J Internet Serv Appl (2011) 2:141–154
DOI 10.1007/s13174-011-0031-1

SI: CLOUD COMPUTING

Autonomic live adaptation of virtual networked environments in a multidomain infrastructure

Paul Ruth · Junghwan Rhee · Dongyan Xu · Sebastien Goasguen · Rick Kennell

Received: 19 November 2010 / Accepted: 27 June 2011 / Published online: 23 July 2011

Abstract The trend toward cloud-based services is creating the need for large scale shared distributed infrastructures. Behind many clouds lay shared distributed infrastructures formed through the federation of many resources residing in multiple domains. Such shared infrastructures enable massive amounts of aggregated computation resources to be shared among large numbers of users. The core technologies enabling these distributed clouds are machine and network virtualization. Virtualization is the technology that enables the execution of arbitrary distributed applications on top of these increasingly popular shared physical infrastructures.

In this paper, we go beyond supporting applications in the cloud and support autonomic adaptation of virtual computation environments as active, integrated entities. More specifically, driven by both dynamic availability of infrastructure resources and dynamic application resource demand, a virtual computation environment is able to automatically relocate itself across the infrastructure and scale its share of infrastructural resources. Such autonomic adaptation is transparent to both users of virtual environments and administrators of infrastructures, maintaining the look and feel of a stable, dedicated environment for the user. As our proof-of-concept, we present the design, implementation, and evaluation of a system called VIOLIN, which is composed of a virtual network of virtual machines capable of live migration across a multidomain physical infrastructure.

Keywords Virtual machines · Cloud computing · Cyberinfrastructure · Autonomics

P. Ruth (✉)
Renaissance Computing Institute (RENCI), University of North Carolina–Chapel Hill, Chapel Hill, NC, USA
e-mail: pruth@renci.org

J. Rhee · D. Xu
Department of Computer Science, Purdue University, West Lafayette, IN, USA

J. Rhee
e-mail: rhee@cs.purdue.edu

D. Xu
e-mail: dxu@cs.purdue.edu

S. Goasguen
School of Computing, Clemson University, Clemson, SC, USA
e-mail: sebgoa@clemson.edu

R. Kennell
Rosen Center for Advanced Computing (RCAC), Purdue University, West Lafayette, IN, USA
e-mail: rick@kennell.org

1 Introduction

In the past decade, we have seen the emergence of services that federate, allocate, and manage resources across multiple network domains, most notably grid computing [3, 11, 12, 25] and distributed laboratories [5]. More recently, we have seen the emergence of more centrally managed cloud services such as Google App Engine [2] and Amazon EC2 [1]. The growth of these infrastructures has led to the availability of unprecedented computational power to a large community of users. Much of the capabilities of these systems were enabled by virtual machine technologies [4, 8, 35], which have been increasingly adopted on top of such shared physical infrastructures [9], and have greatly elevated customization, isolation, and administrator privilege for users running applications inside individual virtual machines.

Going beyond individual virtual machines, our previous work proposed techniques for the creation of virtual distributed computation environments [15, 26, 27] on top

of a shared distributed infrastructure. Our virtual computation middleware, called VIOLIN, deploys environments of virtual machines connected by a virtual network, which provides a layer separating the ownership, configuration, and administration of the VIOLIN virtual environment from those of the underlying infrastructure. Mutually isolated VIOLINs can be created for different users as their "own" private distributed computation environment bearing the same look and feel of customized physical environments with administrative privilege (e.g., their own private cluster). Within VIOLIN, the user is able to execute and interact with unmodified parallel/distributed applications, and can expect strong confinement of potentially untrusted applications. Additional work on VIOLIN has led to distributed virtual environments that can be checkpointed and restarted at arbitrary points during execution [20] as well as many advances in computer security through the use of virtual machine introspection [14] and honey pots [16].

The goal of VIOLIN as a cyberinfrastructure is to federate massive amounts of heterogeneous resources made available through the Internet and make the resources available to users. Unfortunately, existing methods for sharing independently administered resources have limitations. These methods often use restrictive user authentication models that require manual creation and monitoring of accounts within each independent domain. Independent authentication in each domain hinders the ability to federate infrastructures. In addition, the heterogeneity seen across domains limits the portability of most applications. In order to provide for portability, some cyberinfrastructure projects require application programmers to use infrastructure specific APIs and link their codes with infrastructure specific libraries. Most users, especially those who are not computer experts, would prefer to remain unencumbered by the details surrounding access to resources and do not wish to—or may not be able to—modify or recompile existing codes. We argue that the virtualization of cyberinfrastructure resources, through both virtual machines and virtual networks, can allow resources to be used as if they were resources configured and administered locally.

The VIOLIN middleware interacts with the infrastructure to form the foundation of our integrated cyberinfrastructure framework. It manages the creation, destruction, and adaptation of multiple virtual environments sharing the resources of the infrastructure. Each virtual environment is composed of one or more virtual machines connected through an isolated virtual network. The VIOLIN middleware uses machine and network virtualization to decouple the underlying infrastructure from the virtual environment and provides cyberinfrastructure users with a familiar look-and-feel of a customized private local-area network. Users can execute unmodified applications (both new and legacy) as if they were utilizing dedicated local resources. Meanwhile, the cyberinfrastructure is relieved of its responsibility to maintain per-domain user accounts. Domains can participate in the cyberinfrastructure by providing support of virtual machines and agreeing to host virtual machines from the participating domains.

Although enabling VIOLIN virtual environments is a major contribution in itself, the adaptation abilities of virtual environments provide a unique opportunity to maximize the utilization of computational resources. VIOLIN provides fine-grain control of the amount of resources (CPU, memory, and network bandwidth) allocated to each virtual machine within a VIOLIN environment. Further, it provides coarse-grain control by enabling the live migration of individual virtual machines, or whole virtual environments, between physical hosts or domains.

It is possible to realize VIOLIN environments as integrated, autonomic entities that dynamically adapt and relocate themselves for better performance of the applications running inside. This all software virtualization of distributed computation environments presents a unique opportunity to advance the possibilities of autonomic computing [30, 36]. The autonomic adaptation of virtual computation environments is driven by two main factors: (1) the dynamic, heterogeneous availability of infrastructure resources, and (2) the dynamic resource needs of the applications running inside VIOLIN environments. Dynamic resource availability may cause the VIOLIN environment to relocate its virtual machines to new physical hosts when current physical hosts experience increased workloads. At the same time, dynamic applications may require different amounts of resources throughout their execution. The changing requirements can trigger the VIOLIN to adapt its resource capacity in response to the application's needs. Furthermore, the autonomic adaptation (including relocation) of the virtual computation environment is *transparent* to the application and the user, giving the latter the illusion of a well-provisioned, private, networked run-time environment.

To realize the vision of autonomic virtual environments we address the following challenges:

First, VIOLIN must provide application-transparent mechanisms for adapting virtual environments. In order to provide a consistent environment, adaptation must occur without effecting the application or the user. Currently, work has been done to enable resource reallocation and migration within a local-area network [7] and most current machine virtualization platforms support migration. However, we still need to determine how to migrate virtual machines across a multidomain environment without effecting the application. The solution must keep the virtual machine alive throughout the migration. Computation must continue and network connections must remain open.

The necessary cross-domain migration facility requires two features not yet provided by current virtualization techniques. First, virtual machines need to retain the same IP addresses and maintain network accessibility when physi-

cal routers will not know where they were migrated. Second, cross-domain migration cannot rely on NFS to maintain a consistent view of the large virtual machine image files. These files must be transferred quickly across the network. Clearly, current solutions are not yet adequate for multidomain shared infrastructures.

The second challenge is to define *allocation policies*. Our goal is to move beyond the limits of static allocation and provide autonomic environments that have the intelligence to scale resource allocations without user intervention. As such, we need to determine when a virtual machine needs more CPU, which virtual machine should be migrated, and where to migrate the virtual machine when a host can no longer support the memory demands of its guests. Consequently, we must be able to recognize that the best destination could either be the one to which we can quickly migrate or the one with a longer migration time but more adequate resources.

