NORTHWESTERN UNIVERSITY

Automatic, Run-time and Dynamic Adaptation of Distributed Applications Executing in Virtual Environments

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Ananth Inamti Sundararaj

EVANSTON, ILLINOIS

December 2006

Thesis Committee

Peter A. Dinda, Northwestern University, Committee Chair
Fabián E. Bustamante, Northwestern University
Aleksandar Kuzmanovic, Northwestern University
José A.B. Fortes, University of Florida, Gainesville

Abstract

Automatic, Run-time and Dynamic Adaptation of Distributed Applications Executing in Virtual Environments

Ananth Inamti Sundararaj

In the last decade we have seen tremendous increases in computer and network speeds and performance, resulting in the emergence of wide-area distributed computing. However, its potential has not been realized, primarily due to lack of security and isolation, provisioning issues and challenges involved in developing distributed applications.

We present the design, implementation and evaluation of a virtual distributed execution environment consisting of operating system level virtual machines connected by virtual networks. Such environments make possible low-level, application-, developer-, and user-independent adaptation mechanisms such as virtual machine migration, overlay topology configuration and routing, network and CPU reservations. This dissertation argues that automatic, run-time and dynamic adaptation in these environments is the key to addressing issues in traditional distributed computing. In particular, it presents the answer to the question, is there a single adaptation scheme that exploits these mechanisms and is effective for a range of distributed applications.

I formalized the adaptation problem and characterized its complexity and hardness of approximation. We found the problem to be NP-hard and hard to approximate. In response to these results, I designed and implemented fifteen variations of greedy adaptation schemes that heuristically attempt to optimize objective functions. I designed these adapta-

tion schemes using insights gained while studying three classes of distributed applications and evaluated them in the context of a different set of seven application classes. I found that a single adaptation scheme that does a variation of load balancing to determine migration of virtual machines and attempts to optimize a function that is a combination of latency and bandwidth for pairs of communicating virtual machines to be effective for 70% of application classes studied. I present two taxonomies for distributed applications based on their resource demands and their suitability to automatic adaptation driven by my suggested adaptation scheme.

This work fills an important gap in distributed systems research by providing an automatic, run-time and dynamic adaptation scheme that leverages the powerful paradigm of virtualization and is effective for a range of unmodified distributed applications running on unmodified operating systems without requiring any developer interaction.

To my parents, Dr. Pushpa Sundararaj and Late Dr. B.I. Sundararaj

Acknowledgments

I would like to express my deep sense of gratitude to my advisor, Prof. Peter A. Dinda, Department of Electrical Engineering and Computer Science, Northwestern University for his invaluable and constant guidance and inspiration which enabled me to complete this dissertation. I express my profound thanks to Prof. Dinda for his keen insight and helpful suggestions. Prof. Dinda was instrumental in helping me maintain a balance between theoretical and empirical systems research. Further, I would like to acknowledge Prof. Dinda's advice on adopting a spiral model to my research, rather than a waterfall model. This work would not have been possible without his constant support.

Further, I have learned a lot about research, teaching, advising and most importantly about computer systems from Prof. Dinda. Prof. Dinda always encouraged me to think and work on projects independently. It is my ambition to be as good a researcher as Prof. Dinda is.

This work was started in the Department of Computer Science and completed in the newly formed Department of Electrical Engineering and Computer Science. My sincere gratitude to Prof. Ming-Yang Kao, previous Chairman of Computer Science and Prof. Bruce Wessels, current chairman of Electrical Engineering and Computer Science for granting permission to carry out this work.

I would also like to express my thanks to my committee for showing confidence in me and for their encouragement and support. In particular, I would like to thank Prof. Bustamante for the many hours of intellectually stimulating discussions carried out on varied topics related to and beyond my research.

The last six months of this thesis work was completed at Intel Corporation's Corporate Technology Group (CTG) in Hillsboro OR. I would like to thank Dr. Mic Bowman for inviting me to Intel Corporation to complete my thesis work and to collaborate with his group. This work has significantly benefited from constructive feedback, brainstorming sessions and insightful discussions with staff members of the Dynamic Distributed Datacenter group at Intel Research. In particular I would like to express my gratitude to Dr. Mic Bowman, Robert Adams, Rob Knauerhase, Paul Brett, Jeff Sedayao, Vivekanathan Sanjeepan, Dr. Rita Wouhyabi and Younggyun Koh.

Parts of this dissertation have relied on deep insights provided by Jeff Sedayao at Intel Corporation's IT Innovation and Research Center. I would like to express my sincere thanks to Jeff for taking the time to discuss the nature of Enterprise IT applications with me. In particular his insights, based on previous studies conducted by him, helped me gain a better understanding of the resource needs of typical enterprise applications. I would also like to thank Vivekanathan Sanjeepan, Jeff Sedayao and Dr. Rita Wouhyabi for reading portions of an earlier draft and for providing constructive feedback.

I would also like to express my sincere gratitude to Prof. Dan Duchamp, Professor, Stevens Institute of Technology, NJ and Prof. Priti Shankar, Indian Institute of Science, Bangalore, India and Prof. K.C. Navada, Manipal Institute of Technology, Manipal, India, for introducing me to systems research.

I would like to acknowledge that parts of this work are joint with others. I co-designed and co-implemented the first two generations of VNET with my advisor Prof. Dinda. John R. Lange is currently working on the third generation VNET design. VTTIF was developed by Ashish Gupta, VRESERVE was by John R. Lange and VSched by Bin Lin, all at Northwestern University. Wren was designed and implemented by Marcia Zangrilli, Anne I. Huang and Prof. Bruce B. Lowekamp at the College of William and Mary. The studies on the theoretical aspects of the adaptation problem was joint work with John R. Lange at Northwestern University and with Manan Sanghi at Microsoft Corporation. The feedback control based scheduling work was joint with Bin Lin at Northwestern University. Selected parts of the adaptation studies were joint work with Ashish Gupta, Marcia Zangrilli, Anne I. Huang and Prof. Bruce B. Lowekamp. The network reservation studies were carried out jointly with John R. Lange. The verification and validation of the virtualized system simulator was carried out independently by Bin Lin. Bin also generated many of the patterns traces that I studied. My entire dissertation has been a collaborative effort with my advisor, Prof. Peter A. Dinda. I would like to thank all my collaborators. Further, I would like to thank current and past distinguished members of the Northwestern Systems Research Group (NSRG) for the many interesting and intellectually stimulating conversations over the past four years.

My friends have been instrumental in encouraging and supporting me during the entire period of this work. In particular, I would like to express my thanks to Himanshu Maheshwari, Amit Goyal and Vivek Pai. I would also like to thank my long time office mate and friend Jason A. Skicewicz, for all his advice and the many interesting conversations over the years. Additionally, I would like to thank my room-mate and good friend, David A. Davidson, for his understanding and encouragement.

My family has been an important and integral part of this effort. I would like to express my gratitude to my Uncles, S.R. Patri, B.I. Nagaraj and Late Dr. B.I. Gururaj, and Aunts, Prema Patri, Pramila Nagaraj and Tara Gururaj. My cousins, Prashant Patri, Pratima Patri, Srikant Nagaraj, Arati Nagaraj, Laxmi Nagaraj, Madhav Inamti, Anjana Inamti and B.I. Gopalkrishna, have supported me throughout.

I owe a lot to my fiancee, Ashwini S. Pai, for her understanding and tremendous support throughout this long journey. She has been my friend and companian over the past couple of years.

This effort would not have been possible without the ever encouraging support and inspiration provided by my brother, Hemanth I. Sundararaj. Personally, I have always attempted to follow in his footsteps and this entire effort was no exception.

Last, but not the least, I would like to dedicate this work to my Mother, Dr. Pushpa Sundararaj, whose courage, will, sacrifice and love have brought me to this point in life. She is and will always be constantly in my thoughts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last few years we have seen a tremendous increase in computer and network performance. Despite this, there are many applications in science, engineering, business and the arts, that cannot be effectively dealt with the current generation of co-located computer clusters and supercomputers. The popularity of the Internet coupled with the availability of low cost commodity computers and high-speed networks have enabled a shift in how computers are used. Computing resources distributed geographically and under different administrative domains can now be harnessed to provide a wide-area distributed computing environment, much more flexible and powerful than any single supercomputer. Apart from supporting high performance computations, a combination of these distributed resources can also be used to host application services or enterprise services [15]. Variants of this new approach have been known by several names such as metacomputing, scalable computing, global computing, Internet computing, grid computing [55] or simply as wide-area distributed computing.

The potential of wide-area distributed computing has not been fully realized due to its inherent complexity, lack of security and isolation, and because of the challenges involved in developing applications for it. This dissertation argues that resource virtualization solves many of these problems and presents opportunities for solving others by providing the building blocks to build virtualized wide-area distributed computing environments.

In a virtualized model of distributed computing, computation is hosted inside of operating system level virtual machines (VMs). These VMs are then connected to each other and to the outside world via virtual networks. Virtual machines and virtual networks provide the same abstraction as their physical counterparts (i.e. physical machines and physical networks). In addition, they also provide security, isolation, flexibility and better utilization, important features lacking in their physical counterparts.

What is required beyond virtual machines and virtual networks is an application independent adaptation scheme that adapts the virtualized distributed computing system to measured available physical resources such that application performance and/or resource utilization is improved.

This dissertation describes the design, implementation and evaluation of adaptation mechanisms, optimization objective functions and heuristic algorithms that provide for an automatic, run-time and dynamic adaptation scheme that leverages the powerful paradigm of virtualization and is effective for a range of unmodified distributed applications running on unmodified operating systems without requiring any developer or user help.

In the remainder of this chapter we first describe the wide-area distributed computing model in Section 1.1. Section 1.2 discusses some of the major issues in this model that have prevented it from realizing its full potential thus motivating this research. Virtual machines and virtual networks, basic building blocks of our system, are described in Sections 1.3 and 1.4. Adaptive virtual execution environments are introduced in Section 1.5 and the design space for the adaptation problem in such environments is discussed in Section 1.6. Section 1.7 provides a brief tour of the remainder of this dissertation. In particular, it gives a high level description of Virtuoso, the adaptive virtualized computing system developed as part of this dissertation. Secondly, it briefly discusses the design decisions made in the context of the adaptation problem. Finally, Section 1.8 describes the outline of this dissertation.

1.1 Wide-area distributed computing

Under the giant umbrella of wide-area distributed computing, there are different variations of computing models. But in all these computing models what is common is that they are comprised of three main entities, resource providers, resource users and the control system which coordinates the resource usage. Resource providers make physical resources such as CPU, memory, hard disk and network devices available by registering them with the control system.Resource users typically look to execute a distributed computation with inter-node synchronization or look to deploy a distributed service, such as a hosted web service or an enterprise-wide service. Such users request for physical resources from the control system. The control system's information service locates a match between the user's needs and the provided physical resources and allocates an appropriate slice of resources at the abstraction level of an operating system user. The user is then able to connect to these provided resources and configure them appropriately with libraries, software environments, etc. Thus deploying their computational task or application service over a geographically distributed set of computational and networking resources [37].

1.2 Issues with traditional wide-area distributed computing

However, the full potential of wide-area distributed computing systems has not been exploited due to a variety of reasons such as complexity of this model, lack of isolation and security, provisioning issues, and, most importantly, the challenges involved in developing applications for such environments.

1.2.1 Complexity in wide-area distributed computing

Wide-area distributed computing aims to seamlessly multiplex, among resource users, distributed computational and networking resources made available by providers. Operating systems multiplex resources in traditional computer systems. For example, a time sharing operating system multiplexes a single microprocessor among different competing processes. Current wide-area distributed computing systems are built upon the abstraction of an operating system user. Firstly, this approach suffers from the limitations of traditional user account models in crossing administrative domain boundaries [129]. Secondly, there is a lot of complexity on both, the user's and resource provider's, ends. Resource providers have to manage and account for resource usage by remote users and limit the impact their resource usage can have on the other users. Resource user's have to deal with potentially complex middleware solutions. Additionally, users also have to manually configure the provided resources with the correct libraries and associated softwares to support their computation and/or service. This can become a time sink as the number of compute resources grow in size and variability of their basic configuration (kernel version, etc.).

1.2.2 Lack of security and isolation

A key to the success of this model of distributed computation on a remote host owned by a third party provider is to be able to isolate the user's computational and network workload from the physical host and from other user's workloads executing on the same host and/or over the same network. Otherwise, this model will be feasible only in limited cases where a two way trust can be established. Without isolation and mutual trust the integrity of a computation or a service may be compromised by a malicious resource [58], and, conversely, the integrity of the resource may be compromised by a malicious user [16]. Though operating systems attempt to isolate user processes, these are not sufficient as the integrity of the user's jobs is not completely isolated from the resources multiplexed by the operating system. Further, if a user's job requires administrative privileges, there will be only a few provider nodes, if any at all, that will agree to give administrative privileges to an unknown user. Finally, this paradigm necessarily involves communication over the network. Isolation and security are difficult to achieve at the network level without having complete control over the network [15].

1.2.3 Provisioning issues

Almost all enterprises host mission critical business applications in dedicated compute clusters or data centers that are geographically distributed from one another [141, 191]. In this model of wide-area distributed computing, the control system, the resource owners and resource users are affiliated to a single organization. Such resources are typically partitioned into isolated islands, each dedicated to a single business application [191]. Additionally, the resources are statically over-provisioned to account for any sudden surges in the application's resource demands. More often than not this over provisioning is achieved manually by dedicated IT staff. All this leads to low utilization and high cost of ownership for the resources [141, 191].

A second scenario wherein low utilization and cost of ownership become issues are compute clusters running scientific batch parallel applications. The cluster of machines could be co-located or spread over a wide area. Currently, such clusters are space shared. Each batch parallel application is assigned a set of nodes and it is the only application which runs on those nodes. The reason is that if two applications execute on the same set of nodes, each with 50% utilization, then there is no guarantee that their performance will be at least 50% of what it was when they were executing in isolation. The solution would be to time-share the cluster with performance guarantees and control leading to better utilization and lower cost of ownership of the resources from the resource owner's perspective. The resource user would get the option of using less resources, costing him less money, yet assured of performance proportional to the resource usage [11].

1.2.4 Challenges involved in developing applications

The wide range of operating systems, specific versions of supporting software packages or software environment variations, in general, have made developing truly portable distributed applications a challenging task. Further, distributed environments are highly heterogeneous and experience wide fluctuations in available network and computational resources. This requires each application to continuously adapt to available system resources to achieve even reasonable performance. Despite many efforts [13, 103, 174, 188], adaptation mechanisms and control are not common on today's applications. This is because they tend to be both very application-specific and require considerable user or developer effort. Custom adaptation by either the application developer (user) or the resource provider is exceedingly complex as the application requirements, computational and network resources can vary over time.

1.2.5 Virtualization to the rescue

We believe that one solution to these problems is to adopt a new paradigm to wide-area distributed computing, namely, virtualized distributed computing, leveraging operating system level virtual machine (VM) technology coupled with a virtualized abstraction of the network. The three inter-related pieces of this paradigm are virtual machines, virtual networks and adaptation. Resource providers still provide physical resources such as CPU, memory, disk and network and resource users still specify their resource requirements as before. However, the control system, instead of simply providing resource slices at the level of an operating system user, presents users with the abstraction of virtual machines interconnected via virtual networks that dynamically adapt to improve application performance.

1.3 Virtual machines

Virtual machine technology dates back to the 1970s when IBM achieved commercial success by introducing virtualization technology thus allowing its mainframes to be used in a time shared manner [67]. Over the next decade interest in hardware virtualization declined partly due to the introduction of affordable low end computers. Despite this, the general notion of resource virtualization has existed throughout, addressing computer systems problems by adding an additional layer of abstraction. Over the past few years we have seen a resurgence of interest in hardware virtualization technology in the form of commercial products from VMware [182] and IBM [86] and increased activity in the research community [51]. This is primarily due to two reasons. First, computers today are more powerful than ever before. Second, the emergence of computational grids and mega data centers have created a need for security, isolation and better resource utilization that are best addressed by virtualization [52]

The virtual machine technology relevant to this work is discussed in detail in Chapter 9. In brief, my work builds on operating-system level virtual machines, more specifically Virtual Machine Monitors, such as VMware [182] and IBM's VM [86]. These present an abstraction identical to a physical machine. For example, VMware provides the abstraction of an Intel IA32-based PC (including memory, IDE or SCSI disk controllers, disks, network interface cards, video card, BIOS, etc.). On top of this abstraction, almost any existing PC operating system environment can be installed and run. The overhead of this emulation has been shown to be less than 5% deeming this abstraction feasible [52, 164].

Virtualization technology such as VMMs can greatly simplify wide-area distributed computing by lowering the level of abstraction from the traditional units of work, such as jobs, processes, or RPC calls to that of a raw machine. This abstraction makes resource management easier from the perspective of resource providers and results in lower complexity and greater flexibility for resource users. A virtual machine image that includes pre-installed versions of the correct operating system, libraries, middleware and applications can make the deployment of new software and application of patches far simpler. These reasons motivate the use of virtualization in wide-area distributed computing. The first detailed case for distributed computing on virtual machines appeared in a previous paper [52]. We have been developing a middleware system, Virtuoso, for virtual machine distributed computing [153]. The Virtuoso architecture is described in Section 1.7. Others have shown how to incorporate virtual machines into the emerging grid standards environment [102].

1.4 Virtual networks

The term virtual network is a very generic one. For the purposes of this work we define the relationship between a physical network and a virtual network to be similar to that between a physical machine and a virtual machine. A virtual network inter-connects virtual machines and the virtual machines to the outside world. The virtual network abstraction is invisible to the user.

Distributed computing is intrinsically about using multiple sites, with different network management and security philosophies, often spread over the wide area [55]. Running a virtual machine on a remote site is equivalent to visiting the site and connecting a new machine. The nature of the network presence (active Ethernet port, traffic not blocked, routable IP address, forwarding of its packets through firewalls, etc.) the machine gets, or whether it gets a presence at all, depends completely on the policy of the site. Not all connections between machines are possible and not all paths through the network are free.

The impact of this variation is further increased as the number of sites is increased and if we permit the possibility of migrating virtual machines from site to site.

To deal with this network management problem in wide-area distributed computing, I designed and implemented VNET [169], a data link layer virtual network. VNET is described in Chapter 3. Using VNET, virtual machines have no network presence on a remote site. Instead, VNET provides a mechanism to project their virtual network cards onto another network, which also moves the network management problem from a remote network to another network of the user's choosing. Because the virtual network is a data link layer network, a machine can be migrated from site to site without changing its presence—it always keeps the same IP address, routes, etc. VNET maintains the abstraction of a physical LAN between a user's virtual machines and the user's physical network. VNET is publicly available.

1.5 An adaptive virtual execution environment

VNET provides a distributed computation or service developer with the abstraction of multiple physical machines all connected to the user's network that can be easily managed, while in reality they are virtual machines, hosted in different geographic locations under different administrative domains. Virtual networks are not just a required means for providing network connectivity to virtual machines, they have tremendous potential to become adaptive overlay networks.

The core of this dissertation describes the design, implementation and evaluation of adaptation mechanisms, adaptation objective functions and heuristic algorithms that provide for an automatic, run-time and dynamic adaptation scheme that leverages the powerful paradigm of virtualization and is effective for a range of unmodified distributed applications running on unmodified operating systems without requiring any developer or user

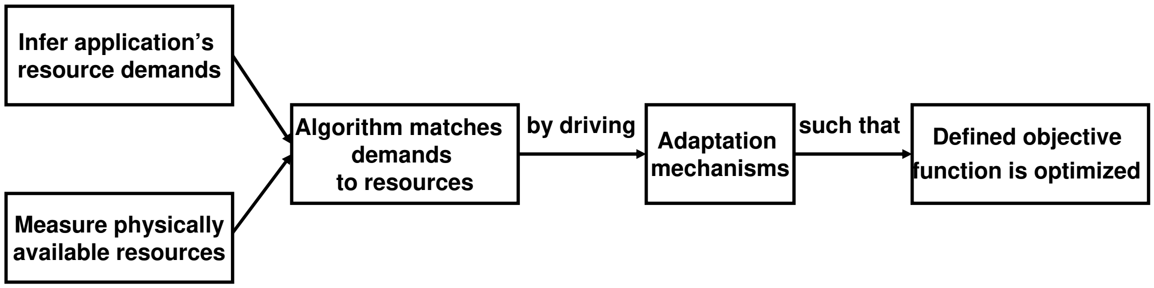


Figure 1.1: The core of a generic adaptive system.

interaction.

1.6 Adaptation problem in virtual execution environments

At a high level, for any system to be adaptive it must comprise of some basic components. Figure 1.1 illustrates a generic adaptation system with its basic building blocks. In this section we explore the design space for the different pieces of the adaptation problem.

1.6.1 Application demands

An application will typically place demands on the available computational resources such as computational speed and utilization, memory, etc. Additionally, it will also make demands on network characteristics such as topology, routing, bandwidth, latency, etc. For example, an application might say that it requires four computational nodes of certain computational speeds each with at least 50% utilization. Further, it might demand that the nodes should be connected via an all-to-all topology wherein the latency among all of them is below a certain threshold. It may also stipulate the expected bandwidth between some of the pairs of communicating nodes.

These demands can either be hard demands or soft demands. Hard demands imply that the specified are essential for the proper operation of the application and that these resources are best reserved for the sole use of the application over a specified duration. Soft demands, as the name implies, are less stringent and act as guidelines that lead to the best operation of the application. Depending on the nature of the system, there might be an additional dimension of cost that the application/user is willing to pay for the demanded resources. Further, these demands could either be explicitly placed by the application/user. Alternatively, the same could be inferred in an invisible manner by the control system. The former has the advantage that the demands will be exact, but requires explicit dialog between the application and the control system. The latter has the advantage that the control system can be invisible to the application, but the inferred demands may not be exact.

1.6.2 Available system resources

In any distributed system, the available computational and network resources keep changing dynamically. It is important for an adaptive system to keep track of all system resources. For a typical distributed system, this has two main components, computational and network resources. Computational resources include the computational speed of the node, memory and disk space available and available node utilization for a particular process. Network resources include the topology between a set of nodes, latency and bandwidth available between pairs of communicating nodes.

The system resource measurement scheme can either be distributed or centralized. A centralized system typically has scalability issues as the system size grows, however, it provides a single global view of the system which makes adaptation decisions easier to make. It should be noted the global view may have a slight lag from the instantaneous resource measurements as all the data has to be routed to a single point in the system. The alternative is to collect resource measurements in a distributed way which eliminates the single point of failure. This allows for a scalable measurement scheme, but the resource information available is always local and an aggregated picture of available resources can be achieved only at a very coarse granularity. Further, the measurements could be made in an active mode or in a passive mode. The former involves making measurements using active probes and the latter achieves the goal using naturally occurring computation and network traffic. Resources, both computational and network, could be available on a best effort scheme or could be available for reservation. Again, similar to the application demands, there might be an additional dimension of cost for all resources on a per unit basis.

1.6.3 Adaptation mechanisms

Beyond the application's demands and the available system resources, the control system must have mechanisms at its command which can change the state of the system. In other words, the system should be inherently adaptable. Adaptation can be either automatic or application driven, runtime or offline and lastly dynamic or static.

- **Automatic or application driven:** Automatic implies that the mechanisms are invisible to the application and driven by the control system. The alternative is to have the application direct and drive the adaptation mechanisms. The former requires no application or user intervention but may not always lead to improved application performance. The adaptation may also be driven by the user. In this model it is the user who gives feedback to the application and directs the control system. Lin et al. showed the effectiveness of user (human) driven adaptation [36, 72, 110, 121].

- **Run-time or offline:** The adaptation mechanisms can be called into play in either an online or an offline fashion. In the former the application is never suspended, while in the latter, the application is suspended for the duration of the issuing and execution of the adaptation instructions. An online scheme leads to better performance and no application downtime, but is significantly more complex than an offline scheme.

- **Dynamic or static:** A dynamic adaptation scheme performs adaptation as soon as change in application demands or system resources is detected or alternatively as soon as adaptation is requested by the application or the user. In a static scheme the change in demands and resource availability and the adaptation timing are asynchronous. On the other hand a dynamic scheme is always susceptible to oscillatory behavior. If the application demands or the system resources change at a rate faster than the reaction time of the adaptation mechanisms, then the system will be driven to oscillations.

1.6.4 Objective function

An adaptation scheme can be constraint driven or a combination of constraints and an optimization objective function. The former is computationally much simpler, where the application places constraints over certain resources and the adaptation system tries to either meet those constraints or notifies the application if any of the constraints are violated. This scheme puts the burden of adaptation on the application with the assumption that the application knows best. This may or may not be the case. Alternatively, the application or the system could define constraints based on application demands and also form an objective function to be optimized. The difference between an application directed objective function and a control system inferred objective function is that the former is known to benefit the application, while the latter is an educated guess which remains to be validated.

It should be noted that adaptation is not always about optimizing a function and going *faster*. Often times adaptation is performed for different reasons such as server consolidation [142], better resource utilization [11, 15] or to meet user requirements on price for resources usage [11]. There is a subtle tension between application centered adaptation and system centered adaptation. System centered adaptation might lead to higher system utilization as a whole, but may end up with reduced performance for a specific application. Reduced per application performance is not always undesired. If the reduction in application performance is proportional to the application's resource usage and is guaranteed, then it creates an opportunity for the application owner to pay less yet have a guaranteed performance bound for his application.

1.6.5 Adaptation algorithm

Finally, what is required to build an effective adaptation scheme is an efficient algorithm that takes into account application demands, system resources and drives the adaptation mechanisms such that all constraints on system resources are met and a given or inferred objective function is optimized. This Ultimately leads to better application and/or system performance.

The specific algorithm is heavily dependent on the exact adaptation problem formulation. Dependent on the problem's computational complexity and hardness of approximation, it may or may not be possible to efficiently solve the adaptation problem. In such a case heuristics will need to be leveraged. If adequate computational resources are available, strategies such as genetic programming or simulated annealing could also be successfully employed.

1.6.6 Generality of adaptation solution

Adaptation schemes can either be application specific or generic that are applicable to a range of application classes. The possibility of a single optimization scheme successfully applicable for every single, distributed application written or to be written is highly unlikely. However, a single optimization scheme with tunable parameters might be applicable to a range of applicable classes. There is a clear trade-off between specificity and

Figure 1.2: Virtuoso front end.

generality. A solution specific to an application is highly potent but not widely applicable. On the other hand, a generic solution is widely applicable but its potency is slightly mitigated for certain classes of applications.

1.7 Virtuoso: A virtualized distributed computing infrastructure

To address the issues in traditional wide-area distributed computing and to realize the full potential of this powerful computing paradigm, we are developing Virtuoso, a virtualized, adaptive, distributed computing infrastructure, which we describe next. A user wishing to perform a computation or looking to deploy a service, instead of purchasing physical hardware from a vendor such as IBM [85] or Dell [31] or compute time on a computational grid or a cycle sharing system [166], can now purchase virtual machines from a website. Figure 1.2 illustrates the Virtuoso web front-end.

In the Virtuoso model of distributed computing, there exists a marketplace for virtual

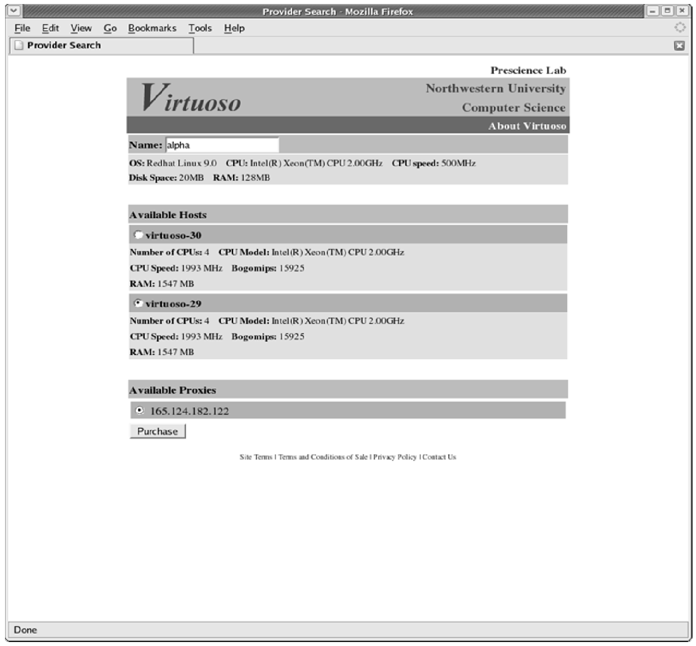


Figure 1.3: Resource provider interface in Virtuoso.

machines. The resource provider registers his physical machine(s) and specifies service rates for its resources such as CPU, memory, disk, etc. on a per unit basis. Figure 1.3 illustrates the interface as seen by a resource provider. A resource user looking to purchase a virtual machine(s) specifies the constraints for his virtual machine(s) in terms of service rates ($/GBhour$, etc). Virtuoso then acts as a broker and matches resource providers and users based on their specified constraints and hosts the user's virtual machines on the matched physical hosts.

1.7.1 User experience in Virtuoso

Virtuoso, for a user, very closely emulates the existing process of buying, configuring, and using an Intel-based computer or collection of computers from a web site, a process with which many users and certainly all system administrators are familiar.

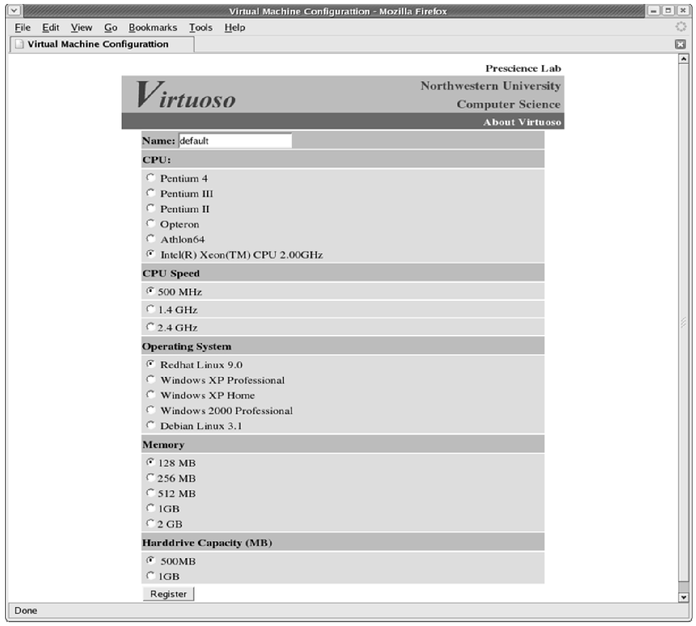


Figure 1.4: Virtuoso user interface.

In our model, the user visits a web site, much like the web site of Dell [31] or IBM [85] or any other company that sells Intel-based computers. Figure 1.4 illustrates the interface a user sees. The site allows him to specify the hardware and software configuration of a computer and its performance requirements, and then order one or more of them. He receives a reference to the virtual machine which he can then use to start, stop, reset, and clone the machine. The system presents the illusion that the virtual machine is right next to the user, in terms of console display, devices, and the network. The console display is sent back to the user's machine, the CD-ROM is proxied to the user's machine's CD-ROM, and the virtual machine appears to be plugged into the network side-by-side with the user's machine. The user can then install additional software, including operating systems. The control system is permitted to move a virtual machine from site to site to optimize its performance or cost, but must preserve the illusion of locality. More details about the current Virtuoso front-end implementation are available in a previous document [153].

We use VMware GSX Server [182] running on Linux as our virtual machine monitor. Although GSX provides a fast remote console, we use VNC [144] in order to remain independent of the underlying virtual machine monitor. We proxy CD-ROM devices using Linux's network block device, or by using CD image files. Network proxying is done using virtual networks, as described later in this section and in Chapter 3. It should be noted that VMware is not a requirement of either Virtuoso or any of the work detailed in this dissertation.

1.7.2 Benefits of Virtuoso's model of distributed computing

Virtuoso's model of distributed computing addresses all the issues with traditional wide-area distributed computing paradigm.

Reduction in complexity

From the resource provider's perspective, it is simple to sell resources packaged as VMs to buyers. This abstraction is lower than the current models of RPC, distributed shared memory, processes, threads and jobs in wide-area distributed computing and cycle stealing [17, 113, 166] systems, hence avoiding software complexity issues at the operating system level. Now all the required libraries, packages, etc. can be conveniently packaged into a virtual machine and sold. Further, checkpointing and cloning of virtual machines is well understood, making creation and customization of virtual machines straightforward.

This abstraction also simplifies life for a resource user. The user does not have to deal with potentially complex middleware solutions and instead is faced with the familiar abstraction of a raw machine connected to his local network and administered by his local network administrator. It is a common belief that lowering the level of abstraction increases performance while increasing complexity. In this particular case, the rule may not apply. Our lowered abstraction for the user is identical to his existing model of a machine or a group of machines.

Better security and isolation

This model also solves the problem of providing security and isolation. From the resource provider's perspective each user is constrained to a virtual machine. Each virtual machine is isolated from each other via mechanisms provided by typical VMMs [9, 182]. Further, malicious code trapped inside of a user's virtual machine now has to break through two levels of security, VMMs and operating systems. Further, accounting for resources consumed by a user, is also simplified [15]. From the user's perspective, this provides freedom from administrative policies at the remote site. The user has administrative privileges and further, can also ask for virtual machines pre-configured with basic software, alternatively, can configure one virtual machine and clone it to produce multiple instances and finally fine-tune each instance as required.

Solution to provisioning issues

Virtuoso allows for resource reservations, both, at the CPU and network levels. The resource reservations are coupled with a feedback control system to guarantee the application's performance to be proportional to its resource usage. This allows for the time-sharing of distributed or co-located compute resources leading to better utilization and lower cost of ownership of the resources.

The bulk of this dissertation explores adaptation schemes that require no user input. However, for certain cases, where application needs cannot be inferred, we illustrate how with minimal application interaction, powerful adaptation schemes can be designed and implemented.

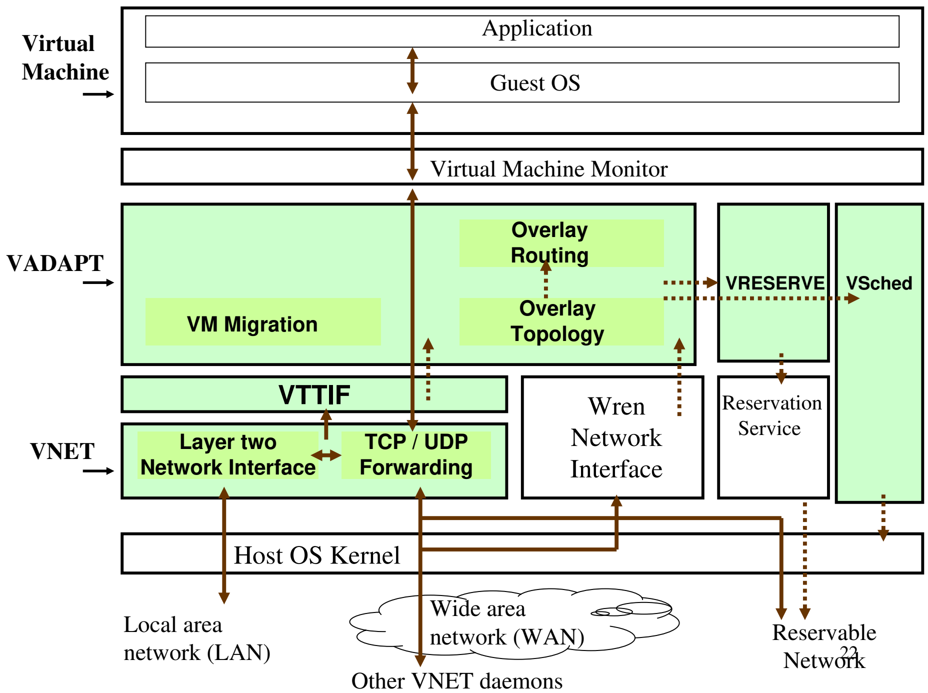


Figure 1.5: System overview. Dashed lines indicate control traffic, solid lines denote actual network traffic. The highlighted boxes are components of Virtuoso.

Automatic, run-time and dynamic adaptation

The core of this dissertation illustrates how automatic run-time and dynamic adaptation of un-modified applications running on un-modified operating systems helps realize the full potential of wide-area distributed computing. The onus of adaptation is moved from the developer/application owner to our control system. This simplifies application development and increases the possibility of wide spread adoption of this powerful computing paradigm in future.

1.7.3 Virtuoso system architecture

We next describe the Virtuoso system architecture. Figure 1.5 illustrates the system schematically.

VNET: Virtual network

VNET is the virtual networking component of Virtuoso. It operates at the Ethernet level and creates the abstraction that all of a user's machines are on the user's LAN. The fact that a user's virtual machines are hosted on third-party foreign LANs is completely invisible to the user. VNET is described in Chapter 3.

VTTIF: Virtual topology and traffic inference framework

VTTIF is integrated with VNET and looks at Ethernet packets to infer the communication traffic matrix. VTTIF can also detect dynamic topology changes and is fitted with mechanisms to prevent oscillations when the topology changes at a rate faster than the reaction time of the adaptation mechanisms. VTTIF is described in Chapter 3 and Appendix A.

Wren: Watching resources from the edge of the network

Wren was developed at College of William and Mary and has been integrated with Virtuoso to provide network measurement estimates. Wren attempts to use the naturally occurring traffic between the virtual machines to make passive network measurements. Wren is described in Chapter 3 and Appendix A.

VADAPT: Adaptation mechanisms

VADAPT is Virtuoso's adaptation engine. It is a collection of application independent adaptation mechanisms that are made possible by the virtualization abstraction and an adaptation scheme to drive them. VADAPT is described in Chapters 3, 7 and 8.

VRESERVE: Network reservations

In the absence of resource reservations, adaptation can be only best effort. VRESERVE is Virtuoso's component, that detects if network reservation is available and leverages the same when possible to improve and guarantee communication characteristics between pairs of communicating VMs. VRESERVE is described in Chapter 5.

VSched: Periodic real-time scheduling

VSched is a periodic real-time scheduler that controls the utilization a VM gets by controlling the *slice* it gets every *period* seconds. The challenge is to ensure that for an application, running inside of a VM, its performance is proportional to the utilization it receives. VSched is described in Chapter 6.

1.7.4 Adaptation in Virtuoso

A system like Virtuoso can be used to execute a variety of computational centric applications and network and application services. We attempt to infer application demands, measure available system resources, adapt the system using available adaptation mechanisms and reservation schemes such that application performance is improved (or some application specified target is met) and/or some objective function is optimized. Figure 1.6 illustrates the Virtuoso adaptation system with its basic building blocks. Notice the similarity with Figure 1.1. This effectively illustrates how Virtuoso has all the building blocks required of an successful adaptive system. We next describe some of the high level contributions of this dissertation. A detailed discussion of the contributions is presented in Chapter 9.

Computational complexity

The generic adaptation problem in such virtual execution environment turns out to be quite complex. This dissertation presents a formalization of the adaptation problem and proofs that it is NP-hard and NP-hard to approximate within a certain factor of the optimal. As far as we are aware, this is the first such formalization of a problem that includes both

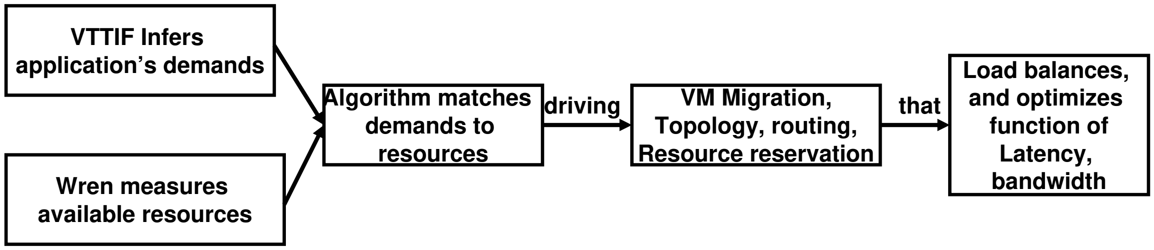


Figure 1.6: The core of Virtuoso adaptive system.

mapping and routing components. Additionally, it is also the first attempt at analyzing the complexity and inapproximability of such problems. This work has the potential to be used as the basis for reduction and analysis for future adaptation problems in virtual execution environments.

Resource reservations

Adaptation schemes that take into account network and CPU reservations were studied in the course of this work. In particular, this dissertation illustrates how Virtuoso can detect if the underlying network provides network reservation mechanisms and leverage them to reserve resources between pairs of communicating VMs. Further, I illustrate how Virtuoso can reserve CPU resources that a particular VM on a physical host gets and ensure that the application inside a set of VMs experience performance proportional to the CPU utilization the VMs receive. This work has the potential to improve utilization and reduce cost of ownership of resources in compute clusters and data centers.

Optimization schemes

In response to the computational complexity results, I have explored the space of heuristic solutions. In particular, I devised six different incarnations of the generic adaptation problem, each with a different objective function. I designed and implemented fifteen different variations of greedy adaptation algorithms that heuristically attempt to optimize the different objective functions. I evaluated these different optimization schemes in the context of ten different classes of distributed applications spanning from scientific applications to transactional web applications to enterprise IT applications. A single optimization scheme with tunable parameters seems to be most effective for majority of the distributed applications studied (seven out of ten). This suggested algorithm does a variation of classic load balancing to determine migration of virtual machines to minimize run-time and load and then attempts to optimize a function that is a combination of latency and bandwidth for pairs of communicating virtual machines. This work has the potential to lead a faster and more wide-spread adoption of the wide-area distributed computing paradigm by moving the onus of adaptation from the developer to the control system, hence simplifying application development.

Application taxonomy

This dissertation provides a taxonomy of the distributed applications studied in the context of this work. In particular the applications are classified as those for which a single optimization scheme is effective (seven out of the ten applications classes studied), those for which the application needs to provide some feedback/input and those for which a single optimization scheme does not work. An additional taxonomy is provided based on the compute and network requirements of the application classes. This work heavily relies on insights and previous work done at Intel Corporation's IT Innovation and Research Group [90]. The results described in this dissertation can be leveraged in the design of future adaptive systems targeted for specific requirements [15].

Experimentation, simulation and analytical studies

There are three main techniques for evaluating and studying a computer system: experimentation, simulation and analytical modeling. Most studies leverage one or two of these techniques. The exact methodology chosen for any case depends on a number of factors: life-cycle stage in which the system is, time available for evaluation, availability of tools, level of accuracy desired and allocated cost [93]. With the aim of studying the adaptation problem comprehensively, this dissertation leverages all three of these techniques. A combination of these techniques is necessitated by the inherent qualities and restrictions of each of the techniques.

1.8 Outline of dissertation

Virtuoso is a distributed adaptive systems for executing distributed applications. We begin this dissertation in Chapter 2 by discussing ten diverse classes of distributed applications. The contents of this chapter draw heavily on insights gained during discussions with researchers at Intel Corporation. We use these application classes to motivate the need for the search of single adaptation scheme with wide applicability to diverse application classes.

The remainder of this dissertation primarily describes the different Virtuoso components depicted in Figure 1.5 moving from the bottom to the top.

Chapter 3 describes the design, implementation and evaluation of our virtual network VNET. It starts off by providing the motivation and justification for the first generation VNET. This is followed by some performance analysis of the same. The second generation of VNET is then described, with particular emphasis on bottleneck elimination, greater performance and enhanced flexibility. We then describe Virtuoso's application inference and network measurement systems, VTTIF and Wren respectively. The focus is on how these have been integrated with VNET and provide the tools to make automatic adaptation invisible to the application and user a reality. Further, this chapter also describes some of the application independent adaptation mechanisms that the abstraction of virtualization makes possible.

The problem formulation is presented in Chapter 4. A generic incarnation of the adaptation problem applicable to most virtual execution environments is described. It then analyzes the computational complexity of a simplified version of the problem and provides proof to show that the problem is NP-hard. It then presents results which show that the problem is also NP-hard to approximate within a factor of $m^{1/2-\delta}$ for any $\delta > 0$, where m is the number of edges in the virtual network graph.

Chapter 5 describes the design, implementation and evaluation of the VRESERVE component of Virtuoso. VRESERVE can automatically detect if the underlying network provides resource reservation mechanisms. In particular we deployed Virtuoso over a real distributed optical testbed and showed the effectiveness of our approach at a proof of concept level.

Though this dissertation is about automatic adaptation that does not require user or application interaction, Chapter 6 studies a scenario wherein application inference and hence automatic adaptation is not possible and what we can still do in such a scenario. In particular, it studies the case wherein a user or system administrator desires a distributed application to consume a specified amount of CPU resource defined in terms of utilization. I tried to answer the question, is it possible to come up with a feedback controlled periodic real-time scheduling scheme such that the application's performance measured in some application defined units is proportional to the utilization its components receive? Though this work was done in the context of parallel applications executing in tightly coupled environments such as clusters, it is also applicable to applications executing inside of virtual machines.

Following up on the complexity and inapproximability results derived in Chapter 4, Chapter 7 describes six different optimization functions and fifteen different variations of adaptation schemes driven by greedy heuristics. Out of the ten application classes discussed in Chapter 2, we used three to design our adaptation schemes. The remainder seven were used for evaluating the adaptation schemes (Chapter 8). Further, we also describe the proof of correctness for some of the algorithms that are not very intuitive. Finally, we present a detailed comparison of all algorithms against each other and the optimal, where the performance of each adaptation scheme is measured in terms of optimizing objective functions.

The bulk of this dissertation's results are presented in Chapter 8. First, I describe our physical distributed testbed and present the overheads of our adaptation system. Second, I describe the experiments conducted on two out of three classes of applications used in the design of the adaptation schemes. Next, I describe, verify and validate a system simulator that mimics the real system within certain limits and based on certain assumptions. This simulator was used to study the effectiveness of the adaptation schemes for the remainder seven application classes. In this case the performance of the adaptation schemes was measured in terms of improving values of application specific metrics. I find that an algorithm with tunable parameters that performs VM migrations based on compute requirements and demands and modifies the overlay topology and routes to meet latency and bandwidth requirements works well for the majority of the application classes studied (seven out of ten). I also present two taxonomies for distributed applications and make recommendations for performing automatic, run-time and dynamic adaptation.

Chapter 9 concludes the dissertation by summarizing the contributions of this dissertation and by comparing and contrasting it against existing literature. It particular I describe how it relates to other work in this area and how it distinguishes itself from the same. Further, we also describe some future work directions. One direction is to relax the notion of automation and involve an user in the adaptation loop to address some of the applications which could not be addressed with the single optimization function [110]. Another direction is to explore combinatorially approximate algorithms with a view to improving the adaptation quality and at the same time being able to provide some performance guarantees. A third possible direction is to create a constraint based system wherein the application provides all the constraint and the system attempts to either meet the same or notify the application upon failure [15, 90].

Appendix A describes VTTIF, the Virtual Topology and Traffic Inference Framework of Virtuoso. In particular, it describes the extensions made to support dynamic topology changes. VTTIF has been fitted with mechanisms which prevent oscillations when the topology changes at a rate faster than what Virtuoso can adapt to.

Wren, a passive network measurement tool is described in Appendix B. Wren was developed by Zangrilli, et al. at the College of William and Mary. Wren has been successfully integrated with Virtuoso. Wren uses the naturally occurring traffic of a user's VMs to make passive network measurements. Wren has an offline and an online mode of operation. Appendix B focuses on the online incarnation of Wren.

Chapter 2

Distributed application classes

This dissertation has a focus on automatic, run-time and dynamic adaptation of distributed applications. In the course of this research we have studied ten diverse classes of distributed applications. These distributed application classes are introduced in this chapter. We briefly describe these application classes and discuss why adaptation is an important issue for all these application classes.

These application classes (listed in Table 2.1) fall into three main categories, high performance scientific computations, transactional web e-commerce applications and enterprise IT applications. We owe our insights and knowledge about enterprise IT applications to Intel Corporation's IT Innovation and Research Group [90] (referred to as Intel IT Research from now onwards). Intel Corporation carried out an extensive study to understand the resource needs of its top ten IT applications [15]. My research draws upon the results gained in that study. I held a series of meetings with researchers at Intel Corporation's Corporate Technology Group (CTG), Hillsboro, OR and the IT Innovation and Research Group, Santa Clara, CA. The aim of these discussions was to understand the diversity of resource demands made by a wide range of distributed application classes.

Virtuoso's model of distributed computation, described in Chapter 1, fits in well with the needs of high performance scientific computations. These applications typically make

Number	Application classes
1.	High performance scientific applications
2.	Transactional web e-commerce applications
3.	Enterprise backup applications
4.	DNS
5.	SMTP mail application
6.	Enterprise wide-area file transfer applications
7.	Enterprise compute-intensive simulations
8.	Enterprise database applications with small transfers
9.	Enterprise database applications with large transfers
10.	Enterprise engineering computing applications

Table 2.1: Ten distributed application classes.

giant computation demands which are difficult to meet with a set of co-located computer clusters. Virtualized, adaptive and distributed execution environments make these computations easier and also create potential to improve performance via dynamic adaptation. Virtuoso's model also fits web transactional e-commerce applications as these applications are currently built using specific orchestrators. Successful adaptation via Virtuoso, we will remove this burden from the shoulder of the developers decreasing the time taken in developing such applications.

At first glance, Virtuoso's model of distributed computation might not seem to fit in with enterprise application needs. However, this is not true. An enterprise can be looked upon as a geographically distributed collection of sites and LANs. The current army of physical machines could be replaced with virtual machines hosted on consolidated servers. These VMs, though physically hosted in different data centers, would still maintain the LAN illusion via VNET. In an enterprise setting resource providers and resource users are typically affiliated with the same organization. Automatic adaptation would complete the picture and create an IT and computing infrastructure that mirrors the existing setup, but comes with the added benefits of less complexity, better security and isolation, better server provisioning and improved performance. With the recent push towards virtualization in enterprises [15], a study like this assumes an important role.

We next describe the ten application classes studied and present the high level insights gained from discussions with researchers at Intel Corporation.

