

APPLICATION ID	APPLICATION	OPERATIONS
1	Firefox	["OPEN", "CLOSE"]
2	Excel	["OPEN", "COMPUTE", "SAVE", "CLOSE", "MINIMIZE", "MAXIMIZE", "ENTRY"]
3	Word	["OPEN", "MODIFY", "SAVE", "CLOSE", "MINIMIZE", "MAXIMIZE"]
4	AdobeReader	["OPEN", "BROWSE", "CLOSE", "MINIMIZE", "MAXIMIZE"]
5	IE_ApacheDoc	["OPEN", "BROWSE", "CLOSE"]
6	Powerpoint	["OPEN", "RUNSLIDESHOW", "MODIFYSLIDES", "APPENDSLIDES", "SAVEAS", "CLOSE", "MINIMIZE", "MAXIMIZE"]
7	Outlook	["OPEN", "READ", "RESTORE", "CLOSE", "MINIMIZE", "MAXIMIZE", "ATTACHMENT-SAVE"]
8	7zip	["COMPRESS"]
10	Video	["OPEN", "PLAY", "CLOSE"]
12	Webalbum	["OPEN", "BROWSE", "CLOSE"]

Table 1: Applications with their IDs and operations

B. Avoiding Synchronized Swimming

Operations in a desktop typically happen at discrete intervals of time, often in bursts that consume many CPU cycles and memory. We do not want all desktops to execute the same sequence of operations for two reasons. It is not representative of a typical VDI deployment, and it causes resource over commitment. To avoid synchronized swimming among desktops, the execution sequence in each desktop is randomized so the desktops perform different things at any given instant of time and the load is distributed evenly distributed (Figure 2).

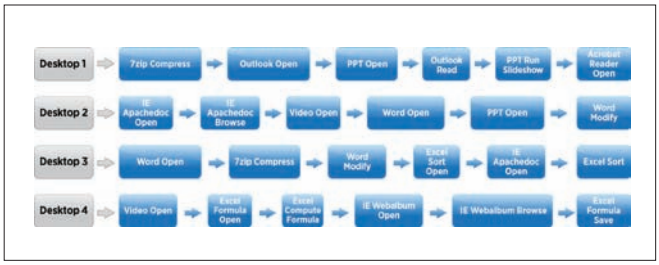

Figure 2. Shows the randomized execution of different operations across different desktop virtual machines. While the figure is not drawn to scale, the box width shows the relative timing of different operations.

C. Timing

The time taken to perform desktop operations ranges from a few milliseconds to tens of seconds. A high resolution timer is needed for operations that complete very quickly. The **Query Performance Counter (QPC)** call in Microsoft Windows cannot be relied upon because it uses the virtualized timestamp counter, which may not be accurate [3]. Consequently, the timestamp counter must be fetched from the performance counter registers on the host CPU. The virtual machine monitor in VMware software enables a virtual machine to issue an **rdpmc()** call into the host machine to fetch hardware performance counters. To measure the latency of an operation, we simply wrap the operation with these **rdpmc()** calls to obtain a much more reliable measurement. Since the **rdpmc()** call is intercepted by the hypervisor, translated, and issued to the host, it can take more cycles than desired. Our measurements indicate this call consumes approximately 150,000 cycles on a 3 GHz processor, or approximately 50 microseconds. The operations measured take at least 50 milliseconds to complete, which means the overhead of using the **rdpmc()** call is less than 0.1 percent.

D. Extending the Workload with Custom Applications

As mentioned earlier, a fixed set of applications is not representative of possible virtual desktop deployments. View Planner is designed to be extensible, enabling users to plug in their applications. Users must follow the same paradigm of identifying operations constituting their application. Since the base AutoIT workload included with View Planner is compiled into an executable, users cannot plug in their code into the main workload. To help this issue, we implemented a custom application framework that uses TCP sockets to communicate between the main workload and the custom application script. Using this feature, users can add

customized applications into their workload mix and identify the applications suite that best fits their VDI deployments. For example, a healthcare company can implement a health related application and mix it with typical VDI user workloads and characterize their platform for VDI deployments.

E. Workload Scalability Enhancements

Virtualized environments make effective use of hardware by allowing multiple operating system instances to run simultaneously on a single computer. This greatly improves utilization and enables economies of scale. While there are innumerable benefits to virtualization, poorly designed virtual environments can cause unpredictability in the way applications behave, primarily due to resource over commitment. The goal of the View Planner workload is to reliably detect and report poor designs in virtual desktop environments. Because the View Planner workload is technically another application running inside virtual desktops, it is susceptible to the same unpredictability and failures under load. To make the process of timing measurement and reporting reliable, we built mechanisms into the workload that ensure the workload runs to completion even under the most stressful conditions.

1) Idempotent Operations and Retries

To make the workload more manageable, we split the operations of each application into the smallest unit possible. We also designed most of these operations to be idempotent, so that failed operations can be retried without disturbing the flow of operations. Our experience indicates many operations fail due to transient load errors and many typically succeed if tried again. As a result, the software retries the operation (just as a normal user) three times before declaring a failure. While the retry mechanism has significantly improved the success rate of individual operations passing under high load, some operations might still fail.

Two options are available when an operation fails. The first option is to fail the workload because one of its constituent operations failed after three retries. Another option is to continue with the workload by ignoring further operations of the application that encountered a failure. We decided to leverage the second approach for two reasons:

- Our workload is composed of many applications and an even greater number of operations. Failing the entire workload for one or two failed applications discards all successful measurements and results in wasted time.
- Since our workload runs on multiple virtual desktops simultaneously, failures in a few desktops do not have a significant impact on the final result if we consider the successful operations of those desktops.

By selectively pruning failed applications from a few virtual machines, we are able to handle failures at a granular level and still count the successful measurements in the final result, resulting in improved robustness and less wasted time for users. We also flag the desktops that failed the run.

2) Progress Checker (View Planner Watchdog)

In situations where extreme reliability and robustness are needed, a watchdog mechanism is needed to ensure things progress smoothly. We use this concept by employing a progress checker process, a very simple user-level process with an extremely low chance of failure. A progress file, a simple text file, keeps track of the workload progress by storing the number of operations completed. The progress checker studies workload progress by reading the progress file.

When the workload starts, it keeps an operations count in a known Microsoft Windows registry location and tries to launch the progress checker process. The workload fails to run if it cannot launch the progress checker. When the workload starts performing regular operations, it increments the count stored in the registry. The progress checker process periodically wakes up and reads the registry. It terminates the workload if progress is not detected. The progress checker sleeps for three times the expected time taken by the longest running operation in the workload. This ensures the workload is not terminated accidentally. Finally, if the progress checker needs to terminate the workload, it does so and reports the timing measurements completed so far so they can be included in the final score.

3. End-User Measurement Framework

The second part of the View Planner framework is the precise measurement of the user experience from the client side. This section describes the novel measurement technique used to measure application response time from the client side. It presents the VDI watermarking approach as well as a brief description of our plugin (client agent) implementation.

A. View Planner Workload Watermarking

In our previous approaches [1, 2], the idea was to use the virtual channel to signal the start and end of events through the display watermarking on the screen. In these techniques, the display watermarking location overlapped with applications, resulting in a chance for the watermarking update to be overlapped by application updates. This results in the particular event being missed and the workload not progressing as expected. We needed a mechanism in which the watermarking location is disjoint to application rendering. To enable better decisions on the client side, we also required metadata encoded in the watermark. This allows us to identify the operation on the client side, as well as drive the workload from the client side to realize true VDI user simulation.

We designed our watermarking approach such that it:

- Uses a new encoding approach that works even under adverse network conditions
- Uses smart watermarking placement to ensure the watermark does not interfere with application updates
- Adapts to larger screen resolutions

To precisely determine application response time on the application side, we overlay metadata on top of the start menu button that travels with the desktop display. This metadata can be any type useful information, such as the application operation type and number of events executed. Using this metadata information timing information for the application operation can be derived on the client side. We detect the metadata and record the timestamps at the start and end of the operation. The difference between these timestamps enables the estimation of application response time. There are many challenges associated with getting accurate metadata to the client side, making it unobtrusive to the application display, and ensuring it always is visible. As a result, the location of metadata display and the codec used to encode metadata is very important.

Table 1 illustrates an example of the encoding of different application operations. The table shows the applications (Firefox, Microsoft Office applications, Adobe Reader, Web Album, and so on) that are supported with View Planner along with their operations. Each application is assigned an application ID and has a set of operations. Using this approach, we can encode a PowerPoint "Open" operation in pixel values by doing the following:

- Note that the Microsoft PowerPoint application has application ID "6"
- See that "Open" is the first operation in the list of operations
- Calculate the encoding of each operation using the following formula: (application_id * NUM_SUBOPS) + (operation_id)
- Determine the encoding of the Microsoft PowerPoint "Open" operation is (6 * 10) + 1 = 61
- The value 10 is used for NUM_SUBOPS since assume the number of operations for a particular application will not exceed 10.