The main contribution of this paper is to increase the performance of shared cyberinfrastructure through the deployment of autonomic adaptation capabilities of VIOLIN virtual environments. These environments retain the customization and isolation properties of existing static VIOLINs, however, they may be migrated to another host domain during run-time. In this way, we can make efficient use of available resources across multiple domains.

This paper describes a prototype adaptive VIOLIN system using Xen [4] virtual machines and its deployment over our local infrastructure. The evaluation of the system shows that we are able to provide increased performance to several concurrently running virtual environments.

The remainder of this paper is organized as follows: Sect. 2 describes the design of VIOLIN autonomic virtual environments, Sect. 3 presents their implementation, Sect. 4 describes the experiments and presents performance results, Sect. 5 compares our study to related works, and Sect. 6 presents the paper's conclusions.

2 VIOLIN autonomic virtual environments

In the VIOLIN system, each user is presented with an isolated virtual computation environment of virtual machines connected by a virtual network. From the user's point of view, a virtual computation environment is a private cluster of machines dedicated to that user. The user does not know where the virtual machines reside. On the other hand, the infrastructure sees the environments as dynamic entities that can move through the infrastructure utilizing as much or as little resources as needed.

The major components of the VIOLIN system are the enabling mechanisms and the adaptation manager.

2.1 Enabling mechanism—distributed virtual switch

The enabling mechanisms include two sets of daemons, the first of which acts as a distributed layer-2 network switch and the second monitors resource utilization and adapts utilization in response to orders from the adaptation manager.

2.1.1 Distributed virtual switch—functionality

The original motivation for VIOLIN was to enable mutually isolated networks connecting large numbers of virtual machines distributed across several hosts. The target platform was User-Mode Linux (UML) [32]. Like most virtualization platforms, UML provides a mechanism for networking virtual machines in either isolated host-only networks or allowing access to an external physical network. The challenge addressed by VIOLIN is to enable networks that are not limited to a single host but remain isolated from the underlying physical network, as well as any other VIOLIN networks. This challenge was addressed by creating a *distributed virtual switch* composed of daemons running on each host.

Figure 1 depicts a high-level view of the distributed virtual switch and the daemons that compose it. From the perspective of the virtual machines, a virtual switch functions like a physical layer-2 Ethernet switch. Each virtual machine connects to the virtual switch through a virtual port in a similar way to how a physical machine connects to a physical Ethernet switch. Each virtual machine's operating system contains its own network stack which provides traditional application-level abstractions and, ultimately, is responsible for transmitting Ethernet frames between it and the virtual switch. Like a physical Ethernet switch, the virtual switch is to accept Ethernet frames from a source virtual machine and forward the frames to the appropriate destination. The virtual switch achieves this goal by inspecting the destination hardware Media Access Control (MAC) address within each frame and forwarding it out the appropriate port toward the destination virtual machine.

In contrast to a physical switch, the distributed virtual switch maintains a presence on each host that supports a virtual machine in a given VIOLIN environment. This presence is achieved by instantiating a virtual switch daemon on each host (see Fig. 2). Each virtual switch daemon manages the virtual ports through which local virtual machines access the distributed virtual switch. Ethernet frames that are generated by a virtual machine are sent to the daemon which inspects the destination MAC address and forwards the frame toward the destination. If the destination virtual machine is also on the local host, the daemon sends the frame directly to the destination through the appropriate virtual port. Alternatively, if the destination virtual machine is not on the local host, the daemon sends the frames to the virtual switch daemon on the host supporting the frame's destination virtual machine which, in turn, forwards it to the destination.

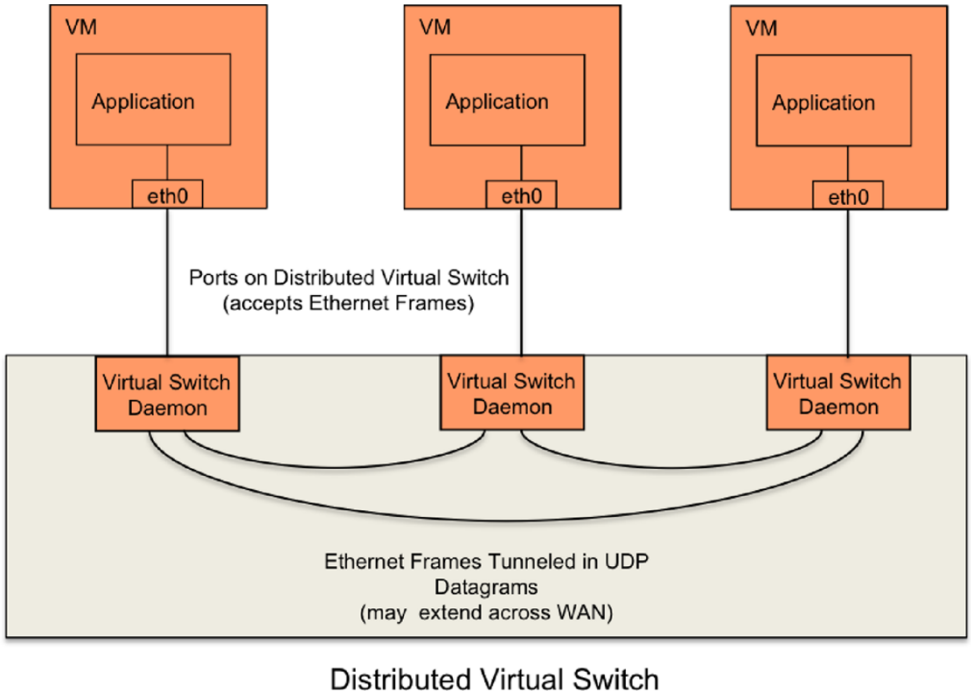

Fig. 1 High-level view of a distributed virtual switch and its internal switch daemons

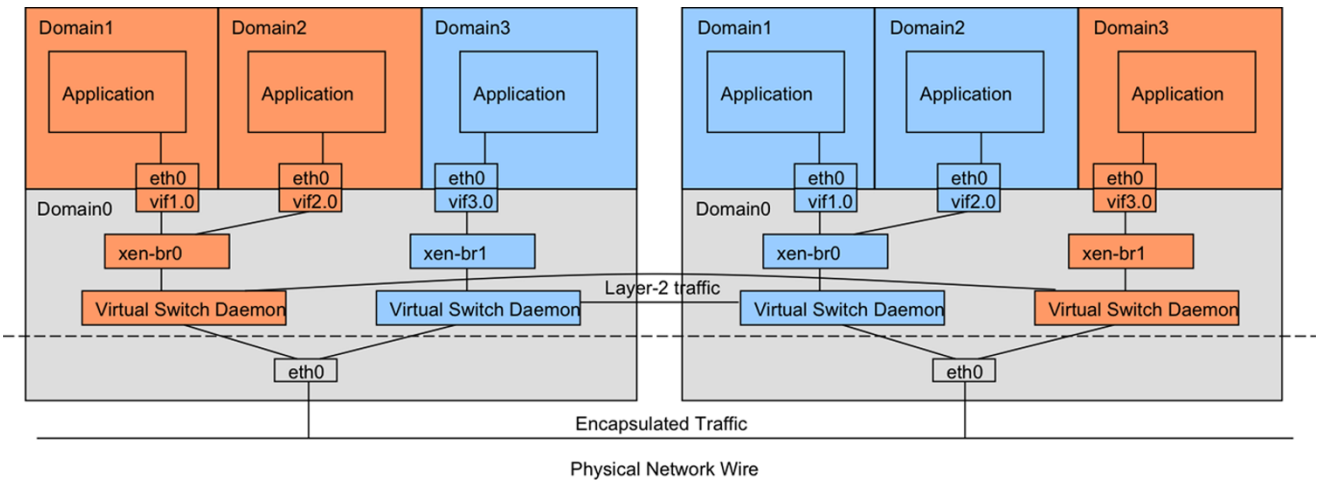

Fig. 2 Multiple mutually isolated VIOLIN virtual environments sharing hosts. Each of two VIOLIN environments is composed of multiple virtual machines called domains in Xen. The virtual machines are connected using distributed virtual switch daemons that act as a single Ethernet switch

2.1.2 Distributed virtual switch—internals

From the point-of-view of the virtual machines the entire distributed virtual switch functions like one layer-2 Ethernet switch; however, coordinating the individual daemons to provided this abstraction requires further explanation. A traditional physical switch contains a table which maps MAC addresses to physical ports. The switch must be able to query this table very fast to achieve good throughput. The switch must also be able to update the table when machines join or leave the network or even move from one physical port to the next. Further, updates are only triggered by observing traditional Ethernet traffic.