2.1 High performance scientific computations

The paradigm of wide-area distributed computing grew out of the ever increasing computational needs of the high performance scientific community. Typical scientific applications and simulations can have computational requirements beyond the capabilities of any single co-located, dedicated cluster of computers. An alternative is to harness the unused computational power of commodity low end computers spread geographically all over the world to create the abstraction of a single machine much more powerful than any Supercomputer [57, 114, 166]. However, as these compute and network resources spread across wide-area networks, adaptation issues become prominent again. Virtuoso's model of distributed computing is ideally placed to service such applications.

We studied a series of real and synthetic high performance computing benchmarks including NASA's NAS benchmark suite [185], persistence of vision ray tracing benchmark POVRAY [33] and a home grown synthetic benchmark that models Bulk Syncronous Parallel (BSP) [180] applications, called Patterns. We next briefly describe each of these applications.

2.1.1 NAS benchmark suite

The NAS (NASA Advanced Supercomputing) benchmark suite was developed to test the performance of parallel supercomputers. The different components of the benchmark suite are designed to mimic the computational and data transfer behavior of large scale computational fluid dynamics (CFD) based applications. This benchmark consists of the following components:

- **EP:** An embarrassingly parallel kernel. It provides an estimate of the upper achievable limits for floating point performance i.e. the performance without significant interprocessor communication [185].
- **MG:** A simplified multigrid kernel. It requires highly structured long distance communication and tests both short and long distance data communication [185].
- **CG:** A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large sparse symmetric positive denite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication employing unstructured matrix vector multiplication [185].
- **FT:** A partial differential equation solution using FFTs. This kernel performs the essence of many "spectral" codes. It is a rigorous test of long distance communication performance [185].
- **IS:** A large integer sort. This kernel performs a sorting operation that is important in particle method codes. It tests both integer computation speed and communication performance [185].

We use the PVM [63] implementation of the NAS benchmarks as developed by Sunderam et al. [187].

2.1.2 Persistence of vision ray tracer

We also studied POVRAY, a popular ray tracing technique. In particular we used the PVM version of the popular ray tracer POVRAY [33]. The PVM version gives it the ability to distribute a rendering across multiple heterogeneous systems [71].

2.1.3 Patterns

Patterns: is a synthetic workload generator that captures the computation and communication behavior of Bulk Synchronous Parallel (BSP) [180] programs. Patterns emulates a BSP program with alternating dummy compute phases and communication phases according to the chosen topology operation, and compute/communicate ratio. The reason for choosing a synthetic benchmark is to allow ourselves the flexibility of quickly changing parameters to study different scenarios in a short period of time. In particular, we can vary the number of nodes, the compute/communicate ratio of the application, and select from communication operations such as reduction, neighbor exchange, and all-to-all on application topologies. Patterns is described in a previous paper [71].

2.2 Transactional web e-commerce applications

With the ever growing popularity of the web, transactional web e-commerce applications are increasing in importance. Most web sites serve dynamic content and are built using a multi-tier model, including the client, the web server front end, the application server(s), cache(s), and database and image server back-ends. Virtuoso, with its simplified, adaptive and distributed model of computing in a perfect position to host such applications.

In particular we deployed and studied TPC-W, an industry benchmark for transactional web ecommerce applications. TPC-W models an online bookstore [157]. We use the University of Wisconsin's PHARM group's implementation [79], particularly the distribution created by Jan Kiefer.

The separable components of a typical can be hosted in separate VMs. Figure 2.1 shows the configuration of TPC-W that we setup, spread over four VMs. Remote Browser Emulators (RBEs) simulate users interacting with the web site. RBEs talk to a web server (Apache) that also runs an application server (Tomcat). The web server fetches images

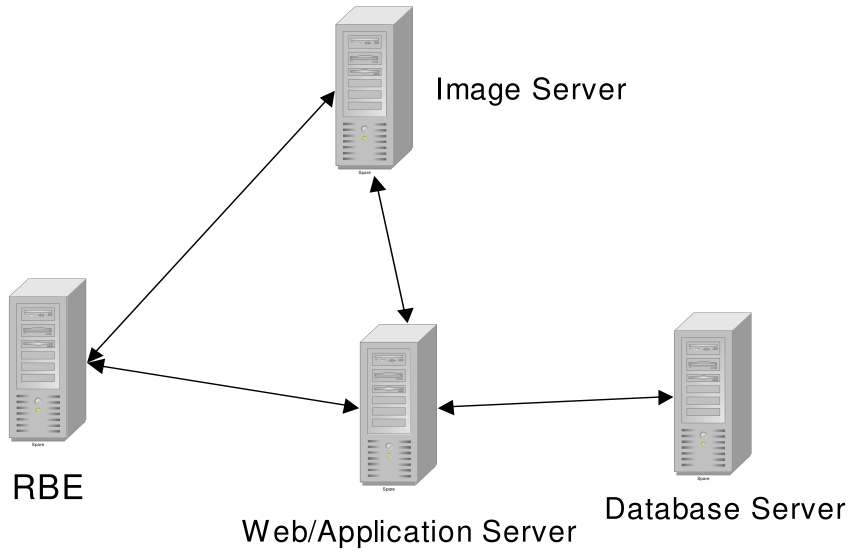


Figure 2.1: The configuration of TPC-W setup and studied.

from an NFS-mounted image server, alternatively forwarding image requests directly to an Apache server also running on the image server. The application server uses a back-end database (MySQL) as it generates content.

2.3 Enterprise backup applications

A typical enterprise, even a modestly sized one, generates tremendous amounts of business data on a daily basis. All enterprises backup data from individual computers on to distributed and replicated backup servers, generally on a daily basis. In large enterprises such as Intel Corporation [89], backup accounts for majority of the traffic flowing in corporate networks [90].

We studied and modeled a typical enterprise backup application based on insights obtained from Intel IT Research. A typical backup scenario is sketched at a high level in Figure 2.2. The backup application acts as a client-server system. The system consists of a single master server, a set of dedicated, replicated backup servers and finally clients that

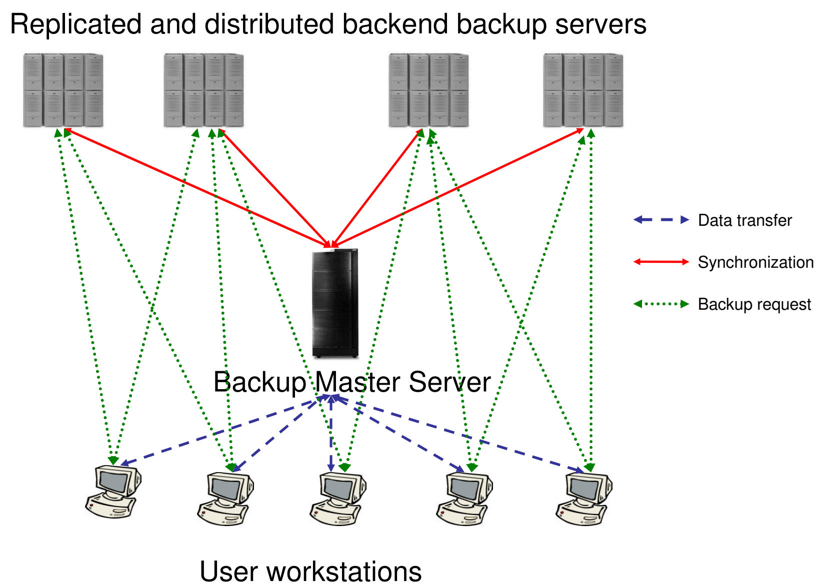


Figure 2.2: A schematic model of enterprise backup applications.

sit on a user's machine. A client communicates with the master server exchanging block hashes to determine the data to be backed up. The master server assigns each client a slice of the backup servers where the user's data is stored in a distributed and replicated fashion. The master server, for each client, maintains a data structure listing the backup servers storing its data.

The backup application is different than the previous two classes studied. Though, it is a critical application, yet it does not affect businesses' bottom line. Once approach to backups can be to complete them as quickly as possible, another would be to utilize networks links such as to avoid network bottlenecks.

2.4 Enterprise DNS

DNS type query services play an important role in an enterprise setting. It has been reported by Intel IT Research that deployment of the DNS service inside Intel typically

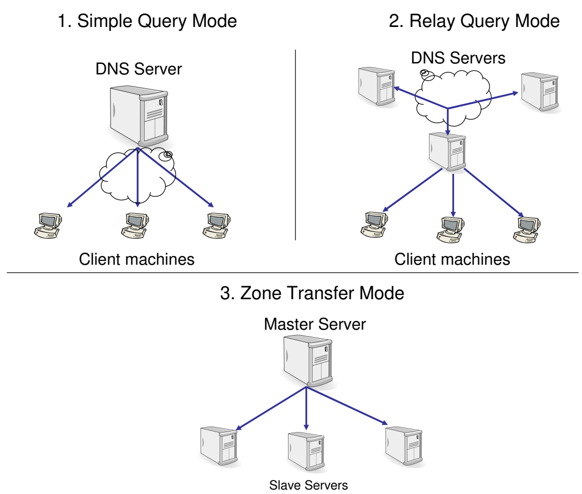


Figure 2.3: A schematic model of DNS.

requires multiple network connections to service a query. Intel sites that have only a single outgoing connection or sites that have less reliable outgoing network connections deploy a DNS server on site [15]. At any given time DNS might be operating in one of multiple modes. First, it might simply be replying to a direct query, in which case low latency is important. Second, it might be servicing a query flowing through a relay of DNS servers, in this case, low latency over wide area networks is important. Third, DNS servers might act as masters for some zones and slaves for others by caching and forwarding data for them. In such a setting, master-slaves often exchange data via zone transfers that requires high bandwidth. Figure 2.3 sketches these modes of operation at a high level.

2.5 Enterprise mail applications

Over the last decade electronic mail (email) has become one of the most important means of communication. This is especially true inside of an enterprise, large or small. Figure 2.4 helps to illustrate the setup of a typical enterprise mail application. A client wishing to send or read mail, contacts an SMTP server. The SMTP server itself talks to DNS and LDAP

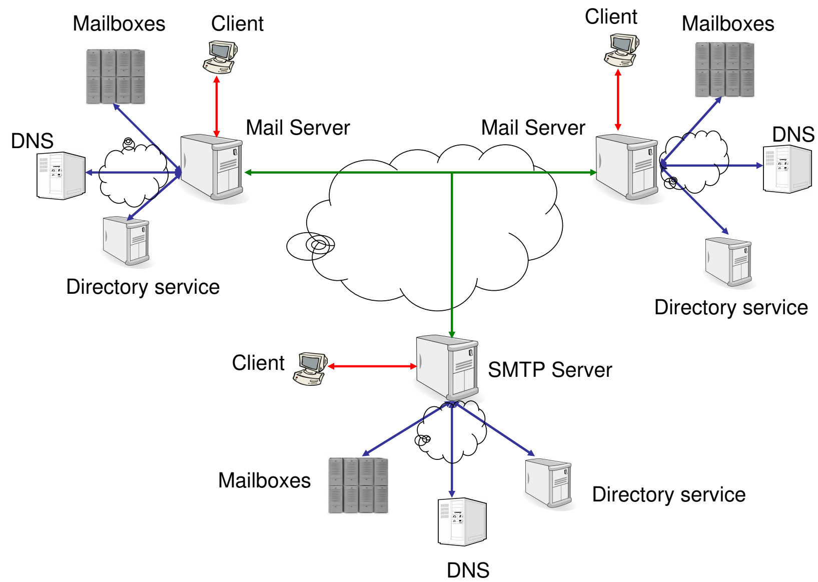


Figure 2.4: A schematic model of an enterprise mail application.

directory services depending on whether it is sending or receiving mail. In the scenario sketched, a client's mailbox is distributed across a set of machines. The initial mail protocol is chatty thus necessitating low latency links. Persistence of TCP connections can cause performance bottlenecks, bandwidth becomes important for larger mails. One possible measure of application performance is messages handled per second, other possibilities are message delays and message queue lengths. Again, the above information is based on discussions held with Intel IT Research.

2.6 Enterprise wide-area file transfers

Intel Corporation has operations in many different geographical locations, spread all over the world. Surprisingly, it does not deploy a global file system such as AFS [82]. Instead when data needs to be accessed from a different location and response time is important, a wide-area file transfer is effected. Such an application primarily places high bandwidth demands [90] and hence can significantly benefited with automatic adaptation.

2.7 Enterprise simulations

Intel Corporation primarily operates in the semiconductor sector. The next generation chip designs are often preceded by long running stand alone simulations to study performance, etc. before they are cast in silicon. Our discussions with Intel IT Research revealed that more often that not, these simulations have high compute requirements.

Intel Corporation uses an in-house wide-area distributed computing system called NetBatch [27, 136]. Intel employees submit simulations to the NetBatch pool which manages idle resources on office workstations and servers. NetBatch includes a hierarchical matchmaker that matches submitted jobs (both engineering computations and simulations) to available resources [136].

It is a well known and independently recorded fact that many of the deployments of these systems (such as NetBatch) manage computers located in several administration sites that are often geographically separated, and are organized in pools containing large number of machines [136]. Thus, NetBatch operates like a traditional wide-area distributed computing system and hence suffers from the deficiencies of this approach as detailed in Chapter 1. This further backs up our statement on the complexities in the traditional model of wide-area distributed computing.

Virtualizing such applications partially solves the problem of over-provisioning and isolation. In a scenario, where such applications execute inside of VMs sitting on server or workstation nodes, there is tremendous potential for performance improvement via automatic and dynamic adaptations.

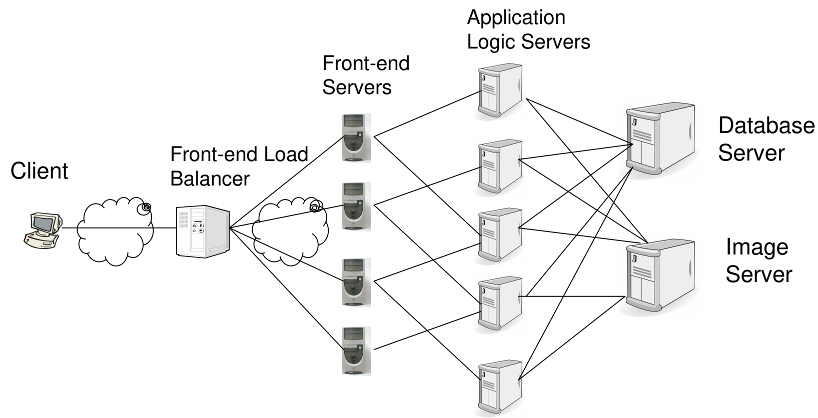


Figure 2.5: A schematic model of an enterprise database application.

2.8 Enterprise database applications - small queries

Database applications are all pervasive. Any organization that stores large amounts of data has to create an efficient database architecture to access the information. Database architectures have been extensively studied and are well understood [61]. A typical multi-tier architecture includes a set of load balanced front end servers, connected to a set of application logic servers. The data itself is stored in a back-end database server, image server and a set of file servers. Figure 2.5 illustrates this typical scenario.

Database systems come in two varieties. Some of them come with a built in orchestration service, which makes resource allocation and provisioning decisions on a stand alone basis. The second category of systems include those do not come with a built in orchestration service. Systems with orchestration services present an interesting challenge for Virtuoso since adaptation can now occur at two levels, inside the application and outside the application driven by Virtuoso. These two levels of adaptation, if occurring simultaneously, can be mutually destructive resulting in oscillatory behavior. For the purposes of this study, we ignore applications that come with built-in orchestration services.

2.9 Enterprise database applications - large queries

This application class is identical to the database application class described above, except for the amount of data being returned at the end of a query. Enterprise often run compute and bandwidth intensive queries at the end of each month and financial quarters to create business summaries [90]. Given the typically tight deadlines associated with such settings, any benefits in performance via automatic adaptation would be welcome [90].

2.10 Enterprise engineering computations

As expected, an engineering company such as Intel Corporation generates a rather large number of engineering computing applications. These are different than the enterprise simulations in the sense that they can have a significant amount of communication between application components. Such engineering jobs are also submitted to NetBatch. The reason these applications are described as a separate class is that their resource demands are very different than the enterprise simulations. Network properties also take center-stage in addition to the computation resources and hence adapting these application presents a bigger challenge, but at the same promises higher returns (since we are provided with additional degrees of freedom).

2.11 Conclusions

In this chapter we have introduced ten distributed application classes. These application classes can be broadly categorized as high performance computing applications, web transactional e-commerce applications and enterprise applications. We notice that applications in all these classes can significantly benefit by executing in distributed virtual environments. Intel Corporation has made a significant push towards introducing virtualization in enterprises [15]. Based on our discussions with researchers at Intel Corporation, we also note that these applications have serious adaptation requirements. In the remainder of this dissertation we study automatic, run-time and dynamic adaptation in the context of these application classes. In particular we pose the question, does there exist a single adaptation scheme that is effective for a majority of applications that are representative of these application classes.

Chapter 3

Virtual networks, inference and measurements

Three important components of an adaptive virtualized computing system are adaptive virtual networks, application demand inference and system resource measurements. These components are the topics of discussion in this chapter.

Virtual networks are not just a means of providing connectivity to virtual machines. They are a powerful means of providing isolation (security, address and performance isolation) and application independent adaptation mechanisms. We begin this chapter by describing the evolution of the design, implementation and evaluation of VNET (for Virtual NETwork), the virtual networking component of Virtuoso. We are currently in the third generation of VNET design. The first two generations will be the focus of discussion of this dissertation.

In the Virtuoso model, when a user purchases a set of virtual machines from the web front-end. The control system places these machines on some foreign provider hosts based on a match between VM and host characteristics and between the price demanded by the providers and the amount that the user is willing to pay. As mentioned previously in Chapter 1, VNET creates the illusion that all of the user's virtual machines are plugged into his local area network.

Section 3.1 describes the design, implementation and evaluation of the first generation VNET whose sole functionality was to create and maintain this illusion. Each VM hosted on a foreign host was connected back to a proxy machine on the user's network, thus creating a bridge with long wires. The overhead of VNET over the wide-area, for which it was designed, was negligible, however, it had significant overhead on 10 and 100 Mbit LANs.

The second generation VNET is described in Section 3.3. In the first iteration of the design, each VM communicated to the outside world and to each other via the user's LAN thus creating a star topology centered on the proxy machine on the user's LAN. We observed that often there is significant communication between a user's VMs and while these VMs might be hosted on the same or close by networks, all traffic first flowed back to the user's LAN and then to the recipient VM, resulting in lowered performance. The second generation design improved on this by allowing, in addition to the star backbone, arbitrary topologies to be formed between VNET daemons running on foreign hosts. The implementation of VNET was also improved. The second generation VNET has negligible overhead on 10 and 100 Mbit LANs. Over a Gigabit LAN it trails the state of the art network virtualization software by only a small margin. It should be noted that VNET operates completely at user level, so there are a wide range of opportunities to further improve the performance of VNET, well beyond the state of the art.

Section 3.4.1 describes the application independent adaptation mechanisms made possible by VMMs and virtual networks including VM migration, virtual topology and routing changes, and resource reservations. Section 3.5 details Virtuoso's application topology and traffic inference framework, VTTIF. VTTIF can infer the traffic load matrices of applications in an invisible manner by looking at the application Ethernet traffic flowing through the VNET daemons. Each VNET daemon forms a local view of the application's traffic matrix, which is then periodically sent to a centralized location where the application communication behavior is inferred by using simple normalization and pruning mechanisms. VTTIF is also fitted with mechanisms to handle scenarios where the application communication topology changes at a rate faster than what Virtuoso can adapt to. In such scenarios VTTIF provides a damped view of the changing application communicating behavior thus preventing itself from being into oscillations.

We describe, in Section 3.6, how a third party passive network measurement scheme, Wren, has been successfully integrated with Virtuoso. For adaptation to be possible in Virtuoso, we need some means of measuring the physical network and host characteristics. There are many tools available that can be used to monitor host characteristics such as CPU speed, current load, etc. [38, 124]. Wren is a tool developed by Zangrilli et al. at the College of William and Mary that measures the physical network in a passive manner by using the naturally occurring VM traffic thus minimizing measurement overheads.

We conclude this chapter in Section 7.7.

3.1 VNET: A virtual network

We are faced with an interesting network management problem in Virtuoso. Unlike traditional units of work in distributed systems, such as jobs, processes, or RPC calls, a virtual machine has, and must have, a direct presence on the network at layer 3 and below. We must be able to communicate with it. VMM software recognizes this need and typically creates a virtual Ethernet card for the guest operating system to use. This virtual card is then emulated using the physical network card in the host machine in one of several ways. The most flexible of these, bridges the virtual card directly to the same network as the physical card, making the virtual machine a first class citizen on the same network, indistinguishable from a physical machine.

Within a single site, this works very well, as there are existing mechanisms to provide new machines with access. Wide-area distributed computing, however, is intrinsically about using multiple sites, with different network management and security philosophies, often spread over the wide area [55]. Running a virtual machine on a remote site is equivalent to visiting the site and connecting a new machine. The nature of the network presence (active Ethernet port, traffic not blocked, routable IP address, forwarding of its packets through firewalls, etc.) the machine gets, or whether it gets a presence at all, depends completely on the policy of the site. The impact of this variation is further complicated as the number of sites is increased, and if we permit virtual machines to migrate from site to site.

To deal with this problem, I have co-designed VNET, a simple data link layer virtual network tool. Using VNET, virtual machines have no network presence at all on a remote site. Instead, VNET provides a mechanism to project their virtual network cards onto another network, which also moves the network management problem from one network to another. For example, all of a user's virtual machines can be made to appear to be connected to the user's own network, where the user can use his existing mechanisms to assure that they have appropriate network presence. Because the virtual network is at the data link layer, a machine can be migrated from site to site without changing its presence—it always keeps the same IP address, routes, etc.

The design of VNET has evolved considerably since its conception. VNET is currently in its third generation of design. As part of my dissertation, I designed and implemented the first two generations of VNET. Lange et al., at Northwestern University, are currently working on the third generation VNET design. In the following sections we first describe VNET's relationship with the virtual networking built into traditional VMMs. We then present the evolution of VNET by starting with the design, implementation and evaluation of the first generation.

As we have developed the first generation VNET, we have come to believe that virtual networks designed specifically for virtual machine distributed computing can be used for much more than simplifying the management problem. This insight led us to design and implement the second generation VNET. In particular, because they see all of the traffic of the virtual machines, they are in an ideal position to (1) measure the traffic load and application topology of the virtual machines, (2) monitor the underlying network, (3) adapt application as measured by (1) to the network as measured by (2) by relocating virtual machines and modifying the virtual network topology and routing rules, and (4) take advantage of resource reservation mechanisms in the underlying network and at the operating system level. Best of all, these services can be done on behalf of existing, unmodified applications and operating systems running in the virtual machines. Later chapters of this dissertation lay out this argument, formalize the adaptation problem, and describe our efforts to solving it.

3.1.1 VNET and traditional VMM networking

VNET is a simple proxying scheme that works entirely at user level. The primary dependence it has on the virtual machine monitor is that there must be a mechanism to extract the raw Ethernet packets sent by the virtual network card, and a mechanism to inject raw Ethernet packets into the virtual card. The specific mechanisms we use are packet filters, packet sockets, and VMWare's host-only networking interface.

Though we use VMware, specifically, VMWare GSX Server 2.5, as our Virtual Machine Monitor, VNET, without modification, has been successfully used with User Mode Linux [32], the VServer extension to Linux [112] and Xen [9].

3.1.2 VMWare networking

VMWare, in its Workstation and Server variants, can connect the virtual network interface to the network in three different ways. To the operating system running in the virtual machine (the VM), they all look the same. By themselves, these connection types are not well suited for use in a wide-area, multi-site environment, as we describe below.

The simplest connection is "bridged", meaning that VMWare uses the physical interface of the Host to directly emulate the virtual interface in the VM. This emulation is not visible to programs running on the Host. With a bridged connection, the VM shows up as another machine on the Local environment, the LAN of the Host. This creates a network management problem for the Local environment (What is this new machine that has suddenly appeared?) and for the User (Will this machine be given network connectivity? How? What's its address? Can I route to it?). Furthermore, if the VM is moved to a Host on a different network, the problems recur, and new ones rear their ugly head (Has the address to the VM changed? What about all its open connections and related state?)

The next form of connection is the host-only connection. Here, a virtual interface is created on the Host which is connected to the virtual interface in the VM. When brought up with the appropriate private IP addresses and routes, this enables programs on the host to talk to programs on the VM. Because we need to be able to talk to the VM from other machines on the user's network and beyond, host-only networking is insufficient. However, it also has the minimum possible interaction with network administration in the local environment.

The final form of connection is via network address translation (NAT), a commonly used technique in border routers and firewalls [44]. Similar to a host-only connection, a virtual interface on the Host is connected to the virtual interface on the VM, and appropriate private IP addresses and routes are assigned. In addition, a daemon running on the Host receives IP packets on the interface. For each outgoing TCP connection establishment (SYN), it rewrites the packet to appear to come from the IP address of the Host's regular interface, from some unused port. It records this mapping from the IP address and port on the VM to the address and port it assigned. Mappings can also be explicitly added for incoming TCP connections or UDP traffic. When a packet arrives on the regular interface

Term	Explanation
User	is the owner of a set virtual machines
Client	is the machine that the user uses to access his virtual machines
Proxy	is a machine for networking on the user's LAN
	Proxy and Client can be the same machine
Host	is a machine that host one or more virtual machines
Local environment	of a VM is the LAN to which its Host is connected
Remote environment	of a VM is the LAN to which the Client and Proxy are connected

Table 3.1: VNET nomenclature.

for the IP and port, it rewrites it using the recorded mapping and passes it to the VM. To the outside world, it simply appears that the Host is generating ordinary packets. To the VM, it appears as if it has a direct connection to the Local environment. For our purposes, NAT networking is insufficient because it is painful to make incoming traffic work correctly as the mappings must be established manually. Furthermore, in some cases it would be necessary for the IP address of the virtual machine to change when it is migrated, making it impossible to maintain connections.

3.2 First generation VNET: A bridge with long wires

In the following, we describe the interface of the first generation VNET, and performance results in the local and wide area. Table 3.1 presents some terminology that is used throughout the remainder of this chapter. The *User* is the owner of the virtual machines (his *VM*s) which he accesses using his *Client* machine. The user also has a *Proxy* machine for networking, although the Proxy and Client can be the same machine. Each VM runs on a *Host*, and multiple VMs may run on each Host. The *Local* environment of a VM is the LAN to which its Host is connected, while the *Remote* environment is the LAN to which the Client and the Proxy are connected. Figure 3.1 illustrates these relationships.

In essence, VNET provides bridged networking, except that the VM is bridged to the

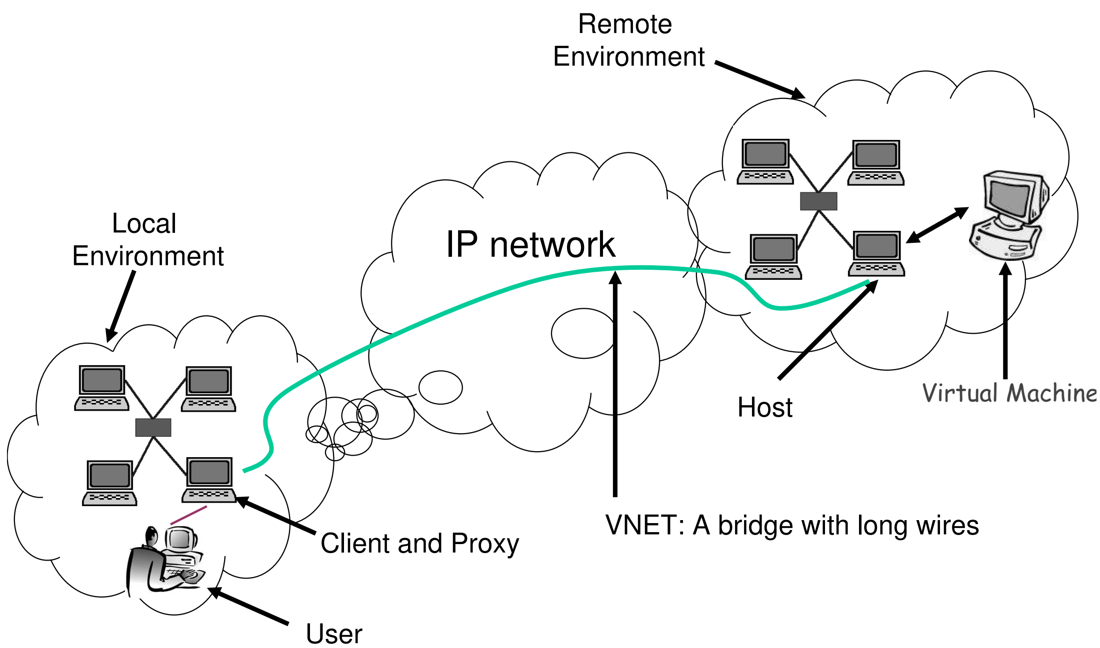


Figure 3.1: Relationship between VNET entities.

Remote network, the network of the user. VNET consists of a client and a server. The client is used simply to instruct servers to do work on its behalf. Each physical machine that can instantiate virtual machines (a Host) runs a single VNET server. At least one machine on the user's network also runs a VNET server. We refer to this machine as the Proxy. The user's machine is referred to as the Client. As mentioned above, the Client and the Proxy can be the same machine. The first generation VNET consists of approximately 4000 lines of C++.

3.2.1 Operation

VNET servers are run on the Host and the Proxy and are connected using a TCP connection that can optionally be encrypted using SSL. The VNET server running on the Host opens the Host's virtual interface in promiscuous mode and installs a packet filter that matches Ethernet packets whose source address is that of the VM's virtual interface. The VNET

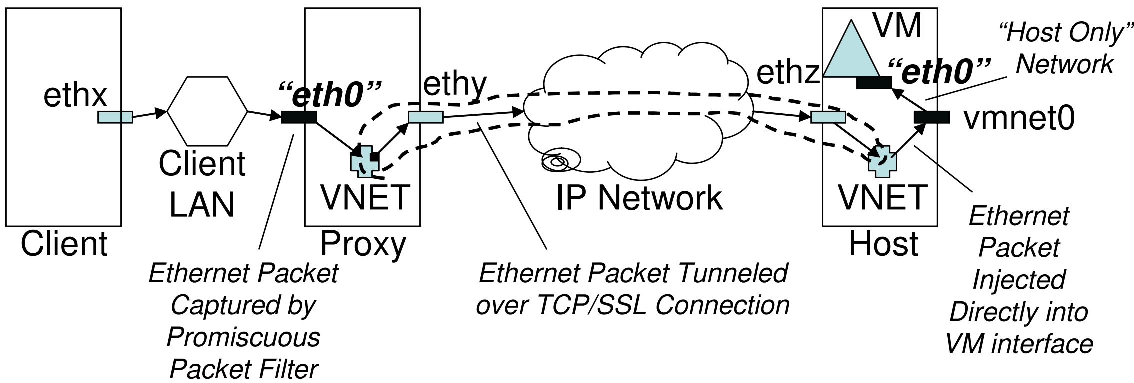


Figure 3.2: VNET configuration for a single remote virtual machine with outbound traffic from the VM.

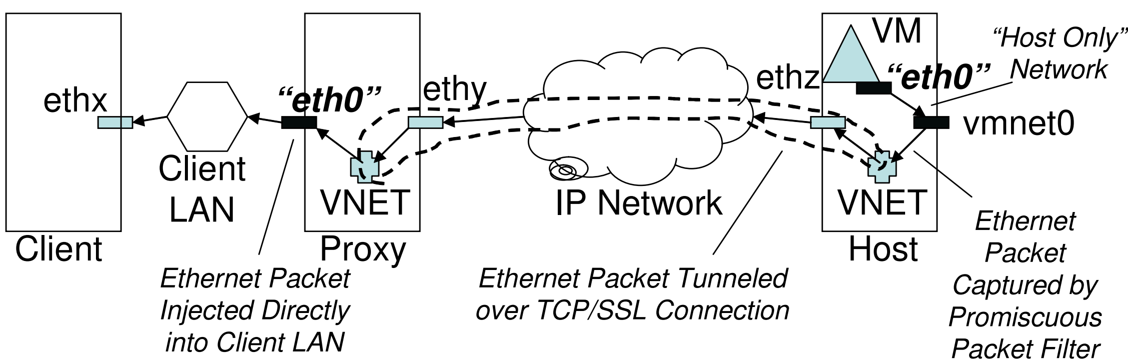


Figure 3.3: VNET configuration for a single remote virtual machine with inbound traffic into the VM.

server on the Proxy opens the Proxy's physical interface in promiscuous mode and installs a packet filter that matches Ethernet packets whose destination address is that of the VM's virtual interface or is the Ethernet broadcast and/or (optionally) multicast addresses. To avoid loops, the packet must also not have a source address matching the VM's address. In each case, the VNET server is using the Berkeley packet filter interface [125] as implemented in libpcap, functionality available on all Unix platforms, as well as Microsoft Windows [126].

When the Proxy's VNET server sees a matching packet, it serializes it over the TCP connection to the Host's VNET server. On receiving the packet, the Proxy's VNET server directly injects the packet into the virtual network interface of the Host (using libnet, which is built on packet sockets, also available on both Unix and Windows) which causes it to be delivered to the VM's virtual network interface. Figure 3.2 illustrates the path of such outbound traffic. When the Host's VNET server sees a matching packet, it serializes it to the Proxy's VNET server. The Proxy's VNET server in turn directly injects it into the physical network interface card, which causes it to be sent on the LAN of the Client. Figure 3.3illustrates the path of such inbound traffic. It should be noted that multiple virtual machines on the Host are possible, as are multiple hosts. Only a single Proxy is needed, and it can be the same as the Client.

The end-effect of such a VNET *Handler* is that the VM appears to be connected to the Remote Ethernet network exactly where the Proxy is connected. A Handler is identified by the following information:

- IP addresses of the Host and Proxy
- TCP ports on the Host and Proxy used by the VNET servers
- Ethernet devices used on the Host and Proxy

- Ethernet addresses which are proxied. These are typically the address of the VM and the broadcast address, but a single handler can support many addresses if needed.

- Roles assigned to the two machines (which is the Host and which is the Proxy)

A single VNET server can support an arbitrary number of handlers, and can act in either the Host or Proxy role for each. Each handler can support multiple addresses. Hence, for example, the single physical interface on a Proxy could provide connectivity for many VMs spread over many sites. Multiple Proxies or multiple interfaces in a single Proxy could be used to increase bandwidth, up to the limit of the User's site's bandwidth to the broader network.

Because VNET operates at the data link layer, it is agnostic about the network layer, meaning protocols other than IP can be used. Furthermore, because we keep the MAC address of the VM's virtual Ethernet adapter and the LAN to which it appears to be connected fixed for the lifetime of the VM, migrating the VM does not require any participation from the VM's OS, and all connections remain open after a migration.

A VNET client wishing to establish a handler between two VNET servers can contact either one. This is convenient, because if only one of the VNET servers is behind a NAT firewall, it can initiate the handler with an outgoing connection through the firewall. If the client is on the same network as the firewall, VNET then requires only that a single port be open on the other site's firewall. If it is not, then both sites need to allow a single port through. If the desired port is not permitted through, there are two options. First, the VNET servers can be configured to use a common port. Second, if only SSH connections are possible, VNET's TCP connection can be tunneled through SSH.

Command	Description
HELLO passwd version	Establish Session
DONE	Finish Session
DEVICES?	Return available network interfaces
HANDLERS?	Return currently running handlers
CLOSE handler	Tear down an existing handler
HANDLE remotepasswd	Establish a handler
local_config	(Described in text)
local_device	
remote_config	
remote_address	
remote_port	
remote_device	
macaddress+	
BEGIN local_config	Establish a handler
local_device	(Described in text)
remote_config	
remote_device	
macaddress+	

Table 3.2: First generation VNET interface.

3.2.2 Interface

VNET servers are run on the Host and the Proxy. A VNET client can contact any server to query status or to instruct it to perform an action on its behalf. The basic protocol is text-based, making it readily scriptable, and bootstraps to binary mode when a Handler is established. Optionally, it can be encrypted for security. Table 3.2 illustrates the interface that a VNET Server presents.

Session establishment and teardown: The establishment of session with a VNET server is initiated by a VNET client or another server using the HELLO command. The client authenticates by presenting a password or by using an SSL certificate. Session teardown is initiated by the VNET client using the DONE command.

Handler establishment and teardown: After a VNET client has established a session with a VNET server, it can ask the server to establish a Handler with another server.

This is accomplished using the HANDLE command. As shown in Figure 3.2, the arguments to this command are the parameters that define a Handler as described earlier. Here, `local_config` and `remote_config` refer to the Handler roles. In response to a HANDLE command, the server will establish a session with the other server in the Handler pair, authenticating as before. It will then issue a BEGIN command to inform the other VNET server of its intentions. If the other server agrees, both servers will bootstrap to a binary protocol for communicating Ethernet packets. The Handler will remain in place until one of the servers closes the TCP connection between them. This can be initiated by a client using the CLOSE command, directed at either server.

Status Enquiry: A client can discover a server's available network interfaces (DEVICES?) and what Handlers it is currently participating in (HANDLERS?).

3.2.3 Performance of first generation VNET

Our goal for the first generation VNET was to make it easy to convey the network management problem induced by VMs to the home network of the user where it can be dealt with using familiar techniques. However, it is important that VNET's overhead not be prohibitively high, certainly not in the wide area. From the strongest to the weakest goal, VNET's performance should be

1. in line with what the physical network is capable of,
2. comparable to other networking solutions that don't address the network management problem, and
3. sufficient for the applications and scenarios where it is used.

In this chapter we will see that the first generation VNET meets the third goal, the second generation VNET meets the third and second goals and the third generation VNET currently under developed by Lange et al. at Northwestern University, is targeted towards the first goal.

Metrics

Latency and throughput are the most fundamental measures used to evaluate the performance of networks. The time for a small transfer is dominated by latency, while that for a large transfer is dominated by throughput. Interactivity, which is often dominated by small transfers, suffers if latencies are either high or highly variable [45]. Bulk transfers suffer if throughput is low. Our measurements were conducted on working days (Monday through Thursday) in the early morning to eliminate time-of-day effects.

Latency: To measure latency, we used the round-trip delay of an ICMP echo request/response pair (i.e., ping), taking samples over hour-long intervals. We computed the average, minimum, maximum and standard deviation of these measurements. Here, we report the average and standard deviation. Notice that this measure of latency is symmetric.

Throughput: To measure average throughput, we use the *ttcp* program. Ttcp is commonly used to test TCP and UDP performance in IP networks. Ttcp times the transmission and reception of data between two systems. We use a socket buffer size of 64 KBytes and transfer a total of 1 GB of data in each test. VNET's TCP connection also uses a socket buffer size of 64 KBytes. TCP socket buffer size can limit performance if it is less than the bandwidth-delay product of the network path, hence our larger-than-default buffers. All throughput measurements were performed in both directions.

Testbeds

Although VNET is targeted primarily for wide-area distributed computing, we evaluated performance in both a LAN and a WAN. Because our LAN testbed provides much lower latency and much higher throughput than our WAN testbed, it allows us to see the overheads

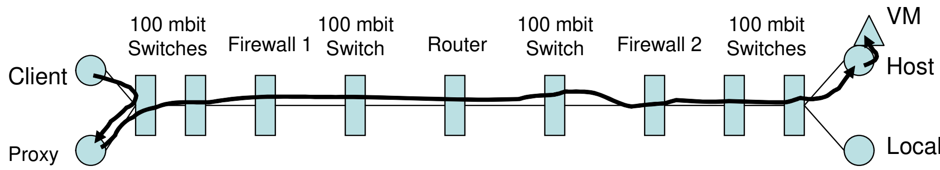


Figure 3.4: VNET test configurations for the local area between two labs in the Northwestern CS Department.

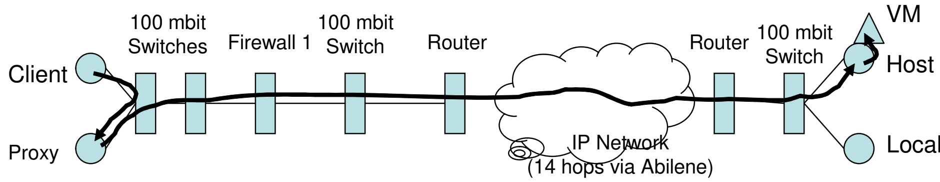


Figure 3.5: VNET test configurations for the wide area between labs in the Northwestern and Carnegie Mellon CS Departments.

due to VNET more clearly. The Client, Proxy, and Host machines are 1 GHz Pentium III machines with Intel Pro/100 adaptors. The virtual machine uses VMware GSX Server 2.5, with 256 MB of memory, 2 GB virtual disk and RedHat 7.3. The network driver used is vmxnet.

Our testbeds are illustrated in Figures 3.4 and 3.5. The LAN and WAN testbeds are identical up to and including the first router out from the Client. This portion is our firewalled lab in the Northwestern CS department. The LAN testbed then connects, via a router which is under university IT control (not ours), to another firewalled lab in our department which is a separate, private IP network. The WAN testbed instead connects via the same router to the Northwestern backbone, the Abiline network, the Pittsburgh Supercomputing Center, and two administrative levels of the campus network at Carnegie Mellon, and finally to an lab machine there. Notice that even a LAN environment can exhibit the network management problem. It is important to stress that the only requirement that VNET places on either of these complex environments is the ability to create a TCP connection between the Host and Proxy in some way.

We measured the latency and throughput of the underlying "physical" IP network, VMWare's virtual networking options, VNET, and of SSH connections:

- *Physical:* VNET transfers Ethernet packets over multiple hops in the underlying network. We measure equivalent hops, and also end-to-end transfers, excepting the VM.
  - *Local* $\leftrightarrow$ *Host*: Machine on the Host's LAN to/from the Host.
  - *Client* $\leftrightarrow$ *Proxy*: Analogous to the first hop for an outgoing packet in VNET and the last hop for an incoming packet.
  - *Host* $\leftrightarrow$ *Proxy*: Analogous to the TCP connection of a Handler, the tunnel between the two VNET servers.
  - *Host* $\leftrightarrow$ *Client*: End-to-end except for the VM.
  - *Host* $\leftrightarrow$ *Host*: Internal transfer on the Host.
- *VMWare:* Here we consider the performance of all three of VMWare's options, described earlier.
  - *Host* $\leftrightarrow$ *VM*: Host-only networking, which VNET builds upon.
  - *Client* $\leftrightarrow$ *VM (Bridged)*: Bridged networking. This leaves the network administration problem at the remote site.
  - *Client* $\leftrightarrow$ *VM (NAT)*: NAT-based networking. This partially solves the network administration problem at the remote site at the layer 3, but creates an asymmetry between incoming and outgoing connections, and does not support VM migration. It's close to VNET in that network traffic is routed through a user-level server.

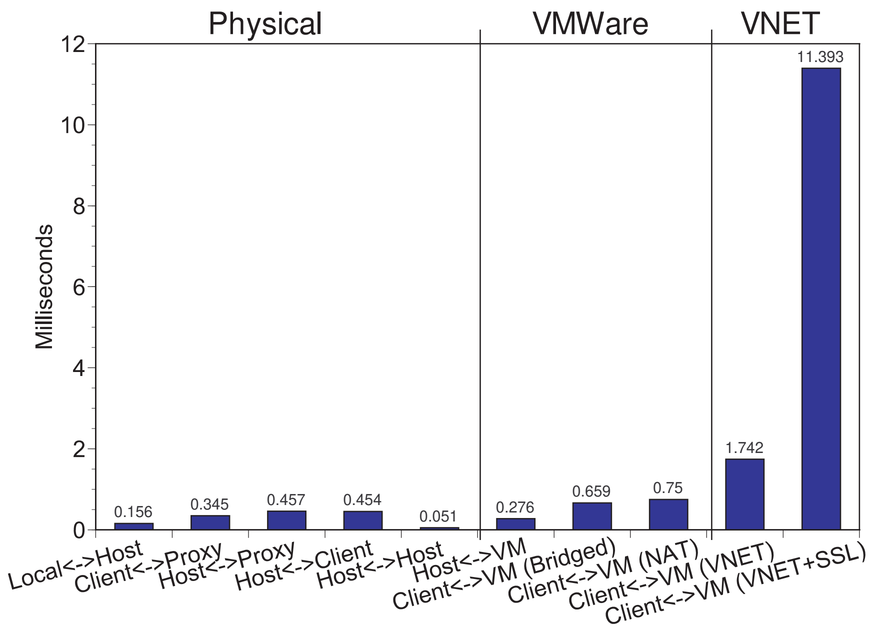


Figure 3.6: Average latency in the local area.

- *VNET:* Here we use VNET to project the VM onto the Client's network.
  - *Client* $\leftrightarrow$ *VM (VNET)*: VNET without SSL
  - *Client* $\leftrightarrow$ *VM (VNET+SSL)*: VNET with SSL
- *SSH:* Here we look at the throughput of an SSH connection between the Client and the Host to compare with VNET with SSL.
  - *Host* $\leftrightarrow$ *Client (SSH)*

3.2.4 Discussion

Average latency:

In Figure 3.6, we see that the average latency on the LAN when using VNET without SSL is 1.742 ms. It is important to understand exactly what is happening. The Client is sending an ICMP echo request to the VM. The request is first intercepted by the Proxy,

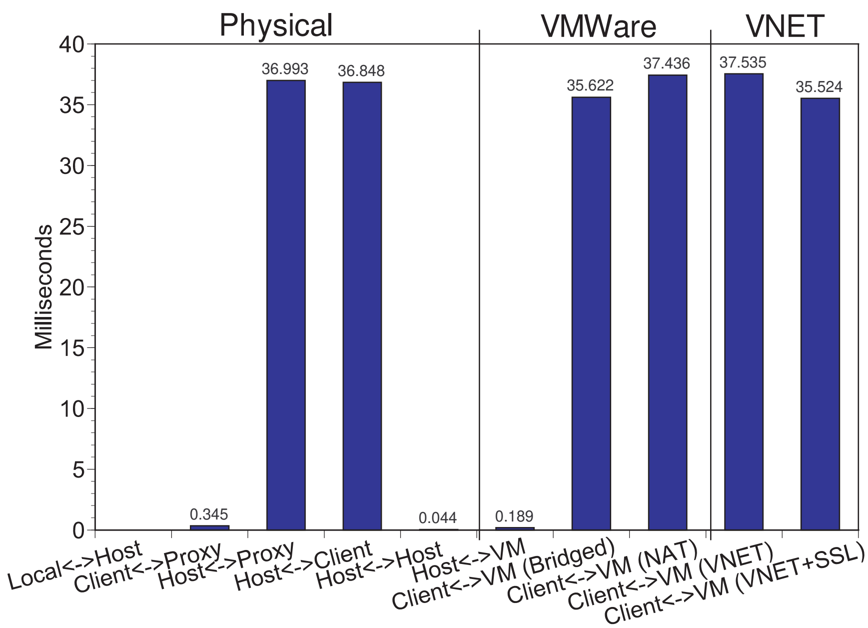


Figure 3.7: Average latency over the wide area.

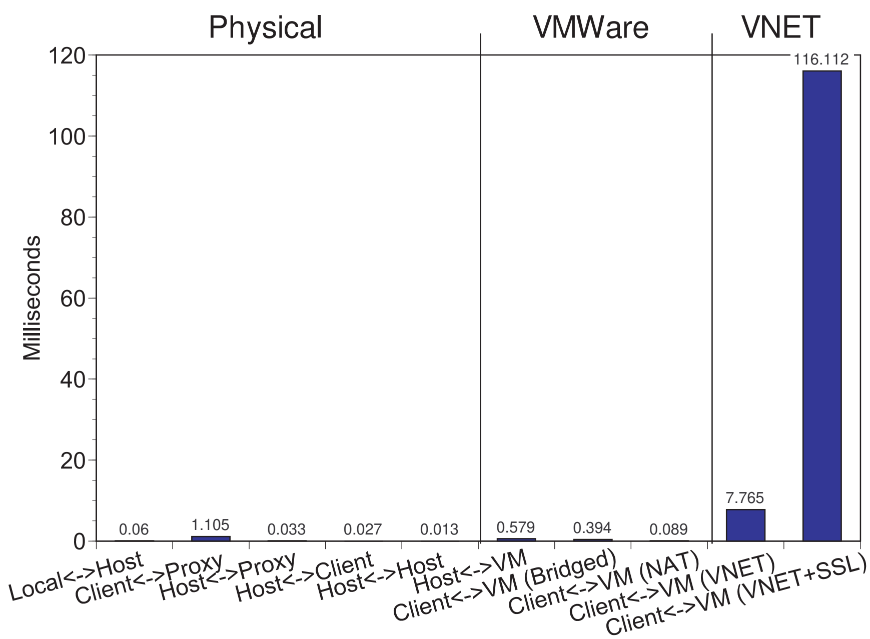


Figure 3.8: Standard deviation of latency in the local area.

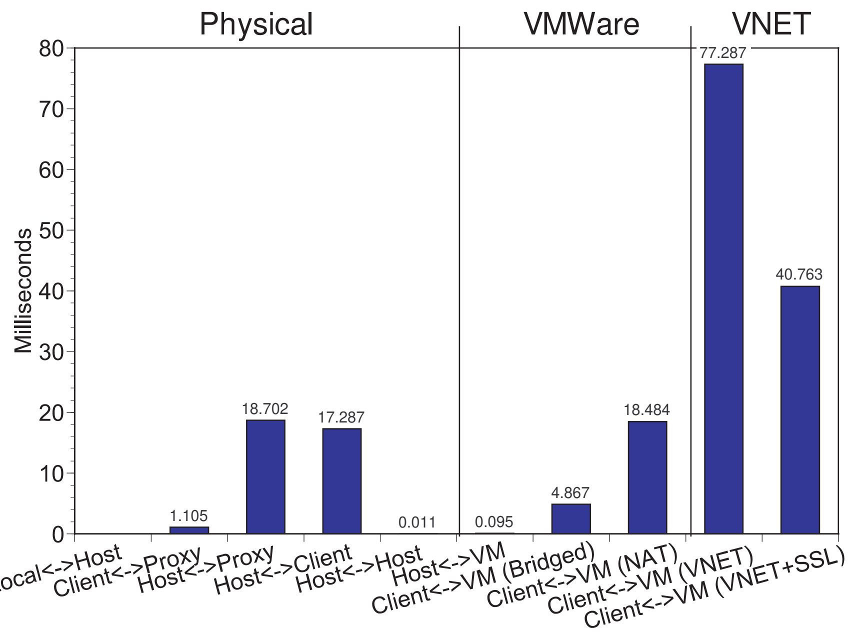


Figure 3.9: Standard deviation of latency over the wide area.