After looking at one encoding example, let's see how we send this encoded data to the client side and how robustly we can infer the code from the client side. As shown in Figure 3, we display the metadata on top of the start menu button since it does not interfere with the rendering of other applications. The watermark is composed of three lines composed of white and black pixel colors, each 20x1 pixels wide. The first line is used to denote the test ID, the event code for the current running operation. The next two markers signal the start and end of a particular operation. Using the example shown at bottom of Figure 3, we can explore how these three lines are used to monitor response time. When the workload executes the Microsoft PowerPoint "Open" operation, the workload watermarks the test ID location with the event code 61, as calculated earlier. The workload puts the sequence number for the number of operations "n" that have occurred (1 in this case) in the start location. The protocol sends the watermark codes to the client when a screen change occurs. The encoded watermarks are decoded as a Microsoft PowerPoint "Open" operation on the client side. When the client observes the start rectangle update, the "start" timestamp is recorded. When the "Open" operation is complete, the workload watermarks the test ID location again with the Microsoft PowerPoint "Open" event code (61) and a code (1000-n) in the end location.

When the client observes the end rectangle update with the sum of the start ID and end ID equaling 1000, the "end" timestamp is recorded. The difference between the timestamps is recorded as the response time of "Open" operation.

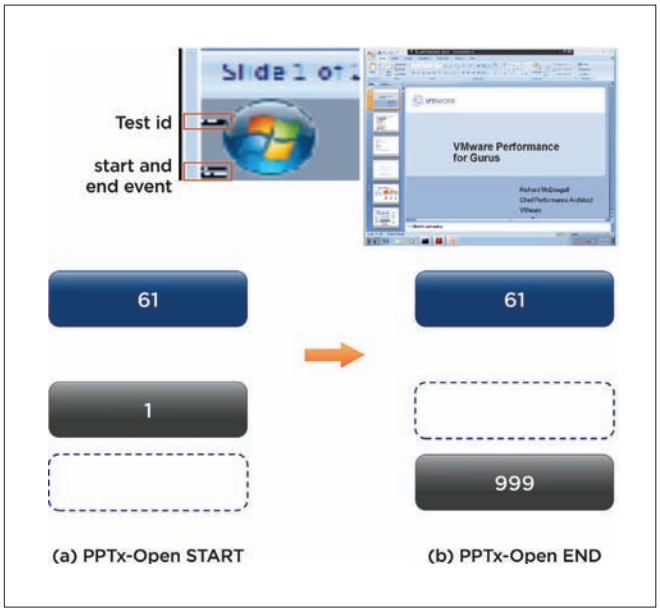

Figure 3. This figure shows the VDI desktop with the Microsoft PowerPoint application running in the background. The watermarking region is near the start menu and is shown on the left. The watermarking codes below show the PowerPoint Open operation. Figure 3a shows the start of the operation and Figure 3b shows the end of the operation.

B. Measurement Plugin Implementation

This section describes the internal details of the measurement plugin. There are common requirements, such as getting rectangle updates, finding pixel color values, sending key events, and using the timer for timestamps. With these four APIs, we can extend the measurement technique to any client device, such as Apple iPads or Android-based tablets. Any mirror driver can be used to get rectangle updates. The function of a mirror driver is to provide access to the display memory and screen updates as they happen. In our implementation, we use the SVGA DevTap interface that we built and implemented as part of the SVGA driver. The software performs approximately 40 scans per second (25 ms granularity) to process incoming display updates to look for encoded watermark events.

On the client side, the plugin runs a state machine. It changes state from sending an event for the next operation, waiting for the start event, waiting for the end event, and finally waiting for the think time. In the "sending event" state, the plugin sends a key event to signal the desktop to start the next operation. The plugin records the time when it sees the end of the event. It continues to iterate through different states of the state machine until the workload "finish" event is sent from the desktop. During video play operation, the main plugin switches to the video plugin [2] and records frame timings. After the video playback is complete, it switches back to the main measurement plugin to measure the response time of other applications. For timing, the host RDTSC is used to read the timestamp counters and divide by the processor frequency to determine the elapsed time.

4. View Planner Architecture

This section provides an overview of the third piece of VDI simulation framework—the View Planner framework to simulate VDI workloads at scale—and discuss its design and architecture. To run and manage workloads at large scale, we designed an automated controller. The central piece in the View Planner architecture, the automated controller is the harness or appliance virtual machine that controls everything, from the management of participating desktop and client virtual machines, to starting the workload and collecting results, to providing a monitoring interface through a web user interface.

The View Planner appliance is essentially a CentOS Linux-based appliance virtual machine that interacts with many VDI server components. It also runs a web server to present a user-friendly web interface. Figure 4 shows the high-level architecture of View Planner. As shown in the diagram, the appliance interacts with a VMware vCenter View connection server or Virtual Center server to control desktop virtual machines. It also communicates with client virtual machines to initiate remote protocol connections. The appliance is responsible for starting the workload simulation in desktop virtual machines. Upon completion, results are uploaded and stored in a database inside the appliance. Results can be viewed using the web interface or extracted from the database at any time.

The harness controller provides the necessary control logic. It runs as a Linux user-level service in the appliance and interacts with many external components. The control logic implements all needed functionality, such as:

- Keeping state and statistics
- Controlling the run and configurations
- Providing monitoring capabilities
- Interacting with the database and virtual machines
- Collecting and parsing results and reporting scores

The View Planner tool uses a robust and asynchronous remote procedure call (RPC) framework (Python Twisted) to communicate with desktop or client virtual machines. Testing shows successful connection handling for up to 4,000 virtual machines.

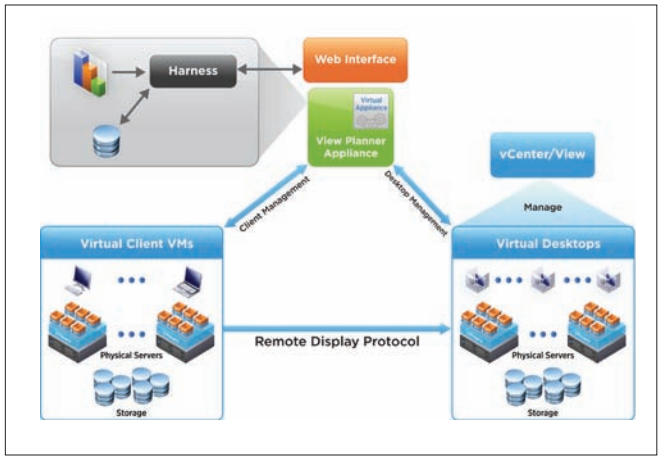

Figure 4. View Planner high-level architecture

A. View Planer Operations

This section discusses the View Planner flow chart and how View Planner operates the full run cycle. Once the harness is powered on, the service listens on a TCP port to serve requests from the web interface. Figure 5 shows the operation flow chart for View Planner. In the first phase, View Planner stores all server information and their credentials. Next, desktop virtual machines can be provisioned (an administrative operation) using the web interface. Following this step, the View Planner user defines the workload profile (applications to run) and the run profile (the number of users to simulate), and so on. Next, the run profile is executed. After the run completes, results are uploaded to the database and can be analyzed. This process can be repeated with a new workload and run profile.

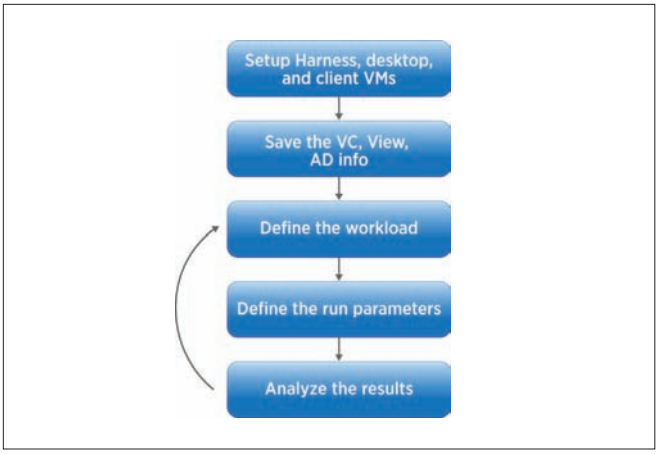

Figure 5. View Planner operational flow chart

Let's discuss in detail what happens in the background, starting when a VDI evaluator executes a particular run profile until the final results are uploaded. This is illustrated in Figure 6 for three different modes of View Planner:

- In "local" mode, the workload executes locally without clients connected.
- In "remote" mode, workload execution and measurement are performed with a remote client connected to one desktop (one-to-one).
- In "passive" mode, one client can connect to multiple desktops (one-to-many).

In these modes, View Planner first resets the test for the previous run, and powers off virtual machines. At this stage, the harness is initialized and ready to execute the new run profile. Next, a prefix match stage finds participating virtual machines based on the desktop or client prefix provided in the run profile. View Planner powers on these participating virtual machines at a staggered rate that is controlled by a configurable parameter. VDI administrators needing to investigate bootstorm issues can increase this value to a maximum, causing View Planner to issue as many power on operations as possible every minute.