The distributed virtual switch works in a similar way, however it is modified to allow for efficient operation across distributed locations. Each virtual switch daemon maintains its own hash table mapping MAC addresses to the port that is the next hop toward the location of the address. As previously stated, if the destination virtual machine is directly connected to the daemon, it sends the frame directly to the destination through the appropriate virtual port. However, if the next hop is a virtual switch daemon on another host, the daemon encapsulates the Ethernet frame in a UDP datagram and sends it to the appropriate remote daemon. VIOLIN can use the low overhead UDP protocol because it emu-

lates a layer-2 Ethernet and is not required to reliably transmit messages. The performance of VIOLIN benefits from using UDP because the connections between switches do not need to be persistent and do not require the transmission and setup overhead of a reliable protocol like TCP.

Internally the switch daemons can be thought of as a set of fully connected independent switches (i.e., each daemon is a switch which has a direct connection to each other switch daemon). Providing a fully connected set of switches daemons limits the maximum path that any frame will travel through the infrastructure to two switch daemons and, therefore, two hosts. Virtual machines residing on the same host will communicate through a single switch while any pair of virtual machines residing on separate hosts will communicate through exactly two switches. This is made possible by the dynamic capabilities of virtualization. Unlike a physically switched network, when a virtual switch daemon is added or removed VIOLIN can dynamically create ports and connections between the new switch and every other switch. VIOLIN can create arbitrary numbers and lengths of connections on-demand. As a result, a set of virtual switches can remain fully connected despite the changing demands of a virtual infrastructure.

Fundamentally, managing theses changes requires updating the hash table within each switch daemon. Just like a physical switch, the distributed virtual switch cannot rely on the host machines to notify the switch when they join or leave a network, or when they move to another port on the switch. Instead, the hash table is updated by inspecting the source MAC address of each frame that passes through the switch. When a frame arrives on a particular port or from another daemon, the MAC address is updated to indicate the, potentially, new next hop. Conveniently, if a virtual machine moves to a new port the mapping will be updated when the first frame arrives on the new port containing the source MAC address.

This property is vitally important to support virtual machine migration. Those familiar with virtual machine migration will know that most virtualization platforms that support live migration transmit unsolicited ARP responses immediately after resuming a virtual machine at its destination. These unsolicited APR responses are sent to the broadcast MAC and will be seen by every switch daemon. Broadcast frames, such as those used to transport ARP messages, are forwarded to all other known virtual switch daemons as well as all local virtual machines except for the source. Since the originating daemon sends broadcast messages to all other daemons, a broadcast message received from another daemon is only forwarded to the local virtual machines and not other daemons (forwarding to other daemons would cause infinite and exponential propagation of the message).

The dynamic nature of the distributed virtual switch and its decentralized passive updates allow it to be efficient and scalable. Previous work has shown that the overhead of using VIOLIN virtual networking with MPI applications is approximately 1.5% [26] when compared with using virtual machines passing traffic through to a physical network. Further, VIOLIN has been shown to scale to tens of virtual machines without an increase in overhead. Although we have not tested VIOLIN on extreme scales, we would not expect the overhead to significantly increase as the number of hosts increases. Communication between virtual machines will travel through at most two hosts and the algorithm for determining the next hop using a hash table is $O(1)$ with respect to the number of virtual switch daemons. The overhead would increase with the number of virtual machines handled by each virtual switch daemon. However, each daemon is only responsible for virtual machines on a single host. Increasing the number of virtual machines per host would add overhead through increased contention on all resources.

It is useful to look at the relationship of VIOLIN virtual networking to the rest of the networking stack. Distributed virtual switch daemons connect to each other through ports that tunnel virtual network traffic (Ethernet frames) within UDP datagrams. With respect to the TCP/IP reference model, the VIOLIN distributed switch comprises the Physical Hardware layer (layer-1) and Data Link layer (layer-2) that support the upper layers in the virtual machine's network stack. Figure 3 shows the network stack as seen by the virtual machines. VIOLIN is implemented at the Link layer and lower with respect to the virtual machines allowing them to deploy any higher-level network protocol (e.g., IP Multicast, IPv6, or experimental protocols).

In contrast, from the point of view of the host infrastructure, the VIOLIN distributed virtual switch and its daemons are at the Application layer. The choice of application layer implementation allows for complete network isolation between the VIOLIN environments and the underlying host network. Network frames are encapsulated within UDP packets and tunneled between the virtual switch daemons. The physical network never directly handles frames from the virtual network. It sends the virtual Ethernet frames as it would any other data.

Isolation between VIOLIN networks is achieved by maintaining strict distinction between the network layers. Figure 2 depicts two hosts, six virtual machines, and two virtual networks. Each host contains two virtual switch daemons, one for each of the virtual networks. Each virtual machine connects to a virtual switch daemon residing on its host determined by the desired network. The switch daemons of each virtual network compose a single distributed virtual switch. This demonstrates how VIOLIN encapsulates virtual network traffic and maintains isolation between the VIOLIN network and the underlying physical network. Network traffic on each virtual Ethernet is isolated from each

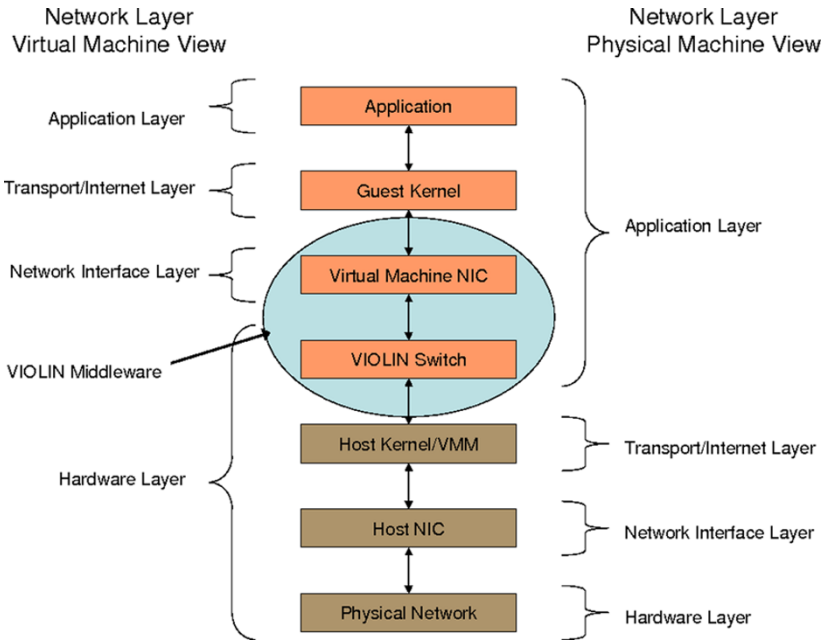

Fig. 3 Location of VIOLIN in the network layers from the view point of both the virtual and physical machines

other as well as the underlying infrastructure. Further, this isolation extends to all higher layers. Each virtual network could create an isolated IP space giving users flexibility to apply any network setting they want without fear of conflict with other virtual networks or the underlying physical network. For example, multiple VIOLIN environments sharing the same infrastructure can use IP addresses (or even MAC addresses) from the same address space.