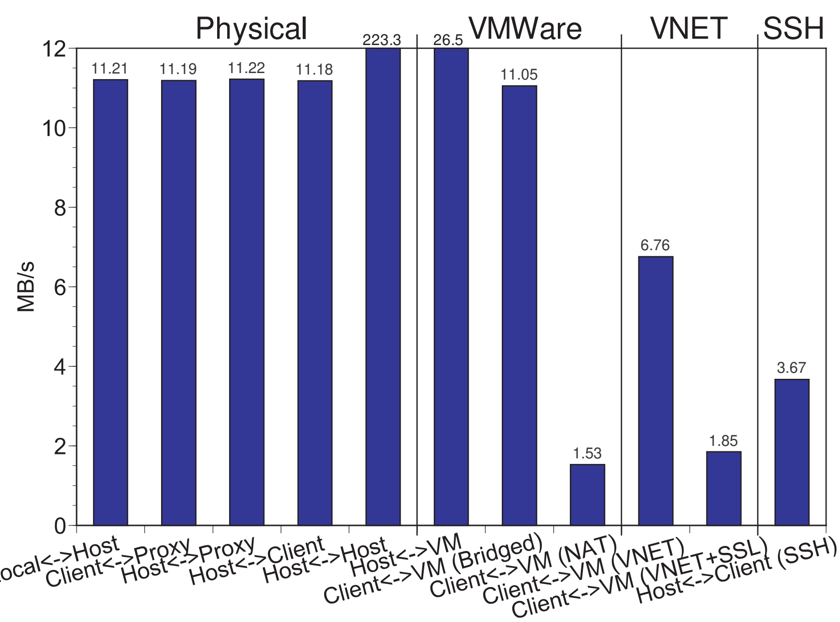


Figure 3.10: Bandwidth in the local area.

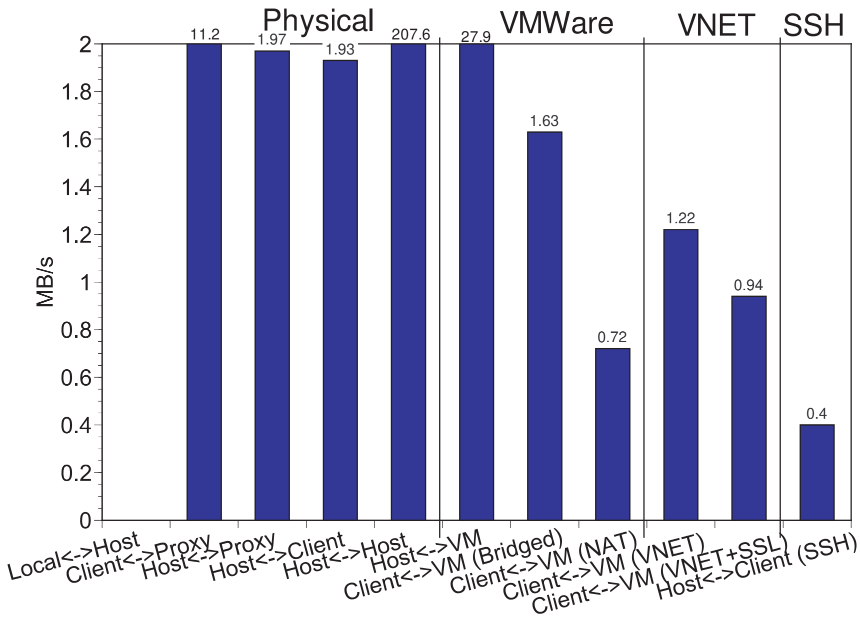


Figure 3.11: Bandwidth over the wide area.

then sent to the Host, and finally the Host sends it to the VM (see Figure 3.4). The reverse path for the echo reply is similar. These three distinct pieces have average latencies of 0.345 ms, 0.457 ms, and 0.276 ms, respectively, on the physical network, which totals 1.078 ms. In the LAN, VNET without SSL increases latency by 0.664 ms, or about 60%. We claim that this is not prohibitive, especially in absolute terms. Hence we note that the operation of VNET over LAN does not add prohibitively to the physical latencies. The VMWare NAT option, which is the closest analog to VNET, except for moving the network management problem, has about 1/2 of the latency. When SSL encryption is turned on, VNET latency grows to 11.393 ms, 10.3 ms and a factor of 10 higher than what is possible on the (unencrypted) physical network.

In Figure 3.7, we note that the average latency on the WAN when using VNET without SSL is 37.535 ms and with SSL encryption is 35.524 ms. If we add up the constituent latencies as done above, we see that the total is 37.527 ms. In other words, VNET with or without SSL has average latency comparable to what is possible on the physical network in the WAN. The average latencies seen by VMWare's networking options are also roughly the same. In the wide area, average latency is dominated by the distance, and we get the benefits of VNET with negligible additional cost.

Standard deviation of latency:

In Figure 3.8, we see that the standard deviation of latency using VNET without SSL in the LAN is 7.765 ms, while SSL increases that to 116.112 ms. Adding constituent parts only totals 1.717 ms, so VNET has clearly dramatically increased the variability in latency, which is unfortunate for interactive applications. We believe this large variability is because the TCP connection between VNET servers inherently trades packet loss for increased delay. For the physical network, we noticed end-to-end packet loss of approximately 1%. VNET packet losses were nil. VNET resends any TCP segment that contains an Ethernet packet that in turn contains an ICMP request/response. This means that the ICMP packet eventually gets through, but is now counted as a high delay packet instead of a lost packet, increasing the standard deviation of latency we measure. A histogram of the ping times shows that almost all delays are a multiple of the round-trip time. TCP tunneling was used to have the option of encrypted traffic. UDP tunneling reduces the deviation seen, illustrating that it results from our specific implementation and not the general design.

In Figure 3.9, we note that the standard deviation of latency on the WAN when using VNET without SSL is 77.287 ms and with SSL is 40.783 ms. Adding the constituent latencies totals only 19.902 ms, showing that we have an unexpected overhead factor of 2 to 4. We again suspect high packet loss rates in the underlying network lead to retransmissions in VNET and hence lower packet loss rates, but a higher standard deviation of latency. We measured a 7% packet loss rate in the physical network compared to 0% with VNET. We again noticed that latencies which deviated from the average did so in multiples of the average latency, supporting our explanation.

Average Throughput:

In Figure 3.10, we see that the average throughput in the LAN when using VNET without SSL is 6.76 MB/sec and with SSL drops to 1.85 MB/sec, while the average throughput for the physical network equivalent is 11.18 MB/sec. We were somewhat surprised with the VNET numbers. We expected that we would be very close to the throughput obtained in the physical network, similar to those achieved by VMWare's host-only and bridged networking options. Instead, our performance is lower than these, but considerably higher than VMWare's NAT option.

In the throughput tests, we essentially have one TCP connection (that used by the ttcps running on the VM and Client) riding on a second TCP connection (that between the two VNET servers on Host and Proxy). A packet loss in the underlying VNET TCP connection will lead to a retransmission and delay for the ttcp TCP connection, which in turn could time out and retransmit itself. On the physical network there is only ttcp's TCP. Here, packet losses might often be detected by the receipt of triple duplicate acknowledgments followed by fast retransmit. However, with VNET, more often than not a loss in the underlying TCP connection will lead to a packet loss detection in ttcp's TCP connection by the expiration of the retransmission timer. The difference is that when a packet loss is detected by timer expiration the TCP connection will enter slow start, dramatically slowing the rate. In contrast, a triple duplicate acknowledgment does not have the effect of triggering slow start.

In essence, VNET is tricking ttcp's TCP connection into thinking that the round-trip time is highly variable when what is really occurring is hidden packet losses. In general, we suspect that TCP's congestion control algorithms are responsible for slowing down the rate and reducing the average throughput. This situation is somewhat similar to that of a *split TCP connection*. A detailed analysis of the throughput in such a case can be found elsewhere [170]. The use of encryption with SSL further reduces the throughput.

In Figure 3.11, we note that the average throughput over the WAN when using VNET without SSL encryption is 1.22 MB/sec and with SSL is 0.94 MB/sec. The average throughput on the physical network is 1.93 MB/sec. Further, we note that the throughput when using VMWare's bridged networking option is only slightly higher than the case where VNET is used (1.63 MB/sec vs. 1.22 MB/sec), while VMWare NAT is considerably slower. Again, as described above, this difference in throughput is probably due to the overlaying of two TCP connections. Notice, however, that the difference is much less than that in the LAN as now there are many more packet losses that in both cases will be detected by ttcp's TCP connection by the expiration of the retransmission timer. Again, the use of encryption with SSL further reduces the throughput.

We initially thought that our highly variable latencies (and corresponding lower-than-ideal TCP throughput) in VNET were due to the priority of the VNET server processes. Conceivably, the VNET server could respond slowly if there were other higher or similar priority processes on the Host, Proxy, or both. To test this hypothesis we tried giving the VNET server processes maximum priority, but this did not change delays or throughput. Hence, this hypothesis was incorrect.

We also compared our implementation of encryption using SSL in the VNET server to SSH's implementation of SSL encryption. We used SCP to copy 1 GB of data from the Host to the Client in both the LAN and the WAN. SCP uses SSH for data transfer, and uses the same authentication and provides the same security as SSH. In the LAN case we found the SCP transfer rate to be 3.67 MB/sec compared to the 1.85 MB/sec with VNET along with SSL encryption. This is an indication that our SSL encryption implementation overhead is not unreasonable. In the WAN the SCP transfer rate was 0.4 MB/sec compared to 0.94 MB/sec with VNET with SSL. This further strengthens the claim that our implementation of encryption in the VNET server is reasonably efficient.

Comparing with VMWare NAT: The throughput obtained when using VMWare's NAT option was 1.53 MB/sec in the LAN and 0.72 MB/sec in the WAN. This is significantly lower than the throughput VNET attains both in the LAN and WAN (6.76 MB/sec and 1.22 MB/sec, respectively). As described previously in Section 3.1.2, VMWare's NAT is a user-level process, similar in principle to a VNET server process. That VNET's performance exceeds that of VMWare NAT, the closest analog in VMWare to VNET's functionality, is very encouraging.

3.2.5 Summary

The following are the main points to take away from our performance evaluation of the first generation VNET:

- Beyond the physical network and the VMWare networking options, VNET gives us the ability to shift the network management problem to the home network of the client.

- The extra average latency when using VNET deployed over the LAN is quite low while the overhead over the WAN is negligible.

- VNET has considerably higher variability in latency than the physical network. This because it automatically retransmits lost packets. If the underlying network has a high loss rate, then this will be reflected as higher latency variability in VNET. Hence, using VNET, in its current implementation, produces a trade: higher variability in latency for zero visible packet loss. This issues has been addressed in the second generation VNET.

- VNET's average throughput is lower than that achievable in the underlying network, although not dramatically so. This was due to an interaction between two levels of

TCP connections. This issue was addressed in the second generation VNET

- VNET's average throughput is significantly better than that of the closest analog, both in terms of functionality and implementation, in VMWare, NAT.
- Using VNET encryption increases average latency and standard deviation of latency by a factor of about 10 compared to the physical network. Encryption also decreases throughput. The VNET encryption results are comparable or faster than those using SSH.

We find that the overheads of VNET, especially in the WAN, are acceptable given what it does. The second generation VNET makes them better. Using VNET, we can transport the network management problem induced by VMs back to the home network of the user, where it can be. readily solved, and we can do so with acceptable performance.

3.3 Second generation VNET: An adaptive overlay

The second generation VNET improves upon the first in both, design and performance. In the first iteration, each VM is connected to the outside world and to other VMs through the user's local area network. This creates a star topology centered on the Proxy sitting on the user's network. This design has a single point of failure. It is also not scalable. Additionally, it also creates a performance bottleneck which leads to reduced performance for applications executing inside of the user's VMs.

We observed that often a user's VMs communicate heavily among themselves. With the aim of optimizing the common case, the second generation VNET allows for, in addition to the star topology, arbitrary topologies to be constructed between the different VNET daemons. To achieve this we designed and implemented an Ethernet layer routing protocol. We also effected a series of performance improvements in this second iteration. We

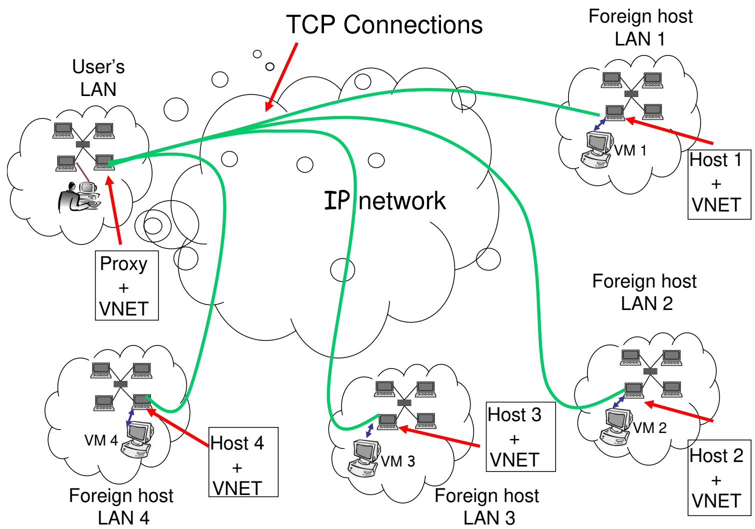


Figure 3.12: VNET startup topology.

first describe the design and operation of this second generation VNET in Sections 3.3.1 through 3.3.3. Its interface is described in Section 3.3.4 and finally we detail the performance improvement in Section 3.3.5.

3.3.1 Operation

Each physical machine that can instantiate virtual machines (a host) runs a single VNET daemon. One machine on the user's network also runs a VNET daemon. This machine is referred to as the Proxy as before. Figure 3.12 shows a typical startup configuration of VNET for four hosts, each of which may support multiple VMs. Each of the VNET daemons running on the foreign hosts is connected by a TCP connection (a VNET link) to the VNET daemon running on the Proxy. We refer to this as the resilient star backbone centered on the Proxy. By resilient, we mean it will always be possible to at least make these connections and re-establish them on failure. It should be noted that the initial setup of the second generation VNET was the only setup possible in the first generation VNET.

The VNET daemons running on the hosts and Proxy open their virtual interfaces in

Source Qualifier	Source Address	Destination Qualifier	Destination Address	Hop Start	Hop End	Interface
not	00:50:56:00:21:01	-	FF:FF:FF:FF:FF:FF	-	-	vmnet1
-	any	-	00:50:56:00:21:01	-	-	vmnet1
-	00:50:56:00:21:01	-	none	Host1	Proxy	-

Source Qualifier : "not" or "-"
Destination Qualifier : "not" or "-"

Source Address : Any valid Ethernet address
Destination Address : Any valid Ethernet address or "None"

Hop Start and Hop End : Physical machines that run VNET daemons
Hop Start – Hop End : TCP connection between those machines

Interface : Interface to which packet has to be injected

For any Ethernet packet multiple rules might be matched at the same time, each match has a priority value and the rule with the highest priority is used.

The rule that has the destination address as "none" is the default rule. This rule is always matched as long as the source address matches, but has the lowest possible priority. The packet would be sent over the TCP connection to the VNET daemon on the Proxy.

Figure 3.13: Portion of a routing table stored on the VNET daemon on a host.

promiscuous mode using Berkeley packet filters [125]. Each VNET daemon has a forwarding table, Figure 3.13 shows one such forwarding table at a VNET daemon. Each packet captured from the interface or received on a TCP connection is matched against this forwarding table to determine where to send it, the possible choices being sending it over one of its outgoing links (TCP / UDP) or writing it out to one of its local interfaces using libnet, which is built on packet sockets, available on both Unix and Windows. If the packet does not match any rule then no action is taken. For each packet multiple rules might be matched at the same time. Each match has a priority value associated with it, calculated dynamically based on the strength of the match. The stronger the match, the higher the priority value. For example, a rule matched with the "any" qualifier will have a lower priority than a rule matched with exact values. The rule with the highest priority is used. The rule that has the destination address as "none" is the default rule. This rule is always matched as long as the source addresses match, but has the lowest possible priority.

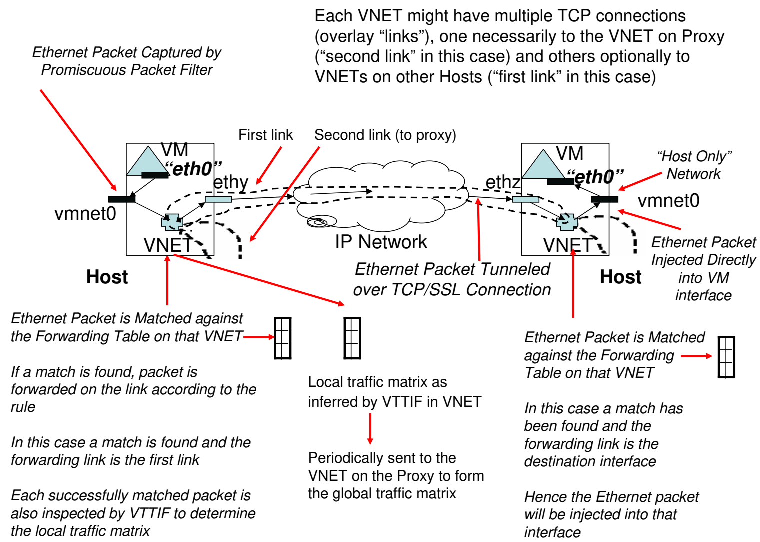


Figure 3.14: A VNET link.

If only the default rule is matched then the packet would be sent over the TCP connection to the VNET daemon on the Proxy.

Figure 3.14 helps to illustrate the operation of a VNET link in its second incarnation. Each successfully matched packet is also passed to VTTIF to determine the local traffic matrix. Each VNET daemon periodically sends its inferred local traffic matrix to the VNET daemon on the Proxy. The Proxy, through its physical interface, provides a network presence for all the VMs on the user's LAN and makes their configuration a responsibility of the user.

The first generation of VNET was limited solely to this star topology [169], thus all traffic among the users' VMs would be forwarded through the central Proxy, resulting in extra latency and a bandwidth bottleneck. The star would be used regardless of the application, as its sole goal was to provide connectivity for the VMs regardless of the security constraints on the various sites.

The second generation VNET removes this restriction. Now, the star topology is simply

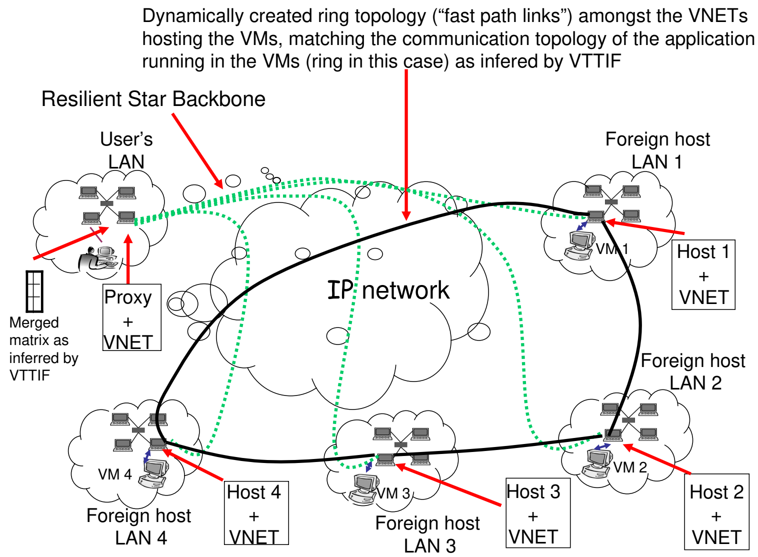


Figure 3.15: As the application progresses VNET adapts its overlay topology to match that of the application communication as inferred by VTTIF leading to an significant improvement in application performance, without any participation from the user.

the initial configuration, again to provide connectivity for the VMs. Additional links and forwarding rules can be added or removed at any time. This makes topology adaptation, as we describe in this thesis, possible. Figure 3.15 shows a VNET configuration that has been dynamically adapted. Also the links can either use TCP or UDP.

3.3.2 VNET primitives

A VNET client can connect to any of the VNET daemons to query status or perform an action. Following are the primitives made available by VNET.

- Add an overlay link between two VNET daemons.
- Delete an overlay link.
- Add a rule to the forwarding table at a VNET daemon.

- Delete a forwarding rule.
- List the available network interfaces.
- List all the links to and from a VNET daemon.
- List all the forwarding rules at a VNET daemon.
- Set an upper bound on VNET configuration time.

The primitives generally execute in about 20 ms, including client time. On initial startup VNET can calculate an upper bound on the time taken to configure itself (or change topology). The last primitive is a means of automatically passing this value to VTTIF, to be used to determine its sampling and smoothing intervals.

3.3.3 A language and its tools

Building on the primitives, we have developed a language for describing the topology and forwarding rules. Figure 3.16 defines the grammar for the language. The tools we use here take the form of scripts that generate or parse descriptions in that language. These tools provide functionality such as:

- Start up a collection of VNET daemons and establish an initial topology among them.
- Fetch and display the current topology.
- Fetch and display the route a packet will take between two Ethernet addresses.
- Compute the differences between the current topology with routing rules and a specified topology with routing rules.

- Reconfigure the topology and routing rules to match a specified topology and routing rules.
- Fetch and display the current application topology using VTTIF (described in Section 3.5).

3.3.4 Interface

The second generation VNET interface is presented in Table 3.3.

As in the first generation, VNET servers are run on the Host and the Proxy. A VNET client can contact any server to query status or to instruct it to perform an action on its behalf.

Session establishment and teardown: The session establishment and teardown mechanism is identical to that of the first generation.

Reaction time: After a VNET client has established a session with a VNET server, it can add/change the recorded time to add/delete a link and its associated routes. This time is specific to each scenario and is recorded at startup when the star topology is created.

VNET link addition and deletion: A VNET client can request a VNET server to create a VNET link (TCP or UDP) between itself and another VNET server. The VNET server then creates this link and updates its and the remote VNET server's data structures. As shown in Table 3.3, the arguments this command are the two end points of the VNET link, its type, TCP or UDP and the password, version and port of the remote VNET server which acts as the endpoint of this requested link.

VNET route addition and deletion: A VNET client can also request a VNET server to add a certain route to its Ethernet level routing table. The arguments to this command are the source and destination Ethernet addresses to be matched, the begining and end point of the link over which the Ethernet packet encapsulated in IP has to be sent out. Notice

$$
\begin{array}{rcl}
\langle\text{program}\rangle & \longrightarrow & \mathit{BEGIN}\,\langle\text{host}\rangle\,\langle\text{config}\rangle\,\mathit{END} \\
\langle\text{host}\rangle & \longrightarrow & \langle\text{host}\rangle\,\mathit{HOST}\,\langle\text{username}\rangle\,\mathit{AT}\,\langle\text{machine}\rangle\,\langle\text{port}\rangle\,\langle\text{interface}\rangle \\
 & & \mid \varepsilon \\
\langle\text{config}\rangle & \longrightarrow & \langle\text{config}\rangle\,\langle\text{action}\rangle\,\langle\text{link}\rangle\,\langle\text{rules}\rangle \mid \varepsilon \\
\langle\text{action}\rangle & \longrightarrow & \mathit{ADD} \mid \mathit{DELETE} \\
\langle\text{link}\rangle & \longrightarrow & \langle\text{link}\rangle\,\mathit{LINK}\,\langle\text{link-type}\rangle\,\langle\text{machine}\rangle\,\langle\text{machine}\rangle\,\langle\text{machine}\rangle \\
 & & \mid \varepsilon \\
\langle\text{link-type}\rangle & \longrightarrow & \mathit{TCP} \mid \mathit{UDP} \\
\langle\text{rules}\rangle & \longrightarrow & \langle\text{rules}\rangle\,\mathit{FORWARD}\,\langle\text{rule-type}\rangle\,\langle\text{machine}\rangle\,\langle\text{qualifier}\rangle \\
 & & \langle\text{macaddress}\rangle\,\langle\text{qualifier}\rangle\,\langle\text{macaddress}\rangle\,\langle\text{destination}\rangle \\
 & & \mid \varepsilon \\
\langle\text{rule-type}\rangle & \longrightarrow & \mathit{EDGE} \mid \mathit{INTERFACE} \\
\langle\text{destination}\rangle & \longrightarrow & \langle\text{machine}\rangle\,\langle\text{machine}\rangle \mid \langle\text{interface}\rangle \\
\langle\text{qualifier}\rangle & \longrightarrow & \mathit{NOT} \mid \mathit{ANY} \mid \varepsilon \\
\langle\text{macaddress}\rangle & \longrightarrow & \textit{Ethernet address} \\
 & & \textit{such as } \mathit{01:02:03:04:05:06} \\
\langle\text{machine}\rangle & \longrightarrow & \textit{Machine name} \\
 & & \textit{such as machine1} \\
\langle\text{username}\rangle & \longrightarrow & \textit{User account on machine} \\
\langle\text{port}\rangle & \longrightarrow & \textit{Port where VNET daemon runs} \\
\langle\text{interface}\rangle & \longrightarrow & \textit{Ethernet interface} \\
 & & \textit{such as eth0} \\
\langle\text{at}\rangle & \longrightarrow & \mathit{AT}
\end{array}
$$

Figure 3.16: Grammar defining the language for describing VNET topology and forwarding rules.

Command	Description
HELLO passwd version	Establish Session
DONE	Finish Session
DEVICES?	Return available network interfaces
LINKS?	Return currently established VNET links
ROUTES?	Return existing VNET routes
REACTION?	Return adaptation reaction time
CHANGEREACTION new_reaction_time	Change the reaction time
ADDLINK vnet_passwd	Add a VNET link
begin_host	
end_host	
type	(TCP or UDP)
end_host_vnet_password	
end_host_vnet_version	
end_host_port	
DELETELINK vnet_passwd	Delete a VNET link
begin_host	
end_host	
type	(TCP or UDP)
end_host_vnet_password	
end_host_vnet_version	
end_host_port	
ADDROUTE vnet_passwd	Add a VNET route
src_address	Ethernet level
dst_address	Ethernet level
hop_begining	
hop_ending	
type	(INTERFACE or EDGE)
DELETEROUTE vnet_passwd	Add a VNET route
src_address	Ethernet level
dst_address	Ethernet level
hop_begining	
hop_ending	
type	(INTERFACE or EDGE)

Table 3.3: Second generation VNET interface.

that the "link" over which this packet has to sent out could also be an interface onto which this packet has to be written. This is specified using the type argument which specifies if the packet is to be sent over a link or is to be written to an interface. If the packet is to be written to an interface then the begining and end point point of the link are identical and refer to the specific interface in question.

Status Enquiry: A client can discover a server's available network interfaces (DEVICES?), what Links it is currently participating in (LINKS?), the snapshot of its current routing table (ROUTES?) and the specific reaction time of the adaptation system (REACTION?).

3.3.5 VNET performance

The overhead of using VNET is mitigated over the wide area. In a 100 Mbps Ethernet LAN, VNET is as fast as state of the art network virtualization software [182], which in turn is almost as fast as the native hardware. However, over faster networks, such as optical networks operating at Giga-bit speeds or higher [91], VNET slightly lagged behind commercial virtual network solutions (in particular VMware GSX Server 2.5.1) [182], which in turn was a factor of 3 slower than the native hardware. We note that since these evaluations were conducted, faster commercial virtual networking solutions have been introduced. It should be noted that VNET currently operates entirely at user level. Hence, there are enormous opportunities to significantly increase VNET's performance beyond that of commercial virtual networking solutions. Below, we first describe the improvements engineered in the second generation VNET and then state our future plans for further optimizing VNET performance.

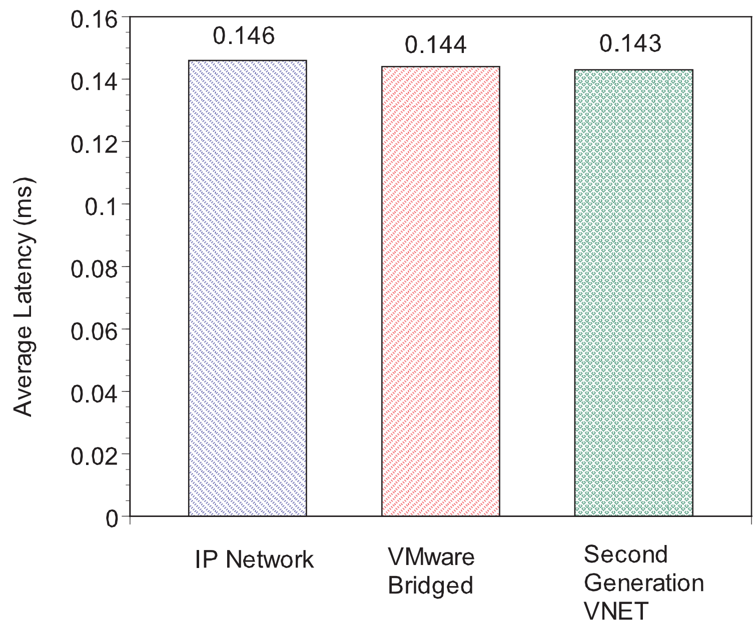


Figure 3.17: Average latency over a 100 Mbit switch with different bottlenecks.

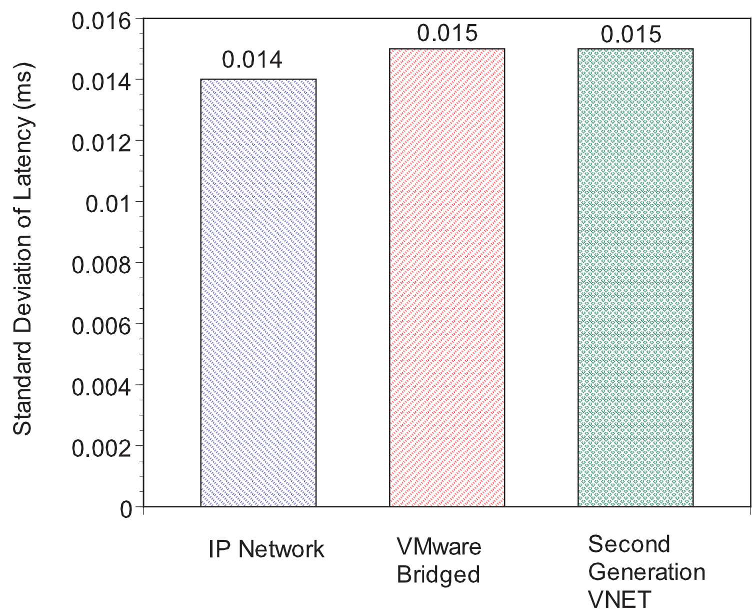


Figure 3.18: Standard deviation of latency over a 100 Mbit switch with different bottlenecks.

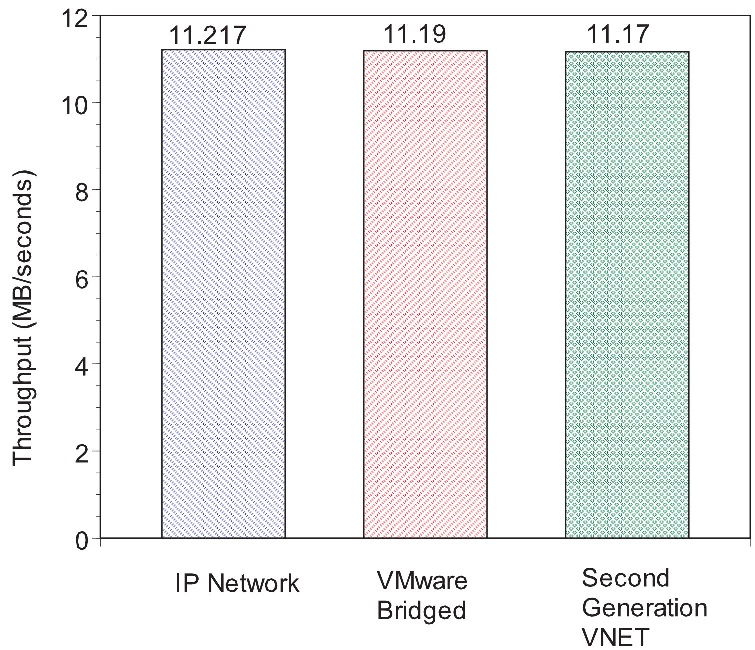


Figure 3.19: Throughput on a 100 Mbit switch with different bottlenecks.

3.3.6 Performance over 100 Mbps Ethernet LAN

In the performance analysis of the first generation VNET, we conducted experiments wherein a physical Client machine interacted with the. user's VM. We looked at the average latency, standard deviation of latency and the throughput experienced for such an interaction when using different methods of network connectivity.

To evaluate the performance of the second generation VNET, we conducted experiments wherein a single user's two VMs interacted with each other. For such an interaction we study the average latency, standard deviation of latency and throughput expericed. We compare three scenarios against each other, connectivity using the IP network, using VMware bridged networking and via the second generation VNET. The latency studies were carried out using ping while the throughput studies were carried out using ttcp.

Figure 3.17 compares the average latency experienced between the two VMs for different scenarios. We notice that there is negligible difference between the different solutions. Figure 3.18 compares the standard deviation of latency for different scenarios. Again we see that both, VMware bridged networking and VNET have negligible overheads in an 100 Mbit LAN. It should be noted that the second generation VNET eliminates the variability in latency that the first iteration had introduced. This is due to the use of UDP links as opposed to the previously used TCP links and is discussed in more detail in Section 3.3.7. Finally, Figure 3.19 compares the throughput as seen in the different scenarios. Once again, we notice that both VMware bridged networking and VNET have virtually no overheads in a 100 Mbit LAN.

3.3.7 UDP overlay links

In the first version of VNET, all VNET overlay links were TCP connections. The primary reason was to make it straightforward to support optional SSL encryption. This is not essential, since a VNET link is a virtual Ethernet layer link and thus needs to provide no guarantees of delivery, ordering, or corruption.

Running VNET over TCP results in lower than necessary throughput for applications running inside the VMs. Interaction between the TCP connection at the application layer and the TCP connection used for the VNET overlay dramatically reduces performance. A packet loss in the underlying VNET TCP connection will lead to a retransmission and hence a delay for the application's TCP connection, which in turn could time out and retransmit itself. The application's TCP connection will always detect a packet loss by the expiration of the retransmission timer rather than by receipt of triple duplicate acknowledgments. This will then always trigger slow start instead of fast retransmit, leading to reduced throughput.

VNET now supports creation of overlay links using UDP in addition to TCP. This increased the throughput seen by application-level TCP by a factor of two.

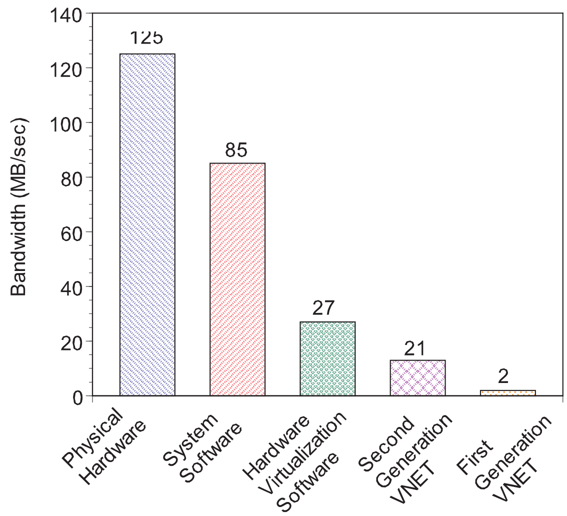


Figure 3.20: Throughput on a gigabit switch with different bottlenecks.

3.3.8 Improved forwarding rule lookup

VNET forwards Ethernet packets. When a packet arrives, it must look up the appropriate forwarding rule based on the packet's destination address. We have improved the lookup mechanism through a forwarding rule cache that gives us constant time lookups on average. This improved performance by a factor of three.

3.3.9 Utilizing memory-mapped I/O support in pcap

VNET performance was substantially improved (factor of two) by utilizing the memory-mapped I/O support in pcap to deliver more data from the VM to VNET for each context switch.

3.3.10 Further improving VNET performance

We have improved the performance of VNET, as measured on a dedicated gigabit Ethernet network, by a factor of ten. However, there is still considerable room for improvement and we are a long way from being able to support gigabit speeds in VNET. Figure 3.20 illustrates where we stand. The hardware virtualization software refers to VMware GSX Server 2.5.1. Some additional possible extensions to VNET are

- To move the forwarding core of VNET into the Linux kernel on the host to avoid context switches in their entirety.
- To write a device driver for use inside the VM that will more efficiently deliver data to VNET.

3.4 Third generation VNET: A virtual transport layer

Lange et al., at Northwestern University, are currently working on a third generation VNET. The third generation VNET includes a framework that makes possible the creation of a Virtual Transport Layer (VTL). At its heart VTL is a framework for packet modification and creation, whose purpose is to modify network traffic to and from a VM in such a way that is undetectable by the VM itself. Lange et al. are currently working on demonstrating the promise of such techniques.

3.4.1 Application independent adaptation mechanisms

This dissertation argues that virtual distributed computing systems provide application independent adaptation mechanisms that can then be used to build an automatic, run-time and dynamic adaptation system. The adaptation engine which drives the adaptation mechanisms is called VADAPT. Virtuoso provides for the following non-reservation based adaptation mechanisms:

- **Virtual machine migration:** Virtuoso allows us to migrate a VM from one physical host to another. Much work exists that demonstrates that fast migration of VMs running commodity applications and operating systems is possible [107, 134, 137, 146] including live migration schemes with downtime on the order of a few seconds [25]. As migration times decrease, the rate of adaptation we can support and my work's relevance increases. Note that while process migration and remote execution has a long history [41, 128, 161, 176, 193], to use these facilities, we must modify or relink the application and/or use a particular OS. Neither is the case with VM migration.

- **Overlay topology modification:** VNET allows us to modify the overlay topology among a user's VMs at will. A key difference between it and overlay work in the application layer multicast community [8, 12, 84] is that the VNET provides global control of the topology, which our adaptation algorithms currently (but not necessarily) assume.

- **Overlay forwarding:** VNET allows us to modify how messages are routed on the overlay. Forwarding tables are globally controlled, topology and routing are completely separated, unlike in multicast systems.

Virtuoso also provides for the following reservations based adaptation schemes:

- **Network reservation:** The VRESERVE component of Virtuoso allows the control system to detect when the underlying network provides for network reservations and then to reserve appropriate networks on behalf of the application for the appropriate period of time. Adaptation using VRESERVE is detailed in Chapter 5.

- **CPU reservation:** The VSched component of Virtuoso allows the control system to reserve CPU resources as a function of *period* and *slice*. VSched is a user level periodic real-time scheduler that can reserve the CPU for *slice* seconds every *period* seconds. Adaptation using VSched is detailed in Chapter 6.

3.5 VTTIF

The Virtual Topology and Traffic Inference Framework (VTTIF) enables topology inference and traffic characterization for applications running inside the VMs in the Virtuoso system. VTTIF infers the application's communication demands which along with the underlying network measurements guide the adaptation mechanisms. As described earlier, such inference is important for automated adaptation where the underlying network and computational resources can be automatically adapted to the application's needs. VNET is ideally placed to monitor the resource demands of the VMs. In earlier work [71], Gupta et al. have demonstrated that it is possible to successfully infer the topology and traffic load matrix of a bulk synchronous parallel application running in a virtual machine-based distributed computing environment by observing the low level traffic sent and received by each VM. VTTIF is integrated with VNET.

VTTIF is configured by three parameters:

- Update rate: The rate at which local traffic matrix updates are sent from the VNET daemons to the VNET daemon running on the Proxy.
- Smoothing interval: The window over which the global traffic matrix on the Proxy is aggregated from the updates received from the other VNET daemons. This provides a low-passed view of the application's behavior.
- Detection threshold: The fraction of traffic intensity on the highest intensity link that must be present on any other link before it is considered to be a part of the topology.

Given some configured update rate, smoothing interval, and detection threshold, there is a maximum rate of topology change that VTTIF can keep up with. Beyond this rate, VTTIF has been designed to stop reacting, settling into a topology that is effectively a union of all the topologies that are unfolding in the network. VTTIF is described in more detail in Appendix A.

3.6 Wren

To successfully build an adaptive system, in addition to inferring the application communication demands, we need to be able to measure the underlying network. The virtual overlay network can then be adapted in the most efficient and productive way. There is abundant work that suggests that underlying network measurements can be accomplished within or without the virtual network using both active [148, 190] and passive techniques [118, 150, 194]. We have shown that the naturally occurring traffic of an existing, unmodified application running in VMs can be used to measure the underlying physical network [73]. In this Section, I will briefly describe this work. Wren is described in greater detail in Appendix B.

3.6.1 Operation

Network measurement using Wren in VNET is based on Zangrilli et al.'s Wren passive monitoring and network characterization system [194, 195]. Many applications adapt to network performance simply by observing the throughput of their own network connections. VNET's natural abstraction of the underlying network makes such application-level adaptation more difficult because the application cannot accurately determine which network resources are in use. However, VNET is in a good position to observe an application's traffic itself. Because VNET does not alter that traffic, it can only observe the amount of traffic naturally generated by the application. Because we are targeting applications with potentially bursty and irregular communication patterns, many applications will not generate enough traffic to saturate the network and provide useful information on the current bandwidth achievable on the network.

Watching Resources from the Edge of the Network (Wren) is designed to passively monitor applications' network traffic and use those observations to determine the available bandwidth along the network paths used by the application. The key observation behind Wren is that even when the application is not saturating the network it is sending bursts of traffic that can be used to measure the available bandwidth of the network. A good example of such an application is a typical scientific computing BSP-style application that sends short messages at regular intervals. Even though the application is not using all of the available bandwidth, we can determine the available bandwidth along that path and use that information to guide adaptation.

Wren consists of a kernel extension and a user-level daemon. Wren can:

- Observe every incoming and outgoing packet arrivals in the system with low overhead.
- Analyze these arrivals using state-of-the-art techniques to derive from them latency and bandwidth information for all hosts that the present host communicates with.
- Collect latency, available bandwidth, and throughput information so that an adaptation algorithm can have a bird's eye view of the physical network, just as it has a bird's eye view of the application topology via VTTIF.
- Answer queries about the bandwidth and latency between any pair of machines in the virtual network.

3.6.2 Wren and Virtuoso

Figure 1.5 shows Virtuoso's interaction with Wren. Virtuoso and Wren are integrated by incorporating the Wren extensions into the Host operating system of the machines running

VNET [73]. In this position, Wren monitors the traffic between VNET daemons, not between individual VMs. Both the VMs and VNET are oblivious to this monitoring, except for a small performance degradation.

It should be noted that Wren requires an operating system kernel modification, while we claim that our adaptation scheme works for un-modified applications executing on um-modified operating systems. In the following we explain this contradiction.

Our adaptation scheme relies on Wren only for physical network measurements. While there are a number of other third party network measurement tools available [159, 190], we chose Wren due its passive measurement architecture. As mentioned above Wren is capable of using the naturally occurring traffic inside of VMs to make its measurement estimates. Other network schemes, though require no modifications to the operating systems, perform active measurements resulting in network traffic overhead. Since our adaptation scheme is not tightly bound to Wren, it can also work with other adaptation schemes that require no operating system modifications. Wren is described in greater detail in Appendix A.

3.7 Conclusions

We described the design, implementation and evaluation of VNET, the virtual networking component of Virtuoso. The evolution of VNET has seen it undergo two generations of design. The first generation VNET was solely concerned with creating and maintaining the illusion that all of a user's VMs are connected to his local area network, where they can be administered in a familiar fashion by the user's local network administrator. We found VNET to have negligible overhead over wide-area networks, but to have significant overheads over a 100Mbit LAN. The first generation VNET, though sufficient for its initial intended purpose (extending the LAN abstraction to a user's VMs), has certain drawbacks.

All of a user's VMs' traffic necessarily flows back through his LAN creating bottlenecks and a single point of failure. TCP VNET links were the only option and UDP support was not provided.

The second generation VNET was developed to address the issues in the first generation VNET. The insight obtained through the first iteration of design was that virtual networks are not just a means of providing connectivity to a user's virtual machines, instead they have the potential to blossom into an adaptive overlay network supporting application independent adaptation mechanisms. The second generation VNET allows for arbitrary topologies to be built between VNET daemons, further UDP support is also provided, thus completing the LAN abstraction. We also considerably improved the implementation of VNET. The second generation has no virtualization overhead over a 100Mbit LAN. Over faster networks operating at gigabit speeds, it only slightly trails the most popular commercial virtual networking solution.

We also described the reservation and non-reservation based application independent adaptation mechanisms, including virtual machine migration, virtual topology and routing changes, and network and CPU based reservation schemes.

Beyond virtual machines, virtual networks and application independent adaptation mechanisms, to achieve successful automatic adaptation, we need some means to infer application demands and measure available system resources. We describe VTTIF and Wren designed and implemented by Gupta et al. and Zangrilli et al. respectively. VTTIF and Wren, both have been integrated with Virtuoso. VTTIF is the topology and traffic inference framework that, while invisible to the application, infers its communication traffic matrix by looking at Ethernet level traffic flowing between the VNET daemons. Wren is a passive network measurement tool that uses the naturally occurring traffic inside of the virtual machines to make physical network measurements. Both, VTTIF and Wren has been shown to have negligible overheads.

Chapter 4

Problem formulation

Virtual execution environments, such as Virtuoso, provide an ideal platform for inferring application resource demands and measuring available computational and network resources. Further, they also make available application independent adaptation mechanisms such as VM migration, overlay network topology and routing changes and resource reservations. However, the key to success is an efficient algorithm to drive these adaptation mechanisms as guided by the measured and inferred data. To gain a better understanding of the adaptation problem and to devise efficient algorithms it is important to first formalize and characterize the adaptation problem.

As mentioned in Chapter 1, three techniques of evaluating a computer system are experimentation, simulation and analytical modeling. This dissertation uses all these three methodologies to study the adaptation problem. This is essential as each technique has its limitations and is best suited to study different aspects of the adaptation problem. Analytical characterization of the adaptation problem is the topic of discussion in this chapter.

In this chapter we provide a rigorous formalization of the adaptation problem that occurs in virtual execution environments. We characterize its computational complexity and hardness of approximation.

Section 4.1 formalizes a real adaptation problem that occurs in virtual execution environments. Though this work has been done in the context of Virtuoso, the formulation is generic enough to apply to other virtual distributed computing systems as well [15]. To the best of our knowledge there currently exists no theoretical analysis for real problems that involve both mapping and routing aspects. Additionally, the formalization is also abstract and generic enough to allow other adaptation problems in many different contexts, such as hardware chip design [77], to possibly map onto it.

We explore the computational complexity of the adaptation problem in Section 4.2. The adaptation problem intuitively appears to be NP-hard. This dissertation takes the intuition all the way through to rigorous formalization and proves that the adaptation problem in virtual execution environment is indeed NP-hard. This work has the potential to act as the basis for the analysis of future virtualized adaptive systems. By proving the problem to be NP-hard we necessitate the search for approximate solutions.

Section 4.3 builds upon the results derived in Section 4.2 by studying the approximability or inapproximability of the adaptation problem. In particular, it characterizes the problem's hardness of approximation by proving that it is NP-hard to approximate within a factor of $m^{1/2-\delta}$ for any $\delta > 0$, where m is the number of edges in the virtual overlay graph. This shows that the adaptation problem is hard to approximate as well.

4.1 Adaptation problem formulation

VNET monitors the underlying network and provides a directed VNET topology graph, $G = (H, E)$, where H are VNET nodes (hosts running VNET daemons and capable of supporting one or more VMs) and E are the possible VNET links. Note that this may not be a complete graph as many links may not be possible due to particular network management and security policies at different network sites. Wren [74] (integrated with VNET [73]) provides estimates for the available bandwidth and latencies over each link in the VNET topology graph. These estimates are described by a bandwidth capacity function, bw : $E \to \mathbb{R}$, and a latency function, lat : $E \to \mathbb{R}$.

In addition, VNET is also in a position to collect information regarding the space capacity (in bytes) and compute capacity made available by each host, described by a host compute capacity function, compute : $H \to \mathbb{R}$ and a host space capacity function, size : $H \to \mathbb{R}$. The set of virtual machines participating in the application is denoted by the set VM. The size and compute capacity demands made by every VM can also be estimated and denoted by a VM compute demand function, vm_compute : $VM \to \mathbb{R}$ and a VM space demand function, vm_size : $VM \to \mathbb{R}$, respectively. We are also given an initial mapping of virtual machines to hosts, $\mathcal{M}$, which is a set of 3-tuples, $M_i = (\mathrm{vm}_i, h_i, y_i)$, $i = 1, 2 \ldots n$, where $\mathrm{vm}_i \in$ VM is the virtual machine in question, $h_i \in H$ is the host that it is currently mapped onto and $y_i \in \{0,1\}$ specifies whether the current mapping of VM to host can be changed or not. A value of 0 implies that the current mapping can be changed and a value of 1 means that the current mapping should be maintained. Additionally, we also obtain an estimate of the execution time remaining for each VM. We denote this by vm_time : $VM \to \mathbb{R}$.