Once the desktops are powered on, they register their IP addresses. Upon meeting a particular threshold, View Planner declares a certain number of desktop or client virtual machines are available for the run. At this stage, View Planner waits for the ramp up time for all virtual machines to settle. Next, it obtains the IP address for each desktop and client virtual machines and uses these IP addresses to initiate the run.

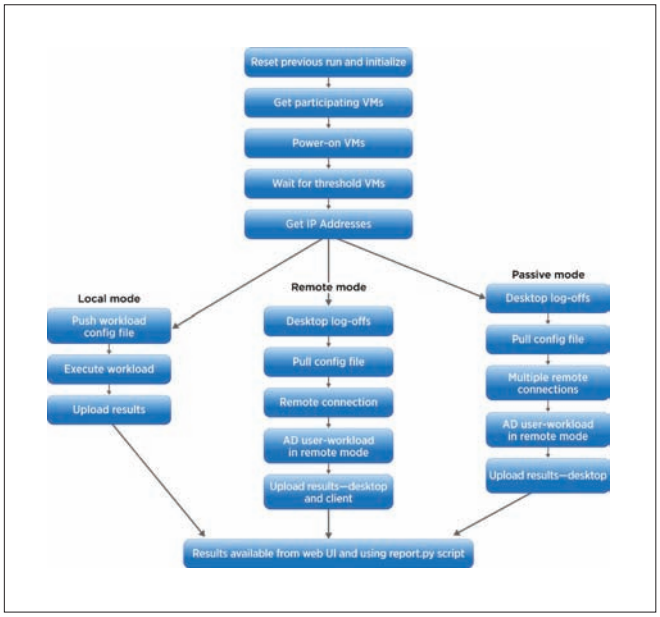

Figure 6. View Planner operations for different modes

In remote and passive mode, View Planner executes an extra phase in which it logs off desktop virtual machines (Figure 6). After the logoff storm, View Planner waits for the virtual machines to settle and CPU usage to return to normal. After this stage, the appliance sends a command to the desktops to execute the workload in local mode. For remote and passive modes, commands are sent to clients to execute logon scripts. This is logon storm is similar to what happens when employees arrive at the office in the morning and log into their desktops. The login storm is an "admin" operation and can be controlled by a configurable parameter. Once the connections are established, they update their status to the harness and View Planner records how many workload runs have started. After the run completes, the desktops upload the results. For remote mode, View Planner finds the matching clients and asks the clients to upload the results. These results are inserted into the database. The View Planner passive mode is good to use when VDI evaluators do not have sufficient hardware to host client virtual machines.

B. Handling Rogue Desktop and Client Virtual Machines

When simulating a large user run, there may be situations in which a few desktops are stuck in the customization state or are unable to obtain an IP address. In this case, a timer runs all the time. After every registration, reset the timer is reset and the software waits for 30 minutes. If more registrations are not seen after 30 minutes, and the required threshold is not met, we kick off the logic to find the bad virtual machines and reset them. After obtaining the IP address of each registered desktop, we use the virtual machine name to IP address mapping to find the rogue virtual machines. Once the rogue virtual machines are reset, they register their IP addresses and, on meeting the threshold, the run starts If the threshold is still not met, the timer is reset a few times and the run is started with the registered number of virtual machines.

C. Scoring Methodology

An invocation of the View Planner workload in a single virtual machine provides hundreds of latency events. As a result, when scaling to thousands of desktops virtual machines using View Planner, the number of latency measurements for different operations grows very large. Hence, a robust and efficient way to understand and analyze the different operational times collected from the numerous desktop virtual machines is required. To better analyze these operations, we divided the important operations into two buckets. Group A consists of interactive operations, Group B consists of I/O intensive operations. Group A and Group B operations are used to define the *quality of service* (QoS), while the remaining operations are used to generate additional load. For a benchmark run to be valid, both Group A and Group B need to meet their QoS requirements. QoS is defined in terms of a threshold time limit, T_h. For each group, 95 percent of the operations must complete within T_h. Limits are set based on extensive experimental results spanning numerous hardware and software configurations. The View Planner benchmark "score" is the number of users (virtual machines) used in a run that pass QoS.

5. Results And Case Studies

This section presents workload characterization results, describes several View Planner use cases, and presents associated results.

For most of the experiments presented in subsequent sections, all of the applications supported in View Planner ran with 20 seconds of think time. Think time is used to simulate the random pause when a VDI user is browsing a page or performing another task. For the 95 percent Group A threshold (T_h), we selected a response time of 1.5 seconds based on user response time and statistical analysis.

A. Workload Characterization

We first characterized the workload based on how many operations View Planner executed and how randomly the operations were distributed across different iterations in different desktop virtual machines.

Event Counts

Figure 7 shows the average number of times each operation is executed per virtual machine. The *-Open and *-Close operations are singleton operations and occur at low frequency, while interactive operations (AdobeReader-Browse, IE_ApacheDoc-Browse, Word-Modify, and so on) typically are executed more than 10 times. This is very close to real-world user behavior: the document is opened and closed once, with many browse and edit operations performed while the document is open.

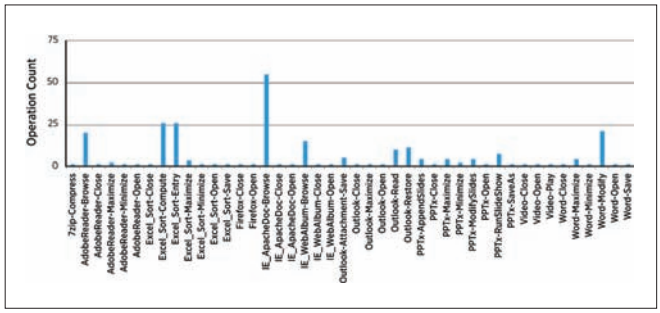

Figure 7. Average event count for each application operation

Response Time

As discussed in Section III.A, using the proposed watermarking technique the response time of applications from the client side can be measured. The response time chart in Figure 8 shows the application latency seen for different operations with the RDP protocol in LAN conditions. As seen from the graph, most of the Open and Save operations take more than two seconds, as they are CPU and I/O intensive operations, while most interactive operations, such as Browse and maximize operations, take less than a second.

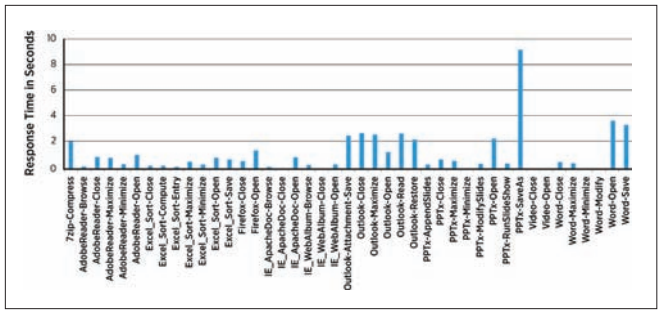

Figure 8. Average response time for each application operation measured from the client side

Iteration Overlap

Figure 9 shows virtual machine execution of operations over each of the seven iterations. The y-axis corresponds to the ID of a particular virtual machine and the x-axis is time. The colors represent different iterations. The data plotted is from a 104 virtual machine run on an 8-core machine. Due to heavy load on the system, there is a skew between the iteration start and stop times across virtual machines, resulting in iteration overlap in the system. We can see that different virtual machines start and finish at different times due to randomized load distribution on the physical host.

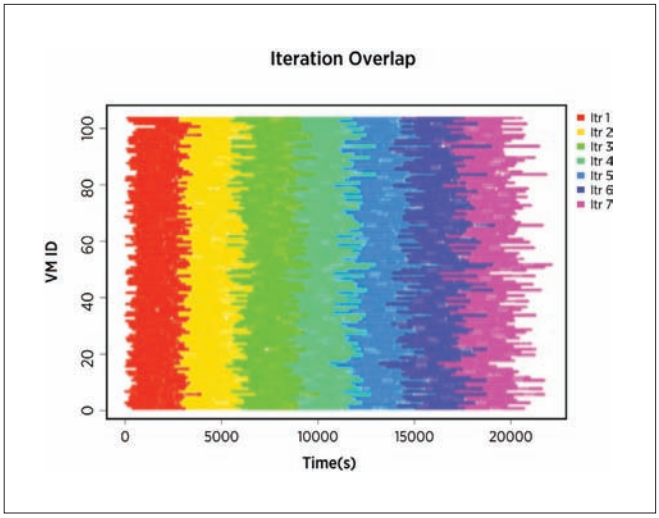

Figure 9. Shows the iteration overlap and how applications run in each virtual machine in 104 virtual machine run.