In addition to the encapsulated network traffic, VIOLIN virtual switch daemons organize themselves using control messages. The set of VIOLIN switch daemons emulates a single layer-2 switch. The control plane is used to organize the daemons and enable them to join and leave the distributed switch. Each switch daemon maintains a control channel to every other switch daemon. The control communication channel is independent of the data plane connections. In order to join the distributed virtual switch, a new daemon must contact a daemon that is a known member of the distributed switch. Once contacted the known daemon will reply with a list of more (most likely all) other daemons. The new daemon will contact each existing daemon to announce its existence.

It is important to note that VIOLIN currently uses a relatively simple peer-to-peer scheme and that many more powerful schemes will work. VIOLIN has all of the benefits and liabilities of modern peer-to-peer techniques. In was not the intention for VIOLIN to contribute in the area of peer-to-peer networking. In terms of performance, the primary contribution of VIOLIN routing is maintaining very short (direct) paths through the data plane between any pair of daemons.

2.1.3 Adaptation mechanism

The *adaptation manager* controls all virtual machines through the *monitoring daemons* (Fig. 4). VIOLIN environments use both memory ballooning and weighted CPU scheduling to achieve fine-grain control over per-host memory and CPU allocation. Both VMware [35] and Xen [4] enable the virtual machine monitor to modify the amount of memory allocated to each virtual machine while the machine is running. At run-time, the *adaptation manager* may decide to modify the memory footprint and percentage of CPU allocated through the monitoring daemons.

An additional contribution of VIOLIN to localized autonomic adaptation is the ability to reallocate resources to virtual machines by migrating them live between network domains. Live virtual machine migration is the transfer of a virtual machine from one host to another without pausing the virtual machine or checkpointing the applications running within the virtual machine. One of the major challenges of live migration is maintaining any network connections the virtual machine may have open. Modern machine virtualization mechanisms provide live virtual machine migration within layer-2 networks [7, 23]. VIOLIN lifts this limitation by creating a virtual layer-2 network that tunnels network traffic end-to-end between remote virtual machines. The virtual network appears to be an isolated physical Ethernet LAN through which migration is possible. As the virtual machines move through the infrastructure, they will remain connected to their original virtual network.

2.2 Adaptation manager

The *adaptation manager* contains the adaptation policy and dictates resource allocation across the infrastructure. The re-

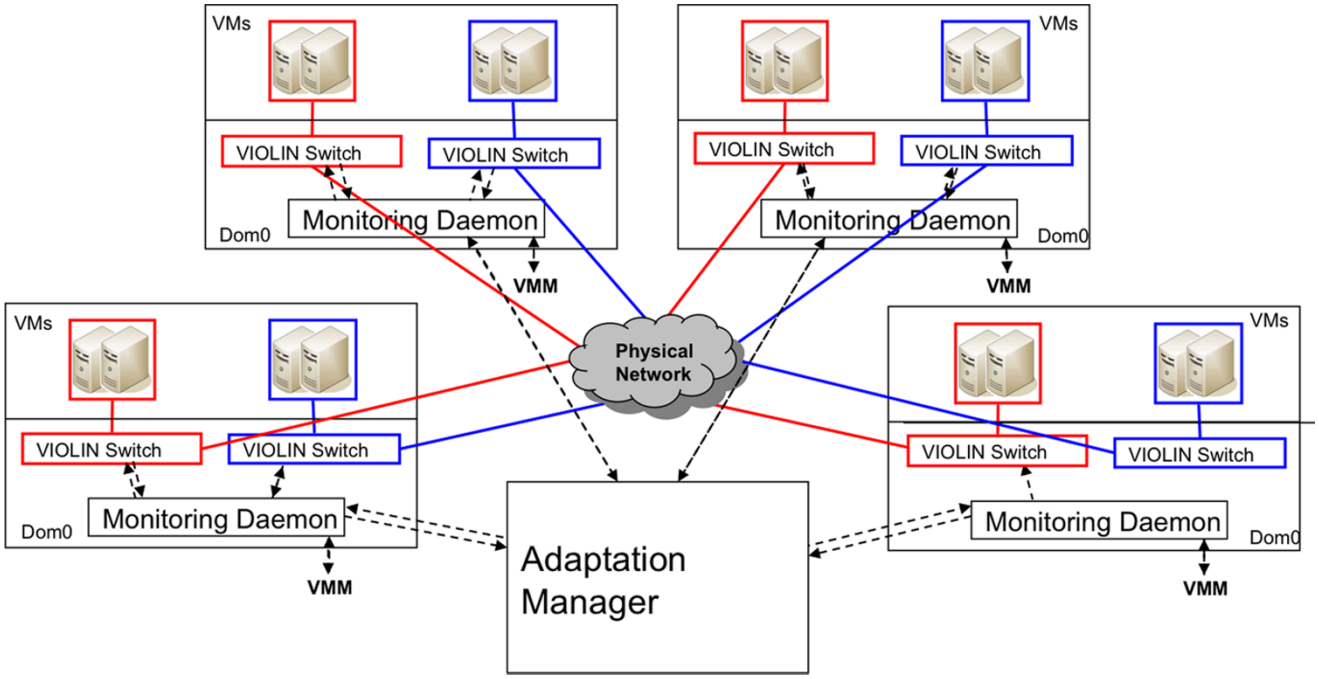

Fig. 4 Multiple VIOLIN environments sharing two hosts. Daemons on each host assist the Adaptation Manager in monitoring and controlling resource allocation

source allocation is determined by a global resource view created from data collected from the resource monitoring daemons.

The *adaptation manager* is the intelligent agent, or "puppeteer" acting on behalf of the users and administrators and making autonomic reallocation decisions. It is appointed two tasks: to compile a global system-view of the available resources and to use this view to transparently adapt the allocation of global resources to virtual environments.

2.2.1 *Infrastructure resource monitoring*

The *adaptation manager* monitors the entire infrastructure by querying the *monitoring daemons* on each host. Via the monitors, it maintains knowledge of all available hosts in addition to the demands of applications running within the VIOLIN environments. Overtime both the resources available in the shared infrastructure and the VIOLIN's utilization of resources will change. Hosts may be added or removed and VIOLINs can be created, destroyed, or enter periods of high or low CPU, memory, or network usage.

2.2.2 *Resource reallocation policy*

The *adaptation manager's* reallocation policy is based on observed host resource availability and virtual machine resource utilization. It uses a heuristic that aims to dynamically migrate overloaded virtual machines between hosts within each domain, and if that is not possible, migrate overloaded VIOLIN environments between domains in the infrastructure. We do not attempt to find the optimal allocation of resources to virtual machines. Instead, we aim at incrementally increasing the performance of the system while minimizing the number of virtual machine migrations and the resulting overhead.

Intuitively, the policy monitors CPU usage of each virtual machine and uses a weighted average to reduce overreacting to temporary spikes in resource demand. The weighted average is used to label each virtual machine as having *high demand* or *low demand* for resources. Each host is given a score equal to the number of *high demand* virtual machines it is supporting divided by the number of cores on the host. The policy's goal is to balance the load on each host by incrementally migrating individual virtual machines when migration reduces the deviation in host scores. In this work, virtual machines are considered to have high demand if their weighted average is over 75% utilization and low demand if it is under 25% utilization.

The heuristic finds over and underutilized host machines and attempts to adjust the virtual machines allocated to the host by first checking other hosts within the local domain. If a local host is supporting more than its share of load, an attempt is made to find another host within the domain to which one or more virtual machines can be migrated. The heuristic first looks at the hosts within the domain that have the lowest utilization level. If no hosts can support the overutilized virtual machine, the whole domain is consid-

ered overloaded and an attempt is made to find another domain which can support the resource needs of one or more of the overloaded domain's VIOLINs. If a destination domain is found, VIOLINs will be migrated live to hosts in that domain.