The bandwidth and compute rate estimates do not implicitly imply reservation, they are random variables that follow a normal distribution with a mean of the estimated value. As mentioned previously Virtuoso provides for network and CPU reservations, in which case the estimates are exactly the resources we get as we can reserve the same. Hence for each edge in E, we define a function nw_reserve : $E \to \{0,1\}$. If the value associated with the edge is 0 then we cannot reserve the link and the actual bandwidth has a normal distribution with a mean of $\mathrm{bw}(E)$ and a variance $\sigma^2_{bw(E)}$, else the link is reservable and the actual bandwidth is $\mathrm{bw}(E)$. Similarly for each host we define a function cpu_reserve : $H \to \{0,1\}$, where a value of 0 means that the compute capacity made available by the host is not reservable and the actual value has a normal distribution with a mean of $\mathrm{compute}(H)$

and a variance $\sigma^2_{\text{compute}(H)}$. The Gaussian model is based on the ideal error distribution from a resource prediction service [35].

VTTIF infers the application communication topology in order to generate the traffic requirements of the application, $\mathcal{A}$, which is a set of 4-tuples, $A_i = (s_i, d_i, b_i, l_i)$, $i = 1, 2 \ldots m$, where s_i is the source VM, d_i is the destination VM, b_i is the bandwidth demand between the source destination pair and l_i is the latency demand between the source destination pair. The bandwidth demand refers to the bandwidth that the application would like to receive between its source destination pairs. The latency demand refers to the latency that the application would not like exceeded between its communicating VM pairs. It should be noted that these demands are not explicitly given by the application, but are inferred.

It should be noted that there is always a cost involved with all the measurements and adaptation mechanisms. Because the overheads of VNET, VTTIF and Wren have been shown to be negligible [74] we do not include them in our formalization. However, the cost of migrating a virtual machine is dependent on the size of the virtual machine, the network characteristics between the corresponding hosts and the specific migration scheme used. These estimates are described by a migration function, migrate: VM x H x H $\rightarrow \mathbb{R}^+$, that provides an estimate in terms of the time required to migrate a virtual machine from one host to another. There is more than one way to take into account the cost of migration, one being to keep the costs of migration for each of the VMs below a certain threshold. Online migration of virtual machines is receiving a lot of interest in the research community [26, 107, 146]. As the migration times are being continually driven down the relevance of our work will continue to increase.

The goal then is to find an adaptation algorithm that uses the measured and inferred data to drive the adaptation mechanisms at hand in order to improve application throughput. In other words we wish to find

1. a mapping from VMs to hosts, $\text{vmap} : VM \to H$, meeting the size and compute capacity demands of the VMs within the host constraints and leveraging CPU reservations where available. Further, the new mapping should also reflect the mapping constraints provided.

2. a routing, $R : \mathcal{A} \to \mathcal{P}$, where $\mathcal{P}$ is the set of all paths in the graph $G = (H, E)$, i.e. for every 4-tuple, $A_i = (s_i, d_i, b_i, l_i)$, allocate a path, $p\big(\text{vmap}(s_i), \text{vmap}(d_i)\big)$, over the overlay graph, G, meeting the application demands while satisfying the bandwidth and latency constraints of the network and leveraging network reservations where available.

Once all the mappings and paths have been decided, we can compute the change in estimated execution times on this new mapping and denote it by $\text{vm\_time\_after} : VM \to \mathbb{R}$. Additionally, each VNET edge will have a residual capacity, rc_e, which is the bandwidth remaining unused on that edge, in that direction

$$\text{rc}_e = \text{bw}_e - \sum_{e \in R(A_i)} b_i$$

For each mapped path, $R(A_i)$, we can also define its bottleneck residual capacity

$$\text{brc}\big(R(A_i)\big) = \min_{e \in R(A_i)} \big\{\text{rc}_e\big\}$$

and its total latency

$$\text{tl}\big(R(A_i)\big) = \sum_{e \in R(A_i)} \big(lat_e\big)$$

. Note that here we are making the assumption that the end-to-end latency is the same as the sum of the individual latencies.

It should be noted that the residual capacity can be spoken of at two levels, at the level of VNET edges and at the level of paths between communicating VMs. The network component of the various objective functions that could be defined would fall into one of two classes, an edge-level or a path-level objective function.

1. Edge-level: a composite function, f, that is a function of, g, a function of the migration costs of all the VMs and h, a function of the total latency over all the edges for each routing and k, a function of the residual bottleneck bandwidths over all the edges in the VNET graph.

2. Path-level: a composite function, f, that is a function of, g, a function of the migration costs of all the VMs and h, a function of the total latency over all the edges for each routing and k, a function of the residual bottleneck bandwidths over all the paths in the routing.

Table 4.1 provides a description of the terms used in the formulation.

Problem 1 (Generic Adaptation Problem In Virtual Execution Environments (GAPVEE))

INPUT:

- A directed graph $G = (H, E)$
- A function bw : $E \rightarrow \mathbb{R}$
- A function lat : $E \rightarrow \mathbb{R}$
- A function compute : $H \rightarrow \mathbb{R}$
- A function size : $H \rightarrow \mathbb{R}$
- A set, $\mathrm{VM} = (\mathrm{vm}_1, \mathrm{vm}_2 \ldots \mathrm{vm}_n)$, $n \in \mathbb{N}$
- A function vm_compute : $\mathrm{VM} \rightarrow \mathbb{R}$
- A function vm_size : $\mathrm{VM} \rightarrow \mathbb{R}$
- A function vm_time : $\mathrm{VM} \rightarrow \mathbb{R}$
- A function migrate : $(VM, H, H) \rightarrow \mathbb{R}$
- A function nw_reserve : $E \rightarrow \{0, 1\}$
- A function cpu_reserve : $H \rightarrow \{0, 1\}$
- A set of ordered 4-tuples $\mathcal{A} = \{(s_i, d_i, b_i, l_i) \mid s_i, d_i \in \mathrm{VM};\ b_i, l_i \in \mathbb{R};\ i = 1, \ldots, m\}$
- A set of ordered 3-tuples $\mathcal{M} = \{(\mathrm{vm}_i, h_i, y_i) \mid \mathrm{vm}_i \in \mathrm{VM};\ h_i \in H;\ y_i \in \{0, 1\};\ i = 1, \ldots, n\}$

OUTPUT: vmap : $VM \rightarrow H$ and $R : \mathcal{A} \rightarrow \mathcal{P}$ such that

- $\sum_{\mathrm{vmap(vm)}=h} \big(\mathrm{vm\_compute(vm)}\big) \leq \mathrm{compute(h)}, \forall\, h \in H$

Symbol	Description
$G = (H, E)$	VNET overlay graph
H	VNET nodes
E	possible VNET links
$\mathrm{bw} : E \to \mathbb{R}$	bandwidth capacity function
$\mathrm{lat} : E \to \mathbb{R}$	latency function
$\mathrm{compute} : H \to \mathbb{R}$	host compute capacity function
$\mathrm{size} : H \to \mathbb{R}$	host space capacity function
VM	set of virtual machines
$\mathrm{vm\_compute} : VM \to \mathbb{R}$	VM compute demand function
$\mathrm{vm\_size} : VM \to \mathbb{R}$	VM space demand function
$\mathcal{M}$	initial mapping of virtual machines to hosts
$M_i = (\mathrm{vm}_i, h_i, y_i)$, $i = 1, 2 \ldots n$	Initial mapping as a set of 3-tuples
$\mathrm{vm}_i \in \mathrm{VM}$	a specific virtual machine in question
$h_i \in H$	host that a VM is currently mapped onto
$y_i \in \{0, 1\}$	whether the current mapping of VM to host can be changed
$\mathrm{vm\_time} : VM \to \mathbb{R}$	execution time remaining for each VM
$\mathrm{nw\_reserve} : E \to \{0, 1\}$	whether an edge can be reserved
$\mathrm{bw}(E)$	expected mean of distribution of actual bandwidth
$\sigma^2_{bw(E)}$	expected variance of distribution of actual bandwidth
$\mathrm{cpu\_reserve} : H \to \{0, 1\}$	whether the CPU can be reserved
$\mathrm{compute}(H)$	expected mean of distribution of actual compute capacity
$\sigma^2_{\mathrm{compute}(H)}$	expected variance of distribution of actual compute capacity
$\mathcal{A}$	application communication topology
$A_i = (s_i, d_i, b_i, l_i)$, $i = 1, 2 \ldots m$	application communication topology as 4-tuples
d_i	destination VM
s_i	source VM
b_i	bandwidth demand between source destination pair
l_i	latency demand between source destination pair
migrate: VM x H x H $\to \mathbb{R}^+$	estimate of migration time
$\mathrm{vmap} : VM \to H$	final mapping from VMs to hosts
$R : \mathcal{A} \to \mathcal{P}$	final routing from application 4-tuples to paths
$p\big(\mathrm{vmap}(s_i), \mathrm{vmap}(d_i)\big)$	path over overlay graph
$\mathrm{vm\_time\_after} : VM \to \mathbb{R}$	change in estimated execution times
rc_e	residual capacity on VNET edges
$\mathrm{rc}_e = \mathrm{bw}_e - \sum_{e \in R(A_i)} b_i$	definition of residual capacity
$\mathrm{brc}\big(R(A_i)\big)$	bottleneck residual capacity for each mapped path
$\mathrm{brc}\big(R(A_i)\big) = \min_{e \in R(A_i)} \{\mathrm{rc}_e\}$	definition of bottleneck residual capacity
$\mathrm{tl}\big(R(A_i)\big)$	total latency over each mapped path
$\mathrm{tl}\big(R(A_i)\big) = \sum_{e \in R(A_i)} \big(lat_e\big)$	definition of total latency

Table 4.1: Description of symbols used in the formalization.

- $\sum_{\text{vmap(vm)}=h} \big(\text{vm\_size(vm)}\big) \leq \text{size}(h), \forall\, h \in H$
- $h_i = \text{vmap}(\text{vm}_i) \quad \forall\, M_i = (\text{vm}_i, h_i) \in \mathcal{M}$ if $y_i = 1$
- $\text{rc}_e \geq 0, \forall e \in E$
- $\big(\sum_{e \in R(A_i)} \text{lat}_e\big) \leq l_i, \forall e \in E$
- For some functions f, g, h, k and l the function $f(g(migrate), h(\text{lat}), k(\text{rc}_e)), l(\text{vm\_time\_after})$ is optimized

It should be noted that for this most generic incarnation we have not specified any particular objective function. The intent of providing this formulation is to provide an abstract description of all the components of the adaptation problem. We next take a significant piece of this generic problem and analyze and characterize it in great detail.

Mapping and routing are the two main components of our adaptation problem. With a view to better understand these two components we define a simpler version wherein we drop the size, compute and latency constraints. We also neglect the cost of migration, which is reasonable as recently migration costs as low as a few seconds have been reported [107]. It should be noted that if the migration is conducted online then the downtime is virtually zero [26]. We also assume that all the links are reservable and that the compute capacity made available is reserved as well.

The specific objective function we choose belongs to the second category mentioned above wherein we consider residual bandwidths of the various paths in the routing. Additionally, we drop the compute aspect in the objective function and restrict it to the extent of constraints. The objective is to maximize the sum of residual bottleneck bandwidths over each mapped path, where residual bottleneck bandwidth is as defined previously in this Section. The intuition behind this objective function is to leave the most room for the application to increase its throughput.

Problem 2 (Mapping and Routing Problem In Virtual Execution Environments (MARPVEE))

INPUT:

- A directed graph $G = (H, E)$
- A function $\mathrm{bw} : E \to \mathbb{R}$
- A set, $\mathrm{VM} = (\mathrm{vm}_1, \mathrm{vm}_2 \ldots \mathrm{vm}_n)$, $n \in \mathbb{N}$
- A set of ordered 3-tuples $\mathcal{A} = \{(s_i, d_i, b_i) \mid s_i, d_i \in \mathrm{VM};\ b_i;\ i = 1, \ldots, m\}$
- A set of ordered 3-tuples $\mathcal{M} = \{(\mathrm{vm}_i, h_i, y_i) \mid \mathrm{vm}_i \in \mathrm{VM};\ h_i \in H;\ y_i \in \{0, 1\};\ i = 1, \ldots, n\}$

OUTPUT: $\mathrm{vmap} : VM \to H$ and $R : \mathcal{A} \to \mathcal{P}$ such that

- $h_i = \mathrm{vmap}(\mathrm{vm}_i) \quad \forall\, M_i = (\mathrm{vm}_i, h_i) \in \mathcal{M}$ if $y_i = 1$
- $\mathrm{rc}_e \geq 0, \forall e \in E$
- $\sum_{i=1}^{m} \left(\min_{e \in R(A_i)} \{\mathrm{rc}_e\} \right)$, where $\mathrm{rc}_e = (\mathrm{bw}_e - \sum_{e \in R(A_i)} b_i)$, is maximized

From now on when we refer to the adaptation problem we will be referring to MARPVEE.

4.2 Computational complexity of the adaptation problem

We first formulate the decision version of the adaptation problem.

Problem 3 (Mapping and Routing Problem In Virtual Execution Environments (MARPVEED))

INPUT:

- A directed graph $G = (H, E)$
- A function $\mathrm{bw} : E \to \mathbb{R}$
- A set, $\mathrm{VM} = (\mathrm{vm}_1, \mathrm{vm}_2 \ldots \mathrm{vm}_n)$, $n \in \mathbb{N}$
- A set of ordered 3-tuples $\mathcal{A} = \{(s_i, d_i, b_i) \mid s_i, d_i \in \mathrm{VM};\ b_i;\ i = 1, \ldots, m\}$
- A set of ordered pairs $\mathcal{M} = \{(\mathrm{vm}_i, h_i) \mid \mathrm{vm}_i \in \mathrm{VM}, h_i \in H;\ i = 1, 2 \ldots r, r \leq n\}$
- $\alpha \in \mathbb{R}$

OUTPUT:

- YES, if there exists a mapping $\mathrm{vmap} : VM \to H$ and a routing $R : \mathcal{A} \to \mathcal{P}$ such that
  - $h_i = \mathrm{vmap}(\mathrm{vm}_i), \forall M_i = (\mathrm{vm}_i, h_i) \in \mathcal{M}$
  - $\mathrm{rc}_e \geq 0, \forall e \in E$
  - $\sum_{i=1}^{m} \left(brc(R(A_i)) \right) \geq \alpha$
- NO, otherwise

To establish the hardness of the problem, we consider a further special case of the problem wherein all the VM to host mappings are constrained by the set of 3-tuples $\mathcal{M}$, leaving us only with the routing problem.

Since the mappings are pre-defined, we can formulate the problem in terms of only the hosts and exclude all VMs. Also, as the latency demands have been dropped, the application 4-tuple reduces to 3-tuple, $A_i = (s_i, d_i, b_i),\ s_i, d_i \in H,\ b_i \in \mathbb{R},\ i = 1, 2 \ldots m$. Notice that now $s_i, d_i \in H$ as VM to host mappings are fixed and VMs are synonymous with the hosts that they are mapped to.

This further constrained version of the adaptation problem with only the routing component is defined as follows.

Problem 4 (Routing Problem In Virtual Execution Environments (RPVEE))

INPUT:

- A directed graph $G = (H, E)$
- A function bw : $E \to \mathbb{R}$
- A set of ordered 3-tuples $\mathcal{A} = \{(s_i, d_i, b_i) \mid s_i, d_i \in H;\ b_i \in \mathbb{R};\ i = 1, \ldots, m\}$

OUTPUT: $R : \mathcal{A} \to \mathcal{P}$ such that

- $\mathrm{rc}_e \geq 0, \forall e \in E$,
- $\sum_{i=1}^{m} \big(brc(R(A_i))\big)$ is maximized

Further, The decision version of RPVEE can be formulated as follows.

Problem 5 (Decision version of Routing Problem In Virtual Execution Environments (RPVEED))

INPUT:

- A directed graph $G = (H, E)$
- A function bw : $E \to \mathbb{R}$

- A set of ordered 3-tuples $\mathcal{A} = \{(s_i, d_i, b_i) \mid s_i, d_i \in H;\ b_i \in \mathbb{R};\ i = 1, \ldots, m\}$
- $\alpha \in \mathbb{R}$

OUTPUT:

- YES, if there exists a routing $R : \mathcal{A} \to \mathcal{P}$ such that
  - $\text{rc}_e \geq 0, \forall e \in E$;
  - $\sum_{i=1}^{m} \big(brc(R(A_i))\big) \geq \alpha$
- NO, otherwise

For the proofs of hardness we will reduce the Edge Disjoint Path Problem to the Routing Problem in Virtual Execution Environments. The edge disjoint problem has been shown to be NP-complete [101] and NP-hard to approximate within a factor of $m^{1/2-\delta}$ [75].

The edge disjoint path problem can be formulated as follows.

Problem 6 (The Edge Disjoint Path Problem (EDPP))

INPUT:

- A graph $G = (H, E)$, $|H| = p$, $|E| = q$
- A set of ordered 2-tuples $\mathcal{S} = \{(s_i, d_i) \mid s_i, d_i \in H;\ i = 1, \ldots, k\}$

OUTPUT:

- The maximum numbers of pairs $(s_i, d_i) \in \mathcal{S}$ that can be connected via edge disjoint paths from s_i to d_i in $G = (H, E)$

Further, the decision version of the edge disjoint path problem can be stated as follows.

Problem 7 (Decision version of Edge Disjoint Path Problem (EDPPD))

INPUT:

- A directed graph $G = (H, E)$, $|H| = p$, $|E| = q$
- A set of ordered 2-tuples $\mathcal{S} = \{(s_i, d_i) \mid s_i, d_i \in H;\ i = 1, \ldots, k\}$

OUTPUT:

- YES, if $\forall (s_i, d_i) \in \mathcal{S}$ there exist edge disjoint paths from s_i to d_i in $G = (H, E)$
- NO, otherwise

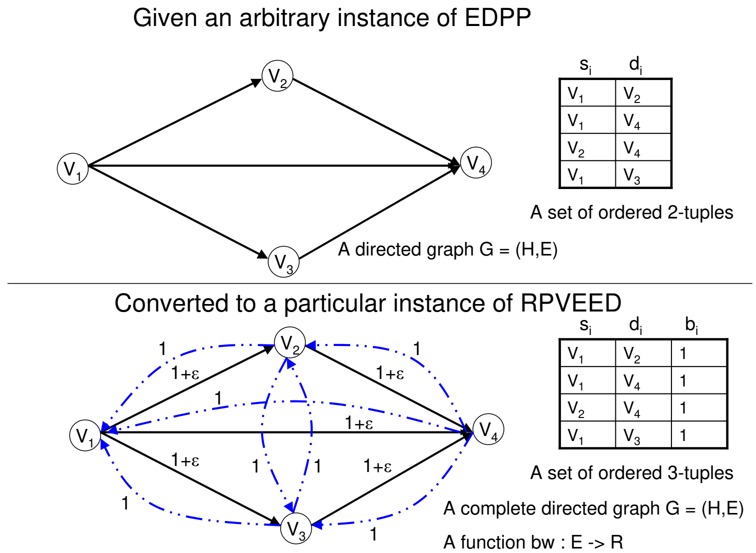


Figure 4.1: Reducing EDPPD to RPVEED. The edge weights are bandwidths as specified by the function bw.

4.2.1 Reduction of the Edge Disjoint Path Problem to the Routing Problem in Virtual Execution Environments

Given an instance $I = \{S, G = (H,E)\}$ of EDPPD or EDPP we reduce it to an instance $R(I)$ of RPVEE or the instance $R_D(I)$ or RPVEED as follows. Construct a complete directed graph $G' = (H,E')$ where $\mathrm{bw}((u,v)) = 1+\varepsilon$ for $\varepsilon < 1$ if $(u,v) \in E$ and $\mathrm{bw}((u,v)) = 1$ if $(u,v) \notin E$. Further for all $(s_i,t_i) \in S$, let $(s_i,d_i,1) \in \mathcal{A}$ (see Figure 4.1) to get the instance $R(I)$ for RPVEE. Let $\alpha = k{\cdot}\varepsilon$ to get the instance $R_D(I)$ for RPVEED. The reductions are trivially accomplished in $O(n^2)$ time.

Theorem 4.1. *MARPVEED is NP-complete.*

Proof. Given an instance $I = \{S, G = (V,E)\}$ of EDPPD, construct the instance $R_D(I)$ of RPVEED as described earlier. We now claim that (a) a YES instance of EDPPD yields a YES instance of RPVEED; and (b) a NO instance of EDPPD yields a NO instance of RPVEED;

The proof for (a) is by construction. Given a YES instance of EDPPD, we know that there exists a set of k edge disjoint paths in G for each of the k (s_i, d_i) tuples in S. Construct the routing R for RPVEED as follows. For every $A_i = (s_i, d_i, 1) \in \mathcal{A}$, let $R(A_i)$ be the edge disjoint path for the corresponding (s_i, d_i) pair in the EDPPD instance. For every edge e included in the routing, $\mathrm{bw}(e) = 1 + \varepsilon$. Further, since the routing consists of edge disjoint paths, each edge is assigned to at most one route. Therefore, $\mathrm{rc}_e = (\mathrm{bw}_e - \sum_{e \in R(A_j)} b_j) = \varepsilon$ for all edges $e \in R(A_i) \forall i$. Thus, $\sum_{i=1}^{k} \left(\min_{e \in R(A_i)} \{\mathrm{rc}_e\}\right) = k \cdot \varepsilon = \alpha$. Hence, the corresponding instance of RPVEED is a YES instance.

The proof for (b) is by contradiction. Suppose a NO instance of EDPPD yields a YES instance of RPVEED. We will use the YES instance of RPVEED to construct a YES instance of EDPPD. Since the weight of every edge in G' is at most $1 + \varepsilon$ and $b_i = 1 \forall i$, an edge could belong to at most one route. This implies that all the routes in R are disjoint. Further, since the bottleneck residual capacity for each route $(\min_{e \in R(A_i)} \{\mathrm{rc}_e\})$ could at most be ε and the total residual capacity is at least $\alpha = k \cdot \varepsilon$, the residual capacity of each route should be exactly ε. This implies that the bandwidth of each edge in the route is $1 + \varepsilon$. Therefore, all the edges included in the routing exist in the graph G and the routes constitute edge disjoint paths in G, thus yielding a YES instance of EDPPD. Hence, the contradiction.

Since RPVEED is a special case of MARPVEED, the NP-completeness of RPVEED immediately implies that MARPVEED is NP-complete. $\square$

4.3 Hardness of Approximation

A natural way to cope with NP-completeness is to seek approximate solutions instead of exact solutions. An algorithm with approximation ratio C computes, for every problem instance, a solution whose cost is within a factor C of the optimum. In this section, we investigate the approximability of MARPVEE. We show that unless P=NP, there does not exist a polynomial approximation algorithm with an approximation ratio better than $m^{1/2-\delta}$ for any $\delta > 0$.

We again use the edge disjoint problem for the purposes of our reduction. It has been previously shown that the problem is NP-hard to approximate within $m^{1/2-\delta}$ [75]. We will prove an essentially matching hardness result on the optimization version of the routing problem RPVEE and then use that result to prove the same bounds for MARPVEE.

4.3.1 Hardness of approximation of RPVEE

For establishing the hardness of approximation for RPVEE, we reduce an instance I of EDPP to instance $R(I)$ of RPVEE as described earlier in Section 4.2.1.

Lemma 4.2. *If the value of the optimal solution to an instance I of EDPP is k^* then the value of optimal solution to the instance $R(I)$ of RPVEE is $k^* \cdot \varepsilon$.*

Proof. Let the value of optimal solution to $R(I)$ be OPT. If there are k^* edge disjoint paths in I the corresponding routes for each of those paths in $R(I)$ will have a bottleneck residual capacity of ε. Therefore, OPT $\geq k^* \cdot \varepsilon$.

Note that for any route in $R(I)$, the bottleneck residual capacity is either 0 or ε. Therefore the total bottleneck residual capacity is a factor of ε. Let OPT $= z \cdot \varepsilon$. We then need to show that $z \leq k^*$. Since a route with a bottleneck residual capacity of ε consists of only the edges in the input graph to I and no two routes share a common edge, there are at least z disjoint paths in I. Since the value of optimal solution to I is k^*, $z \leq k^*$. Hence, we are done. ☐

Theorem 4.3. *For any $\delta > 0$, it is not possible to approximate RPVEE within a factor of $m^{1/2-\delta}$ unless P=NP.*

Proof. We will prove this by contradiction. Let us assume that there exists a polynomial time approximation algorithm A for RPVEE that achieves an approximation guarantee of factor $m^{1/2-\delta}$. Using Lemma 4.2, algorithm A in conjunction with the reduction R yields a polynomial time $m^{1/2-\delta}$-approximation algorithm for EDPP which is not possible unless P=NP [75]. ☐

4.3.2 Hardness of approximation of MARPVEE

We use the inapproximability result obtained above for RPVEE to state the inapproximability result for MAPRVEE with the same bounds. The proof is by contradiction and follows very closely the proof for Theorem 4.3.

Corollary 4.4. *For any $\delta > 0$, it is NP-hard to approximate MARPVEE within $m^{1/2-\delta}$ unless P=NP.*

4.4 Conclusions

It is important to carry out a rigorous formalization and analytical characterization of the adaptation problem. Such an analysis is central to understanding the problem better. Additionally, it also provides direction in exploring the solution space. In this chapter we formalized the adaptation problem in Virtuoso. We also studied its computational complexity. The adaptation problem which occurs in virtual execution environments is NP-hard. The problem is also NP-hard to approximate within $m^{1/2-\delta}$ of the optimal, where m is the number of edges in the virtual overlay graph.

These results set the tone for the remainder of the experimentation and simulations carried out in the course of this dissertation. Since it is hard to efficiently solve the problem and also to approximate it, we explore the space of heuristics solutions. In particular we study if the heuristics solutions developed, though possibly sub-optimal in certain cases, work well for majority of the real world cases in practice. In the next two chapters we consider simplified versions of the adaptation problems and study the effects of introducing resource reservations.

Chapter 5

Automatic network reservations

In the previous chapter (Chapter 4) we formalized the complete adaptation problem. Before we start exploring solutions to the complete problem in Chapters 7 and 8, in this chapter we study automatic adaptation leveraging network reservations. In particular, we formulate a simpler version of the adaptation problem and present a simple greedy heuristic that leverages network reservations as a solution to the adaptation problem. We found that the performance of the application studied (patterns) was significantly improved using this adaptation scheme that included network reservations.

5.1 Simplified version of adaptation problem

Here we present a simplified version of the adaptation problem. This problem is similar to RPVEE, in that it only has a routing component. However, it differs from RPVEE in its objective function. For this simpler version, we wish to maximize application performance.

Problem: Simplified version of adaptation problem

INPUT:

- A directed graph $G = (H, E)$
- A function $\mathrm{bw} : E \to \mathbb{R}$
- A set of ordered 3-tuples $\mathcal{A} = \{(s_i, d_i, b_i) \mid s_i, d_i \in H;\ b_i \in \mathbb{R};\ i = 1, \ldots, m\}$

OUTPUT: $R : \mathcal{A} \to \mathcal{P}$ such that

- application performance is maximized

We next describe why exploiting network reservations is important for an adaptive system.

5.2 Motivation behind this study

Optical networking may dramatically change high performance distributed computing. One reason is that optical networks can support provisioning dynamically configurable lightpaths, a form of circuit switching, through reservations. However, to use it (and all other network reservation mechanisms), the user or developer must modify the application. In this chapter we describe how the notion of automatic, run-time and dynamic adaptation can be extended to take into account network reservations.

Recognition of the potential of high speed optical networks has prompted the creation of private national and international optical networks [19, 130, 131], and the development of new models for using them. For example, the OptIPuter project [156] uses a dedicated optical network to interconnect large compute centers, data storage farms, and visualization centers.

When used with the traditional packet switching paradigm, optical networks operate with extremely high bandwidth but also very high latency [22]. This observation has motivated the optical networking community to investigate supplementing packet switching with capabilities for optical circuit switching. Circuit switching is strongly tied to resource reservations. While considerable work has gone into reservation mechanisms for packet switched networks, there has been little deployment of these mechanisms because of concerns about state size on routers. Circuit switched networks provide an environment that is potentially more suitable for reservation mechanisms. Since the state required in a circuit switch is already per-connection, adding per-connection reservations incurs only a constant factor increase in state size. Network and CPU reservations are powerful mechanisms for *guaranteeing* stable application performance.

Circuit switching requires establishing specific paths and the attributes of those paths, e.g., such as throughput and latency. Currently, this places a new requirement on either the user, who must *manually* reserve the network on behalf of the application, or the developer, who must specifically call a reservation system API. In either case, manual intervention is required to determine what circuits are needed and how to provision them. To date very little work has been done on automatic network reservations based entirely on the application's needs at run time.

In this chapter, we show that it is both feasible and relatively straightforward to *automatically* determine the necessary circuits and reserve them appropriately. Further, we can do so dynamically, changing circuits and reservations at run-time as the communication needs of the application change. Finally, as with our other adaptation schemes, this works with *existing, unmodified applications and operating systems* with *no user or developer intervention*. There is good reason to believe that our work will readily extend to other network reservation schemes as well.

The idea of dynamically creating overlay networks has an analogue in the paradigm of creating optical channels between nodes. Instead of creating an overlay network on top of the existing Internet infrastructure, we request a dedicated light path from the optical network reservation system. For our system we experimented with ODIN [122] (Optical Dynamic INtelligent Signaling), a set of optical network services, including provisioning capabilities, integrated into OMNInet [91] (Optical Metro Network Initiative), an experimental circuit switched optical network. We used VTTIF to monitor the application, and generated ODIN requests based on the inferred topology and traffic load matrix.

VRESERVE is the optical networking component of Virtuoso. VRESERVE alleviates the reservation responsibility for both the user and the developer. In fact the environment experienced by both is exactly the same as when a network without reservations is used. By automatically requesting network reservations at run-time we have enabled any application to transparently and painlessly use dedicated high speed reservable networks to increase communication performance.

In the following, we begin with a discussion of modern optical networks and their light path reservation mechanisms (Section 5.3), including a description of the OMNInet network we use and its ODIN reservation system. Section 8.3 details the experiments conducted to provide a proof of concept that Virtuoso can automatically and dynamically leverage network resource reservations. Section 5.6 concludes this discussion.

5.3 Optical networks

Although optical networking has existed since the 1980s, there has been a recent resurgence of interest for the following reasons:

- Practical optical domain switching and amplification mechanisms have been developed, allowing the majority of a network path to be purely optical [22, 183].
- The throughput possible in existing optical fiber has been growing dramatically [140]. As there is much existing "dark fiber", this means that dedicated or shared wide area optical networks for high performance computing are becoming feasible [57, 155].
- Deployment of network reservation systems that can be used by an end user has stalled. The result is that commodity Internet performance has become increasingly unpredictable, even on dedicated IP networks.

The core of an optical network is built using optical amplifiers, optical switches, and interconnected using fiber-optic cables. Bits are injected into the network by modulating a laser beam feeding a cable. The center frequency of the beam is typically referred to as

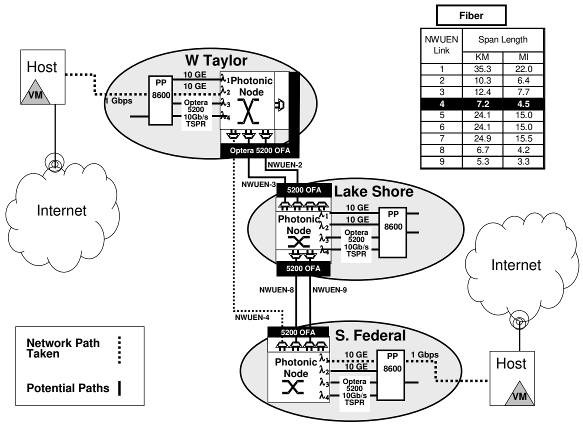


Figure 5.1: Physical topology of the OMNInet testbed network.

the “lambda”. An optical amplifier increases the intensity of a range of frequencies. An optical switch directs light of one frequency (lambda) that arrives on an input port to some output port, possibly changing its frequency in the process. The mapping from input ports and frequencies to output ports and frequencies defines the configuration of the switch. By configuring interconnected switches appropriately, a light path can be established from a source host to a destination host. Such configuration is analogous to call setup in a circuit switched network.

Networks such as OMNInet [91], Canarie [19], and NetherLight [131] allow authorized entities to reserve and provision optical lightpaths. Beyond basic connectivity, this primitive can be used to create logically separate networks (perhaps for different groups) that share the same underlying physical resources. It could also be used directly by applications.

5.3.1 OMNInet and ODIN

In our evaluation, we use the OMNInet network and the ODIN light path reservation system. Figure 5.1 shows the physical topology of a section of OMNInet. OMNInet is an

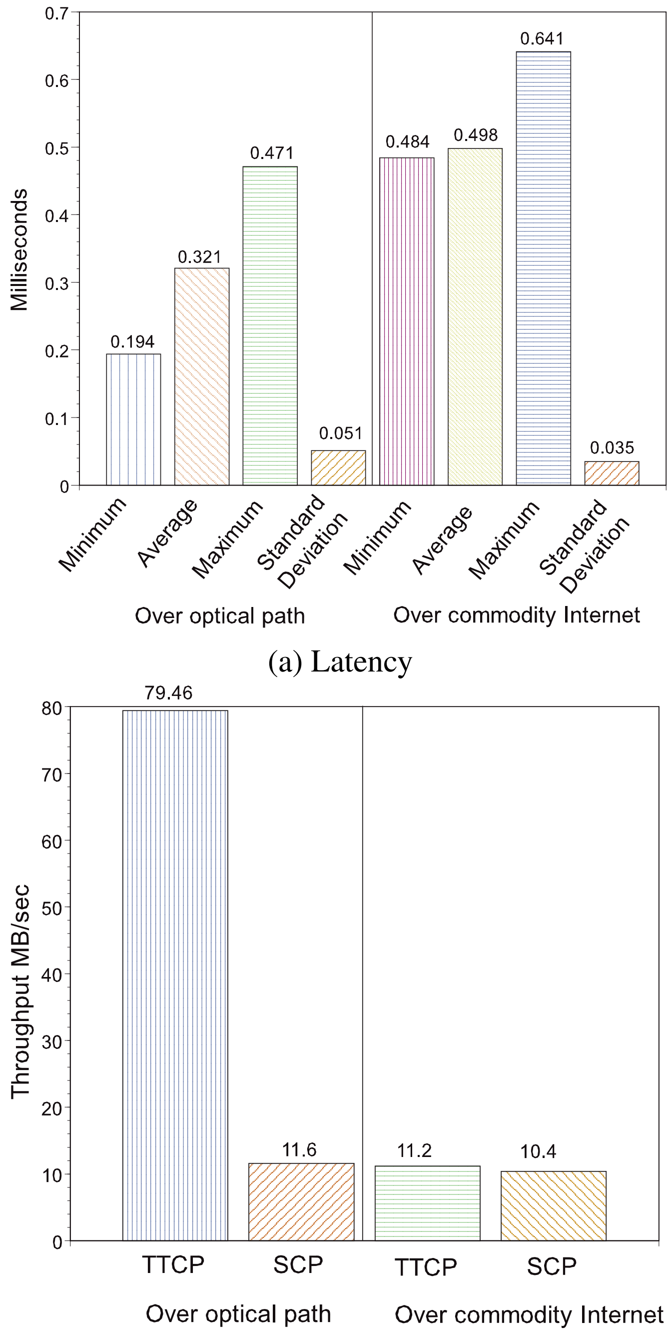
```
Reservation API:
CreatePath(<srcIP>, <dstIP>, <bw>, <lat>);
TeardownPath(<srcIP>, <dstIP>);

ODIN Interface:
oclient -c <srcIP> <dstIP> <lambda#> <flags>
oclient -t <pathID>
```

Figure 5.2: Reservation API.

experimental fully connected network that spans several sites in Chicago. We use machines directly connected to the optical switches at two of the sites. OMNInet is run by the International Center for Advanced Internet Research (iCAIR).

ODIN is a lightpath provisioning system that iCAIR developed for use on OMNInet. Figure 5.2 shows the ODIN provisioning interface. ODIN's interface for establishing a lightpath is very similar to VNET's interface for creating an overlay link. ODIN uses IP addresses to identify network nodes. A path reservation request consists of the source and destination IP addresses, and the required long term average bandwidth and latency of the path. Networks are constructed by making a reservation request for each link. ODIN then uses Ethernet VLANs to ensure that the given network is fully restricted to the hosts in the network graph.

The time to configure the OMNInet network (create a collection of lightpaths) is rather large as the expectation is that this will be done infrequently. The average setup time we observed is $\sim$15 seconds, which includes updating all the switches. Figure 5.3 shows the ideal and measured (using ping and ttcp) latency and throughput of the path denoted on Figure 5.1 as compared to a path over the commodity Internet.



(a) Latency

(b) Throughput

Figure 5.3: Latency and throughput of the optical path as compared to the commodity Internet.

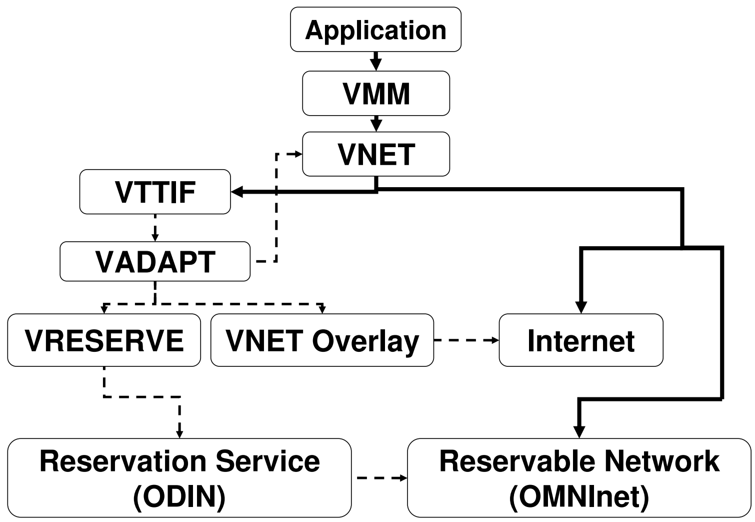


Figure 5.4: System overview. Dashed lines indicate control signals, solid lines denote actual network traffic.

5.3.2 Reservations in other networks

It is important to note that our work is intended to generalize to other network reservation systems, whether they are optical or not. A unifying feature of network reservation systems is that they require the reserver to provide a model of the traffic that will be sent along a path and a specification of its latency and throughput requirements [49]. Our system provides this information, which could be used with, for example, GARA [80].

5.4 Automatic dynamic reservations

A high-level view of the system is shown in Figure 5.4. Each Ethernet packet sent by the application is diverted by the virtual machine monitor into the VNET overlay network system. VNET forwards the packet on an overlay link, which may either be realized over the commodity Internet, or through a network that supports reservations (e.g., OMNInet). VNET also supplies the packet to our inference system, VTTIF, for inspection. Local

VTTIF agents collect data on each host and regularly aggregate the information on each remote VTTIF instance. A lead VTTIF constructs an estimate of the global application topology among its VMs and the corresponding traffic load matrix. This is passed to the adaptation system, VADAPT.

VADAPT attempts to improve application performance using a variety of adaptation mechanisms. One mechanism is to create new overlay links and corresponding overlay forwarding rules. After VADAPT has chosen a set of new overlay links, it passes it to VRESERVE which creates lightpaths for every link where this is possible. For each new light path thus created, VADAPT then changes the forwarding rules to send the data for the link over the lightpath instead of the commodity Internet.

In the following, we provide more details for VRESERVE.

5.4.1 VRESERVE

After VNET has decided which overlay links to create, but before it has created them, VRESERVE analyzes each link to determine if it can be better served using a reservation. Currently this is accomplished through a mapping of default (commodity Internet) interfaces (identified by IP addresses) to interfaces that are connected to a reservable network. If both endpoints of the link share a mapping to the same reservable network, VRESERVE initiates a reservation request for the path between the two corresponding interfaces. If the request succeeds, VADAPT configures the overlay link to use the reserved path. If not successful, the overlay link runs over a path in the commodity Internet.

A key point is that we create an overlay link on top of the reserved path. At first glance this may seem to be redundant, but it allows us to use VNET to perform routing. Without the overlay we would be forced to modify the host machines' routing tables or rewrite the packet headers. With the overlay in place, however, we can perform routing transparently.

Initially, however, this proved to be a substantial performance bottleneck with the first generation VNET. The first generation VNET was designed for the wide area and so did not perform especially well on these very fast links. We redesigned and reimplemented several parts of VNET, described in the second generation VNET, to improve performance enough to warrant the use of the high speed connection. As previously discussed in Section 3.3, VNET now has negligible overhead over a 100 Mbit LAN and only slightly trailed VMware GSX Server 2.5.1 over a Gbit LAN.

The actual implementation of VRESERVE is straightforward. It is a Perl module imported by VNET that implements a procedural interface for the creation and destruction of optical lightpaths. VRESERVE also tracks any changes to the reservable network's state made by a caller. Network reservations are made by interfacing directly to ODIN. ODIN consists of a server running on a trusted host and a command-line client. VRESERVE simply constructs and executes command-lines. Because ODIN does not support deferred scheduling VRESERVE immediately indicates success or failure in creating a lightpath.

5.4.2 An example scenario

A typical execution scenario is as follows. A set of user's virtual machines V, are started on a distributed set of hosts. A VNET star topology is created, with a proxy machine p, to enable communication for every VM in V. A parallel application is then executed inside each VM in V. All intra-VM communication is routed through p, and a traffic matrix is aggregated by VTTIF. From that matrix VTTIF derives a communication topology among the VMs in V. VADAPT uses this topology, combined with the mapping of VMs to hosts, to define a better topology among the VNET daemons. This topology consists of a set of overlay links E. We choose k links with the highest bandwidth requirements from E and place them in H, $H \subseteq E$. VADAPT passes H to VRESERVE for action.

VRESERVE analyzes H and determines a subset of overlay links R for which reservations are possible. VRESERVE then requests reservations for each overlay link in R.

Links that suffer from path reservation failure are removed from R. VNET then creates the overlay network. This is accomplished by creating an overlay link for each element in H and adjusting the forwarding rules to send packets over the reserved paths for the links in R and over the commodity Internet for $H - R$. As the communication pattern changes, a new set H' is created by VADAPT and passed to VNET. VNET and VRESERVE process all the new links identically as before, generating an overlay network of $H \cup H'$. However following the creation process VNET finds the difference $H - H'$, which corresponds to links not needed in the new topology. It then removes those links, as well as any reservations allocated to links in $H - H'$.

The implementation of VRESERVE is about 400 lines of Perl. Half of this is the VRESERVE module, while the other half interfaces VADAPT to VRESERVE. The majority of the implementation involves parsing the output from ODIN. Modifications to VADAPT take the form of VRESERVE API calls and an added IP address mapping service.

5.4.3 Assumptions

To evaluate in isolation, the feasibility and performance of integrating network reservations with Virtuoso, we created a simplified version of the adaptation problem described in Chapter 4.

Our heuristic leverages the information provided by VTTIF to make the VNET overlay topology conform to the inferred application topology by adding and deleting overlay links and forwarding rules and by using network reservation, where possible.

The simplifying assumptions made are:

- Improving application performance amounts to increasing its throughput.
- All routing on the overlay is shortest path first.

The combined evaluation of all the adaptation mechanisms is provided in Chapter 8.

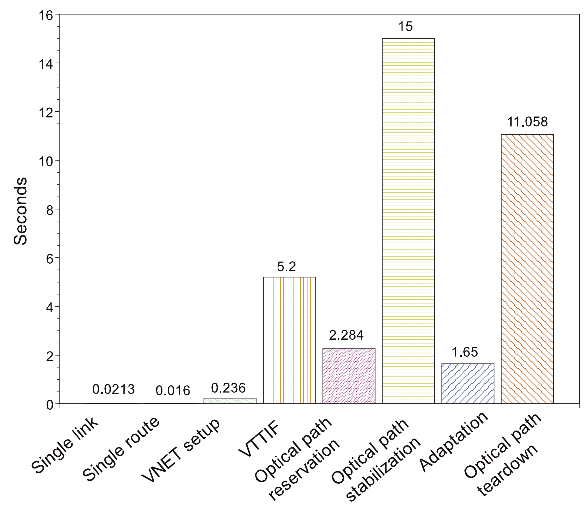


Figure 5.5: The configuration-time costs for the two VM scenario shown in Figure 5.1.

5.5 Experiments

To evaluate our system we ran several experiments to determine its performance. In the following, we first examine the configuration time of our system, and then evaluate the performance of data transfers through the whole system and the performance of our simple parallel application benchmark, Patterns. The result is an existence proof: the system works and there is at least one case where it can lead to enhanced performance.[1]

5.5.1 Configuration time

Figure 5.5 shows the costs involved in configuring the network using our system. The primary cost was the time spent in the reservation system itself. Creating a path entails two delays. The first is a software delay. It took ∼2.5 seconds for ODIN to send the configuration commands to all the switches. The second delay (∼15 seconds) is the time needed for the hardware to reconfigure itself and for the path to stabilize. This stabilization

[1]Our experiments were limited in scope due to a surprise shutdown of the OMNInet network. Furthermore, during the interval in which we were able to use it, only one path could be reserved.

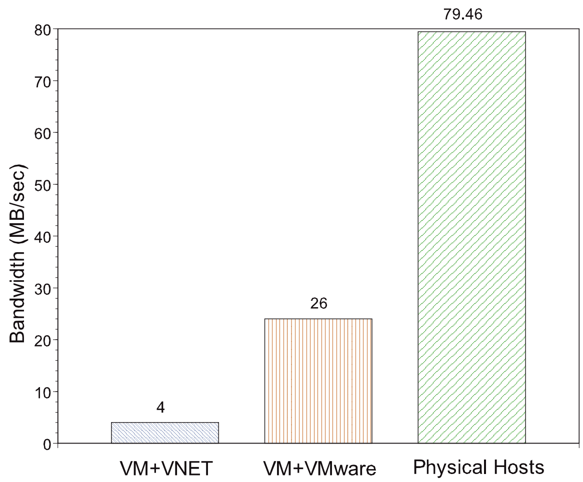


Figure 5.6: Throughput achieved by ttcp on an optical network. VNET here refers to the first generation VNET.

delay is constant, regardless of the complexity of the number of switches being configured. The software delay, however, will grow linearly with the number of switches in a path because ODIN does not currently support parallel configuration. The time taken to tear down an optical path was $\sim$12 seconds.

VTTIF is a significant, but smaller contributor to the setup delay. It must observe traffic for some period of time before it can report a topology. The VTTIF time is a function of a number of parameters and can be as low as one second. The time to execute VRESERVE and to create the individual overlay links is lost in the noise.

The total time from the start of network communication to channeling packets over an overlay link running through a lightpath is < 30 seconds.

5.5.2 VM-to-VM TCP performance

In this experiment, we use the configuration of Figure 5.1 and run the ttcp TCP benchmarking tool in the two VMs. The VNET referred to in the context of these experiments is the first generation VNET. The second generation VNET was still in development at that point in time. By the time the second generation VNET was operational, OMNInet was no longer available for our use. Hence we could not compare the throughput of the second generation VNET with VMware bridged networking over an optical network. However, Section 3.3 provides a comparison between them over a Gigabit LAN.

In this experiments, the system notices the sudden communication between the VMs and establishes a lightpath between their hosts. We would expect a dramatic increase of performance thereafter. The physical network is no longer a bottleneck for the system; it is the first generation VNET and VMware that become the bottlenecks. Figure 5.6 shows the results, comparing the raw throughput between the two hosts with the VMware throughput and the throughput using first generation VNET. While acceptable for communication in the wide area (first generation VNET's original design goal), a $\sim$5 MB/s ceiling is far too low. Note that VMware is the next substantial bottleneck after VNET. As mentioned previously, this VNET bottleneck has been eliminated in the second generation. Now, over high speed networks VNET lags only a little behind VMware's bridged networking.

5.5.3 VM-based BSP benchmark

Here, we use the configuration of Figure 5.1 and run Patters (described in Chapter 2) in the VMs. Patterns uses an all-to-all communication pattern, and we measure its execution rate in iterations/second. Figure 5.7 shows the performance improvement from using a reserved optical network. Performance increases by 170% when the optical path is used. We also see the first generation VNET bottleneck again.

The upshot of this benchmark and the preceding ttcp benchmark is that our system can automatically use reservation systems to improve the performance of distributed and parallel applications.

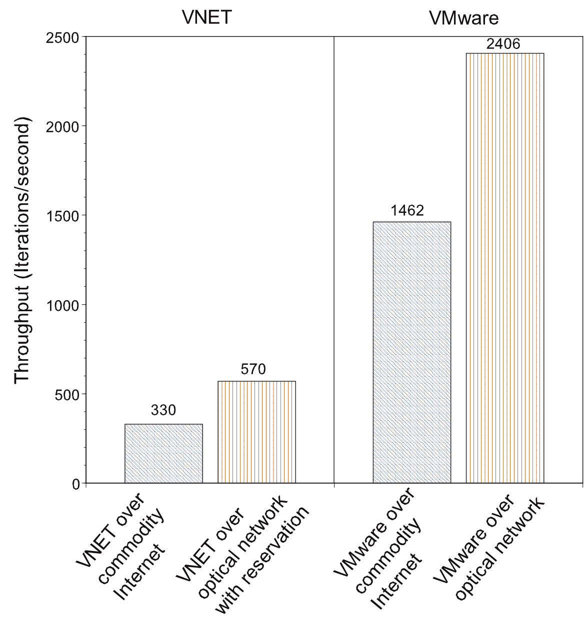


Figure 5.7: Increasing the performance of a BSP benchmark with an all-to-all communication pattern. VNET here refers to the first generation VNET

5.6 Conclusions

We have demonstrated that it is feasible to automatically create network reservations on behalf of unmodified applications, and that such reservations can improve application performance. Specifically, we reserved lightpaths on behalf of applications running in virtual machines by observing their communication traffic over an overlay network and reconstructing their topology from that low-level traffic. Our techniques require no modifications of the application or help from the user or developer.

One question is to what extent our results can generalize. Must we use VMs and an overlay network? Can we support other network reservation models? The answer to the latter question is clearly yes, as the ODIN provisioning model is not qualitatively much different from other per-flow reservation models. We believe that our work can generalize beyond VMs and overlays. For example, traffic monitoring and inference could be done in the kernel and then used to adjust routing tables.