B. Finding the Score (Number of Supported Users)

One of most important use cases of View Planner is determining how many users can be supported on a particular platform. We used View Planner on a single host with the VMware® vSphere® 5 platform and Fibre Channel (FC) storage to determine the maximum number of supported users. Detailed results are shown in Table 2. The simulation started with 96 users. We observed that the Group A 95th percentile was 0.82 seconds, which was less than the threshold value of 1.5 seconds. We systematically increased the number of simulated desktops and looked at the QoS score. When the number of users was increased to 124, we could no longer satisfy the latency QoS threshold. Consequently, this particular configuration can only support approximately 120 users.

TOTAL # VMS	GROUP A 95% (SEC)	QOS STATUS
96	0.82	passed
112	0.93	passed
120	1.43	passed
124	1.54	failed
136	4.15	failed

Table 2: QoS score for different numbers of users

C. Comparing Different Hardware Configurations

View Planner can be used to compare the performance of different platforms:

- Storage protocols, such as the Network File System (NFS), Fibre Channel, and Internet SCSI (iSCSI) protocols
- Processor architectures, such as Intel Nehalem, Intel Westmere, and so on
- Hardware platform configuration settings, such as CPU frequency settings, memory settings, and so on

To demonstrate one such use case, we evaluated the memory over-commitment feature in VMware vSphere [15]. Table 3 shows the 95th percentile QoS threshold of View Planner with different percentages of memory over-commitment. The results show that even with 200 percent memory over-commitment, the system passed the QoS metric of 1.5 seconds.

VIRTUAL MACHINES/ HOST	LOGICAL MEM	PHYSICAL MEM	%MEM OVERCOMMIT	95TH PERCENTILE LATENCY
30	30GB	20GB	50%	0.59
40	40GB	20GB	100%	0.74
60	60GB	20GB	200%	0.86

Table 3: QoS with different memory over-commitment settings

D. Comparing Different Display Protocols

To compare different display protocols, we need to precisely characterize the response time of application operations. This is a capability of our watermarking technique. We simulated different network conditions—using LAN, WAN, and extreme WAN (very low bandwidth, high latency)—to see how the response time increased for different display protocols. Figure 10 shows the normalized response time chart comparing the PC-over-IP (PCoIP), PortICA, and RDP display protocols for three network conditions. These results show that PCoIP provides much better response time in all network conditions compared to other protocols. View Planner enables this kind of study and provides a platform to characterize the "true" end-user experience.

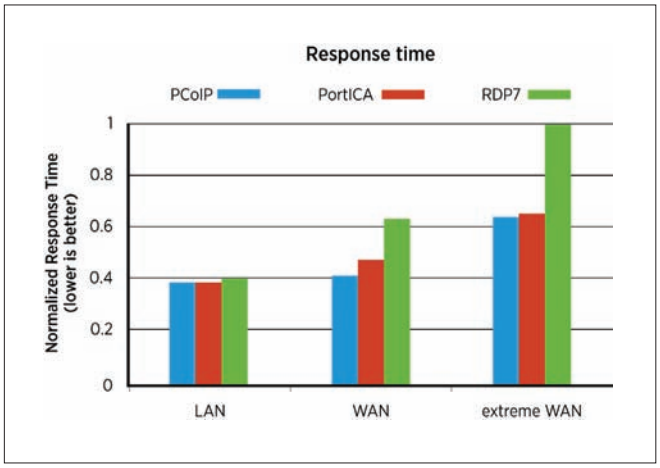

Figure 10. The normalized View Planner response time for diferent display protocols for different network conditions

E. Performance Characterization

View Planner can be used for performance characterization. To illustrate one study, we investigated the differing numbers of users a given platform could support using different versions of VMware View™. Figure 11 shows the 95th percentile response time for VMware View 4.5 and 5.0. We set threshold of 1.5 seconds and required the 95th percentile response time to fall below this threshold. For VMware View 4.5, the response time threshold crossed this threshold at 12 virtual machines (or 12 users) per physical CPU core. Hence, we can support between 11 to 12 users per core on VMware View 4.5. Looking at VMware View 5 result, we see it can easily support 14.5 Windows 7 virtual machines per core. Using View Planner, we were able to characterize the number of users that can be supported on a physical CPU core, and compare two versions of a product to analyze performance improvements (30 percent better consolidation in VMware View 5 compared to VMware View 4.5).

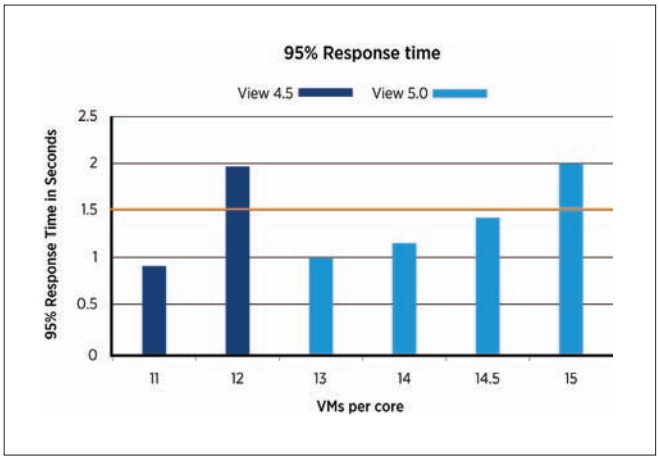

Figure 11. 95th percentile response time for VMware View 4.5 and VMware View 5 as the number of virtual machines per CPU core was increased.

F. Performance Optimizations

View Planner can help users find bottlenecks in the virtualization layer during a scalability study and apply performance optimizations. Using this tool, we can pinpoint the performance bottleneck at a particular component and try to fix the problem. For example, in a particular run, we found that application latencies (user experience) were poor. Upon further analysis, we traced the problem to the storage array, where disk latency was quite high and available I/O bandwidth was fully saturated. We also can study the performance of many protocol features and understand their impact on overall end-user experience. In addition, we identified many performance issues in the CPU and network (downlink bandwidth) usage in various applications during our protocol performance analysis, highlighting the significant potential of this workload in VDI environments.

6. Related Work

There are a number of companies providing VDI test solutions. Some, such as View Planner, focus on the entire VDI deployment [16, 17], while others offer limited scope and focus on a specific aspect of a VDI solution, such as storage [18]. At a high level, the functionality provided by these solutions might appear similar to View Planner at first glance. However, these solutions do not leverage watermarking techniques to accurately gauge the operation latency that is experienced by an end-user. Instead, they rely on "out-of-band" techniques to estimate remote response. For instance, out-out-band techniques include explicitly communicating event start and stop events to the client using separate artificially created events. In this situation, the start and stop events, unlike our watermarking technique, do not piggyback on the remote display channel and may not accurately reflect the operation latency observed by a user. Other approaches involve network layer taps to attempt to detect the initiation and completion of operations. Not only are these approaches potentially inaccurate, they introduce significant complexities that limit portability between operating systems and VDI solutions.

The out-of-band signaling exploited by other VDI test solutions can lead to significant inaccuracies in the results, which in turn c an lead to misleading conclusions about permissible consolidation ratios, protocol comparisons, and result in invalid analysis of VDI deployments. Other approaches include analyzing screen updates and attempting to automatically detect pertinent events (typically used for comparative performance analysis) [19], and inferring remote latencies by network monitoring and slow-motion benchmarking [20, 21]. While these approaches work for a single VDI user on an unloaded system, they can significantly perturb (and are perturbed by) the behavior of VDI protocols under load, making them unsuitable for the robust analysis of realistic VDI workloads at scale.

Finally, a variety of other techniques have been developed for latency analysis and to look for pauses due to events such as garbage collection [22]. These approaches assume (and depend on the fact) the user is running on a local desktop.

7. Conclusion

This paper presented View Planner, a next-generation workload generator and performance characterization tool for large-scale VDI deployments. It supports both types of operations (user and administrative operations) and is designed to be configurable to allow users to accurately represent their particular VDI deployment. A watermarking measurement technique was described that can be used in a novel manner to precisely characterize application response time from the client side. For this technique, watermarks are sent with the display screen as notifications and are piggybacked on the existing display. The detailed architecture of View Planner was described, as well as challenges in building the representative VDI workload, and scalability features. Workload characterization techniques illustrated how View Planner can be used to perform important analysis, such as finding the number of supported users on a given platform, evaluation of memory over-commitment, and identifying performance bottlenecks. Using View Planner, IT administrators can easily perform platform characterization, determine user consolidation, perform necessary capacity planning, undertake performance optimizations, and a variety of other important analyses. We believe View Planner can help VDI administrators to perform scalability studies of nearly real-world VDI deployments and gather useful information about their deployments.

Acknowledgement

We would like to thank Ken Barr for his comments and feedback on the early drafts of this paper, as well as other reviewers. Finally, we thank everyone in the VDI performance team for their direct or indirect contributions to the View Planner tool.