3 Implementation

We have implemented an adaptive VIOLIN system prototype and have deployed the system on a cluster at the University of Mississippi. The cluster has been divided into two partitions each acting as an independent domain.

3.1 Host infrastructure

The virtual machines are hosted on two independent clusters on separate subnets. Both clusters are composed of Dell 1900s each with 8 GB of RAM and dual 4-core Xeon processors running at 2.66 GHz. Both clusters support Xen 3.4.3 virtual machines and VIOLIN virtual networking.

3.2 Virtual environment configuration

Each virtual computation environment is composed of Xen virtual machines connected by a VIOLIN network. One of the virtual machines is a head node and the rest are compute nodes. The head node provides users with access to the VIOLIN environment and, as such, must remain statically located within its original host domain. However, all compute nodes are free to move throughout the infrastructure as they remain connected via the VIOLIN virtual network.

User accounts are managed by a shared Lightweight Directory Access Protocol (LDAP) server and users' home directories are mounted to the local NFS server with the head node acting as a NAT router for the isolated compute nodes, giving a consistent system view to all virtual machines regardless of the physical locations of the virtual machines.

In order to migrate a virtual machine, the following must be transferred to the destination host: A snapshot of the root file system image, a snapshot of the current memory, and the thread of control. Xen's live migration capability supports efficient transfer of the memory and thread of control. It performs an iterative process that reduces the amount of time the virtual machine is unavailable to an almost unnoticeable level. However, Xen does not support the migration of the root file system image. Xen assumes that the root file system is available on both the source and destination hosts -usually through NFS which cannot safely be made available between multiple domains. The shared infrastructure is composed of independently administered domains which cannot safely share NFS servers. In order to perform multidomain migrations, our prototype uses read-only root images that can be distributed without having to be updated. We do this by putting all system files that need to be written to in *tmpfs* filesystems. Since *tmpfs* file systems are resident in memory, Xen will migrate these files with the memory. Initially, we thought of this solution as a workaround to be fixed later, however, our experience has demonstrated that *tmpfs* can be a reasonable solution for a number of applications. In addition to using *tmpfs* for system files, users' home directories are NFS-mounted through the virtual network to server and do not need to be explicitly transferred.

It should be noted that we have deployed similar VIOLIN infrastructures on multiple domains distributed across the Internet. Specifically, we have deployed the VIOLIN middleware on multiple hosts federated from pools at both the University of Mississippi and Purdue University.

4 Experiments

In this section, we present several experiments that show the feasibility of adaptive VIOLIN virtual environments. First, we measure the overhead of live migration of VIOLIN virtual environments, then we demonstrate application performance improvement due to autonomic live adaptation of VIOLINs sharing a multi-domain infrastructure. For all experiments, we use the VIOLIN prototype, an *adaptation manager* employing the heuristic described in Sect. 2.2.2, and running the High Performance Linpack (HPL) cluster benchmarking software [24] within the virtual environments.

4.1 Migration overhead

4.1.1 Objective

Individual machines can perform quite well while being migrated live due to the iterative copy that allows the virtual machine to remain running on the source host during the migration. The virtual machine's state is iteratively copied from the source to the destination. During each iteration only the changes in state need to be transferred. Once a small enough amount of changed state remains, the virtual machine can be paused, the remaining state is transferred, and the virtual machine is resumed on the destination. The overhead of migration is limited to the increased bandwidth used to transfer the virtual machine's state and the very short downtime during the final iteration. Usually, the downtime is short enough that the migration is unnoticed by the applications and users.

Live migration of an entire virtual environment had additional costs. The difference between migrating a single virtual machine and migrating a set of virtual machine comprising a virtual environment is that there is additional time required for the virtual network to be updated and for network traffic to discover its new path. While the network is

Table 1 Overhead of migrating VIOLIN virtual environments as the number of participating virtual machines increases

	Static (1 host)	Migration (1 host)	Overhead (1 host)	Static (2 hosts)	Migration (2 hosts)	Overhead (2 hosts)
1 vm	1 m 33 s	1 m 37 s	4 s (4%)	N/A	N/A	N/A
2 vms	55 s	1 m 22 s	27 s (31%)	53 s	1 m 39 s	46 s (86%)
4 vms	7 m 6 s	7 m 41 s	35 s (8%)	3 m 59 s	5 m 22 s	1 m 23 s (35%)
8 vms	12 m 43 s	13 m 13 s	30 s (4%)	6 m 50 s	8 m 35 s	1 m 45 s (26%)
16 vms	53 m 13 s	55 m 28 s	2 m 15 s (4%)	26 m 42 s	28 m 42 s	2 m 0 s (8%)

stabilizing some network traffic will be lost or delayed causing high-level network protocols such as TCP to reduce the bandwidth it uses. Any reduction in network bandwidth used can have a temporary but lingering effect on the application in the virtual environment.

The objective of this experiment is to find the overhead of live migration of an entire VIOLIN environment that is actively running a resource intensive application. Individual virtual machine migration overheads have been shown to be approximately 165 ms for a virtual machine with 800 MB memory [7]. To meet this objective, we deploy two different configurations. Both configurations migrate virtual environments and measure the effect on the runtime of an application running inside of an environment due to migration. The first configuration measures the effect as the number of virtual machines in an environment is increased. The second configuration measures the effect as the amount of computation and communication in the environment increases for a given number of virtual machines.

For both configurations, we aimed to record the overhead of migrating an entire VIOLIN virtual environment while it is executing a tightly coupled parallel application (HPL). This is an extreme case where the application has both high amounts of communication as well as a high rate of computation. This application represents the worst case for migrating a VIOLIN virtual environment.

Configuration 1 For this experiment, we record the execution time of High-Performance Linpack using each of several VIOLIN virtual environments composed of increasing numbers of virtual machines. Each virtual environment executes an HPL problem size that is appropriate for its aggregate memory size (200 MB per virtual machine).

For each virtual environment size, we record an average of three measured execution times for each of four cases.

– *Static on 1 host*. In this case, all virtual machines are instantiated on a single 8-core host. The environment is not migrated. This case is used as a control.
– *Static on 2 hosts*. In this case, half of the virtual machines are instantiated on each of two 8-core hosts. The environment is not migrated. This case is used as a control.

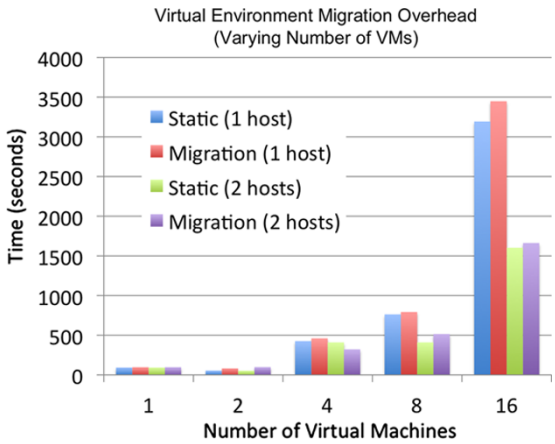

Fig. 5 Configuration 1: Overhead of migrating VIOLIN virtual environments as the number of participating virtual machines increases

– *Migration on 1 host*. In this case, all virtual machines are instantiated on a single 8-core host. The entire environment, including all virtual machines and the virtual switch, is migrated live simultaneously.
– *Migration on 2 hosts*. In this case, half of the virtual machines are instantiated on each of two 8-core hosts. The entire environment, including all virtual machines and the virtual switch, is migrated live simultaneously, however, there are two destination hosts each of which accepts all of the virtual machines from one of the source hosts.

During each run, there is no background load in any of the hosts involved. However, the network is shared and, therefore, incurs background traffic.