The assumptions in Section 5.4.3 need to be relaxed to lead to a more general adaptive environment that includes VM CPU reservations, VM migration, forwarding, and the overlay topology itself. Chapter 8 present our evaluations on fast algorithms that can take into account all the available information and choose among adaptation mechanisms, including network reservations, to optimize application performance.

We believe there are clear benefits to using configurable and reservable networks in concert with adaptive overlay network technologies. Adaptive overlays let us seamlessly integrate new networking technologies into existing applications and into the commodity Internet.

Because VTTIF can provide a holistic view of the application, an entire topology and traffic matrix at once instead of just a link at a time, it should be possible for an optical network reservation system to exploit this higher level, detailed information to schedule reservations across the whole network collectively, providing together with sophisticated time-driven scheduling of the VMs, global communication and computation context switches [47].

Chapter 6

Exploiting CPU reservations

Although the core of this dissertation deals with automatic adaptation that requires no user or application intervention, there are often cases wherein application demands cannot be inferred. Before we describe and evaluate the solutions to the complete adaptation problem in Chapters 7 and 8, in this chapter we study a scenario wherein our automatic adaptation scheme is not applicable due to the lack of automatic and invisible application demand inference. There are times when the adaptation needs to be performed for reasons other than speeding up the application, such load balancing and better utilization of the system as a whole. In other words addressing global scope rather than the local scope of the single application. In such scenarios, it is difficult to infer what exactly the application, user or system administrator demands. The goal of this chapter is to explore what can we still do, in terms of adaptation, in such scenarios.

The specific scenario we study is that of tightly coupled clusters executing batch parallel applications. In particular for a set of parallel applications executing in a cluster, we attempt to constrain the compute utilization for each component of the application on each executing node while ensuring that each application's performance is still proportional to the utilization that the application components receive on the physical hosts. The adaptation mechanism leveraged is CPU reservations. To isolate the focus on CPU reservations, we omitted the use of virtual machines and virtual networks. However, these results are valid even when the applications are executing inside of a user's VMs hosted on a tightly coupled cluster. This is due to the fact that in the latter case we can schedule the VMs as processes instead of scheduling the application processes.

It should be noted that applications do not always desire 100% CPU utilization. If a user's utilization on a third party cluster (resource provider)is tied to the amount he pays, then there can be cases where the user might want to pay less and hence want to use less than 100% CPU. However, a user will be willing to do this only if they are assured that the application's performance is to its CPU resource utilization. Currently, there is no known existing method achieve this goal.

To avoid stalls, destructive interactions and provide predictable performance for users, almost all tightly-coupled computing resources today are space-shared. In space-sharing [163], each application is given a partition of the available nodes, and on its partition, it is the *only* application running, thus avoiding the problem altogether by providing complete performance isolation between running applications. Space-sharing introduces several problems, however. Most obviously, it limits the utilization of the machine because the CPUs of the nodes are idle when communication or I/O is occurring. Space-sharing also makes it likely that applications that require many nodes will be stuck in the queue for a long time and, when running, block many applications that require small numbers of nodes. Finally, space-sharing permits a provider to control the response time or execution rate of a parallel job at only a very course granularity. Though it can be argued theoretically that applications can be always built such that computation and I/O overlap all the time, thus preventing stalls, practically speaking, this is rarely the case.

In this chapter, we propose a new approach to time-sharing parallel applications (running inside or outside of VMs) on tightly-coupled computing resources like clusters, *performance-targetted feedback-controlled real-time scheduling*. The goals of our technique are to provide

- performance isolation within a time-sharing framework that permits multiple applications to share a node, and
- performance control that allows the administrator to finely control the execution rate of each application while keeping its resource utilization proportional to execution rate.

Conversely, the administrator can set a target resource utilization for each application and have commensurate application execution rates follow.

In performance-targetted feedback-controlled real-time scheduling, each node has a periodic real-time scheduler. The local application thread is scheduled with a $(period, slice)$ constraint, meaning that it executes *slice* seconds every *period*. Notice that $slice/period$ is the utilization of the application on the node. Our implementation uses and builds upon a previously described [111] and publicly available VSched tool, developed by Bin et al. at Northwestern University. VSched is a user-level periodic real-time scheduler for Linux that was originally developed to explore scheduling interactive and batch workloads together. Section 6.1 provides an overview.

Once an administrator has set a target execution rate for an application, a global controller determines the appropriate constraint for each of the application's threads of execution and then contacts each corresponding local scheduler to set it. The controller's input is the desired application execution rate, given as a percentage of its maximum rate on the system (i.e., as if it were on a space-shared system). The application or its agent periodically feeds back to the controller its current execution rate. The controller modifies the local schedulers' constraints based on the error between the desired and actual execution rate, with the added constraint that utilization must be proportional to the target execution rate.

In the common case, the only communication in the system is the feedback of the current execution rate of the application to the global controller, and synchronization of the local schedulers through the controller is very infrequent. Section 6.2 describes the global controller in detail.

It is important to point out that our system schedules the CPU of a node, not its physical memory, communication hardware, or local disk I/O. Nonetheless, in practice, we can achieve quite good performance isolation and control even for applications making significant use of these other resources, as we show in our detailed evaluation (Section 6.3.4). Mechanisms for physical memory isolation in current OSes and VMMs are well understood and can be applied in concert with our techniques. As long as the combined working set size of the applications executing on the node does not exceed the physical memory of the machine, the existing mechanisms suffice. Communication has significant computational costs, thus, by throttling the CPU, we also throttle it. The interaction of our system and local disk I/O is more complex. Even so, we can control applications with considerable disk I/O.

6.1 Local scheduler

In the periodic real-time model, a task is run for *slice* seconds every *period* seconds. Using earliest deadline first (EDF) schedulability analysis [115], the scheduler can determine whether some set of $(period, slice)$ constraints can be met. The scheduler simply uses dynamic priority preemptive scheduling with the deadlines of the admitted tasks as priorities.

VSched is a user-level implementation of this approach for Linux that offers soft real-time guarantees. It runs as a Linux process that schedules other Linux processes. Because the Linux kernel does not have priority inheritance mechanisms, nor known bounded interrupt service times, it is impossible for a tool like VSched to provide hard real-time guarantees to ordinary processes. Nonetheless, it has been showed in an earlier paper [111], for a wide range of periods and slices, and under even fairly high utilization, VSched almost always meets the deadlines of its tasks, and when it misses, the miss time is typically very small. VSched supports $(period, slice)$ constraints ranging from the low hundreds of microseconds (if certain kernel features are available) to days. Using this range, the needs of various classes of applications can be described and accommodated. VSched allows changes to task's constraints within about a millisecond.

VSched is a client/server system. The VSched server is a daemon running on Linux that spawns the scheduling core, which executes the scheduling scheme described above. The VSched client communicates with the server over an encrypted TCP connection. In this work, the client is driven by the global controller and we schedule individual Linux processes.

The VSched server uses the admissibility test of the EDF algorithm. The scheduling core is a modified EDF scheduler that dispatches processes in EDF order but interrupts them when they have exhausted their allocated CPU for the current period. If the server admits a task, the core can immediately switch to it.

The scheduling core makes use of the three highest priorities of SCHED_FIFO, the highest priority scheduling class in Linux. The scheduling core itself runs as the highest priority SCHED_FIFO process on the system, assuring that when it becomes runnable, it immediately is given the CPU. The server is run as SCHED_FIFO with the next highest priority so that it will immediately service new requests whenever the scheduling core is not running. The scheduling core assigns the process that it currently wants to run the third highest SCHED_FIFO priority and (optionally) sends it a SIG_CONT. The process that is being switched away from is assigned an ordinary SCHED_OTHER priority. If the administrator has configured hard limiting on its resource use, it is also sent a SIG_STOP, otherwise VSched operates as a work-conserving scheduler.

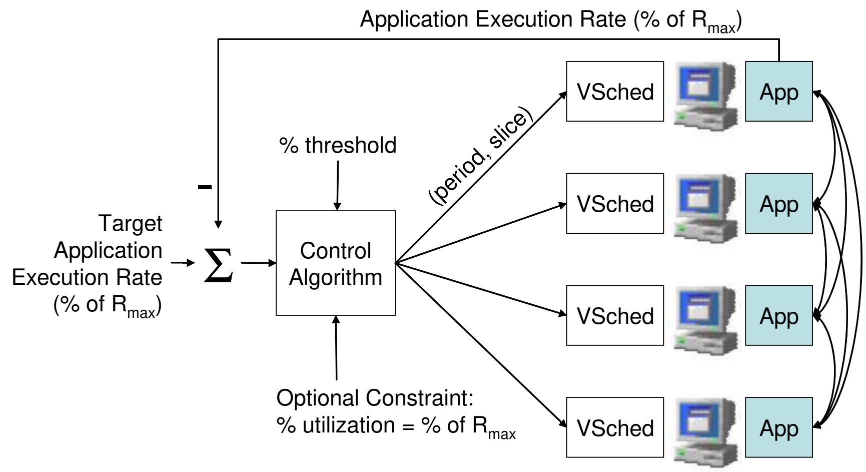


Figure 6.1: Structure of global control.

The performance of VSched has been evaluated on several different platforms. It can achieve very low deadline miss rates up to quite high utilizations and quite fine resolutions. VSched can use over 90% of the CPU even on relatively slow hardware and older kernels (Intel® Pentium® III, 2.4 kernel) and can use over 98% of the CPU on more modern configurations (Intel® Pentium® 4, 2.6 kernel). The mechanisms of VSched and its evaluation are described in much more detail in an earlier paper [111] and the software itself is publicly available.

6.2 Global controller

The control system consists of a centralized feedback controller and multiple host nodes, each running a local copy of VSched, as shown in Figure 6.1. A VSched daemon is responsible for scheduling the local thread(s) of the application(s) under the yoke of the controller. The controller sets $(period, slice)$ constraints using the mechanisms described in Section 6.1. Currently, the same constraint is used for each VSched. One thread of the application, or some other agent, periodically communicates with the controller using non-blocking communication.

6.2.1 Inputs

The maximum application execution rate on the system in application-defined units is R_{max}. The set point of the controller is supplied by the user or the system administrator through a command-line interface that sends a message to the controller. The set point is r_{target} and is a percentage of R_{max}. The system is also defined by its threshold for error, ε, which is given as percentage points. The inputs Δ_{slice} and Δ_{period} specify the smallest amounts by which the slice and period can be changed. The inputs min_{slice} and min_{period} define the smallest slice and period that VSched can achieve on the hardware.

The current utilization of the application is defined in terms of its scheduled period and slice, $U = slice/period$. The user requires that the utilization be proportional to the target rate, that is, that $U = r_{target} \pm \varepsilon$.

The feedback input $r_{current}$ comes from the parallel application we are scheduling and represents its current execution rate as a percentage of R_{max}. To minimize the modification of the application and the communication overhead, our approach only requires high-level knowledge about the application's control flow and only a few extra lines of code.

6.2.2 Control algorithm

The control algorithm (or simply the algorithm) is responsible for choosing a $(period, slice)$ constraint to achieve the following goals

1. The error is within threshold: $r_{current} = r_{target} \pm \varepsilon$, and

2. That the schedule is efficient: $U = r_{target} \pm \varepsilon$.

The algorithm is based on the intuition and observation that application performance will vary depending on which of the many possible $(period, slice)$ schedules corresponding to a given utilization U we choose, and the best choice will be application dependent and vary with time. For example, a finer grain schedule (e.g. (20ms, 10ms)) may result in better application performance than coarser grain schedules (e.g. (200ms, 100ms)). At any point in time, there may be multiple "best" schedules.

The control algorithm attempts to automatically and dynamically achieve goals 1 and 2 in the above, maintaining a particular execution rate r_{target} specified by the user while keeping utilization proportional to the target rate.

We define the error as

$$e = r_{current} - r_{target}.$$

At startup, the algorithm is given an initial rate r_{target}. It chooses a $(period, slice)$ constraint such that $U = r_{target}$ and *period* is set to a relatively large value such as 200 ms. The algorithm is a simple linear search for the largest *period* that satisfies our requirements.

When the application reports a new current rate measurement $r_{current}$ and/or the user specifies a change in the target rate r_{target}, e is recomputed and then the following is executed:

- If $|e| > \varepsilon$ decrease *period* by Δ_{period} and decrease *slice* by Δ_{slice} such that $slice/period = U = r_{target}$. If $period \leq min_{period}$ then we reset *period* to the same value as used at the beginning and again set *slice* such that $U = r_{target}$.
- If $|e| \leq \varepsilon$ do nothing.

It should be noticed that the algorithm always maintains the target utilization and searches the $(period, slice)$ space from larger to smaller granularity, subject to the utilization constraint. The linear search is, in part, done because multiple appropriate schedules may exist. We do not preclude the use of algorithms that walk the space faster, but we have found our current algorithm to be effective.

6.3 Evaluation

In presenting our evaluation, we begin by explaining the experimental framework in Section 6.3.1. Then, in Section 6.3.2, we show the range of control that the scheduling system has made available. This is followed by an examination of using the algorithm described above to prevent the inevitable drift associated with simply using a local real-time scheduler in Section 6.3.3. Next, Section 6.3.4 examines the performance of the algorithm in a dynamic environment, showing their reaction to changing requirements. We then illustrate how the system remains impervious to external load despite the feedback in Section 6.3.7 Next, in Section 6.3.9, show how the system scales as it controls increasing numbers of parallel applications. Finally, in Sections 6.3.10 and 6.3.11, we examine the effects of local disk I/O and memory contention.

6.3.1 Experimental framework

As mentioned previously, Bulk Synchronous Parallel (BSP [65]) model is used to characterize many of the batch parallel workloads that run in tightly coupled computing resources such as clusters. In most of our evaluations we used our synthetic BSP benchmark, Patterns. Patterns is described in more detail in Chapter 2.

In general, we configure Patterns to run with an all-to-all communication pattern on four nodes of our IBM e1350 cluster (Intel® Xeon® 2.0 GHz, 1.5 GB RAM, Gigabit Ethernet interconnect). Each node runs VSched, and a separate node is used to run the controller. Note that all of our results involve CPU and network I/O.

We also evaluated the system using an NAS (NASA Advanced Supercomputing) benchmark. In particular, we use the PVM implementation of the IS (Integer Sort) benchmark developed by White et al. [185]. As described in Chapter 2 IS combines integer computation speed and communication with, unlike Patterns, different nodes doing different

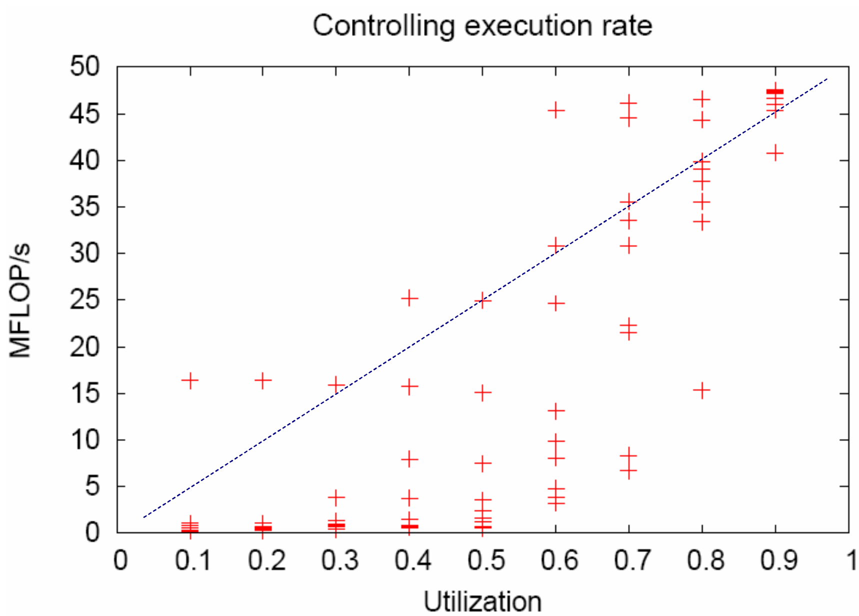


Figure 6.2: Compute rate as a function of utilization for different $(period, slice)$ choices.

amounts of computation and communication.

6.3.2 Range of control

To illustrate the range of control possible using periodic real-time scheduling on the individual nodes, we ran Patterns with a compute/communicate ratio of 1:2, making it quite communication intensive. Note that this configuration is conservative: it is far easier to control a more loosely coupled parallel application with VSched. We ran Patterns repeatedly, with different $(period, slice)$ combinations.

Figure 6.2 shows these test cases. Each point is an execution of Patterns with a different $(period, slice)$, plotting the execution rate of Patterns as a function of Patterns utilization on the individual nodes. Notice the line on the graph, which is the ideal control curve that the control algorithm is attempting to achieve, control over the execution rate of the application with proportional utilization ($r_{current} = r_{target} = U$). Clearly, there *are* choices of $(period, slice)$ that allow us to meet all of the requirements.

6.3.3 Schedule selection and drift

Although there clearly exist $(period, slice)$ schedules that can achieve an execution rate with (or without) proportional utilization, we cannot simply use only the local schedulers for several reasons:

- The appropriate $(period, slice)$ is application dependent because of differing compute/communicate ratios, granularities, and communication patterns. Making the right choice should be automatic.
- The user or system administrator may want to dynamically change the application execution rate r_{target}. The system should react automatically.
- Our implementation is based on a *soft* local real-time scheduler. This means that deadline misses will inevitably occur and this can cause timing offsets between different application threads to accumulate. We must monitor and correct for these slow errors. Notice that this is likely to be the case for a hard local real-time scheduler as well if the admitted tasks vary across the nodes.

Figure 6.3 illustrates what we desire to occur. The target application execution rate is given in iterations per second, here being 0.006 iterations/second. This is Patterns running with a 1:1 compute/communicate ratio on two nodes. The lower curve is that of simply using VSched locally to schedule the application. Although we can see that the rate is correct for the first few iterations, it then drifts downward, upward, and once again downward over the course of the experiment. The roughly straight curve is using VSched, the global controller, and the control algorithm. We can see that the tendency to drift has been eliminated using global feedback control.

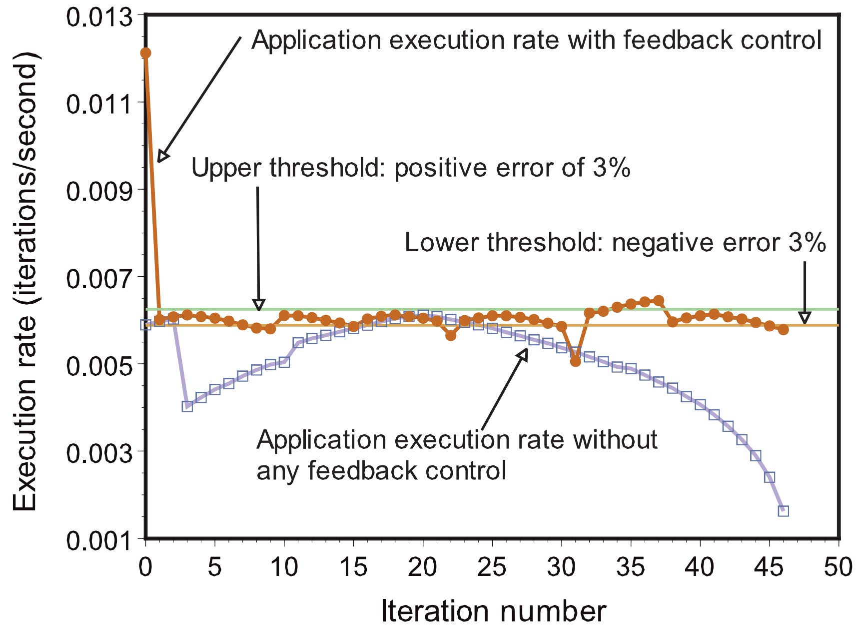


Figure 6.3: Elimination of drift using global feedback control; 1:1 comp/comm ratio.

6.3.4 Evaluating the control algorithm

We studied the performance of the control algorithm using three different compute/communicate ratios (high (5:1) ratio, medium (1:1) ratio, and low (1:5) ratio), different target execution rates r_{target}, and different thresholds ε. In all cases $\Delta_{period} = 2$ ms, where Δ_{period} is the change in period effected by VSched when the application execution rate goes outside of the threshold range, the *slice* is then adjusted such that $U = r_{target}$.

Figure 6.4 shows the results for high, medium, and low test cases with a 3% threshold. We can see that the target rate is easily and quickly achieved, and remains stable for all three test cases. Note that the execution rate of these test cases running at full speed without any scheduling are slightly different.

Next, we focus on two performance metrics:

- Minimum threshold: What is the smallest ε below which control becomes unstable?

- Response time: for stable configurations, what is the typical time between when the target execution rate r_{target} changes and when the $r_{current} = r_{target} \pm \varepsilon$?

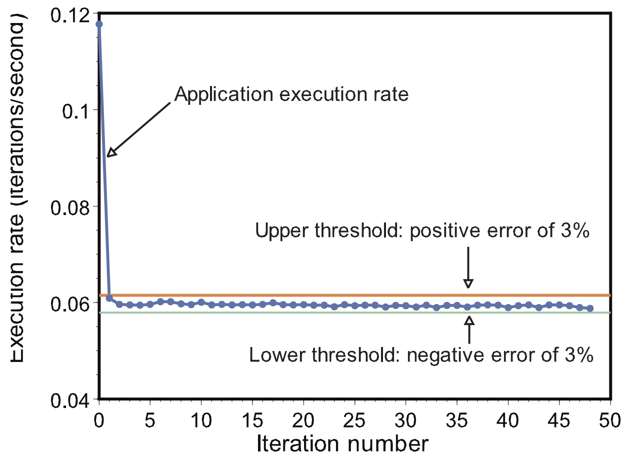
Being true for all feedback control systems, the error threshold will affect the performance of the system. When the threshold ε is too small, the controller becomes unstable and fails because the change applied by the control system to correct the error is even greater than the error. For our control algorithm, when the error threshold is $< 1\%$, the controller will become unstable. Figure 6.5 illustrates this behavior. Note that while the system is now oscillating, it appears to degrade gracefully.

Figure 6.6 illustrates our experiment for measuring the response time. The target rate is changed by the user in the middle of the experiment. Our control system quickly adjusts the execution rate and stabilizes it. It shows that the response time is about 32 seconds, or two iterations, for the case of 1:1 compute/communicate ratio. The average response time over four test cases (1 high, 2 medium, and 1 low compute/communicate ratios) is 30.68 seconds. In all cases, the control algorithm maintains $U = r_{target}$ as an invariant by construction.

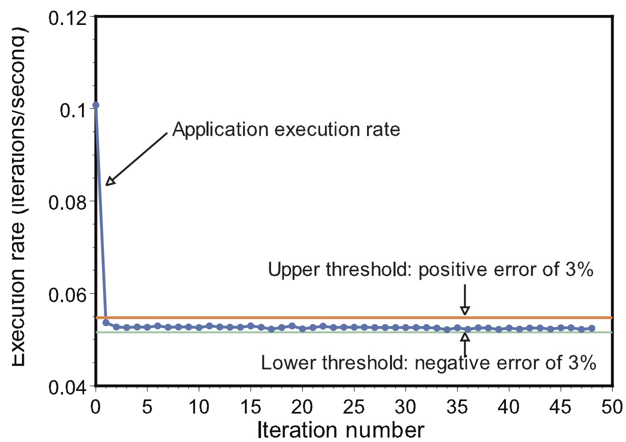
6.3.5 Summary of limits of the control algorithm

Figure 6.7 summarizes the response time, communication cost to support the feedback control, and threshold limits of our control system. Overall we can control with a quite small threshold ε. The system responds quickly, on the order of a couple of iterations of our benchmark. The communication cost is minuscule, on the order of just a few bytes per iteration. Finally, these results are largely independent of the compute/communicate ratio.

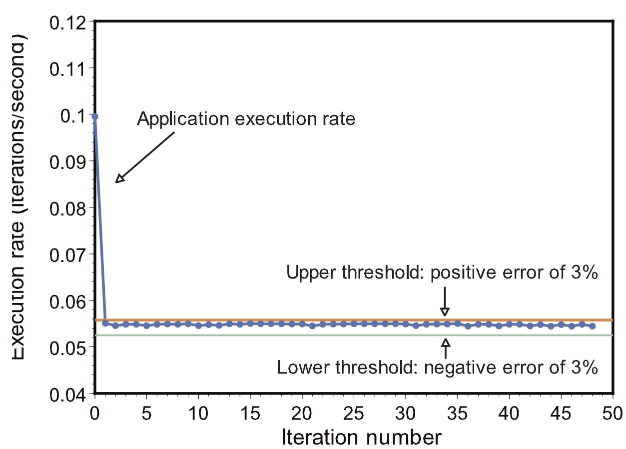
The exceptionally low communication involved in performance-targetted feedback-controlled real-time scheduling is a natural consequence of the deterministic and predictable periodic real-time scheduler being used on each node.



(a) high (5:1) comp/comm ratio



(b) medium (1:1) comp/comm ratio



(c) low (1:5) comp/comm ratio

Figure 6.4: System in stable configuration for varying comp/comm ratio.

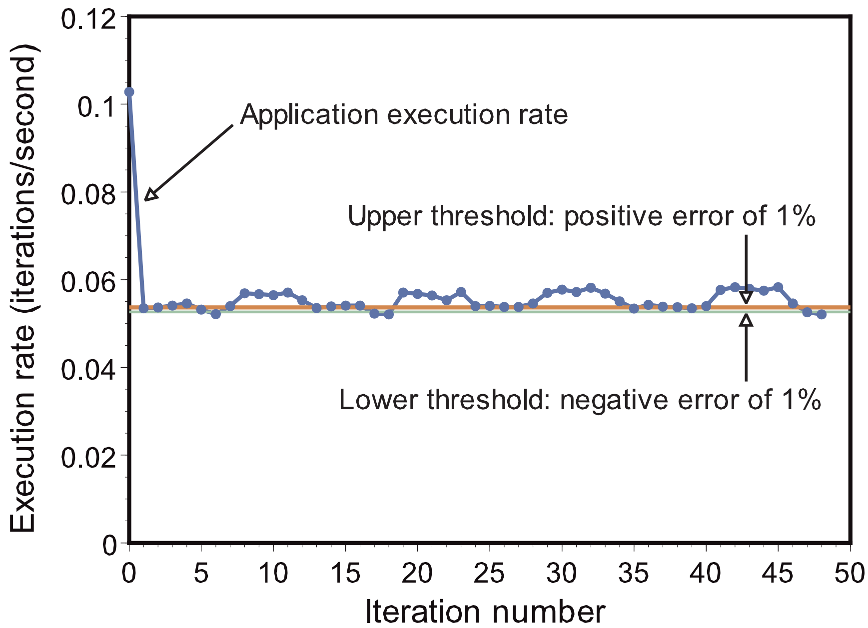


Figure 6.5: System in oscillation when error threshold is made too small; 1:1 comp/comm ratio.

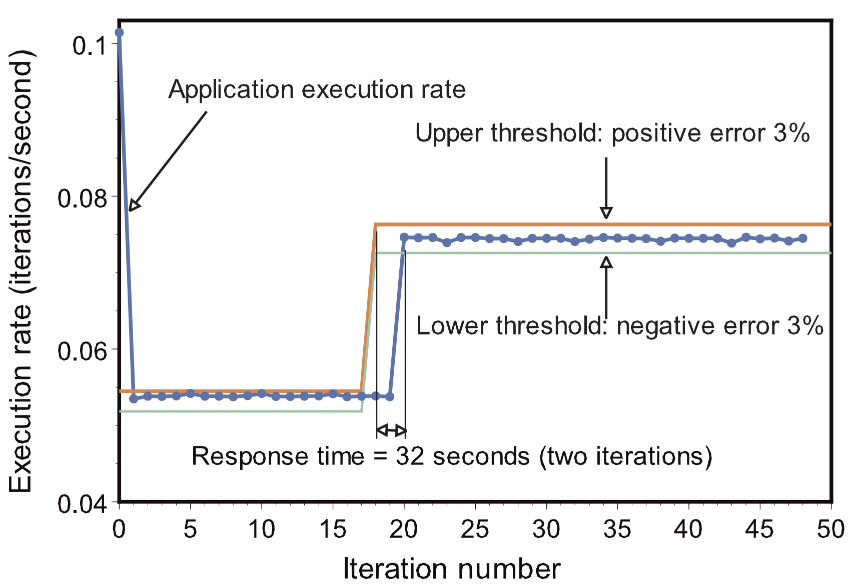


Figure 6.6: Response time of control algorithm; 1:1 comp/comm ratio.

High (5:1) compute/communicate ratio			Medium (1:1) compute/communicate ratio			Low (1:5) compute/communicate ratio		
Response time	Threshold limit	Feedback comm.	Response time	Threshold limit	Feedback comm.	Response time	Threshold limit	Feedback comm.
29.16 s	2 %	32 bytes/iter	31.33 s	2 %	32 bytes/iter	32.01 s	2 %	32 bytes/iter

Figure 6.7: Response time and threshold limits for the control algorithm.

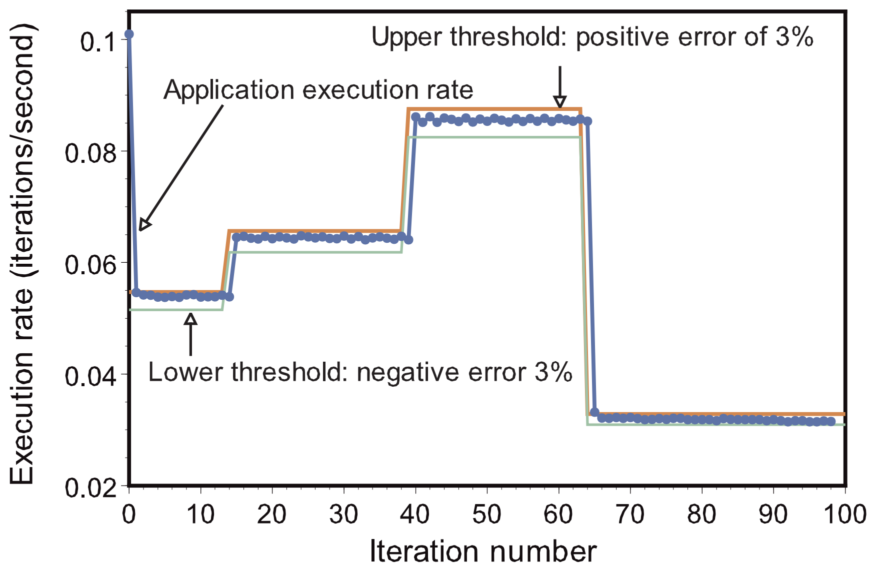


Figure 6.8: Dynamically varying execution rates; 1:1comp/comm ratio.

6.3.6 Dynamic target execution rates

As we mentioned earlier, using the feedback control mechanism, we can dynamically change the target execution rates and our control system will continuously adjust the real-time schedule to adapt to the changes. To see how our system reacts to user inputs over time, we conducted an experiment in which the user adjusted his desired target rate four times during the execution of the Patterns application. As shown in Figure 6.8, the control algorithm works well. After the user changes the target rate, the algorithm quickly adjusts the schedule to reach the target.

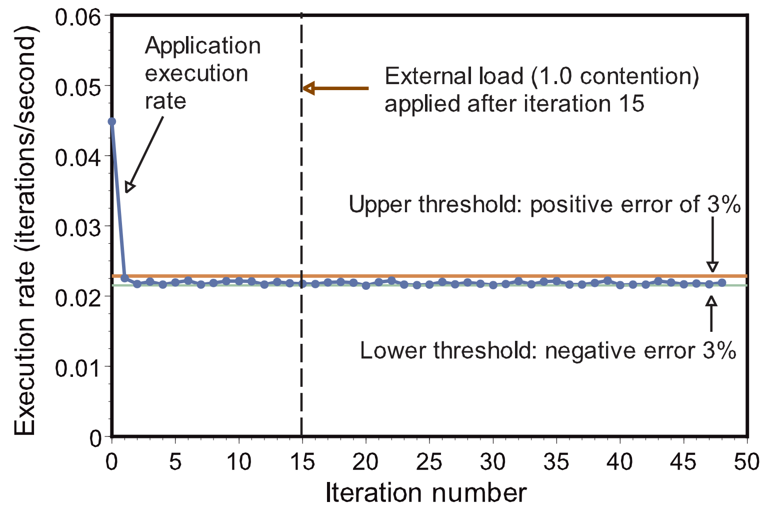


Figure 6.9: Performance of control system under external load; 3:1 comp/comm ratio; 3% threshold.

6.3.7 Ignoring external load

Any coupled parallel program can suffer drastically from external load on any node; the program runs at the speed of the slowest node. We have previously shown that the periodic real-time model of VSched can shield the program from such external load, preventing the slowdown [111]. Here we want to see whether our control system as a whole can still protect a BSP application from external load.

We execute Patterns on four nodes with the target execution rate set to half of its maximum rate. On one of the nodes, we apply external load, a program that contends for the CPU using load trace playback techniques [39]. Contention is defined as the average number of contention processes that are runnable.

Figure 6.9 illustrates the results. At roughly the 15th iteration, an external load is placed on one of the nodes in which Patterns is running, producing a contention of 1.0. We note that the combination of VSched and the feedback controller are able to keep the

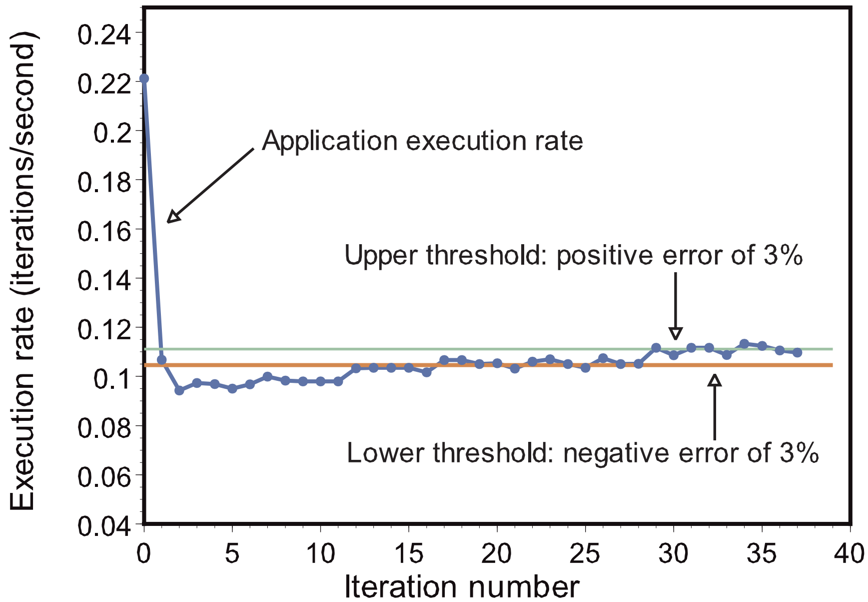


Figure 6.10: Running NAS benchmark under control system; 3% threshold.

performance of Patterns independent of this load. We conclude that our control system can help a BSP application maintain a fixed stable performance under a specified execution rate constraint despite external load.

6.3.8 NAS IS Benchmark

When we run the NAS IS (Integer Sort) benchmark without leveraging our control system, we observe that different nodes have different CPU utilizations. This is very different from the Patterns benchmark, which does roughly the same amount of computation and communication on each node. In our experiment, for a specific configuration of NAS IS executing on four nodes, we observed an average utilization of $\sim$28% for two nodes and $\sim$14% average utilization for the other two nodes.

This variation has the potential to challenge our control system, since in our model we assume the same target utilization U on each node, and we apply the same schedule

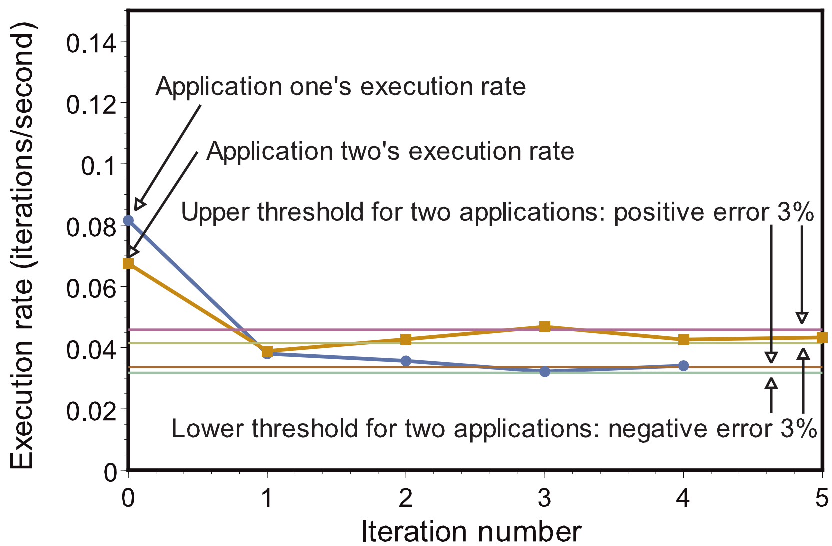


Figure 6.11: Running of two Patterns benchmarks under the control system, 1:1 comp/comm ratio.

on each node. We ran an experiment where we set the target utilization to be half of the maximum utilization among all nodes, i.e. 14%. Figure 6.10 illustrates the performance in this case. We can see that the actual execution rate is successfully brought to within ε of the target rate.

We are currently designing a system in which the global controller is given the freedom to set a different schedule on each node thus making our control system more flexible.

6.3.9 Time-sharing multiple parallel applications

To see how well we can provide time-sharing for multiple parallel applications, we simultaneously execute multiple Patterns benchmarks on the same four nodes of our cluster.

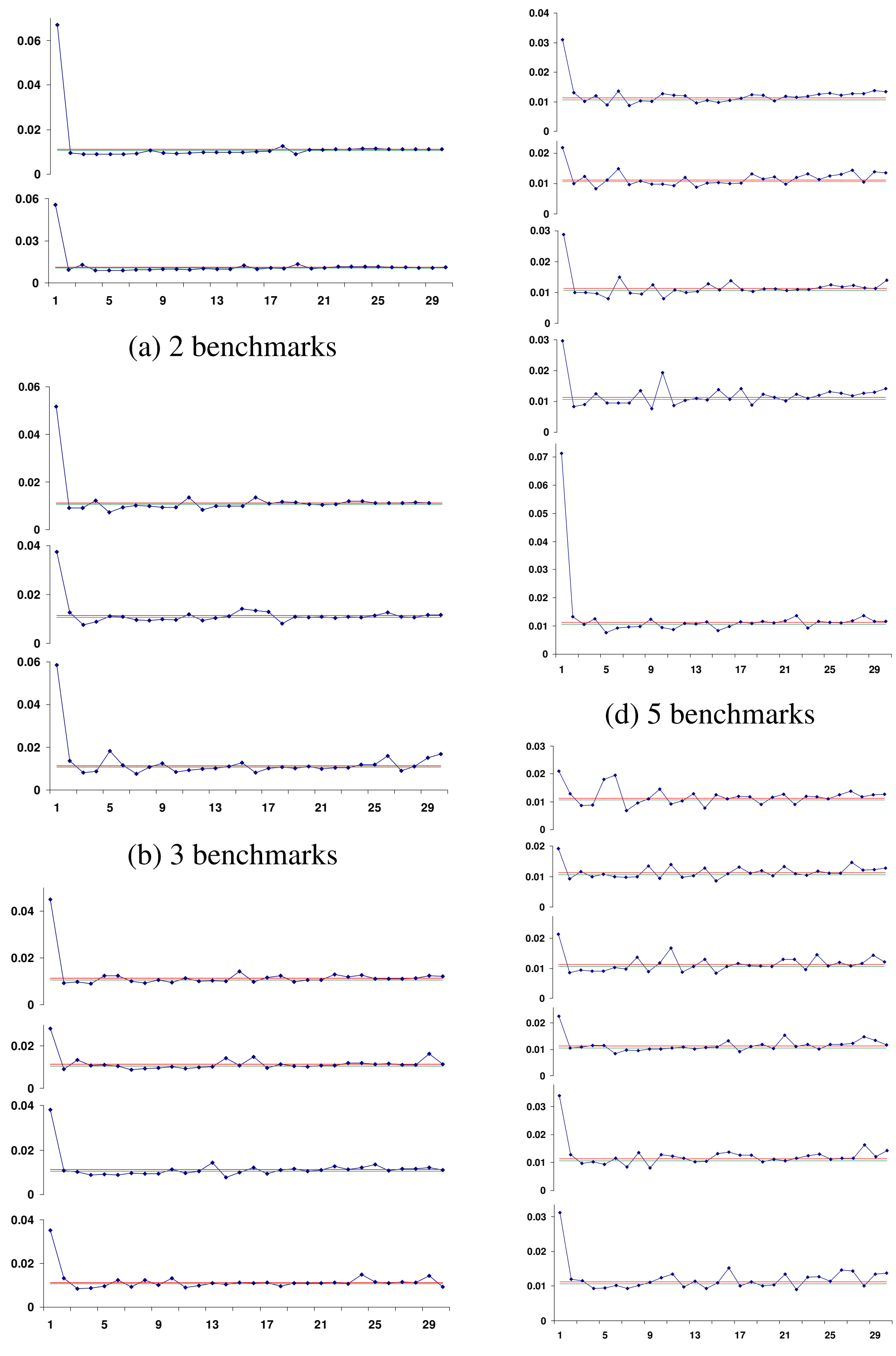
Figure 6.11 shows the results of running two Patterns applications, each configured with a 1:1 compute/communicate ratio. One was configured with a target rate of 30%, with the other set to 40%. We can clearly see that the actual execution rates are quickly brought to within ε of the target rates and remain there for the duration of the experiment.

Next, we consider what happens as we increase the number of Patterns benchmarks running simultaneously. In the following, each Patterns benchmark is set to execute with identical 10% utilization. We run Patterns with a 3:1 compute/communicate ratio. Figure 6.12 shows our results. Each graph shows the execution rate (iterations/second) as a function of the iteration, as well as the two 3% threshold lines. Figure 6.12(a) contains two such graphs, corresponding to two simultaneously executing Patterns benchmarks, (b) has three, and so on.

Overall, we maintain reasonable control as we scale the number of simultaneously executing benchmarks. Further, over the thirty iterations shown, in all cases, the average execution rate meets the target, within threshold.

We do notice a certain degree of oscillation when we run many benchmarks simultaneously. Our explanation is as follows. When VSched receives and admits a new schedule sent by the global controller, it will interrupt the current task and re-select a new task (perhaps the previous one) to run based on its deadline queue. As the number of parallel applications increases, each process of an application on an individual node will have a smaller chance of running uninterrupted throughout its slice. In addition, there will be a smaller chance of each process starting its slice at the same time.

The upshot is that even though the process will continue to meet its deadlines locally, it will be less synchronized with processes running on other nodes. This results in the application's overall performance changing, causing the global controller to be invoked more often. Because the control loop frequency is less than the frequency of these small performance changes, the system begins to oscillate. However, the degradation is graceful, and, again, the long term averages are well behaved.



(a) 2 benchmarks

(b) 3 benchmarks

(c) 4 benchmarks

(d) 5 benchmarks

(e) 6 benchmarks

Figure 6.12: Running multiple Patterns benchmarks; 3:1 comp/comm ratio; 3% threshold.

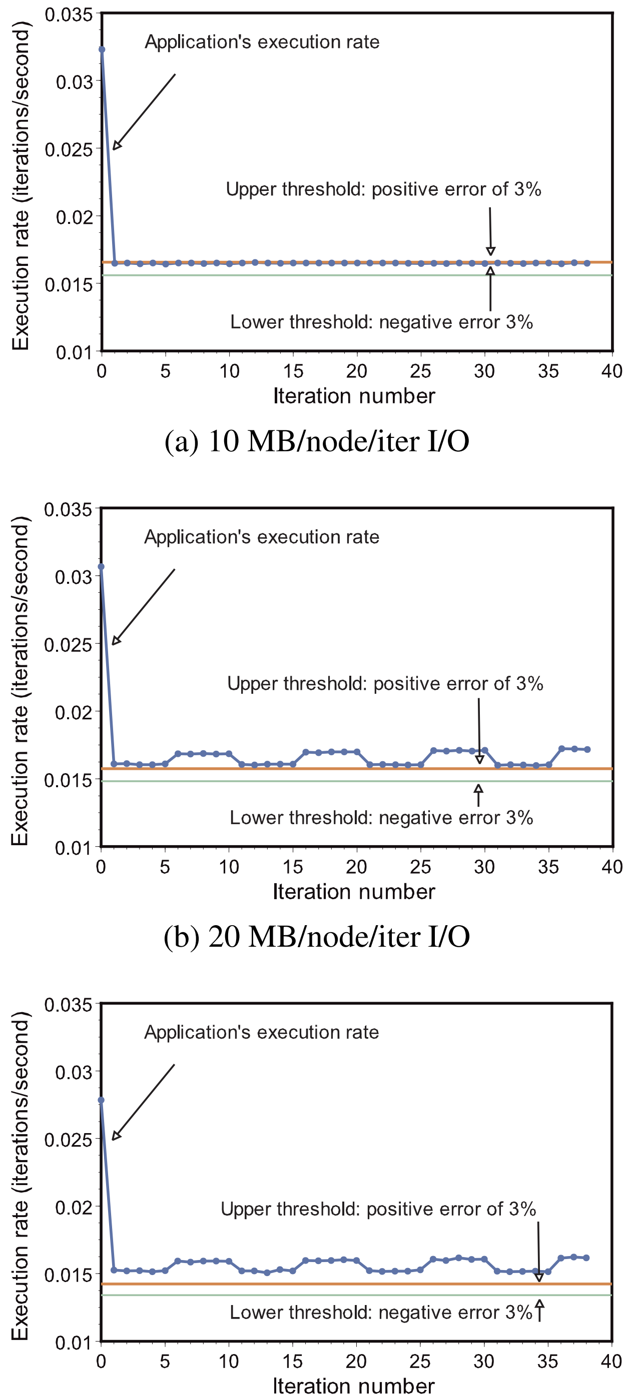
6.3.10 Effects of local disk I/O

Although we are only scheduling the CPU resource, it is clear from the above that this is sufficient to isolate and control a BSP application with complex collective communications of significant volume. Is it sufficient to control such an application when it also extensively performs local disk I/O?

To study the effects of local disk I/O on our scheduling system, we modified the Patterns benchmark to perform varying amounts of local disk I/O. In the modified Patterns, each node writes some number of bytes sequentially to the local IDE hard disk during each iteration. The file is `fsync()`ed, assuring that the data is written through to the physical disk.

In our first set of experiments, we configured Patterns with a very high (145:1) compute/communicate ratio, and 0, 1, 5, 10, 20, 40, and 50 MB per node per iteration of local disk I/O. Our target execution rate was 50% with a threshold of 3%. Figure 6.13 shows the results for 10, 20, and 40 MB/node/iter. 0, 1, 5 are similar to 10, while 50 is similar to 40. For up to 10 MB/node/iter, our system effectively maintains control of the application's execution rate. As we exceed this limit, we develop a slight positive bias; the application runs faster than desired despite the restricted CPU utilization. The dominant part of the time spent on local disk I/O is spent waiting for the disk. As more I/O is done, a larger proportion of application execution time is outside of the control of our system. Since the control algorithm requires that the CPU utilization be equal to the target execution rate, the actual execution rate grows.

In the second set of experiments, we fixed the local disk I/O to 10 MB/node/iter (the maximum controllable situation in the previous experiment) and varied the compute/communicate ratio, introducing different amounts of network I/O. We used a target rate of 50%. We used seven compute/communicate ratios ranging from 4900:1 to

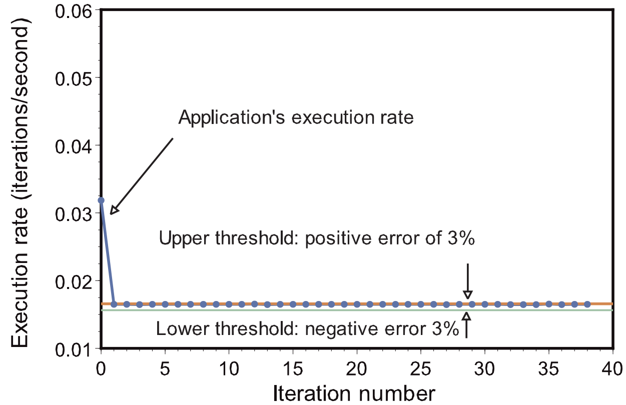


(a) 10 MB/node/iter I/O

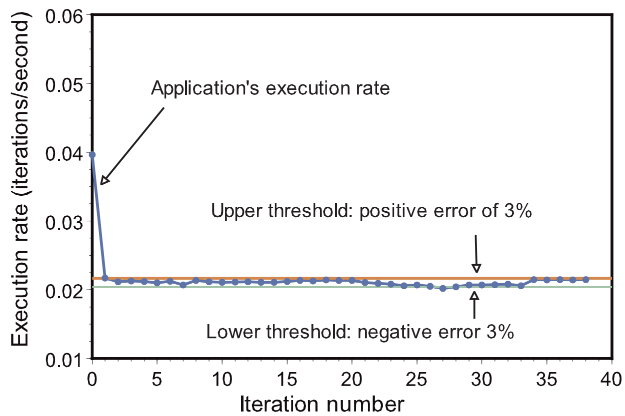
(b) 20 MB/node/iter I/O

(c) 40 MB/node/iter I/O

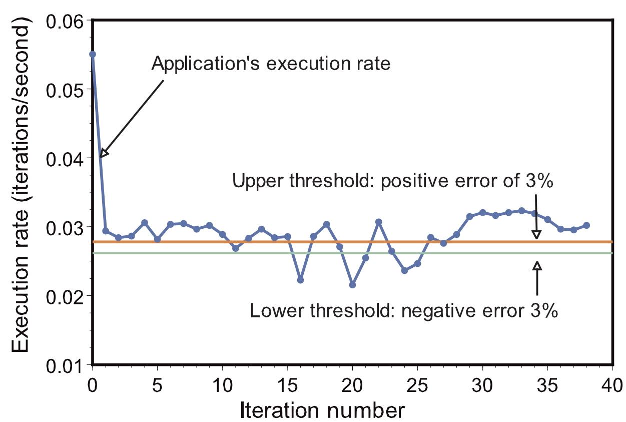
Figure 6.13: Performance of control system with a high (145:1) comp/comm ratio and varying local disk I/O.