Any further information and details about the View Planner tool can be achieved by sending an email to viewplanner-info@vmware.com

References

1 B. Agrawal, L. Spracklen, S. Satnur, R. Bidarkar, "VMware View 5.0 Performance and Best Practices", VMware White Paper, 2011.

2 L. Spracklen, B. Agrawal, R. Bidarkar, H. Sivaraman, "Comprehensive User Experience Monitoring", in VMware Tech Journal, March 2012.

3 VMware Inc., Timekeeping in VMware Virtual Machines, http://www.vmware.com/vmtn/resources/238

4 Python Twisted Framework, http://twistedmatrix.com/trac/

5 Google web toolkit (GWT), http://code.google.com/webtoolkit/

6 Sturdevant, Cameron. VMware View 4.5 is a VDI pacesetter, eWeek Vol. 27, no. 20, pp. 16-18. 15 Nov 2010.

7 VMware View: Desktop Virtualization and Desktop Management www.vmware.com/products/view/

8 Citrix Inc., Citrix XenDesktop 5.0. http://www.citrix.com/English/ps2/products/feature.asp?contentID=2300341

9 J. Nieh, S. J. Yang, and N. Novik, "Measuring Thin-Client Performance Using Slow-Motion Benchmarking", ACM Trans. Comp. Sys., 21:87–115, Feb. 2003.

10 S. Yang, J. Nieh, M. Selsky, N. Tiwari, "The Performance of Remote Display Mechanisms for Thin-Client Computing", Proc. of the USENIX Annual Technical Conference, 2002.

11 VI SDK. https://www.vmware.com/support/developer/vc-sdk/

12 AutoIT documentation http://www.autoitscript.com/autoit3/docs/

13 Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, Joonwon Lee, Task-aware virtual machine scheduling for I/O performance., Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, March 11-13, 2009, Washington, DC, USA

14 Micah Dowty, Jeremy Sugerman, GPU virtualization on VMware's hosted I/O architecture, ACM SIGOPS Operating Systems Review, v.43 n.3, July 2009

15 VMware White Paper. "Understanding Memory Resource Management in VMware® ESX™ Server ", (2010).

16 LoginVSI—Admin Guide of Login Consultants "Virtual Session Indexer" 3.0, http://www.loginvsi.com/en/admin-guide

17 Scapa Test and Performance Platform, http://www.scapatech.com/wpcontent/uploads/2011/04/ScapaTPP_VDI_0411_web.pdf

18 VDI-IOmark, http://vdi-iomark.org/content/resources

19 N. Zeldovich and R. Chandra, "Interactive performance measurement with VNCplay", Proceedings of the USENIX Annual Technical Conference, 2005.

20 A. Lai, "On the Performance of Wide-Area Thin-Client Computing", ACM Transactions on Computer Systems, May 2006.

21 J. Nieh, S. J. Yang, N. Novik, "Measuring Thin-Client Performance Using Slow-Motion Benchmarking", ACM Transactions on Computer Systems, February 2003.

22 A. Adamoli, M. Jovic and M. Hauswirth, "LagAlyzer: A latency profile analysis and visualization tool", International Symposium on Performance Analysis of Systems and Software, 2010.

vSOM: A Framework for Virtual Machine-centric Analysis of End-to-End Storage IO Operations

Sandeep Uttamchandani
VMware, Inc.
suttamchandani@vmware.com

Wenhua Liu
VMware, Inc.
liuw@vmware.com

Samdeep Nayak
VMware, Inc.
samdeep@vmware.com

Abstract

Diagnosis of an I/O performance slow down is a complex problem. The root cause could be one among a plethora of event combinations such as VMware® ESXi misconfiguration, an overloaded fabric switch, disk failures on the storage arrays, and so on. As a virtualization administrator, diagnosing the end-to-end I/O path today requires working with discrete fabric and storage reporting tools, and manually correlating component statistics to find performance abnormalities and root-cause events. To address this pain point, especially for cloud scale deployments, we developed the VMWare SAN Operations Manager (vSOM), a framework for end-to-end monitoring, correlation, and analysis of storage I/O paths. It aggregates events, statistics, and configuration details across the ESXi server, host bus adapters (HBAs), fabric switches, and storage arrays. The correlated monitoring data is analyzed in a continuous fashion, with alerts generated for administrators. The current version invokes simple remediation steps, such as link and device resets, to potentially fix errors such as link errors, frame drops, I/O hangs, and so on. vSOM is designed to be leveraged by advanced analytical tools. One example described in this paper, VMware® vCenter™ Operations Manager™, uses vSOM data to provide end-to-end virtual machine-centric health analytics.

1. Introduction

Consider an application administrator responding to multiple problem tickets: "The enterprise e-mail service has a 40-60 percent higher response time compared to its average response time over the last month." Because the e-mail service is virtualized, the administrator starts by analyzing virtual machines, manually mapping the storage paths and associated logical and physical devices. Today, there is no single tool to assist with the complete end-to-end diagnosis of the Storage Area Network (SAN), starting with the virtual machine, through the HBAs, switches, storage array ports and disks (Figure 1). In large enterprises, the problem is aggravated further with specialized storage administrators performing isolated diagnosis at the storage layer: "The disks look fine; I/O rate for e-mail related volumes has increased, and the response time seems within normal bounds." This to-and-fro between application and storage administrators can take weeks to resolve. The real root cause could be accidental rezoning of the server ports, combined with a change in the storage array configuration that labeled those ports as "low priority traffic."

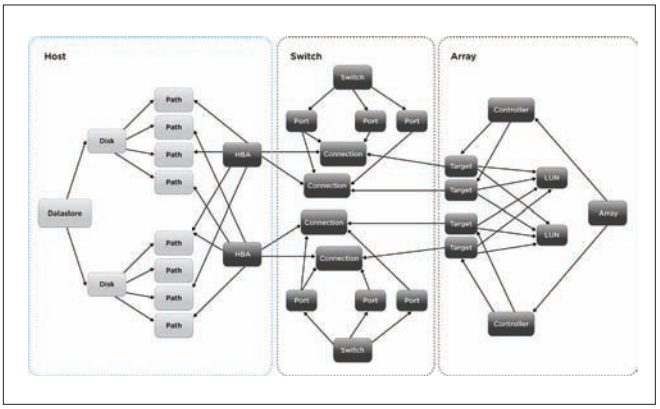

Figure 1: Illustration of the Real-world End-to-end I/O Path in a Virtualization Environment

In summary, end-to-end analysis in a SAN environment is a complex task, even in physical environments. Virtualization makes the analysis even more complex, given the multiplexing of virtual machines on the same physical resources.

This paper describes vSOM, the VMWare SAN Operations Manager, a general-purpose framework for heterogeneous, cloud-scale SAN deployments. vSOM provides a virtual machine-centric framework for end-to-end monitoring, correlation, and analysis across the I/O path including SAN components, namely HBAs, switches, and storage arrays. vSOM is designed to provide correlated monitoring statistics to radically simplify diagnosis, troubleshooting, provisioning, planning, and infrastructure optimization. In contrast, existing tools [1, 2, 3, 4, 5] are discrete in monitoring one or more components, falling short of the end-to-end stack. vSOM internally creates a correlation graph, mapping the logical and physical infrastructure elements, including virtual disks (VMDK), HBA ports, switch ports, array ports, logical disks, and even physical disk details if exposed by the controller. The elements in the correlation graph are monitored continuously, aggregating both statistical metrics and events.

Developing a cloud-scale end-to-end I/O analysis framework is nontrivial. The following are some of the design challenges that vSOM addresses:

- **Heterogeneity of fabric and storage components**: There is no single available standard that is universally supported for out-of-band management of fabric and storage components. SNIA's

Storage Management Interface Specifications (SMI-S)[7] has been adopted by a partial subset of key vendors. Leveraging the VMware vSphere® Storage APIs for Storage Awareness (VASA) [10], provides a uniform syntactic and semantic interface to query storage devices, but is not supported by fabric vendors.

- **Scalability of the monitoring framework**: The current version of vSphere supports 512 virtual machines per host, 60 VMDKs per virtual machine, and 2,000 VMDKs per host. The ratio of VMDKs to physical storage LUNs typically is quite large. vSOM needs to monitor and analyze a relatively large corpus of monitored data to notify administrators for hardware saturation, anomalous behavior, correlated failure events, and so on.
- **Continuous refinement for configuration changes**: In a virtualized environment, the end-to-end configuration is not static. It evolves with events, such as VMware vSphere vMotion®, where either a virtual machine moves to different server, associated storage relocates, or both. The analysis of monitoring data needs to take into account the temporal nature of the configuration and appropriately correlate performance anomalies with configuration changes.

vSOM employs several interesting techniques, summarized as the key contributions of this paper:

1. Discovery and monitoring of the end-to-end I/O path that consists of several heterogeneous fabric and storage components. vSOM stiches together statistical metrics and events using a mix of standards and propriety APIs.
2. Correlation of the configuration details is represented internally as a directed acyclic dependency graph. The edges in the graph have a weight, representing I/O traffic between the source and destination vertices. The graph is self-evolving and updated based on configuration events and I/O load changes.
3. Analysis of statistics and events to provide basic guidance regarding the health of the virtual machine based on health of the I/O path components. Additionally, vSOM plugs into a richer set of analytics, planning, and trending capabilities of VMWare's Operations Manager.