4.1.2 Results (Configuration 1)

Figure 5 and Table 1 show the results. We find that migration does add additional overhead to the application. For each HPL job that is migrated, there is a measurable increase in runtime. The runtime penalty increases as the number of virtual machines participating in the virtual environment increases. However, the penalty, in terms of percentage, decreases as the number of participating virtual machines increases.

Table 2 Overhead of migrating VIOLIN virtual environments as the amount of communication and computation increases

Problem size (HPL)	Static (1 host)	Migration (1 host)	Overhead (1 host)	Static (2 hosts)	Migration (2 hosts)	Overhead (2 hosts)
6 K	54 s	1 m 28 s	22 s (41%)	32 s	1 m 51 s	1 m 30 s (281%)
8 K	3 m 12 s	3 m 52 s	40 s (21%)	1 m 45 s	3 m 44 s	2 m 1 s (115%)
10 K	7 m 3 s	8 m 48 s	1 m 45 s (25%)	3 m 48 s	5 m 01 s	1 m 15 s (33%)
12 K	12 m 43 s	14 m 8 s	1 m 23 s (11%)	6 m 44 s	8 m 2 s	1 m 17 s (20%)
14 K	21 m 52 s	23 m 23 s	1 m 31 s (7%)	11 m 29 s	13 m 55 s	2 m 20 s (20%)

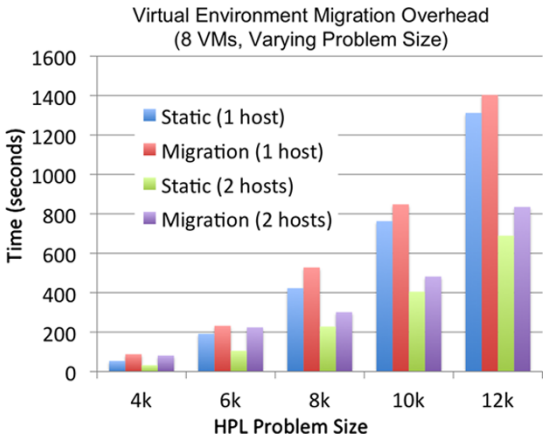

Fig. 6 Overhead of migrating VIOLIN virtual environments as the amount of communication and computation increases

Configuration 2 For this experiment, we record the execution time of High-Performance Linpack using each of several VIOLIN virtual environments composed of eight virtual machines. In this configuration, we do not change the number of virtual machines; instead, we increase the HPL problem size. This aims to find the additional overhead of migration due to increased computation and communication within a virtual environment.

4.1.3 Results (Configuration 2)

Figure 6 and Table 2 show the results. We find that migration does add additional overhead to the application. For each HPL job that is migrated, there is a measurable increase in runtime. The runtime penalty increases as the number of virtual machines participating in the virtual environment increases. However, the penalty, in terms of percentage, decreases as the number of participating virtual machines increases.

4.1.4 Discussion

One requirement of autonomic VIOLIN virtual environments is that there should be little or no effect on the applications due to adaptation. The downtime of migrating an individual machine is minimal due to Xen's iterative live migration mechanism. However, we are interested in the effect downtime has on our application especially with the added overhead of reconfiguring the virtual environment's network.

The data shows that there is a penalty to migrating a VIOLIN virtual environment. However, this penalty is relatively small for large problem sizes and large virtual environment sizes. Additionally, the migration time increases at a much slower rate than the runtime increases as the number of virtual machines increases. One might criticize the experiments for using larger problem sizes in the larger networks to mask the increased overhead of migrating larger environments. It should be noted, however, that the decision to use larger problem sizes on larger environments better represents real usage of such a system. Just as in a traditional cluster, small problem sizes are more efficiently handled by smaller numbers of nodes and are not ideal for large environments. Small problem sizes should not be run on a large number of nodes.

Possibly more significant to the performance of the system is that as the problem size is increased on a given size virtual environment the overhead due to migration remains around 1–2 minutes (at most doubling) for this configuration and application. At the same time, the execution time increases by a factor of twenty. This suggests that the overhead due to reconfiguring the VIOLIN network is not significantly affected by the amount of network traffic and computational demand.

The major effect on application performance is not due to the migration itself but the time to reestablish the VIOLIN virtual network plus application slowdown *during* the migration. This experiment shows that the penalty for migrating a VIOLIN environment is significant but does decrease as a percentage of runtime with increased numbers of virtual machines and larger problem sizes.

4.2 VIOLIN virtual environment adaptation

4.2.1 Objective

This experiment's objective is to demonstrate the effectiveness of the *adaptation manager* and to show how autonomic

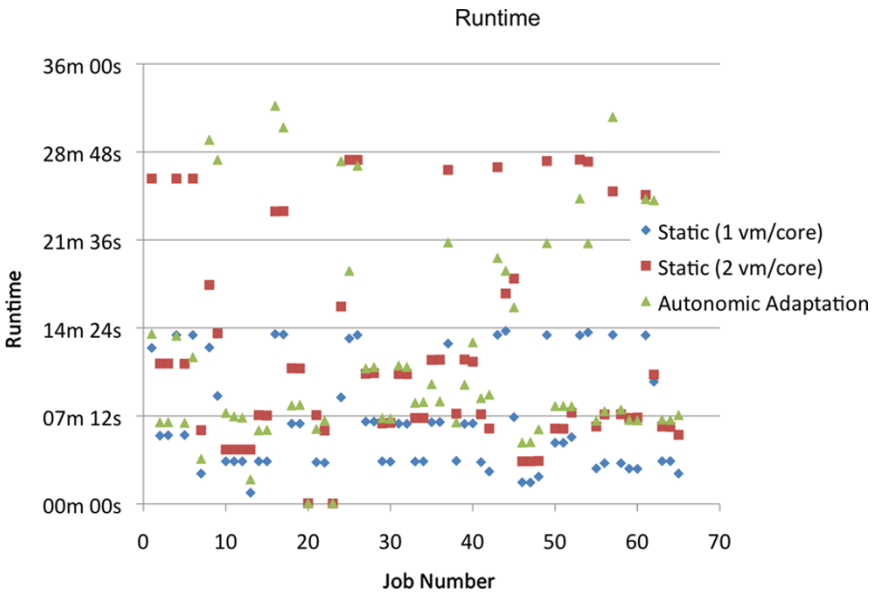

Fig. 7 Runtimes of jobs in each scenario

adaptation can lead to better overall throughput on each VIOLIN virtual environment that shares the infrastructure. Increased throughput is preferred over reduced runtime because users of clouds and other cyberinfrastructures perceive performance as the time between when they submit a job to the time when they receive their results. Users are not affected by the amount of time a job spends running on a machine.

4.2.2 *Configuration*

For this experiment, we launch four VIOLIN virtual environments configured as computational clusters. Each virtual environment deploys an instance of the Torque batch scheduler. We then load each cluster with several HPL jobs of varying sizes and submission times. The goal was to have each cluster run a different mix of HPL jobs. The jobs were chosen to simulate real user submitted jobs of different sizes and submission times. In order to make the experiment more tractable, we chose a number and distribution of jobs such that all would complete in approximately 2 hours of wall-clock time. In this experiment, there were a total of 65 jobs distributed among the four clusters. Each job used between 1 and 16 nodes and ran for between approximately 30 seconds and 30 minutes. Depending on the mix of jobs and the availability of resources, each cluster may go through idle periods.

The virtual environments were deployed on a shared infrastructure that was comprised of two host domains. *Domain 1* has 16 physical cores on two hosts while *Domain 2* has 8 physical cores on a single host. The two domains are subsets of the larger physical cluster that were configured to be on separate subnets. We do not yet have administrative privileges on any machines outside of our campus that can be used for these experiments, therefore, we cannot experiment with truly wide-area infrastructures.

An identical set of jobs and submission times were used for each of three scenarios similar to the previous experiment.