(a) high (4900:1) comp/comm ratio



(b) medium (2:1) comp/comm ratio



(c) low (1:3.5) comp/comm ratio

Figure 6.14: Performance of control system with 10 MB/node/iter of disk I/O and varying comp/comm ratios.

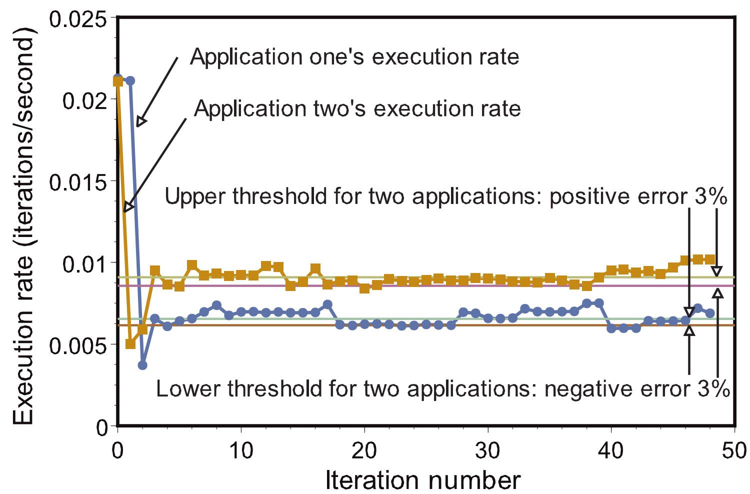


Figure 6.15: Running two Patterns benchmarks under the control system; high (130:1) comp/comm ratio. The combined working set size is slightly less than the physical memory.

1:3.5. Figure 6.14 shows the results for 4900:1, 2:1, and 1:3.5. For high to near 1:1 compute/communicate ratios, our system can effectively control the application's execution rate even with up to 10 MB/node/iteration of local I/O, and degrades gracefully after that.

Our system can effectively control the execution rates of applications performing significant amounts of network and local disk I/O. The points at which control effectiveness begins to decline depends on the compute/communicate ratio and the amount of local disk I/O. With higher ratios, more local disk I/O is acceptable. We have demonstrated control of an application with a 1:1 ratio and 10 MB/node/iter of local disk I/O.

6.3.11 Effects of physical memory use

Our technique makes no attempt to isolate memory, but the underlying node OS certainly does so. Is it sufficient?

To evaluate the effects of physical memory contention on our scheduling system, we modified the Patterns benchmark so that we could control its working set size. We then ran two instances of the modified benchmark simultaneously on the four nodes of our cluster. We configured the first instance with a working set of 600 MB and a target execution rate of 30%, while the second was configured with a working set size of 700 MB and a target rate of 40%. Both instances had a compute/communicate ratio of around 130:1. The combined working set of 1.3 GB is slightly less than the 1.5 GB of memory of our cluster nodes.

We used the control algorithm to schedule the two instances, and Figure 6.15 shows the results of this experiment. We see that despite the significant use of memory by both instances, our system maintains control of both applications' execution rates.

Our results suggest that unless the total working set on the machine is exceeded, physical memory use has little effect on the performance of our scheduling system. It is important to point out that most OS kernels, including Linux, have mechanisms to restrict the physical memory use of a process. These mechanisms can be used to guarantee that the physical memory pressure on the machine does not exceed the supply. A virtual machine monitor such as Xen or VMware provides additional control, enforcing a physical memory limit on a guest OS kernel and all of its processes.

6.4 Conclusions

We have explored automatic, run-time and dynamic adaptation in scenarios wherein it is not possible to infer application demands. In such cases the application, through a thin interface, specifies its resource demands. This work has focused on CPU demands and CPU reservations to study the effectiveness of this adaptation mechanism in isolation. In particular we have described the design, implementation and evaluation of a new approach to time-sharing parallel applications on tightly coupled compute resources such as clusters. This work is also applicable to virtual execution environments. It should be noticed that at the operating system level, a virtual machine resembles an application process.

Our technique, performance-targetted feedback-controlled real-time scheduling, is based on the combination of local scheduling using the periodic real-time model and a global feedback control system that sets the local schedules. The approach performance-isolates parallel applications and allows administrators to dynamically change the desired application execution rate while keeping actual CPU utilization proportional to the application execution rate. Our implementation takes the form of a user-level scheduler for Linux and a centralized controller. Our evaluation shows the system to be stable with low response times. The thresholds needed to prevent control instability are quite reasonable. Despite only isolating and controlling the CPU, we find that memory, communication I/O, and local disk I/O follow.

Chapter 7

Heuristic driven adaptation algorithms

In this chapter and the next (Chapter 8), we present the results of our attempts to solve the complex adaptation problem as a whole. The adaptation problem in virtual execution environments is not only NP-hard, but is also NP-hard to approximate. This insight leads us in the direction of heuristic driven algorithms as solutions to the adaptation problem. A heuristic typically relies on a rule of thumb or intelligent guesswork. For example, a greedy heuristic is based on the assumption that if at each decision point we make a greedy decision then it will ultimately result in global good. A heuristic by definition is sub-optimal in certain cases. The performance of a heuristic in a certain scenario depends on a number of variables such as the specific heuristic in question and the details of the scenario in which it is operating.

A heuristic algorithm could be considered a reasonable solution to a problem if the following four conditions hold:

- The problem is proven to be NP-hard. This implies that is it not possible to find the optimal solution in polynomial time.
- The problem is also proven to be NP-hard to approximate within a a certain factor of the optimal. This implies that combinatorially approximate solutions are not easy

to design.

- The performance of the heuristic, for the most common cases, is empirically within a small factor of the optimal. This implies that though the heuristic could produce sub-optimal solutions at times, for the most common cases, it works pretty well in practice.

- For typical problem sizes the heuristic should complete in a reasonable and acceptable amount of time.

Building upon the NP-hardness and inapproximability results of the adaptation problem in virtual execution environments, in this chapter we introduce a wide range of greedy heuristics. For these greedy heuristics to be acceptable solutions to the adaptation problem they must perform well and efficiently in practice for the most common cases.

Our methodology for the design and evaluation of objective functions and heuristic algorithms has been to study a small set of applications, representative of three of the application classes described in Chapter 2. Based on these studies we designed six different objective functions that different applications might want optimized in different scenarios. Our adaptation problem consists of two main components. First, a VM to host mapping component and second, a communicating VMs to overlay paths routing component. We have designed eight different variations of heuristic algorithms for the mapping problem and six different routing algorithms. Each of these algorithms is heuristically driven by a greedy strategy. At any decision point, the algorithms always make a greedy choice.

These mapping and routing algorithms can be combined in different ways to produce fifteen different adaptation schemes (not all of the 48 possible combinations make sense). It should be noted that not all of the fifteen algorithms attempt to optimize each of the objective functions. Additionally, not all the objective functions are important for every distributed application.

This dissertation studies the effect of these fifteen different adaptation schemes on a range of distributed applications to answer the question, is any one of these schemes generally applicable to a range of distributed application classes? It should be noted that the term "algorithm" is used to refer to the mapping and routing algorithms, while the term "adaptation scheme" refers to the fifteen combinations developed. In places where there is no ambiguity, we often use algorithms and adaptation schemes interchangeably.

There are two possible approaches to comparing adaptation schemes against each other. In the first approach, we define objective functions that the algorithms are attempting to optimize. We then compare performance in terms of values for the objective functions, that each of the adaptation schemes can generate. Such an evaluation tells us algorithm goodness, but only in the context of the objective functions. This evaluation in isolation comes with the big assumption that optimizing the stated objective function *always* results in better application performance for some application defined meaning of performance. The second approach to evaluating algorithms uses the above as only an initial indicator step. It follows it up with an evaluation of the adaptation schemes against the applications where algorithm goodness is directly measured in some application defined terms, this second component is called "closing the loop".

In this chapter we compare the fifteen adaptation schemes against each other and against estimates of optimal solutions in the context of the six objective functions (as opposed to comparison in the context of application performance.) In Chapter 8, we "close the loop" and compare these adaptation schemes for different applications in the context of application performance.

We begin this chapter in Section 7.1 by describing the methodology we adopted to the design and evaluation of the objective functions and heuristic algorithms. The objective functions are described in Section 7.2. Section 7.3 present our mapping algorithms and our routing algorithms are described in Section 7.4. The fifteen combined adaptation schemes are detailed in Section 7.5. Finally, we present a detailed evaluation of these adaptation schemes in the context of the six objective functions and for a wide range of adaptation scenarios in Section 7.6. We conclude this chapter in Section 7.7.

7.1 Design and evaluation methodology

In the course of this research we have studied ten diverse classes of distributed applications. These applications ranged from high performance scientific applications to transactional web applications to enterprise IT applications. These application classes are listed in Table 2.1 and are described in detail in Chapter 2. We divided these ten application classes into two categories. The first was a small category of three application classes that we studied extensively to help us understand the needs of distributed applications and consequently help us in the design of objective functions. The second was a larger category of seven application classes over which we evaluated our fifteen adaptation schemes.

In particular, the first three applications classes listed in Table 2.1 were used to design our objective functions and heuristic algorithms. The last seven application classes listed in Table 2.1 were used to close the loop and to evaluate the effectiveness of the proposed adaptation schemes.

7.2 Objective functions

We designed six different objective functions. Each of these objective functions, individually or in some combination, are important in terms of optimizing different distributed application classes. These six objective functions are by no means exhaustive. They were arrived at as they represent diverse classes of application demands. Further, not all of them are application centric, one of the objective functions attempts to optimize the system as opposed to optimizing the application in isolation.

The two main resources we are trying to optimize for are computational and networking resources. For any application, we would like to reduce the amount of time it spends doing computations and the amount of time it takes for data to flow across the network. Depending on the amount of data flowing across, it would either depend on the bandwidth available or the latency experienced. The six objective functions cover different variations of computational time, bandwidth and latency. The reason they are representative of diverse classes of application demands is that the application classes differ from each other in the computational needs and networks demands (bandwidth and latency) and these different needs (low compute time, high bandwidth and low latency) are captured by the objective functions.

The goal of most adaptation schemes operating in the context of virtualized execution environments is either to optimize the application or to optimize the system or in some rare cases, both. What does it mean to optimize the application? Does it mean that the application must finish earlier? Or does it refer to increased throughput for the application? Or does it means better response time for interactive applications? It could be all of the above for different application classes in different scenarios. The best definition of optimizing an application is to optimize an application defined objective function. Having said that, this dissertation is about automatic, run-time and dynamic adaptation of applications without user or developer interaction.

To achieve our goal we design generalized objective functions that are important for a range of distributed applications. We use these objective functions to design adaptation schemes. The assumption is that adaptation schemes that optimize a majority of these objective functions will also optimize a majority of the applications when evaluated based on application specific metrics and we test this assumption in Chapter 8. Chapter 8 details the motivation behind the search for a single adaptation scheme that is applicable to a range of distributed applications. It should be noticed that there are three steps to evaluating the adaptation schemes:

1. We evaluate each adaptation scheme's solution against an estimate of the optimal solution for each objective function. This gives a general idea regarding the quality of the adaptation scheme.

2. We evaluate the adaptation schemes to see which works best for a range of objective functions. The quality is judged based on objective function values achieved for the different objective functions. This simply gives us an indication as to what might be the behavior of these adaptation schemes for application specific metrics.

3. We evaluate the adaptation schemes to see which works best for a range of distributed applications. The quality is judged based on the values of application defined and application specific metrics.

The specific objective functions defined are stated in Table 7.1 and described below. Based on the insights gained by studying three classes of applications, the aim was to devise objective functions that cover the properties most important for a wide range of application classes. Compute time, bandwidth and latency are three important properties. Applications either wish to minimize computation time or maximize bandwidth and minimize latency such that transfer time is reduced. These objective functions capture such requirements of application classes.

7.2.1 MEETI: Maximize sum of estimated execution time improvement

This objective function attempts to estimate the benefits of preemptive migration in the scenario of virtual execution environments. A lot of related work has been done on preemptive migration in the context of process load balancing [78]. Harchol-Balter et al. measured

Number	Name	Description
1.	MEETI	Maximize Sum of Estimated Execution Time Improvement
2.	MSPL	Minimize Sum of Path Latencies
3.	MSRBB	Maximize Sum of Residual Bottleneck Bandwidths
4.	NSRBB	Minimize Sum of Residual Bottleneck Bandwidths
5.	MSBBL	Maximize Sum of Bottleneck Bandwidth and Latency
6.	MSRBBL	Maximize Sum of Residual Bottleneck Bandwidth and Latency

Table 7.1: Six defined objective functions.

the distribution of lifetimes of UNIX processes and proposed a functional form that fit that distribution well. They answered the question, when should migration occur and which processes should be migrated [78]? The rule of thumb proposed in that work was that the probability that a process with a CPU age of one second uses more than T seconds of total CPU time is 1/T [78]. The MEETI objective function is based on this analysis.

In our scenario when adaptation is called into play, the mapping algorithm determines a new mapping. How do we compare this new mapping to the previous one? Is it better or worse than the previous mapping? We use the following analysis to determine the MEETI objective function.

We denote by H the hosts in the virtual execution system. The set of virtual machines participating in the application is denoted by the set VM. We define an initial mapping from VMs to hosts, vmap1 : $VM \rightarrow H$. We also define a final mapping from VMs to hosts, vmap2 : $VM \rightarrow H$. The compute capacity made available by each host is described by a host compute capacity function, C : $H \rightarrow \mathbb{R}$. The utilization made available by the host for a VM mapped onto it is represented by a host utilization function, U : $H \rightarrow \mathbb{R}$. The migrate function denoted by, M : $VM\ X\ H\ X\ H \rightarrow \mathbb{R}$, provides an estimate of the cost of migration of a VM from one host to another. We are also given a threshold value T. This indicates that we should migrate a VM from one host to another, if and only if the reduction in estimated execution time is more than the value of this threshold. We denote $I = (C(vmap2(VM_i)) * U(vmap2(VM_i)))/(C(vmap1(VM_i)) * U(vmap1(VM_i)))$

The value of the MEETI objective function is defined as

$$\sum_{i=1}^{m}((W(VM_i))/I) - M(VM_i, vmap1(VM_i), vmap2(VM_i)) - T$$

If the value of the MEETI function is greater than zero, than the collective decision of migrating the VMs has proven to be a positive one from the perspective of reducing computation time. We want the value of the MEETI function to be as high as possible, i.e. we wish to maximize the MEETI objective function.

7.2.2 MSPL: Minimize sum of path latencies

The MSPL objective function attempts to study the effect of the adaptation schemes on the latency between communicating pairs of virtual machines. This is particularly important for interactive applications and for applications wherein very small amounts of data are transferred over the network. For each pair of communicating VM, we calculate its path latency based on the specific path from source to destination. The value of MSPL is the sum of this value over all pairs of communicating VMs. We want this sum to be as low as possible, in other words we want to minimize MSPL.

Using the terminology developed in Table 4.1, value of the MSPL objective function is defined as

$$\sum_{i=1}^{m}(lat_{e \in R(A_i)})$$

and is minimized.

7.2.3 MSRBB: Maximize Sum of Residual Bottleneck Bandwidths

The MSRBB objective function is designed to study the effect of adaptation schemes on applications that have high bandwidth demands. Scientific high performance applications often transfer large amounts of data between different nodes, thereby needing high bandwidth. Once all the VMs have been mapped and routes decided. We can calculate the residual bandwidth left remaining on each link. For each pair of communicating VM, we can then calculate the minimum residual value along its path. The value of MSRBB is the sum of this minimum value over all pairs of communicating VMs. We wish to maximize this value. The intuition is that this allows the application maximum scope to utilize more bandwidth in the existing configuration.

Using the terminology developed in Table 4.1, value of the MSRBB objective function is defined as

$$\sum_{i=1}^{m} \Big(\min_{e \in R(A_i)} \{\mathrm{rc}_e\} \Big)$$

where

$$\mathrm{rc}_e = (\mathrm{bw}_e - \sum_{e \in R(A_i)} b_i)$$

and is maximized.

7.2.4 NSRBB: Minimize Sum of Residual Bottleneck Bandwidths

The NSRBB objective function has the exact same value as the MSRBB objective function. However, it is designed to study the effect of minimizing this value as opposed to maximizing this value. The intuition behind minimizing the sum of the residual bottleneck bandwidths across all pairs of communicating VMs is to leave maximum room for other applications to enter the system. Majority of the objective functions are application centric. The NSRBB objective function, on the other hand, is system centric, looking to increase the utilization of the system.

Using the terminology developed in Table 4.1, value of the NSRBB objective function is defined as

$$\sum_{i=1}^{m} \Big(\min_{e \in R(A_i)} \big\{ \mathrm{rc}_e \big\} \Big)$$

where

$$\mathrm{rc}_e = (\mathrm{bw}_e - \sum_{e \in R(A_i)} b_i)$$

and is minimized.

7.2.5 MSBBL: Maximize Sum of Bottleneck Bandwidth and Latency

The MSBBL objective function was designed to study applications for which latency and bandwidth are both important. Many database applications fall into this category. Often times the queries are evenly divided between those where the data returned is very small making latency important and those that return large amount of data making bandwidth the bottleneck. Since latency and bandwidth are dimensionally not the same, we introduce a constant, called the latency constant with units of MB. The sum of latency and bandwidth can now be represented as $bandwidth + (constant/latency)$. The latency constant is a tunable parameter, larger is its value, larger is the weight assigned to latency. Notice that when the latency constant is set to zero MSBBL degenerates into MSRBB.

Using the terminology developed in Table 4.1, value of the MSBBL objective function is defined as

$$\sum_{i=1}^{m} \Big(\min_{e \in R(A_i)} \big\{ \mathrm{bl}_e \big\} \Big)$$

where

$$\mathrm{bl}_e = (\mathrm{bw}_e + c/\mathrm{lat}_e)$$

and is maximized.

7.2.6 MSRBBL: Maximize Sum of Residual Bottleneck Bandwidth and Latency

The MSRBBL objective function, like the MSBBL objective function, attempts to capture distributed applications for which bandwidth and latency, both are important. The MSBBL objective function did not take into account the bandwidth and latency demands made by the application. It simply attempted to find the paths that had the maximum bandwidth and minimum latency. The MSRBBL on the hand, also takes into account the application bandwidth and latency demands and attempts to find paths that have high bottleneck bandwidth and low latency in relation to the bandwidth and latency demands expressed by the application.

Using the terminology developed in Table 4.1, value of the MSBBL objective function is defined as

$$\sum_{i=1}^{m} \Big(\min_{e \in R(A_i)} \{\mathrm{bl}_e\} \Big)$$

where

$$\mathrm{bl}_e = (\mathrm{bw}_e + c/\mathrm{lat}_e - \sum_{e \in R(A_i)} (b_i + c/l_i))$$

and it is maximized

7.3 VM to physical host mapping algorithms

We have devised eight algorithms driven by greedy heuristics for mapping VMs to hosts. Table 7.3 provides a listing of these mapping algorithms.

In all the algorithms, VMs are mapped onto physical hosts and the input to the algorithm is the application communication behavior as captured by VTTIF and available

Number	Name	Description
1.	CLB	Computational Load Balancing
2.	GOBM	Greedy One-pass Bandwidth Map
3.	GOBBM	Greedy One-pass Bottleneck Bandwidth Map
4.	GOELM	Greedy One-pass End-end Latency Map
5.	GOPLM	Greedy One-pass Path Latency Map
6.	GOBLM	Greedy One-pass Bandwidth Latency Map
7.	GOBBLM	Greedy One-pass Bottleneck Bandwidth Latency Map
8.	GTBM	Greedy Two-pass Bandwidth Map

Table 7.2: Eight greedy VM to host mapping algorithms.

bandwidth and latency between each pair of VNET daemons, as reported by Wren, both expressed as adjacency lists. Further, some of the algorithms (Algorithm 1) also take additional inputs such as host characteristics like speed and utilization, and a threshold to determine when to start migrating virtual machines. Some of the algorithms (Algorithms 11 and 13) also take a latency constant c as an input. This is to create a objective function that takes into account, both bandwidth and latency. The constant is used to give weights to latency and bandwidth measurements.

All of the algorithms are slight variations of greedy algorithms. In each, we order the VMs or pairs of communicating VMs in increasing or decreasing order of a certain property. We also order the physical hosts or pairs of physical hosts in increasing or decreasing order of the same property and then we start mapping VMs to hosts in a greedy fashion. The primary difference between the different algorithms is the property used for ordering and the exact nature of the ordering, increasing or decreasing. The algorithms are described in English and in pseudo code in Algorithms 1 through Algorithms 16.

Algorithm 1 Computational load balancing (CLB) in English

Order the VMs in decreasing order of compute demands
Order the VNET daemons in decreasing order of the product of compute capacity and utilization available
while There are unmapped VMs **do**
  if The VMs has already not been considered for re-mapping **then**
    Map it to the first hosts which currently have no VMs mapped onto them and the estimated execution time is reduced by at least a certain pre-defined threshold as detailed in Section 7.2.1
  end if
end while
Compute the difference between the current mapping and the new mapping and issue VM migration instructions to achieve the new mapping.

Algorithm 2 Computational Load Balancing in pseudo code CLB(VM,VNET,Map,ReverseMap,Util,Speed,T,M,NewMap,NewReverseMap)

$Quicksort(VM, 1, length[VM])$
$Quicksort(VNET, 1, length[VNET])$
$Q \leftarrow VM$
$P \leftarrow VNET$
while $Q \neq \emptyset$ **do**
  $v \leftarrow ExtractMax(Q)$
  $R \leftarrow P$
  while $R \neq \emptyset$ **do**
    $w \leftarrow ExtractMax(R)$
    if $((Util(w) * Speed(w)/Util(ReverseMap(v)) * Speed(ReverseMap(v))) + T + M \geq 0)$ **then**
      $NewMap(w) \leftarrow v$
      $NewReverseMap(v) \leftarrow w$
      $break$
    end if
  end while
end while
$Diff(Map, NewMap)$

Algorithm 3 Greedy one-pass bandwidth map (GOBM) in English

Order the VM adjacency list by decreasing traffic intensity
Order the VNET daemon adjacency list by decreasing throughput
while There are unmapped VMs **do**
  if both the VMs for a communicating pair are not mapped **then**
    Map them to the first pair of hosts which currently have no VMs mapped onto them
  else
    Map the VM to a VNET daemon such that the throughput estimate between the VM and its already mapped counterpart is maximum
  end if
end while
Compute the difference between the current mapping and the new mapping and issue VM migration instructions to achieve the new mapping.

7.4 Inter VM routing algorithms

We have devised six algorithms driven by greedy heuristics for mapping pairs of communicating VMs to paths in the VNET overlay network. Table 7.3 provides a listing of these routing algorithms.

It should be noted that although for ease of discussion we have broken the adaptation problem into a mapping and a routing component, it is not necessary that first the mapping algorithm finishes execution for all VMs and then only the routing algorithm is run. The mapping and the routing algorithms could also be interleaved with each other. In other words, there are two modes in which the adaptation algorithms could operate, sequential or interleaved mode. In the former, all the mappings are effected first and then all the routing changes are made. In the latter, for each mapping change effected, the routing change is also made before the next mapping can be changed.

Algorithm 4 Greedy one-pass bandwidth map in pseudo code GOBM(VNET,VMD,VNETD,Map,RevereseMap,NewMap,NewReverseMap)

```
Quicksort(VMD, 1, length[VMD])
Quicksort(VNETD, 1, length[VNETD])
P ← VNET
while VMD ≠ ∅ do
   w ← ExtractMax(VMD)
   if ((NewReverseMap(w(i)) == ∅) and (NewReverseMap(w(j)) == ∅)) then
      while P ≠ ∅ do
         v ← ExtractMax(P)
         if ((NewMap(v(i)) == ∅)and(NewMap(v(j)) == ∅)) then
            NewReverseMap(w(i)) ← v(i)
            NewReverseMap(w(j)) ← v(j)
            NewMap(v(i)) ← w(i)
            NewMap(v(j)) ← w(j)
            break
         end if
      end while
   else if (NewReverseMap(w(i)) ≠ ∅) then
      NewReverseMap(w(j)) ← MAX(NewReverseMap(w(i)), VNET)
      NewMap(MAX(NewReverseMap(w(i)), VNET) ← w(j)
   else
      NewReverseMap(w(i)) ← MAX(NewReverseMap(w(j)), VNET)
      NewMap(MAX(NewReverseMap(w(j)), VNET) ← w(i)
   end if
end while
Diff(Map, NewMap)
```

Algorithm 5 Greedy one-pass bottleneck bandwidth map (GOBBM) in English

Order the VM adjacency list by decreasing traffic intensity
Extract an ordered list of VMs from the above with a breadth first approach, eliminating duplicates
For each pair of VNET daemons, find the maximum bottleneck bandwidth (the widest path) using the adapted Dijkstra's algorithm described in Section 7.4.2
Order the VNET daemon adjacency list by decreasing bottleneck bandwidth
Extract an ordered list of VNET daemons from the above with a breadth first approach, eliminating duplicates
Map the VMs to VNET daemons in order using the ordered list of VMs and VNET daemons obtained above
Compute the differences between the current mapping and the new mapping and issue migration instructions to achieve the new mapping

Algorithm 6 Greedy one-pass bottleneck bandwidth map in pseudo code GOBBM(G,B,VNET,VMD,VNETD,Map,RevereseMap,NewMap,NewReverseMap)

$Quicksort(VMD, 1, length[VMD])$
$P \leftarrow BFS(VMD.VMD[0](i))$
for all vnetd in VNETD **do**
  $vnetd - b(i, j, k) \leftarrow (vnet(i), vnet(j), adapted\_dijkstra(G, B, vnet(i))$
end for
$Quicksort(VNETD - B, 1, length[VNETD - B])$
$Q \leftarrow BFS(VNETD - B, VNETD - B[0](i))$
for all p[i] in P **do**
  $NewReserveMap(p[i]) = q[i]$
  $NewMap(q[i]) = p[i]$
end for
$Diff(Map, NewMap)$

Algorithm 7 Greedy one-pass end-end latency map (GOELM)

Order the VM adjacency list by increasing traffic latency a
Order the VNET daemon adjacency list by increasing latency
while There are unmapped VMs **do**
  if both the VMs for a communicating pair are not mapped **then**
    Map them to the first pair of hosts which currently have no VMs mapped onto them
  else
    Map the VM to a VNET daemon such that the latency estimate between the VM and its already mapped counterpart is minimum
  end if
end while
Compute the difference between the current mapping and the new mapping and issue VM migration instructions to achieve the new mapping.

Number	Name	Description
1.	OHR	Overlay Hop Reduction
2.	WBP	Widest Bandwidth Path
3.	NBP	Narrowest Bandwidth Path
4.	WBLP	Widest Bandwidth Latency Path
5.	WRBLP	Widest Residual Bandwidth Latency Path
6.	STLP	Shortest Total Latency Path
	S	Sequential ordering of mapping and routing algorithms
	I	Interleaved ordering of mapping and routing algorithms

Table 7.3: Six greedy communicating VM pairs to VNET overlay network paths routing algorithms.

Algorithm 8 Greedy one-pass end-end latency map (GOELM)(VNET,VMD,VNETD,Map,RevereseMap,NewMap,NewReverseMap)

```
Quicksort(VMD, 1, length[VMD])
Quicksort(VNETD, 1, length[VNETD])
P ← VNET
while VMD ≠ ∅ do
   w ← ExtractMin(VMD)
   if ((NewReverseMap(w(i)) == ∅) and (NewReverseMap(w(j)) == ∅)) then
      while P ≠ ∅ do
         v ← ExtractMin(P)
         if ((NewMap(v(i)) == ∅)and(NewMap(v(j)) == ∅)) then
            NewReverseMap(w(i)) ← v(i)
            NewReverseMap(w(j)) ← v(j)
            NewMap(v(i)) ← w(i)
            NewMap(v(j)) ← w(j)
            break
         end if
      end while
   else if (NewReverseMap(w(i)) ≠ ∅) then
      NewReverseMap(w(j)) ← MIN(NewReverseMap(w(i)), VNET)
      NewMap(MAX(NewReverseMap(w(i)), VNET) ← w(j)
   else
      NewReverseMap(w(i)) ← MIN(NewReverseMap(w(j)), VNET)
      NewMap(MAX(NewReverseMap(w(j)), VNET) ← w(i)
   end if
end while
Diff(Map, NewMap)
```

Algorithm 9 Greedy one-pass path latency map (GOPLM)

Order the VM adjacency list by increasing traffic latency
Extract an ordered list of VMs from the above with a breadth first approach, eliminating duplicates
For each pair of VNET daemons, find the minimum latency path (the shortest path) using Dijkstra's shortest paths algorithm
Order the VNET daemon adjacency list by increasing path latencies
Extract an ordered list of VNET daemons from the above with a breadth first approach, eliminating duplicates
Map the VMs to VNET daemons in order using the ordered list of VMs and VNET daemons obtained above
Compute the differences between the current mapping and the new mapping and issue migration instructions to achieve the new mapping

Algorithm 10 Greedy one-pass path latency map in pseudo code GO-PLM(G,L,VNET,VMD,VNETD,Map,RevereseMap,NewMap,NewReverseMap)

$Quicksort(VMD, 1, length[VMD])$
$P \leftarrow BFS(VMD.VMD[0](i))$
for all vnetd in VNETD **do**
  $vnetd - l(i, j, k) \leftarrow (vnet(i), vnet(j), dijkstra(G, L, vnet(i))$
end for
$Quicksort(VNETD - L, 1, length[VNETD - B])$
$Q \leftarrow BFS(VNETD - L, VNETD - L[0](i))$
for all p[i] in P **do**
  $NewReserveMap(p[i]) = q[i]$
  $NewMap(q[i]) = p[i]$
end for
$Diff(Map, NewMap)$

Algorithm 11 Greedy one-pass bandwidth latency map (GOBLM)

Order the VM adjacency list by decreasing traffic intensity, where traffic intensity is calculated as a function of latency and bandwidth (bandwidth + c/latency)
Order the VNET daemon adjacency list by decreasing function of latency and bandwidth
while There are unmapped VMs **do**
  if both the VMs for a communicating pair are not mapped **then**
    Map them to the first pair of hosts which currently have no VMs mapped onto them
  else
    Map the VM to a VNET daemon such that the bandwidth latency function estimate between the VM and its already mapped counterpart is maximum
  end if
end while
Compute the difference between the current mapping and the new mapping and issue VM migration instructions to achieve the new mapping.

7.4.1 Algorithms OHR, WBP and STLP

We use greedy heuristic algorithms to determine a path for each pair of communicating VMs. As above we use VTTIF and Wren outputs expressed as adjacency lists as inputs.

Algorithm OHR is a simple hop reduction algorithm that tries to create direct overlay links between communicating source-destination VMs based on their host mappings. The algorithm is described in Algorithm 17. It should be noted that we make the simplifying assumption that all direct paths are possible. However, in reality, this may not always be the case. In such cases the current version of the algorithm would simply fail. A more robust way of accounting for such cases would be to build a policy aware routing scheme. Such a scheme would have the following pieces of functionality, first, detecting if some policy exists, second, finding a workaround the same.

Algorithm STLP is similar to algorithm OHR, except that instead of connecting the source and destination VMs by a direct path, we connect them via a path which has the lowest latency. In other words, we find the shortest path. We use Dijkstra's shortest path algorithm for the same [28]. The algorithm is described in Algorithm 19.

Algorithm 12 Greedy one-pass bandwidth latency map in pseudo code GOBLM(VNET,VMD,VNETD,Map,RevereseMap,NewMap,NewReverseMap)

```
Quicksort(VMD, 1, length[VMD])
Quicksort(VNETD, 1, length[VNETD])
P ← VNET
while VMD ≠ ∅ do
   w ← ExtractMax(VMD)
   if ((NewReverseMap(w(i)) == ∅) and (NewReverseMap(w(j)) == ∅)) then
      while P ≠ ∅ do
         v ← ExtractMax(P)
         if ((NewMap(v(i)) == ∅)and(NewMap(v(j)) == ∅)) then
            NewReverseMap(w(i)) ← v(i)
            NewReverseMap(w(j)) ← v(j)
            NewMap(v(i)) ← w(i)
            NewMap(v(j)) ← w(j)
            break
         end if
      end while
   else if (NewReverseMap(w(i)) ≠ ∅) then
      NewReverseMap(w(j)) ← MAX(NewReverseMap(w(i)),VNET)
      NewMap(MAX(NewReverseMap(w(i)),VNET) ← w(j)
   else
      NewReverseMap(w(i)) ← MAX(NewReverseMap(w(j)),VNET)
      NewMap(MAX(NewReverseMap(w(j)),VNET) ← w(i)
   end if
end while
Diff(Map,NewMap)
```

Algorithm 13 Greedy one-pass bottleneck bandwidth latency map (GOBBLM)

Order the VM adjacency list by decreasing traffic intensity where traffic intensity is calculated as a function of latency and bandwidth (bandwidth + c/latency)
Extract an ordered list of VMs from the above with a breadth first approach, eliminating duplicates
For each pair of VNET daemons, find the maximum bottleneck function of latency and bandwidth (the widest path taking into account both latency and bandwidth) using a variant of the adapted Dijkstra's algorithm described in Section 7.4.2
Order the VNET daemon adjacency list by decreasing bottleneck function of latency and bandwidth
Extract an ordered list of VNET daemons from the above with a breadth first approach, eliminating duplicates
Map the VMs to VNET daemons in order using the ordered list of VMs and VNET daemons obtained above
Compute the differences between the current mapping and the new mapping and issue migration instructions to achieve the new mapping

Algorithm 14 Greedy one-pass bottleneck bandwidth latency map in pseudo code GOBBLM(G,BL,VNET,VMD,VNETD,Map,RevereseMap,NewMap,NewReverseMap)

$Quicksort(VMD, 1, length[VMD])$
$P \leftarrow BFS(VMD.VMD[0](i))$
for all vnetd in VNETD **do**
  $vnetd - bl(i, j, k) \leftarrow (vnet(i), vnet(j), adapted\_dijkstra\_variant(G, BL, vnet(i))$
end for
$Quicksort(VNETD - BL, 1, length[VNETD - BL])$
$Q \leftarrow BFS(VNETD - BL, VNETD - BL[0](i))$
for all p[i] in P **do**
  $NewReserveMap(p[i]) = q[i]$
  $NewMap(q[i]) = p[i]$
end for
$Diff(Map, NewMap)$

Algorithm 15 Greedy Two-pass Bandwidth Map (GTBM)

Order the VM adjacency list by decreasing traffic intensity
Order the VNET daemon adjacency list by decreasing throughput
/* First pass */
while There is a pair of VMs neither of which has been mapped **do**
  Locate the first pair of communicating VMs such that neither of them have been mapped
  Map them to the first pair of hosts which currently have no VMs mapped onto them
end while
/* Second pass */
while There is an unmapped VMs **do**
  Locate a VM that have not been mapped
  Map the VM to a VNET daemon such that the throughput estimate between the VM and its already mapped counterpart is maximum.
end while
Compute the difference between the current mapping and the new mapping and issue VM migration instructions to achieve the new mapping.

Algorithm WBP tries to find for each pair of communicating VMs, the widest path defined in terms of the residual bottleneck bandwidth. We have adapted Dijkstra's shortest path algorithm [28] that now finds the widest path for an unsplittable network flow.

7.4.2 Adapted Dijkstra's algorithm

We use a modified version of Dijkstra's algorithm [28] to select a path for each 4-tuple that has the maximum bottleneck bandwidth. This is the "select widest" approach.

We adapt Dijkstra's algorithm for single source shortest path to find the maximum bottleneck bandwidth between each VNET daemon and to find for each 3-tuple $A(s_i, d_i, c_i)$, the widest path $p(i, j)$ with respect to the residual capacity.

Dijkstra's algorithm solves the single-source shortest paths problem on a weighted, directed graph $G = (H, E)$. We have created a modified Dijkstra's algorithm that solves the single-source widest paths problem on a weighted directed graph $G = (H, E)$ with a

Algorithm 16 Greedy Two-pass Bandwidth Map in pseudo code GTBM(VNET,VMD,VNETD,Map,RevereseMap,NewMap,NewReverseMap)

```
Quicksort(VMD, 1, length[VMD])
Quicksort(VNETD, 1, length[VNETD])
P ← VNET
while VMD ≠ ∅ do
  w ← ExtractMax(VMD)
  if ((NewReverseMap(w(i)) == ∅) and (NewReverseMap(w(j)) == ∅)) then
    while P ≠ ∅ do
      v ← ExtractMax(P)
      if ((NewMap(v(i)) == ∅)and(NewMap(v(j)) == ∅)) then
        NewReverseMap(w(i)) ← v(i)
        NewReverseMap(w(j)) ← v(j)
        NewMap(v(i)) ← w(i)
        NewMap(v(j)) ← w(j)
        break
      end if
    end while
  end if
end while
while VMD ≠ ∅ do
  w ← ExtractMax(VMD)
  if (NewReverseMap(w(i)) ≠ ∅) then
    NewReverseMap(w(j)) ← MAX(NewReverseMap(w(i)),VNET)
    NewMap(MAX(NewReverseMap(w(i)),VNET) ← w(j)
  else
    NewReverseMap(w(i)) ← MAX(NewReverseMap(w(j)),VNET)
    NewMap(MAX(NewReverseMap(w(j)),VNET) ← w(i)
  end if
end while
Diff(Map,NewMap)
```

Algorithm 17 Overlay hop reduction (OHR)

Order the set $\mathcal{A}$ of VM to VM communication demands in descending order of communication intensity (VTTIF traffic matrix entry)
while There are unmapped 4-tuple in $\mathcal{A}$ **do**
  Map it to a direct single hop path from the host that source VM is mapped to, to the host that destination VM is mapped on to
end while

Algorithm 18 Overlay hop reduction in pseudo code OHR(A,Path,NewPath,ReverseMap)

$Quicksort(A, 1, length[A])$
while $A \neq \emptyset$ **do**
  $w \leftarrow ExtractMax(A)$
  $NewPath(w(i), w(j)) \leftarrow (ReverseMap(w(i)), ReverseMap(w(i)))$
end while
$Diff(Path, NewPath)$

Algorithm 19 Shortest total latency path(STLP)

Order the set $\mathcal{A}$ of VM to VM communication demands in ascending order of latencies (VTTIF traffic matrix entry)
while There are unmapped 4-tuple in $\mathcal{A}$ **do**
  Map it to the shortest path (minimum path latency) from the host that source VM is mapped to, to the host that destination VM is mapped on to, using Dijkstra's shortest path algorithm
end while

Algorithm 20 Shortest total latency path in pseudo code STLP(G,L,A,Path,NewPath,ReverseMap)

$Quicksort(A, 1, length[A])$
while $A \neq \emptyset$ **do**
  $w \leftarrow ExtractMax(A)$
  $NewPath(w(i), w(j)) \leftarrow dijkstra(G, L, w(i))$
end while
$Diff(Path, NewPath)$

Algorithm 21 Widest bandwidth path (WBP)

Order the set $\mathcal{A}$ of VM to VM communication demands in descending order of communication intensity (VTTIF traffic matrix entry)
while There are unmapped 4-tuple in $\mathcal{A}$ **do**
  Map it to the widest path possible, using an adapted version of Dijkstra's algorithm described in Section 7.4.2
  Adjust residual capacities in the network adjacency list to reflect the mapping
end while

Algorithm 22 Widest bandwidth path in pseudo code WBP(G,B,A,Path,NewPath,ReverseMap)

$Quicksort(A, 1, length[A])$
while $A \neq \emptyset$ **do**
  $w \leftarrow ExtractMax(A)$
  $NewPath(w(i), w(j)) \leftarrow adapted\_dijkstra(G, B, w(i))$
end while
$Diff(Path, NewPath)$

weight function $c : E \to \mathbb{R}$ which is the available bandwidth in our case.

As in Dijkstra's algorithm we maintain a set U of vertices whose final widest-path weights from source u have already been determined. That is, for all vertices $v \in U$, we have $b[v] = \gamma(u,v)$, where $\gamma(u,v)$ is the widest path value from source u to vertex v. The algorithm repeatedly selects the vertex $w \in H - U$ with the largest widest-path estimate, inserts w into U and relaxes (we slightly modify the original Relax algorithm) all edges leaving w. Just as in the implementation of Dijkstra's algorithm, we maintain a priority queue Q that contains all the vertices in $H - U$, keyed by their b values. This implementation too assumes that graph G is represented by adjacency lists.

Similar to Dijkstra's algorithm we initialize the widest path estimates and the predecessors by the procedure described in Algorithm 23.

The modified process of relaxing an edge (w, v) consists of testing whether the bottle-

Algorithm 23 Initialize(G,u)

```
for each vertex v ∈ H[G] do
    b[v] ← 0
    π[v] ← NIL
end for
b[u] ← ∞
```

neck bandwidth decreases for a path from source u to vertex v by going through w, if it does, then we update $b[v]$ and $\pi[v]$. This procedure is described in Algorithm 24

Algorithm 24 ModifiedRelax(w,v,c)

```
if b[v] < min(b[w], c(w,v)) then
    b[v] ← min(b[w], c(w,v))
    π[v] ← w
end if
```

We can very easily see the correctness of ModifiedRelax. After relaxing an edge (w,v), we have $b[v] \geq min(b[w],c(w,v))$. As, if $b[v] \leq min(b[w],c(w,v))$, then we would set $b[v]$ to $min(b[w],c(w,v))$ and hence the invariant holds. Further, if $b[v] \geq min(b[w],c(w,v))$ initially, then we do nothing and the invariant still holds.

Algorithm 25 is the adapted version of Dijkstra's algorithm to find the widest path for a single tuple.

7.4.3 Correctness of adapted Dijkstra's algorithm

At first glance the correctness of the adapted Dijkstra's algorithm is not intuitive. Hence, we present a proof of correctness. Similar to the proof of correctness for Dijkstra's shortest paths algorithm, we can prove that the adapted Dijkstra's algorithm is correct by proving

Algorithm 25 AdaptedDijkstra(G,c,u)

$Initialize(G,u)$
$U \leftarrow \emptyset$
$Q \leftarrow H[G]$
while $Q \neq \emptyset$ **do** {loop invariant: $\forall\, v \in U, b(v) = \gamma(u,v)$}
  $w \leftarrow ExtractMax(Q)$
  $U \leftarrow U \cup w$
  for each $vertex\ v \in Adj[w]$ **do**
    $ModifiedRelax(w,v,c)$
  end for
end while

by induction on the size of set U that the invariant, $\forall v \in U$, $b[v] = \gamma(u,v)$, always holds.

Base case: Initially $U = \emptyset$ and the invariant is trivially true.

Inductive step: We assume the invariant to be true for $|U| = i$.

Proof: Assuming the truth of the invariant for $|U| = i$, we need to show that it holds for $|U| = i+1$ as well.

Let v be the $(i+1)^{th}$ vertex extracted from Q and placed in U and let p be the path from u to v with weight $b[v]$. Let w be the vertex just before v in p. Since only those paths to vertices in Q are considered that use vertices from U, $w \in U$ hence by the inductive step we have $b[w] = \gamma(u,w)$.

Next, we can prove that p is the widest path from u to v by contradiction. Let us assume that p is not the widest path and instead p^* is the widest path from u to v. Since this path connects a vertex in U to a vertex in $H-U$, there must be a first edge, $(x,y) \in p^*$ where $x \in U$ and $y \in H-U$. Hence the path p^* can now be represented as $p_1.(x,y).p_2$. By the inductive hypothesis $b[x] = \gamma(u,x)$ and since p^* is the widest path, it follows that $p_1.(x,y)$ must be the widest path from w to y, as if there had been a path with higher bottleneck bandwidth, that would have contradicted the optimality of p^*. When the edge x was placed in U, the edge (x,y) was relaxed and hence $b[y] = \gamma(u,y)$. Since v was the $(i+1)^{th}$ vertex chosen from Q while y was still in Q, it implies that $b[v] \geq b[y]$. Since we do not have any negative edge weights and $\gamma(s,v)$ is the bottleneck bandwidth on p^*, that combined with the previous expression gives us bottleneck bandwidth of $p^* \leq b[v]$ which is the bottleneck bandwidth of path p. This contradicts our first assumption that path p^* is wider than path p.

Since we have proved that the invariant holds for the base case and that the truth of the invariant for $|U| = i$ implies the truth of the invariant for $|U| = i+1$, we have proved the correctness of the adapted Dijkstra's algorithm using mathematical induction. □

7.4.4 Complexity of adapted Dijkstra's algorithm

Similar to Dijkstra, it can be shown that the running time of the adapted Dijkstra's algorithm is $O(H^2 + E)$. This bound can be reduced by a faster implementation of the data structures.

7.4.5 Algorithms NBP, WBLP and WRBLP

Algorithms NBP, WBLP and WRBLP, all use slightly different variants of the adapted Dijkstra's algorithm. We next describe these variations.

Variation on adapted Dijkstra for NBP

The adapted Dijkstra version used in NBP is slightly more complex than the Adapted Dijkstra's widest path algorithm. In the widest path algorithm as more nodes are discovered, the estimate of the width of a path from a source to destination is changed, if by going through an additional node actually increases the bottleneck width of the path. But, in the adapted Dijkstra version for NBP, the aim is to find the narrowest path from source to destination, such that the width of the path is greater, than the bandwidth demand made by those pairs of communicating nodes. So now when we walk the path discovering nodes, the estimate of the width of a path from a source to destination is changed, if by going through an additional node actually decreases the width of the path, as long as this width is greater than the demand placed by those pairs of communicating nodes. Further, we also constantly check if we have created a loop in the path. Another time when we change the width estimate is if the current estimate is less than the bandwidth demand and going through a node actually increases the width beyond the bandwidth demand. Again we also constantly check to see if we have created any loops.

Variation on adapted Dijkstra for WBLP and WRBLP

The variation on the adapted Dijkstra for WBLP and WRBLP is identical. Adapted Dijkstra finds the widest path from a single source to all the destinations. The width of a link is defined as its bandwidth. This variation on the adapted Dijkstra is identical in all but one respects to adapted Dijkstra's algorithm. The width of a link, instead of being bandwidth, is now the function $(bandwidth + c/latency)$.

Algorithm 26 Narrowest bandwidth path (NBP)

Order the set $\mathcal{A}$ of VM to VM communication demands in descending order of communication intensity (VTTIF traffic matrix entry)
while There are unmapped 4-tuple in $\mathcal{A}$ **do**
  Map it to the narrowest path possible, using a variant of the adapted version of Dijkstra's algorithm, the algorithm now finds the narrowest path instead of the widest path
  Adjust residual capacities in the network adjacency list to reflect the mapping
end while

Algorithm 27 Narrowest bandwidth path in pseudo code NBP(G,B,A,Path,NewPath,ReverseMap)

$Quicksort(A, 1, length[A])$
while $A \neq \emptyset$ **do**
  $w \leftarrow ExtractMax(A)$
  $NewPath(w(i), w(j)) \leftarrow adapted\_dijkstra\_variant\_NBP(G, B, w(i))$
end while
$Diff(Path, NewPath)$

Algorithm 28 Widest bandwidth latency path (WBLP)

Order the set $\mathcal{A}$ of VM to VM communication demands in descending order of communication intensity (VTTIF traffic matrix entry)
while There are unmapped 4-tuple in $\mathcal{A}$ **do**
  Map it to the widest path possible, using a variant of the adapted version of Dijkstra's algorithm, the width of path is now defined as a function of both bandwidth and latency (bandwidth + c/latency)
end while

Algorithm 29 Widest bandwidth latency path in pseudo code WBLP(G,BL,A,Path,NewPath,ReverseMap)

$Quicksort(A, 1, length[A])$
while $A \neq \emptyset$ **do**
  $w \leftarrow ExtractMax(A)$
  $NewPath(w(i), w(j)) \leftarrow adapted\_dijkstra\_variant\_WBLP(G, BL, w(i))$
end while
$Diff(Path, NewPath)$

Algorithm 30 Widest residual bandwidth latency path (WRBLP)

Order the set $\mathcal{A}$ of VM to VM communication demands in descending order of communication intensity (VTTIF traffic matrix entry)
while There are unmapped 4-tuple in $\mathcal{A}$ **do**
  Map it to the widest path possible, using a variant of the adapted version of Dijkstra's algorithm, the width of path is now defined as a function of both bandwidth and latency (bandwidth + c/latency)
  Adjust residual capacities in the network adjacency list to reflect the mapping
end while

Algorithm 31 Widest residual bandwidth latency path in pseudo code WR-BLP(G,BL,A,Path,NewPath,ReverseMap)

$Quicksort(A, 1, length[A])$
while $A \neq \emptyset$ **do**
  $w \leftarrow ExtractMax(A)$
  $NewPath(w(i), w(j)) \leftarrow adapted\_dijkstra\_variant\_WBLP(G, BL, w(i))$
  $Modify(BL, w(k), w(l))$
end while
$Diff(Path, NewPath)$

7.5 Combinations of mapping and routing algorithms

Since our adaptation problem has a mapping and routing component, we came up with fifteen different adaptation schemes, each with a different combination of mapping and routing algorithms. Table 7.4 lists fifteen adaptation schemes. For each scheme we list the mapping algorithm, routing algorithm and the objective functions targetted.

Since we are combining our mapping algorithms with the routing algorithm and because we have 8 mapping algorithms and 6 routing algorithms, the natural question to ask is why were 48 adaptation schemes not studied? Why only fifteen? The reason these 15 fifteen adaptation schemes were chosen were that these alone made sense. For example, one theoretically possible combination could have been GOBM + STLP. If we had chosen this algorithm than what we would be doing would be migrating VMs to hosts that are connected via high bandwidth links and then trying to find low latency paths among them. This is not meaningful for either locating high bandwidth paths, nor is it suitable for locating low latency paths and nor is it suitable for locating high bandwidth and low latency paths. Hence we ignored considering such possibilities.