The rest of the paper is organized as follows: Section 2 gives a bird's eye-view of vSOM. Sections 3, 4, and 5 cover details of the monitoring, correlation, and analysis modules respectively. The conclusion and future work are summarized in Section 6.

2. A Bird's Eye View of vSOM

The objective of vSOM is to monitor, correlate, and analyze the health of the virtual machine, as a function of the SAN I/O components (HBA, switches, and storage array). This section describes the system model and a high-level overview of the vSOM architecture.

2.1 System Model

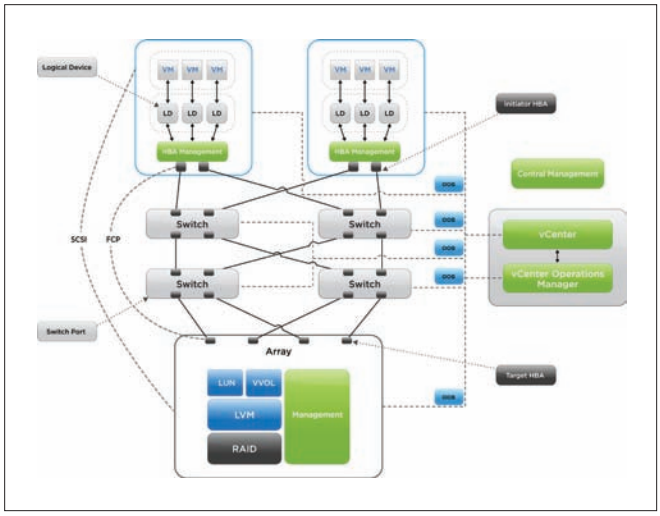

Figure 2: vSOM System Model

The overall system model is shown in Figure 2. The hypervisor abstracts physical storage LUNs (also referred to as Datastore), and exports Virtual Disks (VMDKs) to virtual machines. The hypervisor supports a broad variety of storage protocols, such as Fibre Channel (FC), Internet SCSI (iSCSI), Fibre Channel over Ethernet (FCoE), the Network File System (NFS), and so on. A VMDK is used by the Guest operating system either as a raw physical device abstraction (referred to as Raw Device Mapping or RDM), or a logical volume derived from VMFS or a NFS mount point. RDMs are not a common use-case, since they bypass the hypervisor I/O stack and key features of the hypervisor such as vMotion, resource scheduling, and so on, cannot be used. As illustrated in Figure 2, a typical end-to-end I/O path from the virtual machine to physical storage consists of Virtual machine → VMDK → HBA Port → Switch Port →Array Port → Array LUN → Physical device. The current version of vSOM supports block devices only. NFS volumes are not supported.

Multiple industry-wide efforts try to standardize the management of HBAs, switches, and storage arrays. The most popular and widely adopted standard is SNIA's Storage Management Initiative Specifications (SMI-S)[7]. The standard defines profiles, methods, functions, and interfaces for monitoring and configuring system components. This standard is widely adopted by switch and HBA vendors, with limited adoption by storage array vendors. The standard is built on the Common Information Model (CIM)[8] that defines the architecture to query and interface with system management components. In the context of CIM, a CIM Object Manager (CIMOM) implements the management interface, accessible locally or through a remote connection.

2.2 vSOM Overview

vSOM tracks the end-to-end I/O path and collects data across ESXi hosts, VMware Virtual Center™, fabrics, and storage arrays. component is monitored continuously to collect configuration details, performance statistics, and events. Data collected from the components is correlated to create a virtual machine-centric analysis including VMDKs, HBAs, switches, and storage arrays. The preciseness of the end-to-end correlation depends on the configuration. For a virtual machine with a raw mapped VMDK, there is a one-to-one mapping between VMDK and the physical LUN—the statistics gathered from the LUN and HBA paths can be attributed directly to the virtual machine. Conversely, with a virtual machine using VMDKs carved on a VMFS volume, the statistics of the storage array LUN and HBA paths reflect the status of a set of virtual machines sharing the LUN.

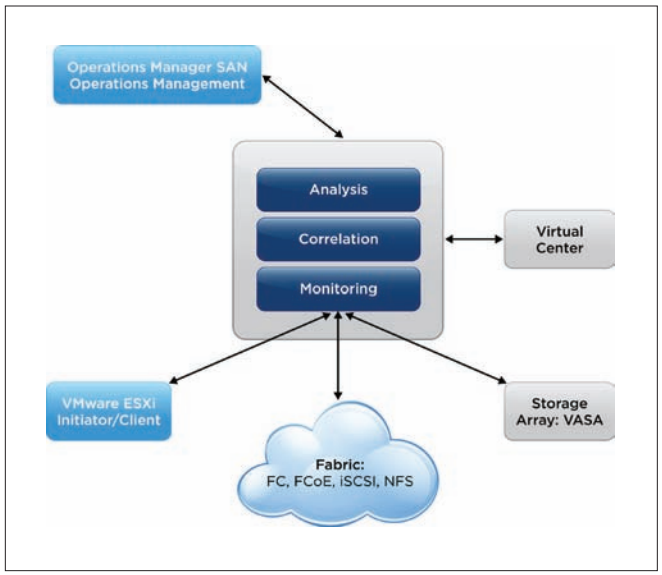

Figure 3: High-level vSOM Architectural Modules

The vSOM architecture consists of three key building blocks: Monitoring, Correlation, and Analysis Modules (Figure 3).

- **Monitoring Module**: The monitoring framework consists of Agents and a centralized Management Station. Agents collect statistics and events on the components. The initial component discovery uses a combination of vSphere configuration details from vCenter, combined with the Service Location Protocol (SLP)[11]. Agents collect monitoring data in real time, using a combination of SMI-S standards and vSphere APIs. The Management Station aggregates data collected by the individual component agents, and internally uses different protocols to connect with the host, switch, and storage array agents.
- **Correlation Module**: The configuration details collected from the component agents are used to create an end-to-end dependency graph. The graph is updated continuously for events such as vMotion, Storage vMotion, Storage DRS, HA failover, and so on. An update to the I/O path configuration is tracked with a unique Configuration ID (CID). The historical statistical data from each component is persisted by tagging with the corresponding CID. This enables statistical anomaly detection and the effect of changes to the configuration.
- **Analysis Module**: The monitored data is analyzed to determine the health of individual components. Statistical anomaly analysis of monitored data can be absolute or relative to other components. The current version of vSOM also supports rudimentary remediation actions that are triggered when an erroneous pattern is observed over a period of time, such as an increased number of loss sync or parity errors. vSOM plugs into Operations Manager, and provides data for end-to-end virtual machine-centric analysis.

3. Monitoring Module

As mentioned earlier, the Monitoring module consists of Agents and the centralized Management Station. Agents implement different mechanisms to collect data. The ESXi agent uses CIM, fabric agents uses SMI-S, storage array agents use either SMI-S or VMWare's API for Storage Awareness (VASA). Besides the agents, the Management Station also communicates with vCenter Server to collect event details.

The monitoring details collected from Agents are represented internally as software objects. The schema of these objects leverages CIM-defined profiles wherever possible. The objects are persisted by the Management Station using a circular buffer implementation. The size of the circular buffer is configurable, and corresponds to the amount of history. Component monitoring is near-real-time, with the monitoring interval typically being 30-60 seconds.

This section describes the steps involved in the monitoring bootstrapping process, as well as details of the internal software object representation. vSOM implements a specialized CIM-based agent for ESXi hosts, and this section covers the key implementation details.

3.1 Bootstrapping Process

Bootstrapping involves the discovery of entities associated with a given virtual machine. This is accomplished using a combination of vSphere configuration analysis and the Service Location Protocol (SLP). The steps involved in the discovery process are summarized as follows:

1. vSOM queries the vCenter Server to identify all hosts and datastores that host active virtual machines and VMDKs, respectively. Querying the vCenter Server acts as white-box knowledge, limiting the search space and enabling faster convergence. In contrast, black-box discovery of all devices with a vSphere cluster and SAN setup would be much slower to complete.
2. For each VMDK, the ESXi CIM provider is queried to provide the logical device details.
3. Using the logical device details, vCenter is queried to retrieve the associated storage port IDs at the host and storage array (commonly referred to as initiator and target ports). Each port is uniquely identified with a World Wide ID (WWID). At the end of this step, for each VMDK, the corresponding initiator and target port WWIDs are discovered. For VMDKs mapped on the same logical device (datastore), the initiator and target ports are the same.