- *Static (1 vm/core)*. The first scenario uses static virtual machines (i.e., no migration), allocated one virtual machine per core with two virtual environments in each of the two domains. Virtual environments 1 and 2 were deployed with 8 virtual machines each on Domain 1. Virtual environments 3 and 4 were deployed with 4 virtual machines each on Domain 2.
- *Static (2 vms/core)* The second uses static virtual machines allocated at two virtual machines per core with two virtual environments in each of the two domains. Virtual environments 1 and 2 were deployed with 16 virtual machines each on Domain 1. Virtual environments 3 and 4 were deployed with 8 virtual machines each on Domain 2.
- *Autonomic adaptation*. The final scenario uses 16 virtual machines per virtual environment, each of which will autonomically be migrated through the infrastructure by the adaptation manager in accordance with the adaptation policy.

The experiment compares the execution time and submission-to-completion time of each job when using each of the three scenarios.

4.2.3 *Results*

The chart in Fig. 7 shows a scatter plot of the runtime of each of the 65 jobs in each of the three scenarios. The x-axis is the job ID while the y-axis shows the runtime. For any particular job in the x-axis, we can see the runtime for each of

Table 3 Average runtime penalty and submission-to-completion time savings due to autonomic adaptation

	Static (1 vm/core)	Static (2 vms/core)
Relative effect of adaptation on runtime	192% (92% penalty)	108% (8% penalty)
Relative effect of adaptation on submission-to-completion time	77% (23% savings)	90% (10% savings)

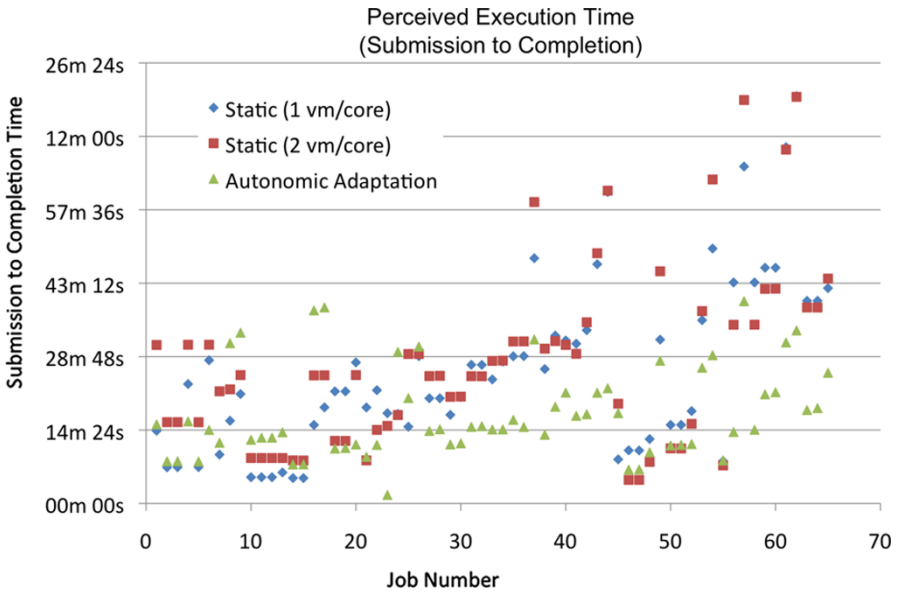

Fig. 8 Submission-to-completion time of jobs in each scenario

the three scenarios. The general trend is that the runtime of the jobs is increased by autonomic adaptation. Table 3 quantifies this observation by showing that the average runtime penalty using autonomic migration is 92% when compared with static allocation of one virtual machine per core and 8% when compared with static allocation of two virtual machines per core.

The chart in Fig. 8 shows a scatter plot of the time from submission of a job to the job's completion. Again, the x-axis is the job ID while the y-axis shows the submission-to-completion time. For any particular job in the x-axis, we can see the submission-to-completion time for each of the three scenarios. Although the previous plot shows that the runtime of any particular job is usually increased, the submission-to-completion time of a job usually decreases when using autonomic adaptation. Table 3 quantifies this observation by showing that the average submission to completion savings using autonomic migration is 23% when compared with static allocation of one virtual machine per core and 10% when compared with static allocation of two virtual machines per core.

4.2.4 Discussion

The interesting idea demonstrated by this experiment is that the average time from when a user submits a job to when he or she gets the result can be reduced significantly by a simple adaptation policy. In this example, each job finished on average between 10% and 23% sooner than it would without adaptation. In what initially seems to be contradictory data, the same jobs ran for significantly longer, however, the shared nature of the infrastructure provided more aggregate throughput and, therefore, better perceived performance from the users' point-of-view.

More advanced adaptation policies are expected to achieve even better results. Specifically, if the application—in this case the Torque scheduler—communicated with the adaptation manager each could potentially make smarter decisions, further increasing the throughput of the system.

5 Related works

Currently, most techniques for federating and managing wide-area shared computation infrastructures apply meta-scheduling of Grid resources as in Globus [10], Condor [31], and In-VIGO [37]. All of these solutions provide access to large amounts of computational power without incurring the cost of full ownership. However, common to all of these systems is that arbitrary parallel/distributed applications cannot run unaltered and jobs run on nodes over which the users do not have administrative control.

In-VIGO is a distributed Grid environment supporting multiple applications that share resource pools. The In-VIGO resources are virtual machines. When a job is submitted, a virtual workspace is created for the job by assigning existing virtual machines to execute it. During the execution of the job, the virtual machines are assigned to the user who has access to his or her unique workspace through the NFS-based distributed virtual file system. Provided with In-VIGO is an automatic virtual machine creation service called VM-Plant [21]. VMPlant is used to automatically create custom root file systems to be used in In-VIGO workspaces. The developers of In-VIGO developed WOW [13] a similar network overlay that utilizes more advance peer-to-peer mechanisms for joining and leaving the systems. Although the networking overlay used by WOW is similar to VIOLIN, WOW does not address autonomic adaption of virtual environments.

Cluster-on-Demand (COD) [6] allows dynamic sharing of resources between multiple clusters. COD reallocates resources by using remote-boot technologies to reinstall preconfigured disk images from the network. The disk image that is installed determines which cluster the nodes will belong to upon booting. In this way, COD can redistribute the resources of a cluster among several logical clusters sharing those resources.

Virtual networking is a fundamental component of VIOLIN. Currently, available machine virtualization platforms do not supply advanced virtual networking facilities. UML, VMware, and Xen all provide networking services by giving the virtual machines real IP addresses from the host network. PlanetLab [5] uses a technique to share a single IP address among all virtual machines on a host by controlling access to the ports. These techniques allow virtual machines to connect to a network but do not create a virtual network. Among the network virtualization techniques are VIOLIN, VNET [29], and SoftUDC [19], all of which create virtual networks of virtual machines residing on distributed hosts. Of particular interest and relevance is VNET, which supports adaptation of network resources [28].

The base VIOLIN infrastructure has been in use for several years and, to the best of our knowledge, was one of the first virtual network underlays for virtual environments. Since the development of VIOLIN several network underlays have been developed and used for a variety of purposes. One of the most interesting applications of these technologies includes the monitoring and adaption of cloud computing resources with the goal of reducing the total amount of electrical power used for computation. Some of these projects include VirtualPower [22], pMapper [33], Brown-Map [34], and Mistral [17]. Although these projects all have different goals VIOLIN addresses a different problem by enabling federation of resources from independent domains. The level of virtualization and isolation provided by VIOLIN allows virtual environments to be migrated live across domain boundaries without the need for remote administrators to manage users and applications within the virtual environments.

The desire to create autonomous environments that adapt to optimize the use of shared resources has lead to the development of many systems that are steps toward this goal. VIOLIN has made a significant contribution to this area but there still remain many open topics in this area of research. One of the primary focus areas of this research is in the development of autonomic algorithms that decide when the benefits of a new resource allocation outweighs the cost of migration. Most of this work is limited to intracluster adaption toward satisfying existing service level agreements [18].