The computational complexity of the adaptation schemes listed in Table 7.4 is presented in Table 7.5. The running times of the algorithms can be improved by using better data structures.

Number	Name	Targetted objective function
0.	BF	Brute force optimal
1.	OHR	-
2.	CLB	MEETI
3.	CLB + OHR	MEETI
4.	GOBM + OHR	-
5.	GOBM + WBP + S	MSRBB
6.	GOELM + STLP	MSPL
7.	GOBM + NBP	NSRBB
8.	GOBLM + WBLP	MSBBL
9.	GOBBM + WBP	MSRBB
10.	GOBBLM + WRBLP	MSRBBL
11.	GOPLM + STLP	MSPL
12.	CLB + WBLP	MEETI + MSBBL
13.	GOBM + WBP + L	MSRBB
14.	GTBM + WBP + S	MSRBB
15.	GTBM + WBP + I	MSRBB

Table 7.4: Fifteen different adaptation schemes.

Number	Name	complexity
0.	BF	-
1.	OHR	$O(LlgL)$
2.	CLB	$O(NlgN+HN)$
3.	CLB + OHR	$O(HN+LlgL)$
4.	GOBM + OHR	$O(EL+ElgE)$
5.	GOBM + WBP + S	$O(ElgE+H^2L)$
6.	GOELM + STLP	$O(EL+H^2)$
7.	GOBM + NBP	$O(ElgE+H^2L)$
8.	GOBLM + WBLP	$O(ElgE+H^2L)$
9.	GOBBM + WBP	$O(ElgE+H^2L)$
10.	GOBBLM + WRBLP	$O(ElgE+H^2L)$
11.	GOPLM + STLP	$O(EL+H^2L)$
12.	CLB + WBLP	$O(ElgE+H^2L)$
13.	GOBM + WBP + L	$O(ElgE+H^2L)$
14.	GTBM + WBP + S	$O(ElgE+H^2L)$
15.	GTBM + WBP + I	$O(ElgE+H^2L)$

Table 7.5: Computational complexity of the fifteen adaptation schemes. The numbers (H,E,N,L) refer to the number of VNET hosts, number of edges in VNET graph, number of VMs and number of edges in VM graph, respectively.

7.6 Evaluation

Here, we are interested in evaluating two different aspects of the adaptation schemes. First, in Section 7.6.2 we wish to compare the adaptation schemes against each other and against estimates of the optimal. All evaluation in this chapter is in the context of the objective functions. We are trying to answer the question: which adaptation scheme works the best for a wide range of objective functions and adaptation scenarios? This will give us an indication of which adaptation scheme might be effective for a range of distributed applications, where effectiveness is measured in terms of application defined metrics.

Second, in Section 7.6.3, we want to study the time taken for the different adaptation schemes to execute under different scenarios. Table 7.5 lists the complexity of the adaptation schemes. However, beyond this, we also want to get a sense of the empirical executions of the adaptation schemes. This will indicate the feasibility and practicality of the solution (and specific implementation). The reason we have directed our attention to heuristics is since for reasonably large adaptation scenarios, generating optimal solutions is an intractable problem. Hence, the heuristic solutions must not suffer from the same failings.

7.6.1 Generating random adaptation scenarios

In this chapter, we use synthetically constructed adaptation scenarios. The reason is that we would like to study the behavior of the adaptation schemes for many different cases spanning from small setups to large scenarios. It is challenging to get real applications running on a large scale in a distributed test bed that is not completely under our control. Further, though constructing synthetic traces for large scenarios is possible, validating and verifying the can get to be a challenging task. Hence, to study the effectiveness of the adaptation schemes in a wide range of adaptation scenarios, we randomly generate them.

The random adaptation scenario generator is modeled on our experience with real applications and measurements of physical systems. It consists of a physical system measurements generator (to mimic Wren and Ganglia [124]/RPS's [38] output), an application demand generator (to mimic VTTIF output coupled with computational demands) and an initial mapping generator (to mimic Virtuoso's default mapping and routing through the Proxy).

The physical system measurements generator generated CPU speed measurements between 1.2 and 3.6 with a normal distribution with a mean at 2.6. It generated utilizations with a normal distribution with a mean at 50%. The latency estimates between hosts were generate were generated to be between 0.00004 and 0.01 seconds with a uniform distribution. The bandwidth estimates between the hosts were generated to between 200 KB/sec and 21 MB/sec with an uniform distribution. The application demand generator generated compute demands in the range of 2 and 21 Giga operations with a normal distribution. The latency demands were in the range 0.00009 and 0.05 seconds with a normal distribution and the bandwidth demands were in the range 100 KB/sec and 11 MB/sec, also with an uniform distribution. We assumed a moderately fast migration scheme that on an average took 300 seconds to migrate a 1 GB VM [153]. This is a reasonable assumption based on the recent progresses made on virtual machine migration schemes [107, 134, 137, 146]. In all the scenarios described in this chapter, application communication topology is all-to-all with both compute and network demands. Further, the physical network is also assumed to be completely connected with all paths through it possible. We also make the assumption that we have complete information with regard to physical resource measurements.

However, it should be carefully noted that we still need to close the loop by carrying out evaluations with real applications, real application traces and synthetic, verified and validated, application traces. This is the topic of discussion in Chapter 8.

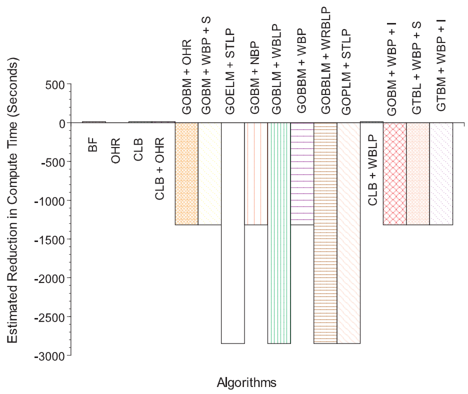
7.6.2 Comparison of adaptation schemes

As mentioned above, we randomly constructed adaptation scenarios involving 4 VMs 8 Hosts, 8 VMs 16 Hosts, 16 VMs 32 Hosts, 32 VMs 64 Hosts, 64 VMs 128 Hosts and 128 VMs 256 Hosts. In this section we present results for these scenarios. We also constructed and studied larger scenarios up to 1024 VMs and 2048 Hosts.

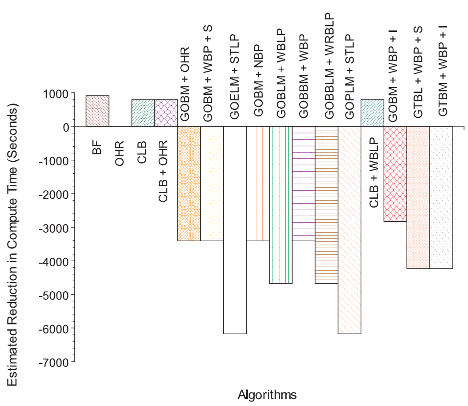
It should be noted that Virtuoso's adaptation engine is centralized in nature. Though this provides the most accurate global system and hence allows for the most effective system, it is important to understand the subtleties involved in a centralized system. For very large scenarios, even an efficient adaptation algorithm can take long periods to complete. Then the system is faced with the question: should the adaptation scheme be run? A decentralized adaptation scheme that localizes many of the adaptation decisions makes adaptation possible for very large scales (thousands of nodes), however, the potency of the adaptation scheme reduces with the loss of centralized and global control.

MEETI: Maximize Sum of Estimated Execution Time Improvement

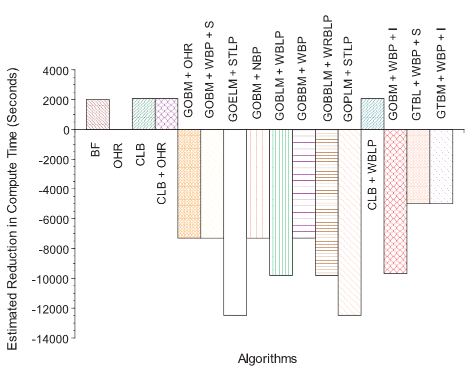
Figure 7.1 shows the performance of all the adaptation schemes as measured against the MEETI objective function. For smaller scenarios up to 32 hosts, we were also able to calculate an estimate of the optimal. We did this using a brute force approach. We kept generating random configurations and always kept track of the best value so far for the MEETI objective function. We ran the optimal solution estimator in parallel on 10 different machines on our cluster for 72 hours. It should be noted that these are simply estimates of the optimal, due to the explosion of possibilities as the adaptation scenarios grow, the estimates of the optimal might be sub-optimal themselves and in certain cases lower than what the greedy heuristic driven algorithms can achieve. From the figures we notice that the best performing adaptation schemes find the optimal solution. We notice CLB, CLB +



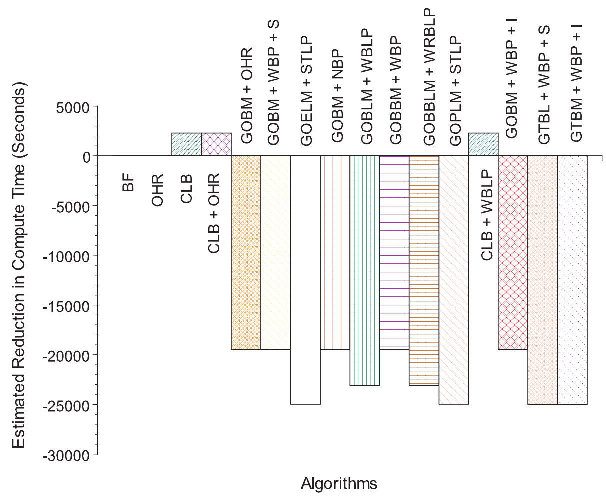
(a) 4 VMs 8 Hosts



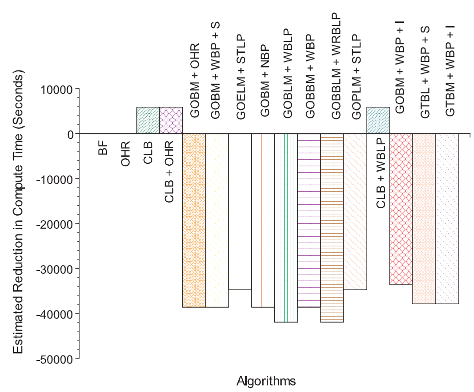
(b) 8 VMs 16 Hosts



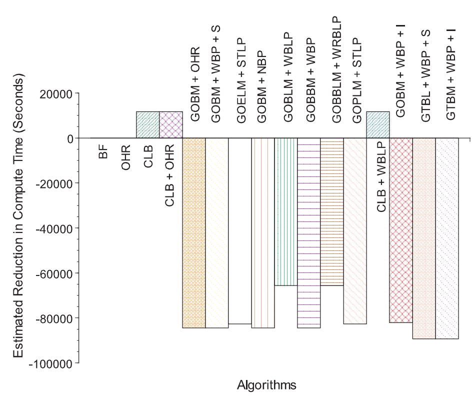
(c) 16 VMs 32 Hosts



(d) 32 VMs 64 Hosts



(e) 64 VMs 128 Hosts



(f) 128 VMs 256 Hosts

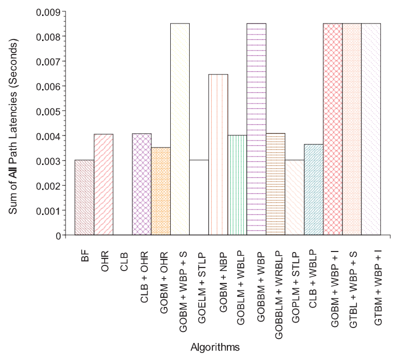
Figure 7.1: Comparison of adaptation schemes for different scenarios, in the context of MEETI objective function. Higher is better.

OHR and CLB + WBLP are the adaptation schemes that perform consistently well. This should not come as a surprise since the MEETI objective function is all about reducing the computational time and CLB, CLB + OHR and CLB + WBLP share this objective. We note that OHR has a zero value for MEETI objective function as it does not have a mapping component and hence no VMs get migrated.

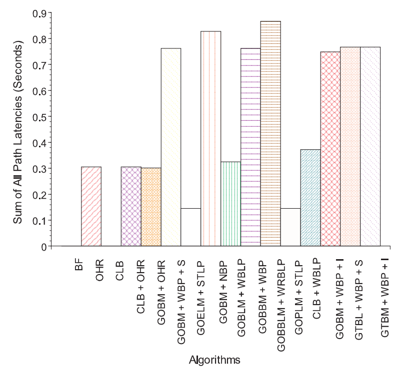
Another interesting observation is that when performance is measured in terms of the MEETI objective function, it is very easy for a network centric adaptation scheme to do the "wrong" thing, migrating VMs to hosts resulting in an increase in computation time. The values of MEETI that are less than zero indicate the total increase in computation time overall migrated VMs. Since computation is central to many distributed applications, a generally applicable adaptation scheme must take it into consideration.

MSPL: Minimize Sum of Path Latencies

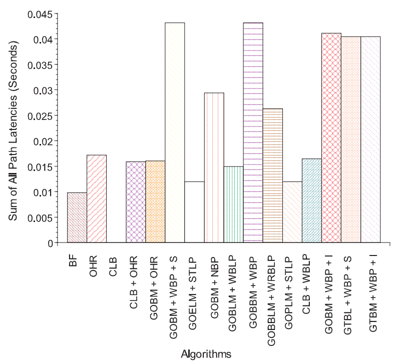
Figure 7.2 shows the performance of all the adaptation schemes as measured against the MSPL objective function. We notice that OHR, CLB + OHR, GOBBM + OHR perform reasonably well in most cases. GOELM + STLP and GOPLM + STLP perform the best as they are specifically targeted to scenarios where minimizing latency is of prime importance. These achieve values very close to the optimal. We also notice CLB + WBLP to perform well, though sub-optimal. Notice that for the 16 VM 32 Host scenario, the MSPL value generated by the optimal estimator is worse than what the best adaptation schemes can achieve. This is due to the fact the optimal generator is simply an estimator and in this case it did not find the optimal. Adaptation schemes which have a bandwidth bias do not do that well, with the exception of CLB + WBLP, which has both, a latency and a bandwidth component.



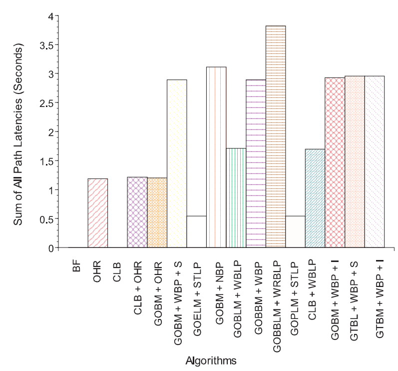
(a) 4 VMs 8 Hosts



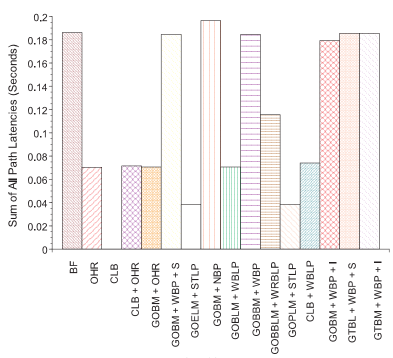
(d) 32 VMs 64 Hosts



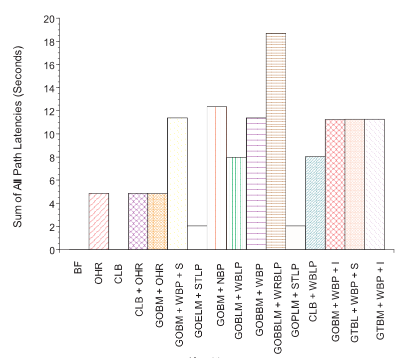
(b) 8 VMs 16 Hosts



(e) 64 VMs 128 Hosts

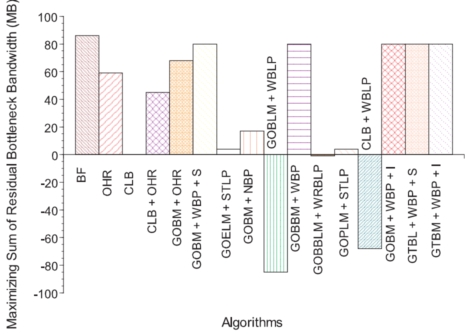


(c) 16 VMs 32 Hosts

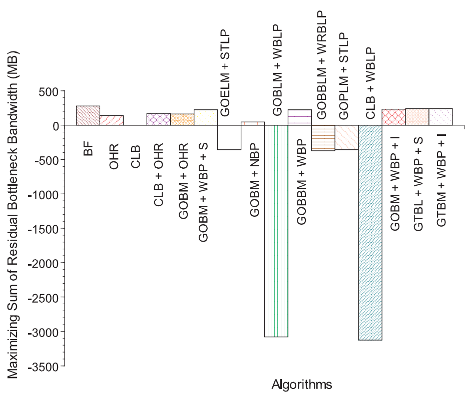


(f) 128 VMs 256 Hosts

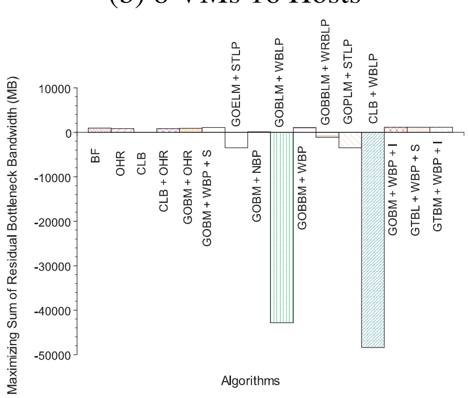
Figure 7.2: Comparison of adaptation schemes for different scenarios, in the context of MSPL objective function. Lower is better.



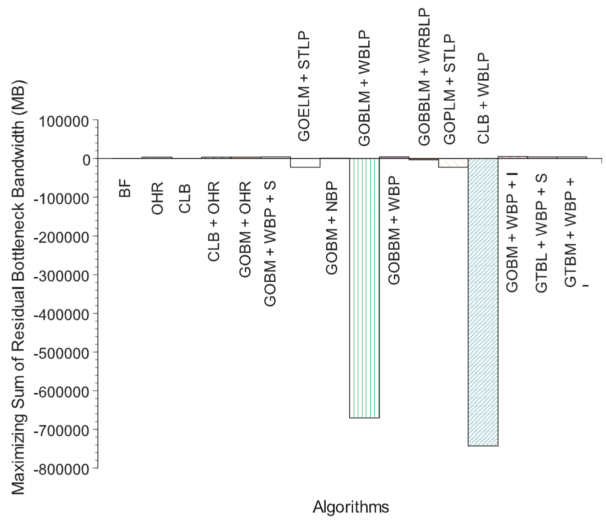
(a) 4 VMs 8 Hosts



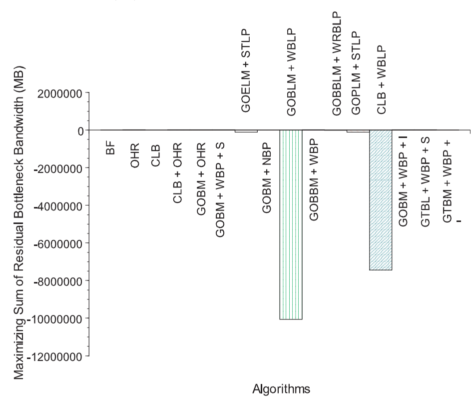
(b) 8 VMs 16 Hosts



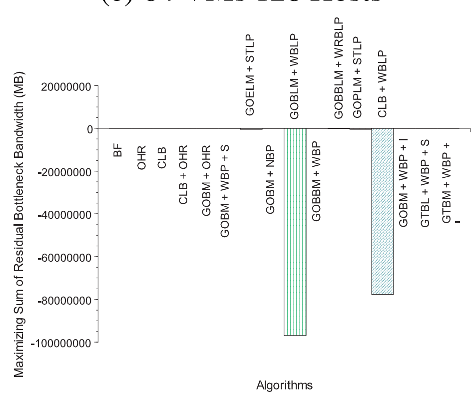
(c) 16 VMs 32 Hosts



(d) 32 VMs 64 Hosts



(e) 64 VMs 128 Hosts



(f) 128 VMs 256 Hosts

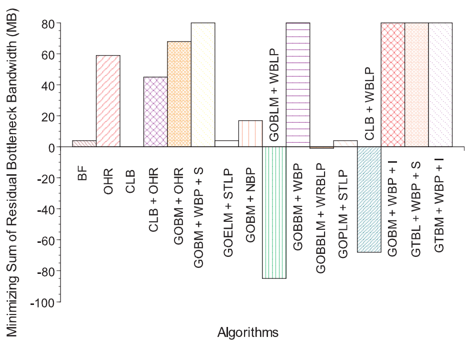
Figure 7.3: Comparison of adaptation schemes for different scenarios, in the context of MSRBB objective function. Higher is better.

MSRBB: Maximize Sum of Residual Bottleneck Bandwidths

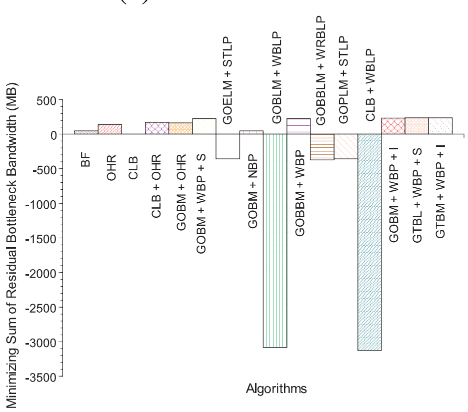
Figure 7.3 shows the performance of all the adaptation schemes as measured against the MSRBB objective function. This objective function has a bias towards adaptation schemes that optimize bandwidth. MSRBB is very receptive to GOBBM + WBP + S, GOBBM + WBP + I, GTBM + WBP + S and GTBM + WBP + I. These algorithms have a significant bandwidth component. The objective function values these algorithms achieved were very close to those generated by optimal solution estimator. Indicating that these adaptation schemes, though sub-optimal at times, work well in practice. We were surprised to notice the poor performance of CLB + WBLP. CLB + WBLP optimizes for both bandwidth and latency. Our explanation is that for the particular value of the latency constant, the adaptation scheme chose a low bandwidth, low latency path, thus resulting in a lower value for the MSRBB function. It should be noted that algorithms which generate a negative value for this objective function will not be always able to find a mapping and a routing.

NSRBB: Minimize Sum of Residual Bottleneck Bandwidths

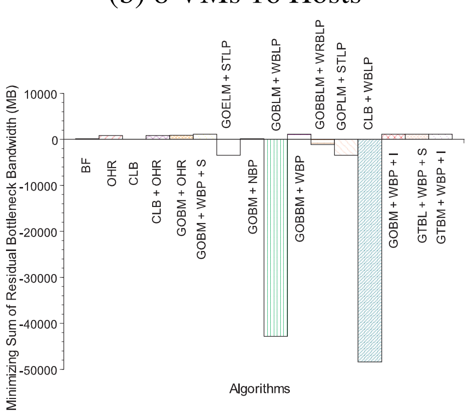
We present the performance of the adaptation schemes as measured against the NSRBB objective function in Figure 7.4. The GOBM + NBP adaptation scheme performs the best in this context. This adaptation scheme was specifically designed to study the effect of minimizing the sum of the residual bottleneck bandwidths. We also notice that some of the schemes that attempt to maximize it also perform reasonably well. The values achieved by GOBM + NBP are very close to those generated by the optimal estimator. Again, we notice that CLB + WBLP suffers from choosing paths which a lower latency, but also a lower bandwidth.



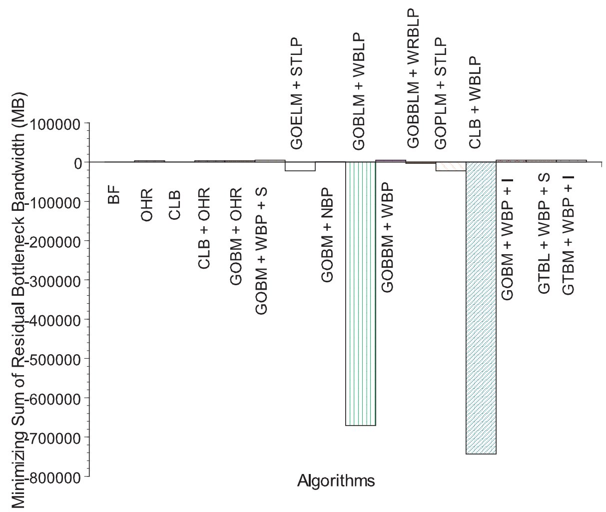
(a) 4 VMs 8 Hosts



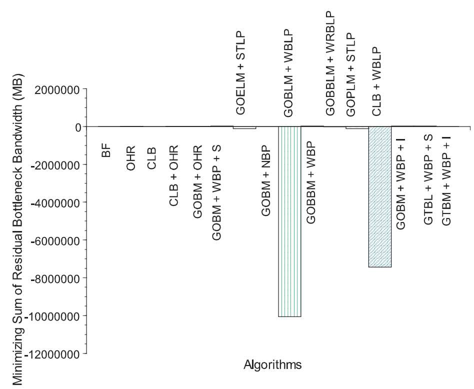
(b) 8 VMs 16 Hosts



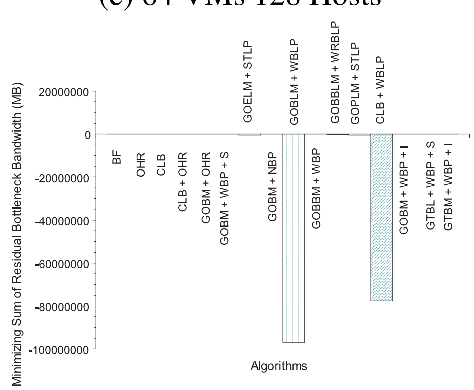
(c) 16 VMs 32 Hosts



(d) 32 VMs 64 Hosts

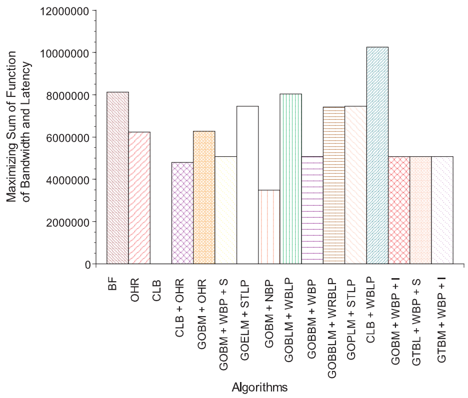


(e) 64 VMs 128 Hosts

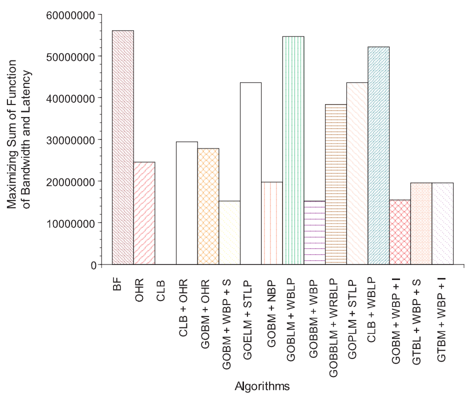


(f) 128 VMs 256 Hosts

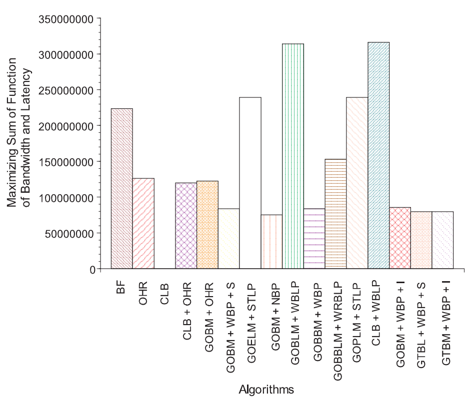
Figure 7.4: Comparison of adaptation schemes for different scenarios, in the context of NSRBB objective function. Lower is better.



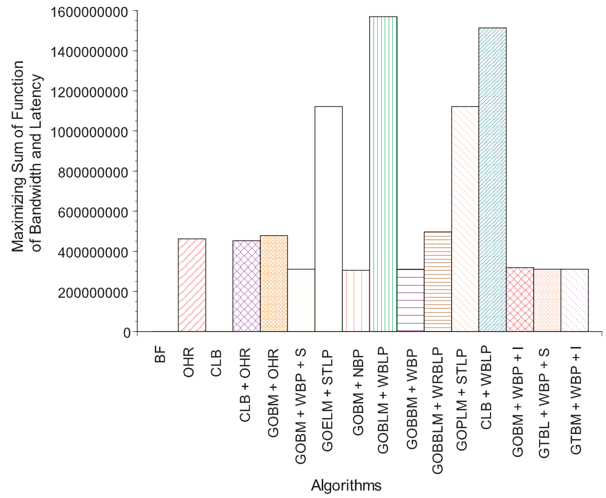
(a) 4 VMs 8 Hosts



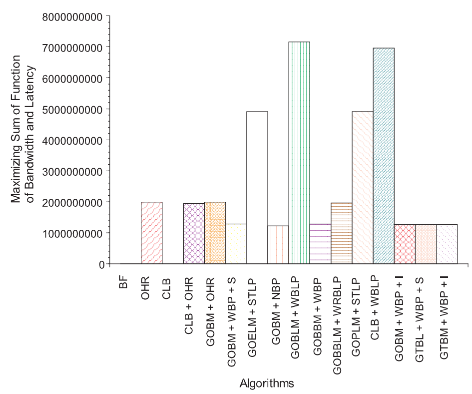
(b) 8 VMs 16 Hosts



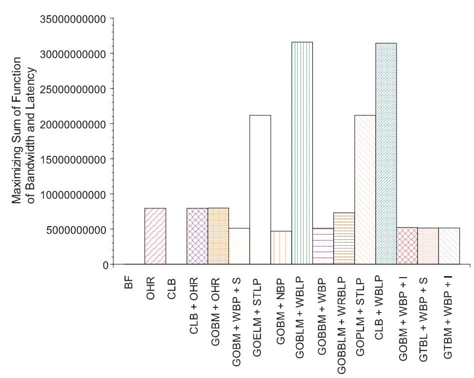
(c) 16 VMs 32 Hosts



(d) 32 VMs 64 Hosts



(e) 64 VMs 128 Hosts



(f) 128 VMs 256 Hosts

Figure 7.5: Comparison of adaptation schemes for different scenarios, in the context of MSBBL objective function. Higher is better.

MSBBL: Maximize Sum of Bottleneck Bandwidth and Latency

Figure 7.5 shows the performance of all the adaptation schemes as measured against the MSBBL objective function. This objective function tests the effectiveness of adaptation schemes that attempt to optimize both, latency and bandwidth. The two adaptation schemes that consistently perform well are GOBLM + WBLP and CLB + WBLP. Both these algorithms optimize both, latency and bandwidth, based on a tunable constant, which is used to assign weights to latency and bandwidth measures. Additionally, we also notice, that latency centric algorithms perform reasonably well. The values achieved by GOBLM + WBLP and CLB + WBLP are very close to those calculated optimal solution estimator.

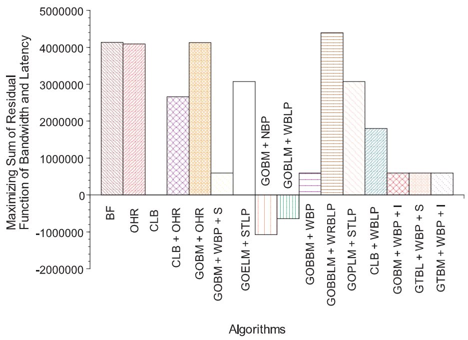
MSRBBL: Maximize Sum of Residual Bottleneck Bandwidth and Latency

We present the performance of the adaptation schemes as measured against the MSRBBL objective function in Figure 7.6. The MSRBBL objective function taken into account, both, latency and bandwidth. Further, it also takes into account the latency and bandwidth demands made by the application. We find the GOBBLM + WRBLP adaptation scheme to perform the best. Notice that algorithms which generate a negative value for this objective function will not be always able to find a mapping and a routing.

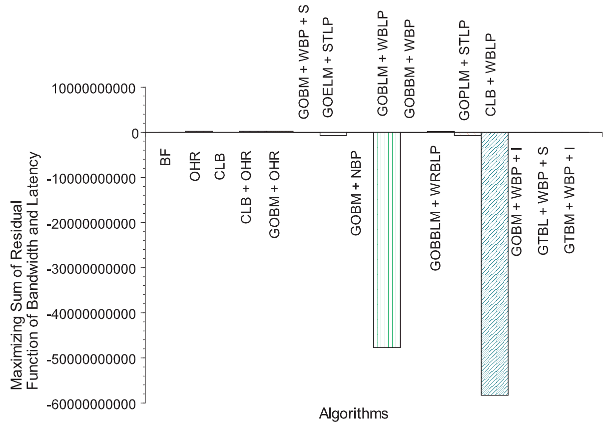
7.6.3 Execution time of adaptation schemes

The main motivation for exploring the space of heuristic algorithms was that even for small scenarios, optimal solution generation is an intractable problem. Hence, it is important to study the execution times of these proposed heuristics to ensure that for reasonably large adaptation scenarios, they complete in an acceptable amount of time. We have already presented the complexities of all the adaptation schemes in Table 7.5.

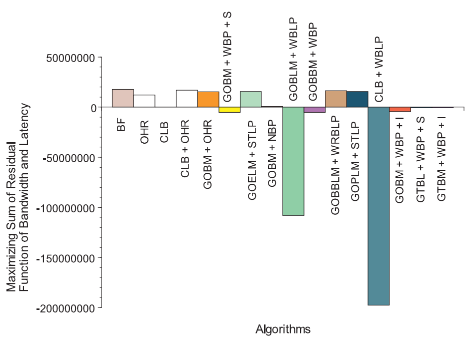
Figure 7.7 shows the algorithm completion times for different scenarios from 4 VMs and 8 physical hosts to 32 VMs and 64 physical hosts. These experiments were conducted



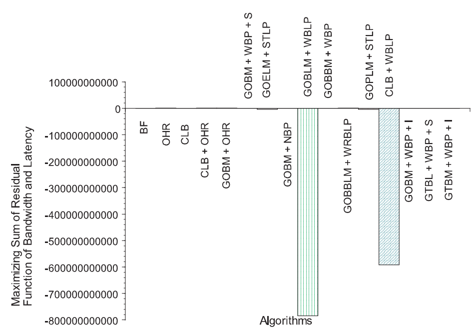
(a) 4 VMs 8 Hosts



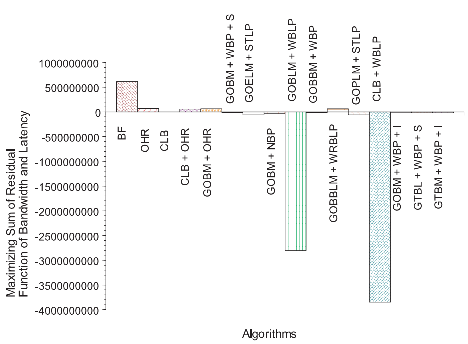
(d) 32 VMs 64 Hosts



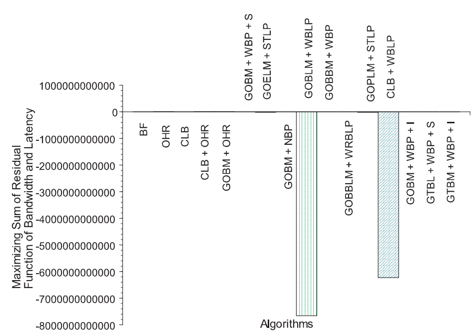
(b) 8 VMs 16 Hosts



(e) 64 VMs 128 Hosts

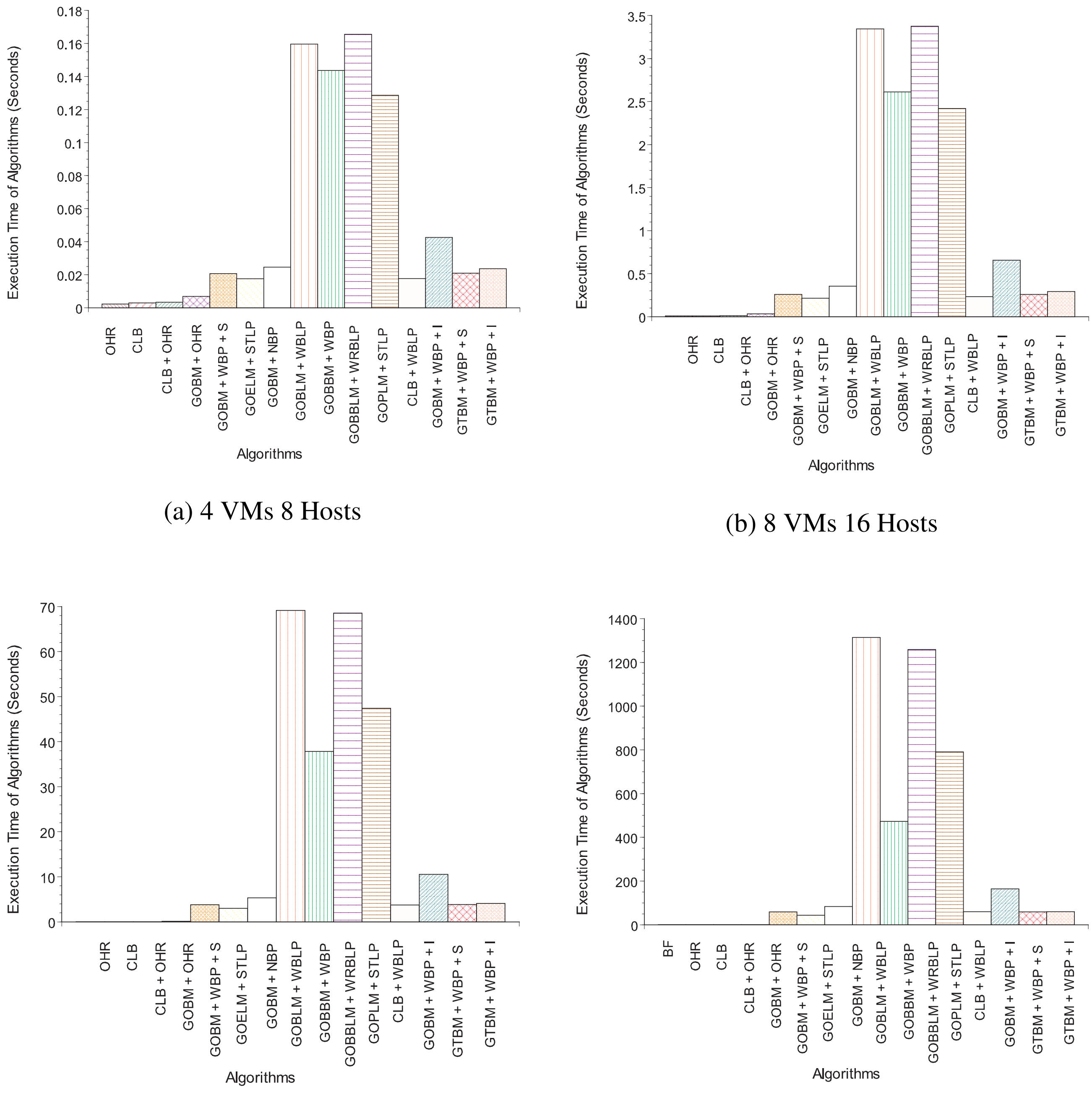


(c) 16 VMs 32 Hosts



(f) 128 VMs 256 Hosts

Figure 7.6: Comparison of adaptation schemes for different scenarios, in the context of MSRBBL objective function. Higher is better.



(a) 4 VMs 8 Hosts

(b) 8 VMs 16 Hosts

(c) 16 VMs 32 Hosts

(d) 32 VMs 64 Hosts

Figure 7.7: Execution time of different adaptation schemes for different scenarios.

on one node of our IBM e1350 cluster, whose nodes are dual 2.0 GHz Intel® Xeon® with 1.5 GB RAM running Red Hat Linux 9.0 The algorithms themselves were implemented in Perl 5.8.

We notice that, even for the largest scenarios, almost all the algorithms complete in less than one minute. GOBLM + WBLP, GOBBM + WBP, GOBBLM + WRBLP and GOPLM + STLP take significantly longer to complete than their counterparts. Even among these GOBLM + WBLP and GOBBLM + WRBLP take over 20 minutes to complete. These two algorithms have the same complexity as some of the others, yet they take longer to complete. It is my opinion that this is an artifact of our specific implementation of these algorithms

It should be noted that the change that these adaptation schemes are reacting to must persist for at least 20 minutes to make this adaptation meaningful. These algorithm completion times can be significantly reduced by using faster implementations of data structures. Additionally, it should be noted that these algorithms have been implemented in Perl. By moving this implementation to C++ and optimizing it, we expect to achieve significant performance improvements.

But the key point is that even by very conservative standards (Perl, unoptimized, code, etc.) for a reasonably large Virtuoso setup, the adaptation scheme completion times are on the order of a few seconds.

7.7 Conclusions

Building upon the inapproximability results of Chapter 4, we explored the space of heuristically driven adaptation algorithms. We analyzed some distributed applications ranging from high performance distributed applications to enterprise IT applications to web e-commerce transactional applications. In particular we studied the resource demands made by these application classes. Based on the information gathered, we devised six different objective functions to cover a wide range of distributed application classes. Each objective function represents a metric important for some application is some scenario. The natural question that then arises is, is there a single adaptation scheme that works well for a range of applications? This question also translates to, is there a single adaptation scheme that works well for a range of objective functions. This second question was the center of our evaluation in this chapter.

Since the adaptation problem consists of both, a mapping and a routing component, we designed and implemented eight different greedy mapping algorithms and six different routing algorithms. We then combined these in different combinations to form fifteen different adaptation schemes.

We presented a detailed evaluation comparing the fifteen adaptation schemes among themselves and to an estimate of the optimal (for small input sizes). Different adaptation algorithms target different objective functions. However, we found that on the whole CLB + WBLP adaptation scheme was most widely applicable. Further, even for cases where its performance was sub-par, it could easily be improved by changing the values of its tunable parameters. However, it remains to be seen if this adaptation scheme has wide applicability in the context of real application traces where the goodness measure is in application defined terms (and not in terms of objective functions). This is the focus of Chapter 8.

Based on the evaluations performed in this chapter the adaptation scheme CLB + WBLP, because it is fast (empirically measured), scales (linearithmic scaling) and covers the widest range of objective functions.

Chapter 8

Experimentation and simulation

This dissertation studies automatic, run-time and dynamic adaptation of distributed applications executing in virtual environments. This scheme, in the most part (see Chapter 6), requires no interaction with the application, developer or user, and works on un-modified applications executing on un-modified operating systems. This is achieved via application independent adaptation schemes, application demand inference and physical resource measurement schemes, that are all invisible to the application.

In response to the complexity and inapproximability of the adaptation problem, we have explored the space of greedy heuristic driven adaptation algorithms. In particular we have devised fifteen different adaptation schemes. Each of these schemes is applicable to a set of distributed application classes in different scenarios.

At the highest level, this dissertation answers the question, does there exist a single adaptation scheme that is applicable for a wide range of distributed applications? The reason this question is important is that in the past there have been numerous attempts at performing application dependent or application driven adaptation that expect active participation from the application/developer/user [13, 103, 174, 188]. Despite all these efforts adaptation mechanisms and control are not common in today's applications. This is in part due to the complexity of such approaches. When computation and communication is spread over wide-area networks, complex adaptation needs to be manually performed by the developer or user. The reason for this is that resource availability varies considerably over the wide-area. Our approach to this problem is to hide the complexity from the application/developer/user and maintain the simple local area abstraction that users are familiar and comfortable with today. A natural culmination of this effort is an answer to this thesis question.

8.1 Evaluation methodology

In this chapter we leverage both, physical experiments and simulations, to answer the question posed above. In Chapter 2 we stated the different application classes studied in the course of this dissertation (Table 2.1). We divided these applications into two categories. The first category, consisting of the first three application classes listed in Table 2.1, were used in designing and implementing the objective functions and the fifteen adaptation schemes. We closed the loop and tested the effectiveness of all the adaptation schemes developed against each of the remaining seven application classes.

For two of the three application classes used to develop the adaptation schemes, we present some of the results that led us to the design of these fifteen adaptation schemes. These evaluations were conducted on our physical real-world distributed testbed. The "closing the loop" evaluations were conducted in simulation using a verified and validated virtualized system simulator.

Our evaluation is divided into the following three categories:

- **System overheads:** We carried out a series of experiments to understand the overheads of the system in general and in particular the overheads associated with the application independent adaptation schemes.

- **Experimentation results:** We conducted a series of adaptation experiments using two different classes of applications. These experiments were conducted on our physical wide-ares testbed to better understand automatic, run-time and dynamic adaptation. We used the knowledge gained in the course of these experiments to design the fifteen adaptation schemes.

- **Simulation results:** We finally closed the loop and evaluated the fifteen adaptation schemes against the remainder application classes that were not used in the design and development of the algorithms. Due to practical problems in getting these diverse application classes to execute in our wide-area testbed, we conducted this part of the evaluation in simulation using a verified and validated simulator.

Section 8.2 details the overheads of our adaptation system, in particular the application independent adaptation schemes. We detail, in Section 8.3, evaluation results obtained via physical experimentation for two classes of distributed applications and how these results provided insights that led to the development of the adaptation schemes. Since our "closing the loop" evaluations were conducted in simulation, we introduce our independently verified and validated virtualized system simulator in Section 8.5. The evaluation of the fifteen adaptation schemes against diverse application classes is described in Section 8.6. We conclude the chapter in Section 8.7 by summarizing our evaluation results.

8.2 System overheads

We carried out a series of experiments to understand the overheads of our adaptive system. In particular we studied the following:

- Overheads of the migration system
- Overheads of the network reservation system (Results discussed in Chapter 5 (Figure 5.5))

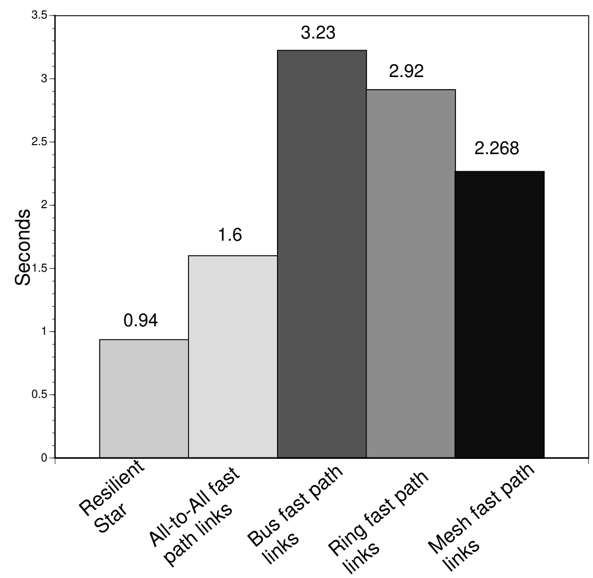


Figure 8.1: Time to set up the backbone star configuration and to add fast path links for different (inferred) topologies.

- Overheads of the CPU reservation system (Results discussed in Chapter 6).
- Overheads of the non-reservation adaptation mechanisms (Results discussed in Chapter 5 (Figure 5.5).
- Configuration times for setting up some common topologies

In this section we discuss the overheads of our migration system (Section 8.2.2) and the configuration times for setting up some common topologies representing application communication behaviors (Section 8.2.1).

8.2.1 Configuration times for setting up some common topologies

We present some more results for the configuration times of our adaptive system.

Figure 8.1 shows the time required to create different VNET topologies among eight VNET daemons each hosting a single VM. Here, all the hosts are in single cluster (our

IBM e1350 cluster described in Section 8.3). The Proxy and the user are located on a network separated by a metropolitan area network (MAN). This setup helps emphasize overheads and eliminate other factors such as wide area latency, etc.

It takes 0.94 seconds to create the resilient star topology among the VNET daemons, including time to add the links and populate the forwarding tables. It takes a further 1.6 seconds to add all the fast path links and corresponding forwarding rules for an all-to-all topology. Adding fast path links for a bus topology takes longer (3.23 seconds), even though there are fewer links. This is because VNET does not use hierarchical routing. Since VNET operates at the link layer, virtual machine migration would punch holes in hierarchical routing tables. Hence, VNET forwards packets based on a source and destination address match rather than just the destination address match, which leads to an increase in the number of forwarding rules for some topologies such as the bus topology.

8.2.2 Overheads of the migration system

Virtuoso allows us to migrate a VM from one physical host to another. Though our migration scheme is not the fastest, much work exists that demonstrates that fast migration of VMs running commodity applications and operating systems is possible [107, 134, 137, 146], including live migration schemes with downtime on the order of a few seconds [25]. As migration times decrease, the rate of adaptation we can support and my work's relevance increases.

8.3 Physical experimentation results

Out of the ten application classes studied in the course of this research, we used the first three (High performance computing applications, transactional web e-commerce applications and IT backup) to design our fifteen adaptation schemes. We studied the first two of

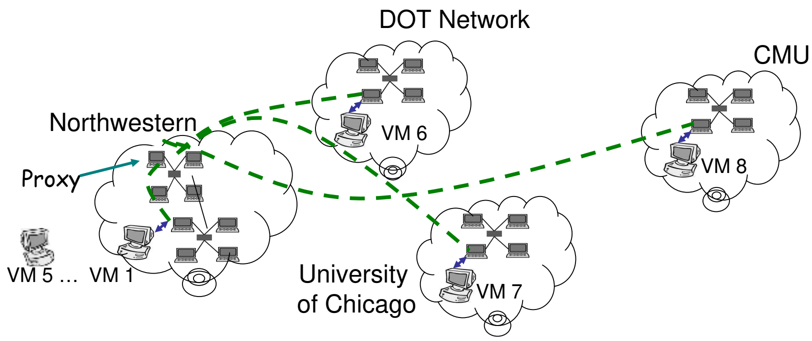


Figure 8.2: Wide area testbed.

these extensively, carrying out numerous evaluations on our physical wide-area testbed.