4. The following schemes are adopted to discover the fabric topology, depending on whether the storage protocol is FC, FCoE, or iSCSI.

    a. FC and FCoE fabrics implement CIM. vSOM uses SLP to discover the CIMOM for each switch, followed by validation of support for the SMI-S profile. If the SMI-S profile is supported by the CIMOM, vSOM queries the switch ports associated with the initiator and target WWID and identifies any inter-wwitch links that might be connected between the host and the target.

    b. For iSCSI, vSOM uses fast traceroute to identify the fabric topology between the host and the target. vSOM queries the individual port details using the Simple Network Management Protocol (SNMP).

3.2 End-to-end I/O Monitoring

The end-to-end I/O path is represented as a combination of Host, Switch, and Array objects. Each object stores a combination of performance statistics and events.

3.2.1 Host Object

The Host object includes monitoring of the VMDKs, SCSI disks, and HBA. The host object is exported as a CIM profile, accessed by the central Management Station. Instead of defining a new data model, the Host object follows the SMI-S *Block Server Performance Subprofile* [7]. The profile defines classes and methods for managing performance information, and was originally designed for storage arrays, virtualization engines, and volume managers. In designing the data model (Figure 4),

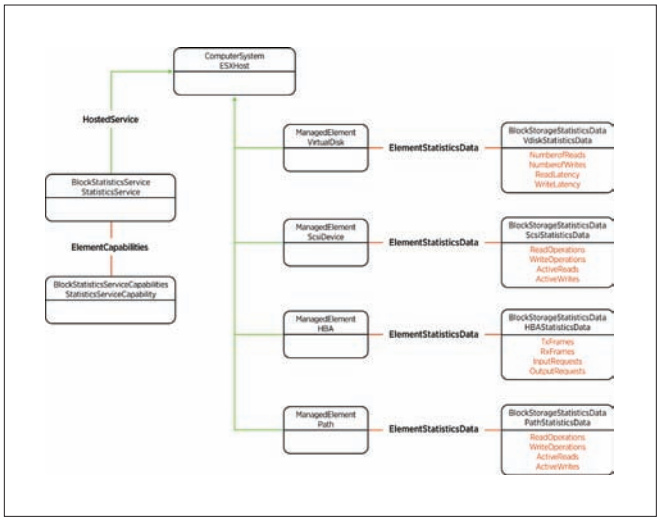

Figure 4: Data Model for ESXi Host Object Derived from the SMI-S Block Server Performance Sub-profile

we observed a one-to-one mapping between the abstractions on an ESXi server and a storage array: Virtual machines within ESXi are equivalent to hosts of a storage array. VMDKs are equivalent to LUNs exported by a storage array. SCSI devices on the host are similar to physical drives on an array, HBAs as the back-end ports, datastores as the storage pools of a storage array. In other words, a Block Server Performance Subprofile maps well with vSOM requirements. The Host CIM provider implements only a subset of the classes, associations, properties, and methods of the profile, as required for the vSOM monitoring framework. For the HBA object, the data collected includes generic SCSI performance data and transport-specific performance data.

3.2.2 Switch Object

The Switch object consists of a collection of ports across one or more switches that are in the I/O path that serves the virtual machine's storage traffic. Major SAN switch vendors have implemented SMI-S compliant CIM providers. vSOM uses these CIM providers as data collection agents. In the context of storage, switches typically use Fibre Channel or standard Ethernet. For each FC port, vSOM switch objects use the data model defined in CIM schema 2.24 (CIM_FCPortRateStatistics and CIM_FCPortStatistics). In SCSI, only Class 3 service is used on Fibre Channel. As a result, the attributes in the CIM profile containing Class 1 or Class 2 are ignored. Performance data is collected and persisted only for the ports on which ESX hosts or storage arrays are connected.

3.2.3 Storage Array Object

The Storage Array is the final destination of the I/O, and is referred to as the Target. A typical storage array consists of array ports, controllers, and logical volumes. Additionally, storage arrays can export details on physical disks and back-end storage ports. In vSOM, two mechanisms are available to collect data from the storage array: VMWare's VASA profile or the CIM storage profile. The latter provides a limited set of attributes, generalizing the differentiated capabilities of the storage arrays. The Management Station connects with the Array agents using the WSDL-based protocol[9] for VASA or a Generic Storage Adapter for CIM. Note that for the purpose of vSOM, the existing VASA specifications have been extended with certain performance attributes.

3.3 Internals for Data Collection from Host

As a part of the vSOM initiative, the ESXi host has been extended with I/O Device Management (IODM). This module provides functionality to configure, monitor, and deliver Storage object I/O details to the vSOM Management Station. This subsection describes details of the IODM implementation within ESXi (Figure 5).

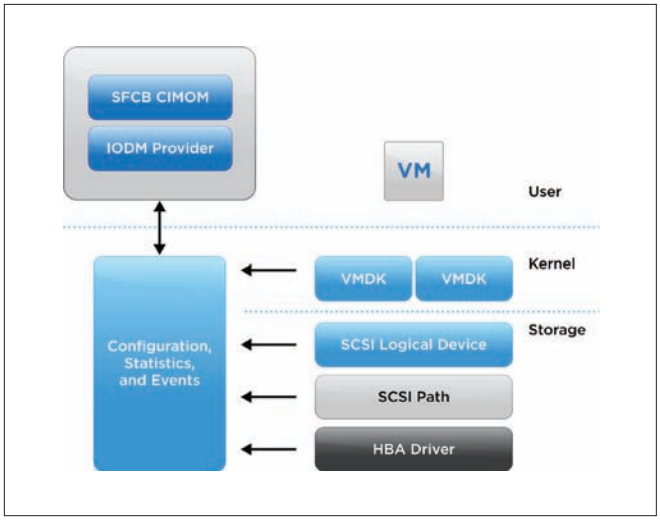

Figure 5: I/O Device Management (IODM) Implementation for ESXi

IODM implementation is split into two parts: an IODM Kernel Module and an IODM CIM Provider for the user world. The kernel module captures statistics and events about VMDKs, SCSI logical devices, SCSI paths, and HBA transport details. The kernel module also presents an asynchronous mechanism to deliver events to the user world.

The IODM CIM provider implemented consists of two parts: Upper layer and Bottom layer. The Upper layer is the standard CIM interface, with both the intrinsic and extrinsic interfaces implemented. The intrinsic interface includes **EnumInstances**, **EnumInstanceNames**, and **GetInstance**. This interface is used to get I/O statistics and error information for virtual machines, devices, and HBAs. The extrinsic interface controls the IODM behavior with functions such as start/stop data collection. CIM indication for events also is part of the Upper layer. It interacts with the IODM kernel modules to get events and alerts.

4. Correlation Module

The Correlation Module maintains the dependency between the components in the I/O path. The dependency details are persisted as an acyclic directed graph. Vertices represent the I/O path components, and edges represent the correlation weight, as illustrated in Figure 6. The steps involved in discovering correlation details between the components is similar to the bootstrapping process covered in Section 3.1.

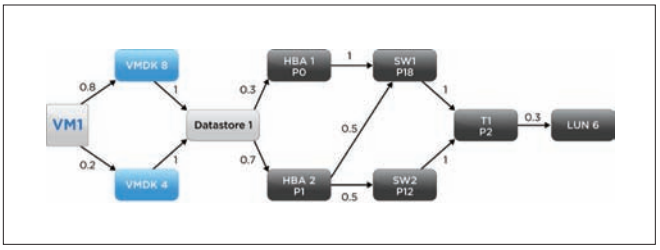

Figure 6: Representation of the Correlation between I/O Path Components

Figure 6 shows the correlation details between VM1 and LUN 6. VM1 has configured VMDK 4 and VMDK 8 as storage volumes. These VMDKs are mapped to logical SCSI device (Datastore 1). The Datastore is connected to HBA 1 and HBA 2 on ports P0 and P1 respectively. These ports are mapped to Switches 1 and 2 (SW1 and SW2) on ports 18 (P18) and 12 (P12), respectively. Finally, the switches connect to the Storage Array Target (T1 on Port 2), accessing LUN 6. With respect to correlation granularity, VMDK 4 and VMDK 8 merge into Datastore 1. The outgoing traffic from the Datastore can be a combination of other VMDKs as well (in addition to VMDK 4 and 8). The weights for an edge are normalized to a value between 0 and 1. The summation of the outgoing edges from a vertex should typically be 1. Note that this might not always be the case. For instance, the traffic from Target T1 Port 2 is mapped to other LUNs besides LUN 6. As a result, the total of the outgoing edges from T1 P2 is shown in Figure 6 as 0.3.

vSOM continuously monitors for configuration changes and updates the dependency graph. Each version of the configuration is tracked using a unique 32-byte Configuration ID. Maintaining the versions of the dependency graph help in the time travel analysis of configuration changes, and their corresponding impact on configuration changes. As mentioned earlier, the performance statistics are tagged with the Configuration ID.

The dependency graph is updated in response to either configuration change events or updates to the edge weights in response to workload variations. Configuration change events such as vMotion and HA, among others, are tracked from vCenter, while CIM indications from individual components indicate the creation, deletion, and other operational status change of switches, FC ports, and other components. The edge weights in the dependency graph are maintained as a moving average and are updated over longer time windows (3-6 hours).