6 Conclusion

We have presented the design and implementation of adaptive VIOLIN virtual environments on top of a multi-domain shared infrastructure. Using VIOLIN's adaptation mechanisms and policies, virtual computation environments can move through the multidomain shared infrastructure and adapt to the needs of their applications and availability of infrastructure resources. The *adaptation manager* acts on behalf of the users and infrastructure administrators to dynamically control the allocation of resources to the virtual environments. Our experiments with deployment of VIOLIN have shown significant improvement in perceived application performance by reducing the submission to completion time by as much as 23%.

Acknowledgements We would like to thank the anonymous reviewers for their constructive comments and suggestions. This work was supported in part by NSF Grants OCI-0438246, OCI-0504261, and CNS-0546173.

References

1. Amazon ec2. http://aws.amazon.com/ec2/
2. Google app engine. http://code.google.com/appengine/
3. Anderson DP (2004) Boinc: a system for public-resource computing and storage. In: Proceedings of the 5th IEEE/ACM international workshop on grid computing, GRID '04, Washington, DC, USA. IEEE Computer Society, Los Alamitos, pp 4–10
4. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In: SOSP '03: proceedings of the nineteenth ACM symposium on operating systems principles, New York, NY, USA. ACM, New York, pp 164–177
5. Bavier A, Bowman M, Chun B, Culler D, Karlin S, Muir S, Peterson L, Roscoe T, Spalink T, Wawrzoniak M (2004) Operating system support for planetary-scale network services. In Proceedings of the 1st conference on symposium on networked systems design and implementation (NSDI'04), vol 1. USENIX Association, Berkeley, p 19

6. Chase JS, Irwin DE, Grit LE, Moore JD, Sprenkle SE (2003) Dynamic virtual clusters in a grid site manager. In: HPDC '03: proceedings of the 12th IEEE international symposium on high performance distributed computing (HPDC'03), Washington, DC, USA. IEEE Computer Society, Los Alamitos, p 90
7. Clark C, Fraser K, Hand S, Hansen JG, Jul E, Limpach C, Pratt I, Warfield A (2005) Live migration of virtual machines. In: Proceedings of USENIX NSDI
8. Dike J (2000) User-mode port of the Linux kernel. In: Proceedings of the USENIX annual Linux showcases and conference
9. Figueiredo RJ, Dinda PA, Fortes JAB (2003) A case for grid computing on virtual machines. In Proceedings of the 23rd international conference on distributed computing systems (ICDCS '03). IEEE Computer Society, Washington, p 550
10. Foster I, Kesselman C (1997) Globus: a metacomputing infrastructure toolkit. Int J Supercomput Appl 11(2)
11. Foster I, Kesselmann C (1999) Globus: a toolkit-based grid architecture. In: The grid: blueprints for a new computing infrastructure, pp 259–278
12. Foster I (2001) The anatomy of the grid: enabling scalable virtual organizations. In IEEE international symposium on cluster computing and the grid (CCGrid'01), IEEE Computer Society, Los Alamitos, p 6
13. Ganguly A, Agrawal A, Boykin PO, Figueiredo RJO Wow: self-organizing wide area overlay networks of virtual workstations. In: IEEE international symposium on high performance distributed computing, pp 30–42
14. Jiang X, Buchholz F, Walters A, Xu D, Wang Y-M, Spafford EH (2008) Tracing worm break-in and contaminations via process coloring: a provenance-preserving approach. IEEE Trans Parallel Distrib Syst 19(7). doi:10.1109/TPDS.2007.70765
15. Jiang X, Xu D (2003) Violin: virtual internetworking on overlay infrastructure. Technical report, Purdue University
16. Jiang X, Xu D, Wang Y-M (2006) Collapsar: a VM-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. J Parallel Distrib Comput 66(9), 1165–1180
17. Jung G, Hiltunen MA, Joshi KR, Schlichting RD, Pu C (2010) In: Mistral: dynamically managing power, performance, and adaptation cost in cloud infrastructures. IEEE Press, New York, pp 62–73
18. Jung G, Joshi KR, Hiltunen MA, Schlichting RD, Pu C (2009) A cost-sensitive adaptation engine for server consolidation of multitier applications. In: Proceedings of the 10th ACM/IFIP/USENIX international conference on middleware, middleware '09, New York, NY, USA. Springer, New York, pp 9:1–9:20
19. Kallahalla M, Uysal M, Swaminathan R, Lowell DE, Wray M, Christian T, Edwards N, Dalton CI, Gittler F (2004) SoftUDC: a software-based data center for utility computing. IEEE Comput 37(11):38–46
20. Kangarlou A, Eugster P, Xu D (2009) Vnsnap: taking snapshots of virtual networked environments with minimal downtime. In: Proceedings of the 39th IEEE/IFIP international conference on dependable systems and networks
21. Krsul I, Ganguly A, Zhang J, Fortes JAB, Figueiredo RJ (2004) Vmplants: providing and managing virtual machine execution environments for grid computing. In: SC '04: proceedings of the proceedings of the ACM/IEEE SC2004 conference (SC'04), Washington, DC, USA. IEEE Computer Society, Los Alamitos, p 7
22. Nathuji R, Schwan K (2007) Virtualpower: coordinated power management in virtualized enterprise systems. In: Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles, SOSP '07, New York, NY, USA. ACM, New York, pp 265–278
23. Nocentino A, Ruth P (2009) Toward dependency-aware live virtual machine migration. In: The proceedings of the 3rd international workshop on virtualization technology in distributed computing (VTDC)
24. Petitet A, Whaley RC, Dongarra J, Cleary A (2004) A portable implementation of the high-performance Linpack benchmark for distributed-memory computers, No. 2387600, http://academic.research.microsoft.com/
25. Pordes R, Petravick D, Kramer B, Olson D, Livny M, Roy A, Avery P, Blackburn K, Wenaus T, Wirthwein F, Foster I, Gardner R, Wilde M, Blatecky A, McGee J, Quick R (2007) The open science grid. J Phys Conf Ser 78(1):012057
26. Ruth P, Jiang X, Xu D, Goasguen S (2005) Virtual distributed environments in a shared infrastructure. IEEE Comput 38(5):63–69
27. Ruth P, McGachey P, Xu D (2005) Viocluster: virtualization for dynamic computational domains. In: CLUSTER 2005
28. Sundararaj A, Gupta A, Dinda P (2005) Increasing application performance in virtual environments through run-time inference and adaptation. In: Proceedings of the 14th IEEE international symposium on high performance distributed computing (HPDC 2005)
29. Sundararaj AI, Dinda PA (2004) Towards virtual networks for virtual machine grid computing. In: Virtual machine research and technology symposium, pp 177–190
30. Tesauroa G, Chess DM, Walsh WE, Das R, Segal A, Whalley I, Kephart JO, White SR (2004) A multi-agent systems approach to autonomic computing. In: Proceedings of the third international joint conference on autonomous agents and multiagent systems, pp 464–471
31. Thain D, Tannenbaum T, Livny M (2004) Distributed computing in practice: the condor experience. In: Concurrency and computation: practice and experience
32. http://user-mode-linux.sourceforge.net
33. Verma A, Ahuja P, Neogi A (2008) pmapper: power and migration cost aware application placement in virtualized systems. In: Proceedings of the 9th ACM/IFIP/USENIX international conference on Middleware, Middleware '08, New York, NY, USA. Springer, New York, pp 243–264
34. Verma A, De P, Mann V, Nayak KT, Purohit A, Dasgupta G, Kothari R (2010) Brownmap: enforcing power budget in shared data centers. In: Middleware, pp 42–63
35. VMware. http://www.vmware.com
36. White SR, Hanson JE, Whalley I, Chess DM, Kephart JO (2004) An architectural approach to autonomic computing. In: Proceedings of the IEEE international conference on autonomic computing
37. Xu J, Adabala S, Fortes JAB (2005) Towards autonomic virtual applications in the in-vigo system. In: Proceedings of the 2nd IEEE international conference on autonomic computing (ICAC-05), pp 15–26