5. Analysis Module

The goal of the Analysis module is to use monitoring and correlation details to provide an intuitive representation of virtual machine health as a function of the heath of the individual components (Host machine, HBAs, Fabric, and Storage Arrays). The Analysis module categorizes the health of each component into green, yellow, orange, or red:

- Green indicates normal, with the component behaving within expected thresholds
- Yellow means attention is needed, primarily based on reported error events
- Orange indicates an increasingly degrading condition, based on a combination of statistical analysis and error events
- Red means I/O can no longer flow and virtual machines cannot operate

Based on the dependency graph, the health of individual components is rolled up at the virtual machine level (Figure 7).

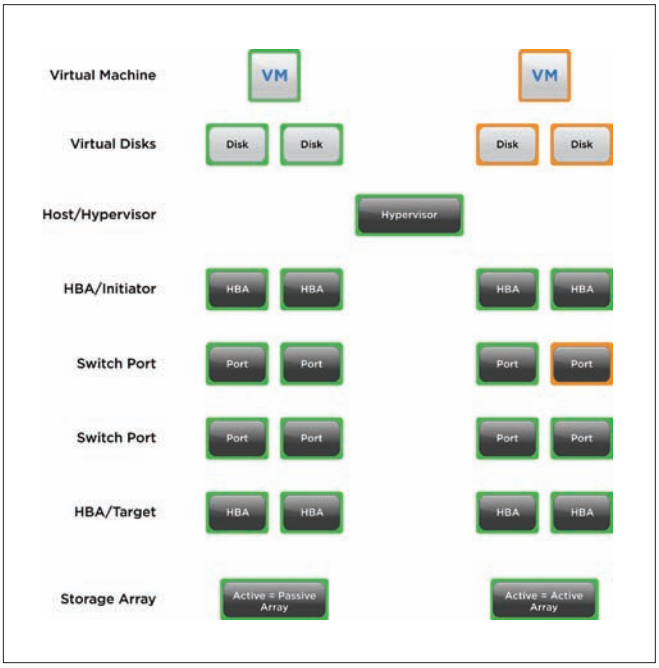

Figure 7: Virtual Machine Health is a Cumulative Roll Up of Individual Components in the End-to-end Path

As shown, degradation in the Switch Port affects the virtual machine, based on its high correlation weight for host to storage array connectivity.

The health of a component can be deduced using different approaches. Traditionally, administrators are expected to define alert thresholds. For example, when capacity reaches 70 percent, the health of the disk is marked yellow or orange. Defining these thresholds typically are nontrivial and arbitrary. Further, given the scale of cloud systems, it is unrealistic for administrators to define these thresholds.

vSOM employs two different techniques to determine component health. The first approach is referred to as Absolute Anomaly Analysis. The history of monitored data is analyzed to determine whether current component performance is anomalous. There are standard data mining techniques for anomaly detection. vSOM uses the K-means Clustering approach that detects an anomaly and associates a weight to help categorize the anomaly as yellow, orange, or red. The second approach is based on Relative Analysis. In this approach, peer components (such as ports on the same switch, or events on different ports of the same HBA) are analyzed to determine if the observed behavior anomaly is similar to other components. In large-scale deployments, Relative Analysis is an effective approach, especially if the available history of monitored data is not sufficient for Absolute Anomaly Analysis.

Analysis can help pinpoint the root cause of the problem and be used to trigger automated remediation. Automated root-cause analysis is nontrivial, especially in large-scale, real-world deployments where the cause and effect might not always be on the same component, or the problem might be a result of multiple correlated events. vSOM implements a limited version of auto remediation, using link or device resets. Complex remediation actions, such as changing the I/O path or vMotion, is beyond the scope of the current version of vSOM.

Reset is an effective correction action for a common set of link-level and device-level erroneous patterns. For errors observed over a period of time, such as an increased number of loss sync or parity errors, failure of protocol handshaking, and so on, are commonly fixed with a link reset. A link reset can reinitialize the link and put I/O back on track. If several link resets do not fix the problem, the path can be disabled to trigger a path failover to a backup path if multiple paths are available.

vSOM correlates events from components in the end-to-end path. This helps in determining the root cause of events such as link down, frame drops, I/O or virtual machine hangs, and similar events. For instance, link down events are collected from the ESX host by subscribing to CIM indications, helping to isolate the root cause on the ESX host versus a specific HBA or switch port.

The vCenter Operations Manager is an existing VMware product that provides rich analytical capabilities for managing performance and capacity for virtual and physical infrastructures. It provides analytics for performance troubleshooting, diagnosis, capacity planning, trending, and so on. Using advanced statistics analysis, Operations Manager currently associates a health score to resources such as compute, and continuously tracks the health to raise alerts for abnormal behavior, sometimes even before a problem exhibits any symptoms. vSOM plugs into Operations Manager using its standard adapter interface. vSOM complements the existing analysis of Operations Manager for virtual machine and datastore level, with details of the SAN components (HBAs, Fabric, and Storage Array). Operations Manager stores historic statistical data in a specialized database and implements anomaly detection algorithms for historic data analysis. In addition to end-to-end monitoring and troubleshooting, Operations Manager can help with planning and optimization use cases, such as balancing workloads across all controllers and switch ports.

6. Conclusion and Future Work

This paper describes an end-to-end SAN management framework implemented for vSphere. It addresses the pain points associated with monitoring I/O components from the viewpoint of virtual machine-centric performance. While problem diagnosis is the most intuitive use case, vSOM is applicable to other use cases, such as planning, single-pane of glass monitoring, load balancing, and more.

We plan to extend this work in several dimensions. Automated remediation has significant value for administrators. We plan to extend beyond the current reset action to support complex multistep actions. For root-cause analysis, we plan to combine our current black-box anomaly analysis with rule-based techniques, particularly to correlate error events across different components in the I/O path. Finally, we are exploring monitoring module extensions to include application-level statistics in the end-to-end I/O path.

References

1 HP Systems Insight Manager Overview, http://h18013.www1.hp.com/products/servers/management/hpsim/index.html?jumpid=go/hpsim

2 Dell OpenManage Systems Management, http://www.dell.com/content/topics/global.aspx/sitelets/solutions/management/en/openmanage?c=us&l=en&cs=555

3 IBM Tivoli Storage Management Solutions, http://www-01.ibm.com/software/tivoli/solutions/storage/

4 Shen, K., Zhong, M., and Li, C. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies (FAST) (2005), pp. 23–23.

5 Pollack, K. T., and Uttamchandani, S. Genesis: A Scalable Self-Evolving Performance Management Framework for Storage Systems. In IEEE International Conference on Distributed Computing Systems (ICDCS) (2006), p. 33.

6 VMWare vCenter Operations Manager, http://www.vmware.com/support/pubs/vcops-pubs.html

7 Storage Management Initiative Specification (SMI-S), http://www.snia.org/tech_activities/standards/curr_standards/smi

8 Common Information Model, http://en.wikipedia.org/wiki/Common_Information_Model_(computing)

9 Web Services Description Language (WSDL), http://www.w3.org/TR/wsdl

10 vSphere Storage API for Storage Awareness (VASA), http://blogs.vmware.com/vsphere/2011/08/vsphere-50-storage-features-part-10-vasa-vsphere-storage-apis-storage-awareness.html

11 Service Location Protocol, http://www.ietf.org/rfc/rfc2608.txt

vmware®

VMware Academic Program

The VMware Academic Program advances the company's strategic objectives through collaborative research and other initiatives. The Program works in close partnerships with both the R&D and University Relations teams to engage with the global academic community.

A number of research programs operated by VMAP provide academic partners with opportunities to connect with VMware. These include:

- Annual Request for Proposals (see front cover pre-announcement), providing funding for a number of academic projects in a particular area of interest.
- Conference Sponsorship, supporting participation of VMware staff in academic conferences and providing financial support for conferences across a wide variety of technical domains.
- Graduate Fellowships, recognizing outstanding PhD students conducting research that addresses long-term technical challenges.
- VMAP Research Symposium, an opportunity for the academic community to learn more about VMware's R&D activities and existing collaborative projects.

We also support a number of technology initiatives. The VMAP licensing program (http://vmware.com/go/vmap) provides higher education institutions around the world with access to key VMware products. Our cloud technology enables the Next-Generation Education Environment (NEE, http://labs.vmware.com/nee/) to deliver virtual environments for educational use. New online education providers, such as EdX, use VMware solutions to deliver a uniform learning environment on multiple platforms.

Visit http://labs.vmware.com/ to learn more about the VMware Academic Program.

SPECIAL THANKS

Pat Gelsinger
Chief Executive Officer

Terry Anderson
Vice President of Global Corporate Communications

Alex Garthwaite

Rean Griffith

Margaret Lowe

Questions and comments can be sent to vmtj@vmware.com

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com