Since, these three applications classes were used as an aid to designing the adaptation schemes, it is not very meaningful to extensively evaluate the performance of these applications in the context of all the fifteen adaptation schemes. However, in this section we do discuss the physical experiments conducted in the context of the first two application classes (high performance computing (Section 8.3.1) and web e-commerce transactional applications (Section 8.3.2)).

Our experimental testbed, as illustrated in Figure 8.2, is spread across four sites. At Northwestern University, IL, we have two clusters and some additional machines spread across the campus network. The first cluster is a IBM e1350 cluster, whose nodes are dual 2.0 GHz Intel® Xeon® with 1.5 GB RAM running Red Hat Linux 9.0 and VMware GSX Server 2.5, connected by a 100 Mbit switched network. The second is a slightly slower cluster, whose nodes are dual 1 GHz (Intel® Pentium® III with 1 GB RAM running Red Hat 7.3 and VMware GSX Server 2.5, connected by a 10 Mbit switched network. These two clusters are inter-connected via two firewalls and a campus network. Performance diverse machines at Carnegie Mellon University (CMU), PA, University of Chicago, IL and the DOT research network [40] make up the remainder of the testbed.

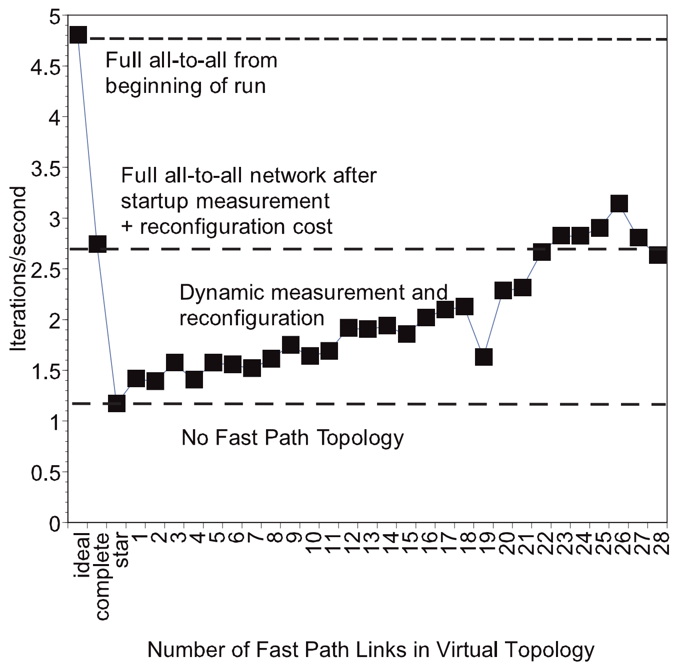


Figure 8.3: All-to-all topology with eight VMs, all on the same cluster.

8.3.1 Adaptation of high performance computing applications

We present results for the cases where we adapt using only topology based adaptation schemes (topology and routing changes) and cases where we leverage both, VM migration and topology centric adaptation schemes.

Topology centric adaptation

If we add c of the n inferred links using the OHR adaptation scheme, how much do we gain in terms of throughput, measured as iterations/second of patterns? We repeated this experiment for a variety of physical configurations. Additionally, we also experimented with different application topology and communication patterns such as neighbor exchange on a bus, ring, 2D mesh, and all-to-all. In the following, we show representative results.

Figure 8.3 gives an example for the single cluster configuration (all VMs on the IBM

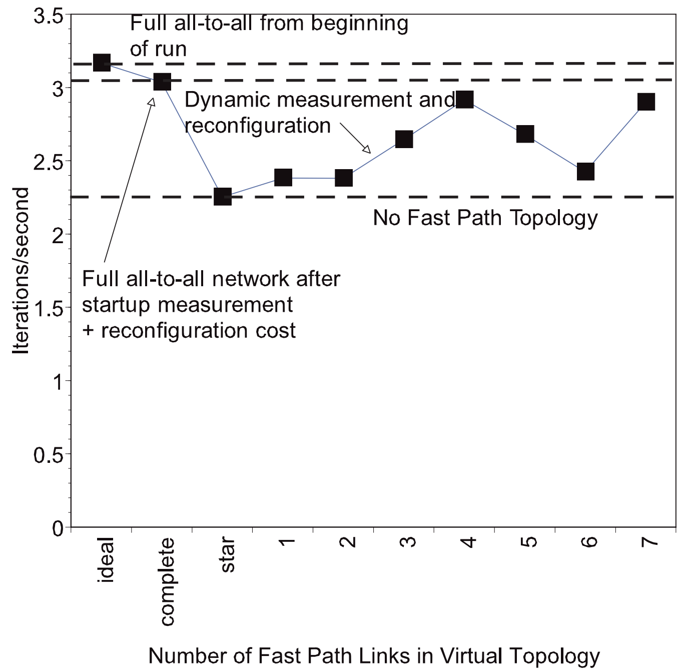


Figure 8.4: Bus topology with eight VMs, spread over two clusters over a MAN.

e1350 cluster), here running an 8 VM all-to-all communication. Using only the resilient star, the application has a throughput of ∼1.25 iterations/second, which increases to ∼1.5 iterations/second when the highest priority fast path link is added. This increase continues as we add links, improving throughput by up to factor of two.

Figure 8.4 illustrates the worst performance we measured, for a bus topology among machines spread over our two clusters separated by a MAN. Even here, our adaptation scheme did not decrease performance.

Figure 8.5 shows performance for 8 VMs, all-to-all, in the WAN scenario, with the hosts spread over the WAN (3 on the IBM e1350 cluster at Northwestern, 2 in slower cluster at Northwestern, one in a third Northwestern campus network, one at University of Chicago, and one at CMU). The Proxy and the user are located on a separate network at Northwestern. Again, we see a significant performance improvement as more and more

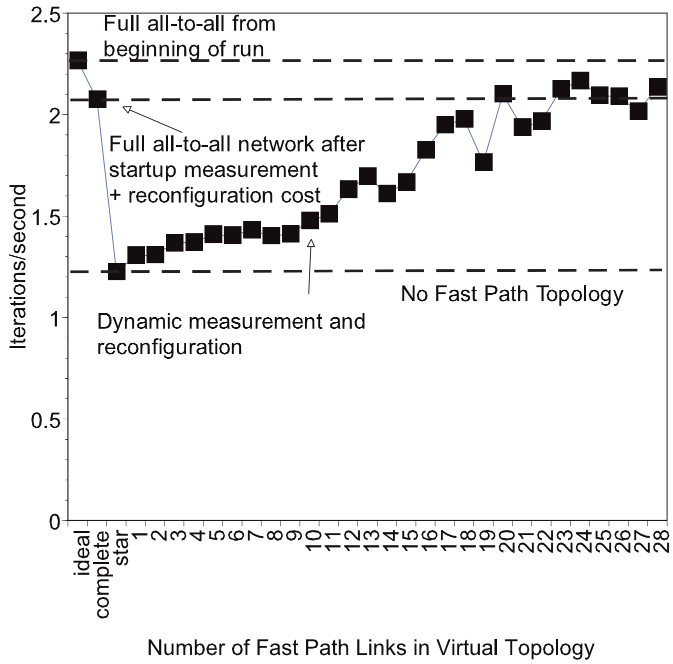


Figure 8.5: All-to-all topology with eight VMs, spread over a WAN.

fast path links are added.

Based on the learnings gained in these studies we designed the remainder of the routing algorithms detailed in Chapter 7. The most important insight was that maybe we can do better than IP routes, by routing intelligently over the VNET overlay. This idea was first introduced in the Resilient Overlay Network (RON) project [4].

Adaptation schemes based on multiple adaptation mechanisms

For the patterns application, we studied the following scenarios:

1. Adapting to compute/communicate ratio: Patterns was run in 8 VMs spread over the WAN (4 on Northwestern's IBM e1350 cluster, 3 on the slower Northwestern cluster, and 1 at CMU). The compute/communicate ratio of patterns was varied.

2. Adapting to external load imbalance: Patterns was run in 8 VMs, all on Northwest-

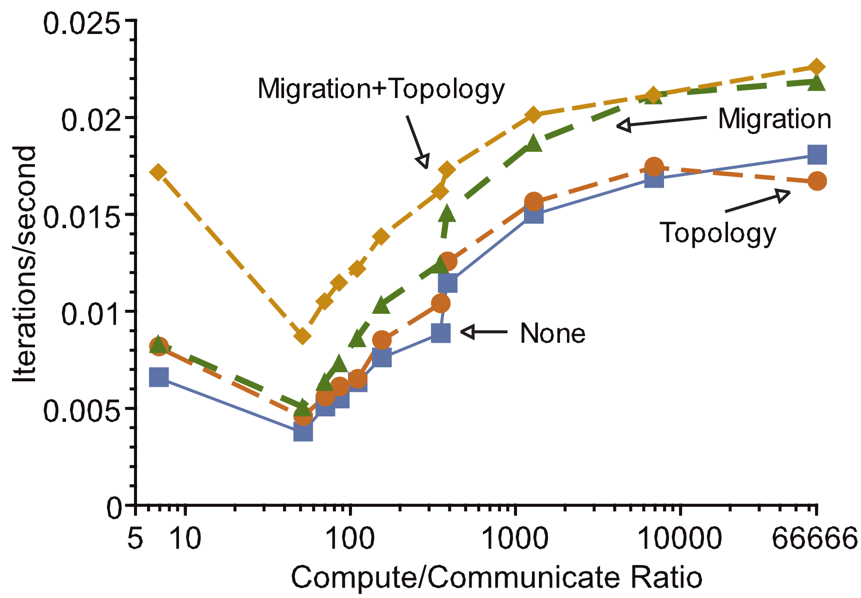


Figure 8.6: Effect on application throughput of adapting to compute/communicate ratio.

ern's IBM e1350 cluster. A high level of external load was introduced on one of the nodes of the cluster. The compute/communicate ratio of patterns was varied.

In both cases, patterns executed an all-to-all communication pattern and we compare results for when we performed no adaptation, topology only adaptation, migration only adaptation and combined topology and migration based adaptation. It should be carefully noted that in the first scenario, topology only adaptation is OHR, migration only adaptation is GOBM, combined topology and migration adaptation is GOBM + OHR. In the second scenario, topology only adaptation is OHR, migration only adaptation is CLB, combined topology and migration adaptation is CLB + OHR.

For an application with a low compute/communicate ratio, we would expect that migrating its VMs to a more closely coupled environment would improve performance. We would also expect that it would benefit more from topology adaptation than an application with a high ratio.

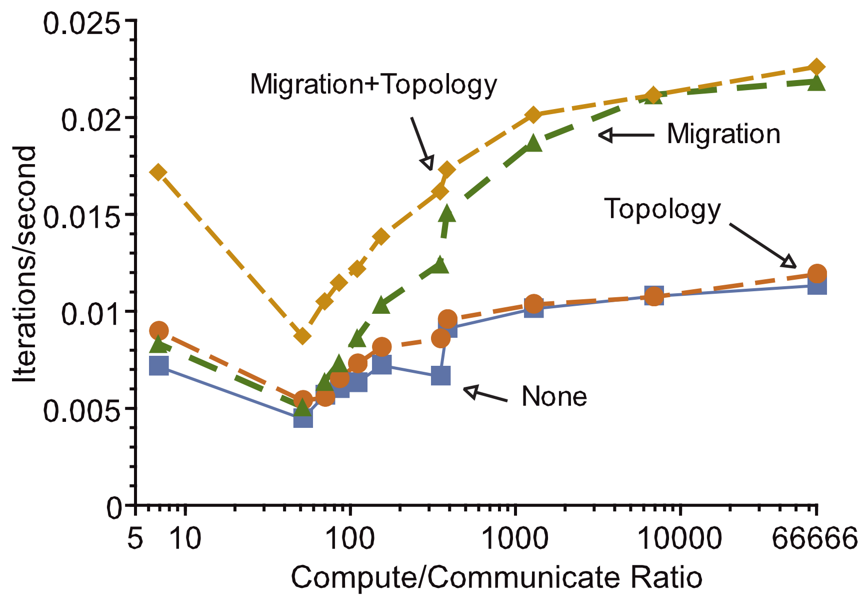


Figure 8.7: Effect on application throughput of adapting to external load imbalance.

Figure 8.6 illustrates our scenario of adapting to the compute/communicate ratio of the application. For a low compute/communicate ratio, we see that the application benefits the most from migration to a local cluster and the formation of the fast path links. In the WAN environment, adding the overlay links alone doesn't help much because the underlying network is slow. Adding the overlay links in the local environment has a dramatic effect because the underlying network is much faster.

As we move towards high compute/communicate ratios, migration to a local environment results in significant performance improvements. The hosts that we use initially have diverse performance characteristics. This heterogeneity leads to increasing throughput differences as the application becomes more compute intensive. Because BSP applications run at the speed of the slowest node, the benefit of migrating to similar-performing nodes increases as the compute/communicate ratio grows.

	No topology adaptation	Topology only adaptation (OHR)
No migration based adaptation	1.216	1.76
Migration only adaptation (CLB)	1.4	2.52

Figure 8.8: Web throughput (WIPS) with image server facing external load under different adaptation approaches.

Figure 8.7, shows the results of adapting to external load imbalance. We can see that for low compute/communicate ratios, migration alone does not help much. The VMs are I/O bound here and do not benefit from being relieved of external CPU load. However, migrating to a lightly loaded host *and* adding the fast path links dramatically increases throughput. After the migration, the VM has the CPU cycles needed to drive network much faster.

As the compute/communicate ratio increases, we see that the effect of migration quickly overpowers the effect of adding the overlay links, as we might expect. Migrating the VM to a lightly loaded machine greatly improves the performance of the whole application.

The results from this study indicated that it is important to study diverse adaptation schemes for diverse application scenarios. Section 8.6 is the topic of such a discussion.

8.3.2 Transactional web e-commerce applications

The purpose of this study was to understand the effectiveness of our approach for non-parallel applications. In particular we carried out studies on the TPC-W benchmark. TPC-W models an online bookstore and we have described it in Chapter 2. The separable components of the site can be hosted in separate VMs. We run the browsing interaction job mix (5% of accesses are order-related) to place pressure on the front-end web servers and the image server.

The primary TPC-W metric is the WIPS (Web Interactions Per Second) rating. Figure 8.8 shows the sustained WIPS achieved under different adaptation approaches. We are adapting to a considerable external load being applied to the host on which the image server is running. When we migrate this VM to another host in the cluster, performance improves. Reconfiguring the topology also improves performance as there is considerable traffic outbound from the image server. Using both adaptation mechanisms simultaneously increases performance by a factor of two compared to the original configuration.

8.4 Scaling

We tested topology adaptation scenarios with all-to-all traffic among up to 28 VMs, the maximum possible on a single one of our clusters. We used a pre-defined VM to host mapping to study the scalability of the overlay adaptation. While the cost of VM migration to meet an adaptation goal grows with the number of VMs, the number of links in the overlay topology can grow with the square of the number VMs, thus the system will scale as VNET scales, not as migration scales. The number of forwarding rules per node can also grow with the square of the number of VMs, although the worst topology for this is a linear one, which is unlikely to be used. For an all-to-all, it grows linearly with the number of VMs.

At 28 VMs, we can create our initial star topology in about about 2.9 seconds, with 84% of the time spent loading forwarding rules into VNET daemons. The total number of links and forwarding rules in the system for a star grows linearly with the number of VMs. Adding the full all-to-all topology takes 20.5 seconds, of which 67% involves loading forwarding rules. The inference time remains roughly the same as with the smaller scenarios we described previously.

Not surprisingly, the benefit of adapting the topology to the application grows as the number of VMs grows.

8.5 Virtualized system simulator

Analytical modeling, physical experimentation and simulations are three means of studying computer systems. In the course of this dissertation we have extensively used all three techniques. In particular, we have used these techniques in a sequential manner. First, we analytically modeled the adaptation problem. We found the problem to be hard to solve and approximate. This analysis suggested that we study heuristic driven algorithms as solutions rather than attempt theoretical combinatorially approximate solutions. Following this insight we carried out extensive physical experiments to study different aspects of our adaptation mechanisms and problem. We used the insights obtained from these studies to design fifteen different adaptation schemes as possible solutions to the adaptation problem. Beyond this, we experienced a lot of problems in maintaining our wide-area testbed limiting our productivity. To carry out a more extensive evaluation of our system we turned our attention to simulations. Simulations when carried out using a verified and validated simulator can provide powerful indications. In the remainder of this section we present the design, implementation, verification and validation of our virtualized system simulator.

8.5.1 Assumptions and limitations

These simulation results are not a substitute for experimental results, but should be used in conjunction with the modeling and experimental results. The simulator described in this section is modeled on the Virtuoso system. The simulator is written in Perl and consists of approximately 5000 lines of code. The high level design (described next) is generic and we hope that such a design with minor modifications would also be applicable to other virtualized execution environments [15].

As with any simulator, we need to be cognizant of its limitations. The simulator does not model computer system memory or disks, what it models is computation, communication and the computation costs of communication. The simulator uses real world network measurements as input data, however, it does not uses a real model for time of day effects. Though the simulator takes into account cross traffic, it does not account for sudden instantaneous spikes in CPU or network traffic (unless fed in at the begining of the simulation).

Despite its limitations the simulator can be useful in multiple ways:

- It allows us to study scenarios that are difficult or impossible to setup in the physical world.
- Since it operates in virtual time, it reduces the time to complete adaptation studies, thereby allowing us to study many more scenarios than otherwise possible.
- Provides means for creating a plug and play system to study the effects of different CPU and network traffic models, and adaptation schemes.
- Simulation studies help us understand the adaptation problem better and in certain cases can also provide insights that can be used to further refine its physical deployed counterpart.
- With minor modifications, it can simulate multiple systems.

8.5.2 Simulator design

The simulator at the highest level consists of two components, an application execution component and an adaptation component. Figure 8.9 helps to explain the structure and working of the simulator.

Simulator input

The simulator takes in four categories of inputs:

Application trace:

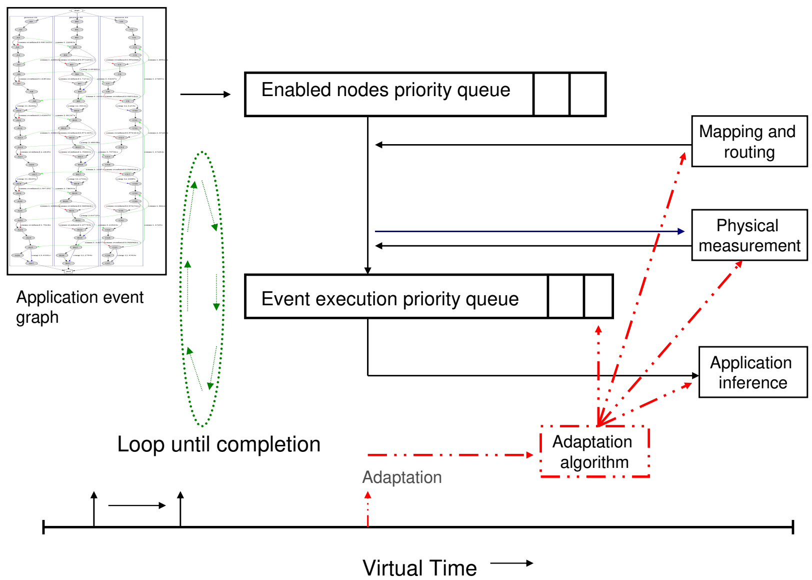


Figure 8.9: Working of the virtual system simulator.

An application trace is an event graph represented as a directed acyclic graph (DAG). Figure 8.10 shows a visual representation of a very tiny trace generated for a tiny run of patterns executing on three VMs (named A, B and C). Trace generation is described in Section 8.5.3. The event graph consists of nodes and edges. It starts at a node called *start* and ends at a node called *end*. The progression of each application component executing inside a VM is represented by a series of nodes connected by plain or annotated edges. In Figure 8.10, each such application component is enclosed in a rectangular box.

There are four possible types of edges in the event graph:

1. *Compute edge:* Such an edge is labeled with the amount of compute operations that are to be performed on that node starting at that point in time. The origination node of the edge represents the VM state before the start of the computation and the destination node of the edge represents the state of the VM at the end of the computation.

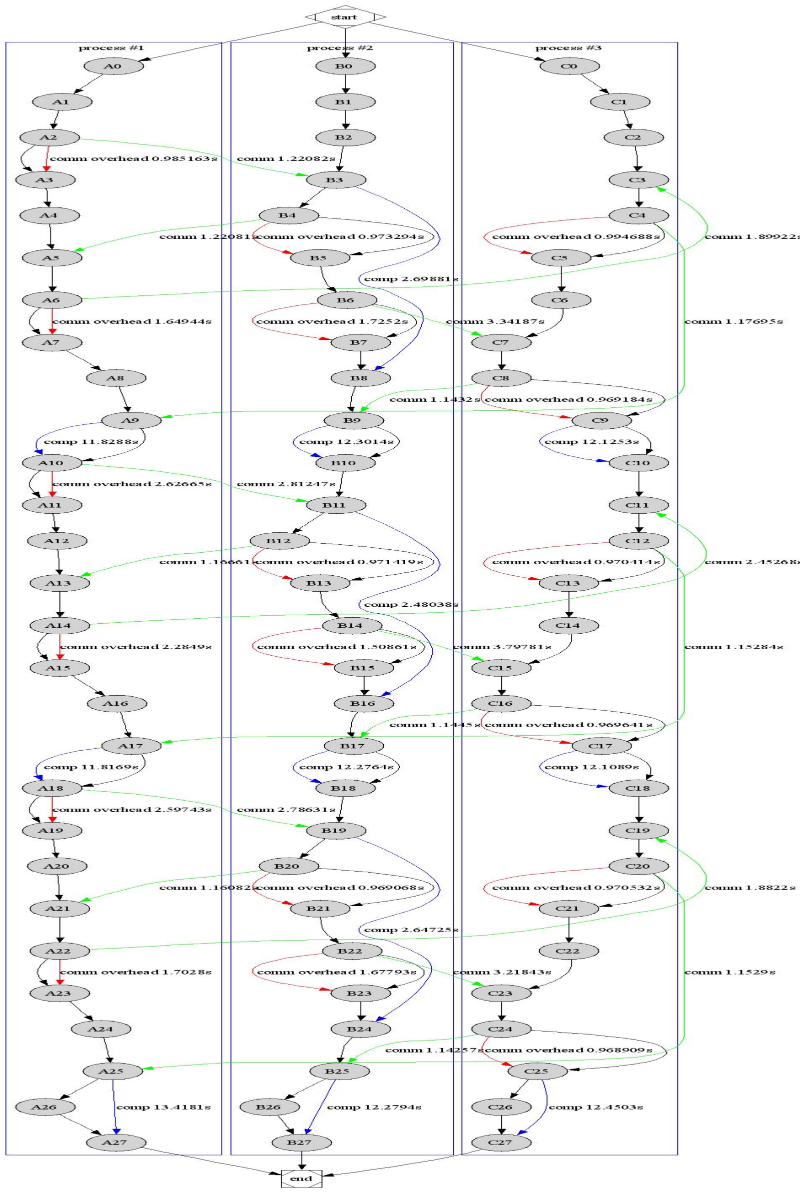


Figure 8.10: Input application trace for a tiny patterns application.

2. *Communicate edge:* This edge is labeled with the amount of bytes that are to be sent from one VM to another. Again, the origination node represents the state of the sending VM before it starts sending the data and the destination node represents the state of the receiving VM after it has received all the data.

3. *Communication overhead edge:* This edge captures the computation overhead of communication. The amount of time between starting of "send" operation on a VM until its completion.

4. *Plain edge:* Some of the edges are not annotated (are instead plain). These edges simply make the graph visually understandable. They are ignored by the simulator.

Physical system measurement data: In our physical system, tools such as Wren and Ganglia [124] can provide system resource measurements. One of the inputs to the simulator is a file containing such data. The file is populated with previously measured data. On the network end, it contains bandwidth and latency measurements between VNET hosts in the system. It also contains the CPU speed and utilization that a VM gets on that host. This information can change at any point in time to reflect a change in available system resources.

Initial VM to host mapping and routing information: Since, the simulator models the Virtuoso system, it is fed with an initial VM to host mapping and the star topology routing. As the application progresses and the system performs adaptation, this information is modified to reflect the latest mapping and routing.

Command line parameters: The simulator also takes in a few command line parameters to specify the adaptation algorithm to be used, values for certain algorithm parameters among others. The interface to the simulator is described in Table 8.1.

Command line parameters	Description
a	Specifies adaptation related parameters
adapt_at	0 means no adaptation, else time of first adaptation
interval	periodic intervals (from adapt_at), 0 means no repetitions
times	number of periodic adaptation
s [0 or 1]	0 means execute in one step, 1 means execute in multiple steps
p [0 or 1]	0 means print results to file, 1 means print to screen and file
m measurement_file	specifies file containing physical host and network measurement
g mapping_file	file containing initial mapping
b vttif_file	file containing VTTIF data
i input_file	file containing application trace
o output_file	file containing application specific output
l log_file	file containing complete system simulator log
c cost_file	file containing parameters for costs of adaptation
r algorithm	adaptation algorithm to be used
v [0 or 1]	0 means offline adaptation, 1 means online adaptation
t threshold	threshold in seconds for algorithms CLB and CLB + OHR
y latency_constant	constant for algorithms GOBLM + WBLP

Table 8.1: Virtualized system simulator interface.

Simulator data structures

During the course of its operation, the simulator maintains three main data structures.

Incoming edges priority queue: At startup the simulator reads in the input event graph. It creates a priority queue containing all the nodes in the graph. The priorities are the number of incoming edges in the input application event graph. Lower the number of incoming edges, higher is the priority of that node. Nodes with the highest priority are serviced first.

Event priority queue: The simulator also maintains a priority queue containing events to be executed. These events correspond to one of the three annotated edges in the event graph. An event can either be a computation event, a communication event or a computation overhead of communication event. The events in this priority queue are indexed by their estimated completion times. Earlier is the completion time of an event, higher is its priority. Events with the highest priority are serviced first

Application state hashes: As the simulator executes the application, for each application component executing inside of a VM, we maintain state. This consists of time spent by the component computing (user time), time spent in system overheads (system time), time spent waiting for other events (idle time). A global wallclock for the entire application is also maintained. The efficiency of an application component inside of a VM is calculated as $efficiency = usertime/wallclock$.

Simulator operation

At startup the simulator loads in the application event graph. It performs a topological sort on the nodes and creates the incoming edges priority queue. The pseudo code shown in Figure 8.11 is at the core of the simulator.

The simulator walks the event graph looking for nodes that have zero incoming edges.

This means that the node has no existing dependencies and that we can process all its outgoing edges. For each out going edge, based on whether it is compute, communicate or compute overhead of communicate, we calculate its estimated completion time. This calculation is based on the current VM to host mapping and overlay routing, and the physical system measurement data. We add an event to the event priority queue indexed by this estimated completion time. If the outgoing edge is a plain edge, we then reduce that particular node's incoming edge count by one. Adding an event to the event queue implies that it has started execution (or communication), but has not yet completed. If the event is a communication event, then we modify the physical system measured data to account for the application's own communication.

When we have no nodes with zero incoming edges to process, we execute the next event in the event priority queue. We increment the wallclock to represent the progression in time and we also modify the application data structures (user time, system time, etc.) based on the specifics of the event executed. The incoming edge count of the concerned node is decreased by one. Further, for a communication event we modify the physical system measured data to reflect the completion of this communication event.

We then go back to see if we have "enabled" any nodes, i.e. if any nodes have zero incoming edges and then repeat the above mentioned steps.

Once we finished walking the entire graph, we execute the remaining events in the event priority queue and perform the associated actions as above.

Adaptation model

We have described at a very high level the basic functioning of the system simulator. Here we present a description of the simulator's adaptation model. As shown in Table 8.1, times at which adaptation is requested is supplied from the command line at the begining of the simulation. Before execution of any event from the event priority queue, we check to see if

```
// 1. Initially all nodes are marked ``unserviced''
//
// 2. A Node becomes ``enabled'' when number of incoming edges
//    to it is zero
//
// 3. A Node is ``serviced'' after we have finished
//    processing all its going edges

while(there exists an unserviced node){
    remove the enabled nodes from incoming edges queue;
    foreach (enabled node) {
        foreach (outgoing edge) {
            process the event associated with it;
            add the appropriate event to event priority
            queue to finish at appropriate time;
            if (outgoing edge is plain) {
                reduce the number of incoming edges to the node by 1;
            }
        }
        mark the node as serviced;
    }
    if (there exists an enabled node) {
        next;
        // takes us to begining of top-level while loop
    }
    remove the next event from the event priority queue;
    execute the event;
    modify wallclock and application data structures;
    reduce the number of incoming edges to the node by 1;
}
while (there exists an event in event priority queue) {
    remove the next event from the event priority queue;
    execute the event;
    modify wallclock and application data structures;
}
```

Figure 8.11: The simulator core.

the adaptation time is enabled. If so, then based on the adaptation scheme specified on the command line (Table 8.1), we execute the related adaptation algorithms. The algorithms modify the then current mapping and routing, thus creating a new mapping and routing.

The remainder of the application now has to execute on this new mapping and routing. It should be noted that the events already present in the event queue were estimated to finish at their respective times based on the previous mapping. For each such event in the priority queue, we break up its execution into two components. First, from the time they started until the end of the previous mapping. Second, from the time the new mapping takes into effect until their completion time as calculated on the new mapping. We finish executing the first component and the second component is added to the event queue to replace this event's previous entry.

The simulator has the option of performing the adaptation offline or online. In offline adaptation, we assume the application to stall until the adaptation completes. For example, an adaptation scenario where we perform VM migration. The migration semantics could be suspend-resume, where the application stalls for during migration. On the other hand, online adaptation assumes that the application continues execution while adaptation is performed. In this model, until the adaptation is in progress the application executes on the old configuration (mapping and routing). Once the adaptation is complete, the application continues execution without interruption, but now on the new mapping.

Simulator output

In Virtuoso, VTTIF infers the application's network resource demand inference. In simulation, inferring application resource demand is tremendously simplified as the simulator has access to the application trace and hence knows the ground truth. The simulator mimics VTTIF's behavior in that when it detects a change in application demands, it records the same which is then available as input for the adaptation schemes. At the end of a simulator

```
0.087789298245614: analyze the outgoing edge "B4 -> A5 comm 1000000"

0.087789298245614: In current configuration transferring 1000000 bytes between vm "B" mapped on host "virtuoso-2" and vm "A" mapped on host "virtuoso-3" takes 0.087789298245614 seconds

0.087789298245614: insert "B4 -> A5 comm 1000000" into event queue to finish at time 0.175578596491228

0.087789298245614: Begining of event queue
        0.163966:       A2      ->      A3      commoverhead    0.163966        0       1       0.163966        0.163966
        0.175578596491228:      B4      ->      A5      comm    1000000         0.087789298245614       11390910.0537771        1000000         0.175578596491228
         End of event queue

0.087789298245614: analyze the outgoing edge "B4 -> B5 commoverhead 0.163976"

0.087789298245614: In current configuration computing 0.163976 operations on vm "B" mapped on host "virtuoso-2" takes 0.163976 seconds

0.087789298245614: insert "B4 -> B5 commoverhead 0.163976" into event queue to finish at time 0.251765298245614

0.087789298245614: Begining of event queue
        0.163966:       A2      ->      A3      commoverhead    0.163966        0       1       0.163966        0.163966
        0.175578596491228:      B4      ->      A5      comm    1000000         0.087789298245614       11390910.0537771        1000000         0.175578596491228
        0.251765298245614:      B4      ->      B5      commoverhead    0.163976        0.087789298245614       1       0.163976        0.251765298245614
         End of event queue

0.087789298245614: Look to execute the events at the next priority level 0.163966

0.087789298245614: pop "A2 -> A3 commoverhead 0.163966 0 1 0.163966 0.163966" from the event priority queue

0.087789298245614: In current configuration computing 0.163966 operations on vm "A" mapped on host "virtuoso-3" takes 0.163966 seconds

0.163966: Finished executing "A2 -> A3 commoverhead 0.163966 0 1 0.163966 0.163966" and set the node's user time to 0.081983

0.163966: Begining of event queue
        0.175578596491228:      B4      ->      A5      comm    1000000         0.087789298245614       11390910.0537771        1000000         0.175578596491228
        0.251765298245614:      B4      ->      B5      commoverhead    0.163976        0.087789298245614       1       0.163976        0.251765298245614
         End of event queue

0.163966: Decrease the number of incoming edges for node "A3" from 1 to 0
```

Figure 8.12: A portion of the log file for the execution of a patterns application.

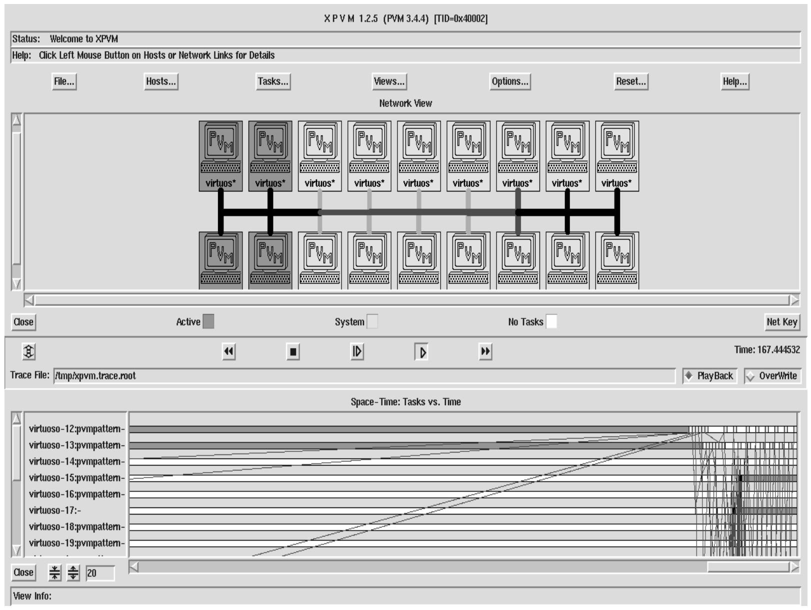


Figure 8.13: XPVM output for PVM patterns application executing on 18 hosts.

run, we print out the user time, system time, idle time and efficiency for each application component executing in a VM. In addition, we also print out application specific metrics, such as iterations per second (throughput) for the patterns application, messages per second for a mail application, etc. Additionally, we also maintain a detailed log that records everything that happens in the lifetime of the simulator. Figure 8.12 shows a portion of such a log file.

8.5.3 Real trace generation

Our simulator is a trace driven simulator. The idea is to collect traces of real applications under a tightly controlled setup and then to replay the traces under real world scenarios to study effects of adaptation. The tightly controlled setup refers to our IBM e1350 cluster. We use isolated and unloaded nodes interconnected via a Gigabit switch.

Since we use PVM versions of high performance computing applications, we were

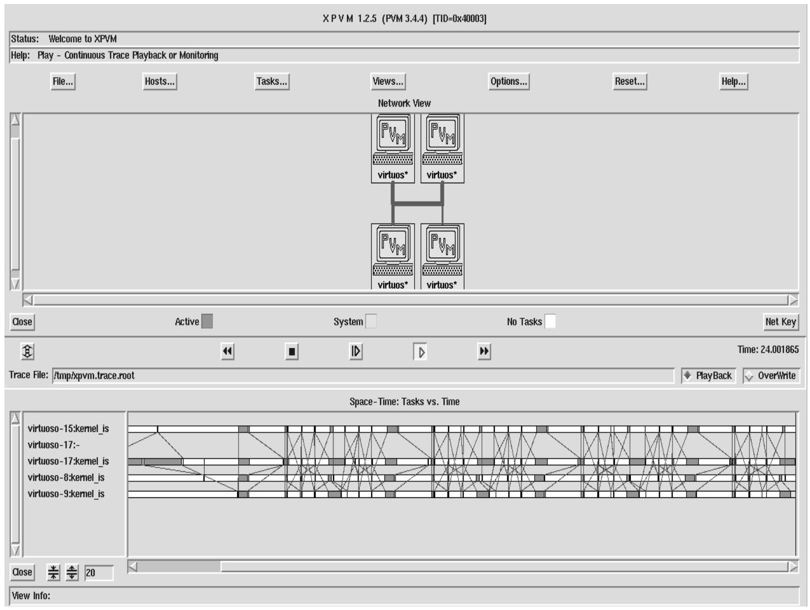


Figure 8.14: XPVM output for PVM NAS IS benchmark executing on 4 hosts.

able to obtain application trace data using the XPVM [105] tracing facility. XPVM is a tracing facility and tool for PVM [63]. The XPVM tracing system supports a buffering mechanism to reduce the perturbation of user applications caused by tracing, and a more flexible trace event definition scheme which is based on a self-defining data format. The tracing instrumentation is built into the PVM library [105].

Figure 8.13 shows a snapshot of the XPVM trace for the patterns application executing on eighteen nodes of our cluster. Figure 8.14 is a snapshot of the XPVM trace for the NAS IS benchmark executing on four nodes of our cluster. We converted the raw XPVM traces into the input format accepted by the simulator.

8.5.4 Verification and validation

Verification refers to the process of ensuring that the simulator is implemented correctly and validation refers to the process of ensuring that it is representative of the real system.

The verification and validation was performed independently by a person who was familiar with the idea behind this simulator, but was not directly involved in the development of the simulator.

Verification

The simulator was independently verified using the following techniques [93]:

- **Structured walk-through:** I explained the code of the simulator in detail to multiple people. Several bugs were identified and addressed through this process.
- **Running simplified cases:** A synthetic small trace constructed by hand was used as input. We compared the output of the simulator with the manually calculated output. They were very similar in terms of the wall-clock time, total computation time and communication time. In addition, the log file of the simulator was carefully checked line by line to catch and fix errors.
- **Antidebugging:** We included additional checks and outputs in the simulator and the log file respectively to point out bugs.
- **Degeneracy tests:** We checked the working of the simulator for extreme values (multiple VMs mapped on same hosts, applications with only computation or only communication, adaptation completing instantaneously, adaptation taking a long time to completion, etc.).
- **Consistency tests:** To verify the adaptation functionality of the simulator, the physical system measurement file was modified by adding two idle nodes with higher bandwidth and associated all-to-all routes while we reduced the utilization limits on two nodes that initially hosted VMs. The application trace executed was for a patterns application executing on four nodes of our cluster. The result showed that both

VMs were correctly migrated with routes among VMs being changed correctly. As the result of the adaptation, both computation and communication time decreased when compared with the case of no adaptation.

Validation

The simulator was validated using the following techniques [93]:

- **Expert intuition:** This is the most practical and common way to validate a model [93]. Multiple brainstorming meetings of people knowledgeable about virtualized execution systems were called. Some of these meetings were at Northwestern University and others were help at Intel Corporation (where parts of this research was conducted). We validated the assumptions, inputs and outputs.

- **Real system measurements:** This is the most reliable and preferred way to validate a simulation model [93]. Using XPVM [105] and our task-graph converter (to convert XPVM trace to a form compatible with our simulator), we gathered a patterns trace on 4 nodes of our cluster. According to patterns output, the ratio of computation over communication was high. The trace contained 5 iterations and the communication pattern was set to be all-to-all. The data in the measurement file was collected via physical measurement tools such as ttcp, ping and top to reflect the current state of the cluster nodes and the isolated network between them. The comparison between patterns output and the simulator output showed that the simulator correctly and closely simulates the running of the trace. We then modified the measurement file by reducing the utilization limits on 2 of those 4 nodes from 100% to 50%. Then, as expected, the total computation time in the output of the simulator nearly doubles, which is what we expect since the entire patterns application is slowed down to the speed of the slowest node. This result also agreed with the output of patterns, with the utilization being throttled using VSched. A validation was also carried out for a slightly larger application executing among 6 VMs with a low computation to communication ratio.

8.5.5 Synthetic trace construction

We were able to obtain, using XPVM, real traces for applications belonging to the high performance computing class. For the seven classes of applications that were used to evaluate the effectiveness of the adaptation schemes, we constructed synthetic traces by hand. These traces were constructed and verified in our discussions with Intel's Corporate Technology Group and Intel Corporation's IT Innovation and Research Group from June 2006 until November 2006. The last six months of this research was conducted on site at Intel Corporation's Corporate Technology Group, Hillsboro, OR.

Over the past two years, Intel Corporation, has conducted extensive studies to understand enterprise application resource demands. This study was conducted specifically with a view to understand the applicability of virtualizing these enterprise applications in operating system virtual machines connected via virtual networks [15].

The aim of our discussions was to better understand the previously conducted study [15], to build application traces that were modeled on the application demands as understood by the above study and to discuss these constructed traces so as to establish if they were reasonable representatives of real applications. This process included brainstorming sessions with a group of researchers, reading Intel technical journals and reports, one on one discussions with researchers in Hillsboro OR and Santa Clara, CA, conducting measurement studies on the Intel IT Research Overlay [15], and individual interviews conducted with the researchers.

The constructed traces have the same syntax as real XPVM traces, but differ significantly in their computation and communication behaviors.

8.6 Evaluations to close the loop

Up and until now, we collected application classes, studied a subset of those and used the insights gained therein to design fifteen different adaptation schemes. We evaluated these adaptation schemes against each other and against estimates of the optimal in terms of optimizing the different objective functions. However, based on the evaluations presented so far, we do not know the effectiveness of these adaptation schemes in terms of application specific metrics. In other words, we know what these adaptation schemes can do for specific objective functions, but we do not know what these adaptation schemes can do for the applications.

We fill this gap and close the loop for our adaptation studies by studying the effectiveness of our fifteen adaptation schemes for application traces that model typical applications for the application classes listed in Table 2.1. In particular we try to find the answer to the question we posed in the begining of this dissertation, is there a single adaptation scheme that has wide applicability among a diverse range of application classes.

Though the simulator can model offline and online adaptation schemes, in this section the model of adaptation is online, unless stated otherwise. In the fast few years, there have been significant efforts in reducing migration times and in realizing the ideal case of fast live migrations [25, 107, 134, 137, 146]. Migration times have a number of variables, such as the size of the VM in question, the network and path from the source host to the destination host, the amount of caching at the destination VM, etc. Currently, there are no published models that predict migration times, given values for this set of parameters. We make a simplifying assumption and use a single migration time to be representative of the average of the migration times for different scenarios. In particular, we model a live migration time of 50 seconds.The cost of network adaptation is modeled on our previously measured adaptation configuration times. The migration threshold was set to 50 seconds and the latency constant was set to 0.001 MB.

Over a period of time while building and maintaining our distributed testbed, we made extensive physical system measurements. We complemented these by making additional measurements on Intel Corporation's IT Research Overlay Network. This overlay network extends between ten sites spread geographically across the US. The sites that we had access to were in Hillsboro, OR, Santa Clara, CA and Hudson, MA. The network measurement data fed as input to the simulator represents the statically measured physical network (latency using ping and bandwidth using ttcp) between all these hosts spread across the testbed. The CPU utilization that each VM on a host got was synthetic data to create the notion of multiple users per host. This was done, as at the time of the measurements, almost all the nodes were lightly loaded. The synthetic utilization was created using the generator described in Chapter 7.

In all our evaluations we compare the algorithms against each other and also against the case where we do not perform adaptation. The initial VM to host mapping and routing through proxy is created randomly. This (the concept of an initial mapping and not the fact that it is randomly generated for these simulations) mimics Virtuoso's front-end's behavior while acting as a broker of virtual resources. The initial mapping has a significant impact on the improvement brought about by the adaptation algorithm. If the initial mapping is not suited to the application then the adaptation algorithms will improve application performance significantly. However, if the initial mapping is perfectly suited to the needs of the adaptation, then the adaptation schemes will seem to be ineffective. What is important from the point of view of adaptation is, if the initial mapping is not good, or if physical conditions change such that the current mapping is no longer well suited, then in such cases are the adaptation schemes able to change things for the better.

In this section, we present our evaluation results and in the following section (Section 8.7), we present a summary of our results and a set of recommendations.

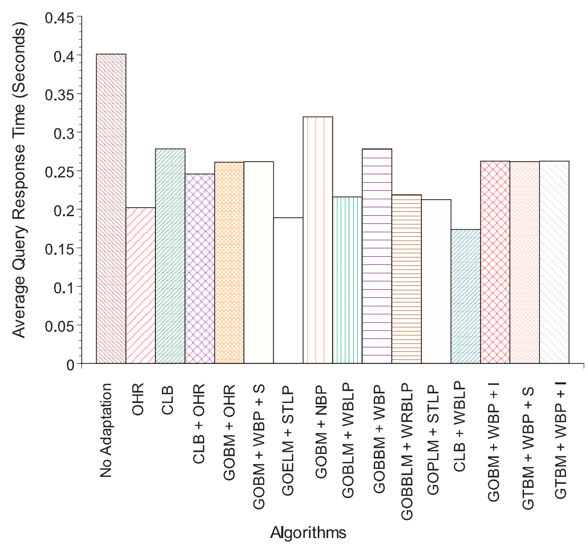


Figure 8.15: Performance of adaptation schemes for an application trace modeling an adaptive DNS.

8.6.1 Application trace representative of an enterprise DNS service

Figure 8.15 shows the effectiveness of the different adaptation schemes for an application trace modeling an enterprise DNS service. The trace is representative of a series of query response pairs between end host VMs and VMs hosting DNS servers and a set of zone transfers. The interactions presented represent a mixture of:

- Host queries DNS server and the server responds directly (small queries with quick small replies).
- Host queries DNS server and the DNS server, in turn, queries another DNS server, creating a relay, before response gets back to the host (small queries with more delayed replies).

- A DNS server acts as a master server for one zone, but as a slave server, caching contents for a second zone, at periodic intervals of time the master server for a zone conducts a mapping transfer with the slave servers for that zone (bulk transfers).

The application specific metric that we are interested in is the average query response time. From Figure 8.15 we notice that CLB + WBLP provides us with the lowest average query response times. The trace contains a mixture of interactions which are affected negatively by high latency and low bandwidth paths. CLB + WBLP attempts to find paths that have low latencies and high bandwidths. We also notice that algorithms that do not take latency into account (such as GOBM + NBP, GOBBM + WBP) result in higher average query response times. The performance of CLB is better than expected. This application does not have a significant computation component, hence one would expect that migrating VMs to improve execution time would not reduce the average query response time. One possible explanation is that the high compute capacity nodes that the VMs were migrated to also connected using low latency and high bandwidth links. We notice that GOELM + STLP and GOPLM + STLP also significantly reduce the average query response times. These algorithms are latency centric and since low latency is an important factor for such applications we see the improvement in performance. Finally, note that all the algorithms provide some improvement over the case wherein we rely on Virtuoso's default mapping and perform no adaptation.

8.6.2 Application trace representative of an enterprise mail service

Enterprise mail application is an important application with multiple dependencies between different components. Our trace is representative of three mail servers, each speaking to a directory service, a machine hosting mailboxes and a DNS server. The application specific metric that is kept track of is messages per second handled by the overall system.

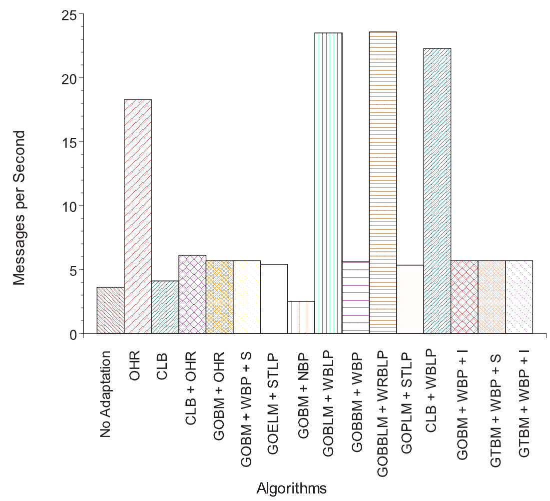


Figure 8.16: Performance of adaptation schemes for an application trace modeling a typical mail server setup.

The trace begins with a collection of chatty protocols interspersed with large data transfers representative of large mails flowing through the system.

Figure 8.16 shows the effectiveness of the adaptation schemes in terms of the application defined metric. Algorithms GOBLM + WBLP and GOBBLM + WRBLP perform the best in terms of increasing the average number of messages handled. Algorithm CLB + WBLP also performs reasonably well. These algorithms look to optimize both latency and bandwidth. Since different stages have different needs, optimizing for one, but not the other does not lead to significant performance improvement. It is interesting to note that adaptation schemes that optimize only latency and those that optimize for only bandwidth, perform very similarly. Though the number of large transfers handled is only a small fraction of the number of smaller chattier interactions, the payback of optimizing for the same is larger. We also see that a single pass mapping performs the same as a two pass mapping

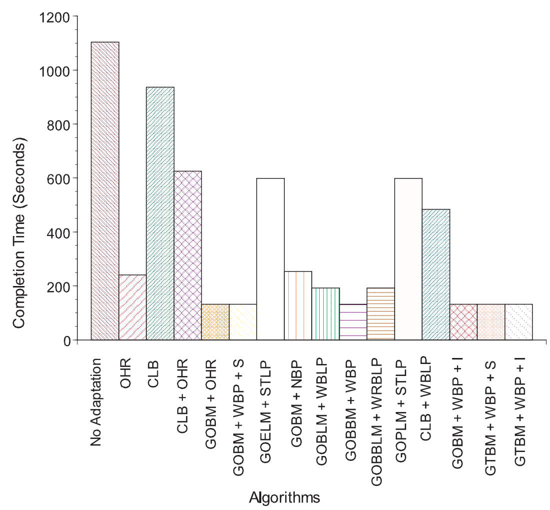


Figure 8.17: Performance of adaptation schemes for an application trace modeling an enterprise wide-area file transfer.

(GOBM + WBP + S and GTBM + WBP + I), further, we do not see any specific gains of interleaving mapping and routing (GOBM + WBP + S and GOBM + WBP + I). The use of GOBM + NBP actually reduces the performance of the application as compared to the case where we perform no adaptation. This is unexpected, as the mapping for the no adaptation case is not suitable for the application and GOBM + NBP manages to create a mapping and routing, which is even worse. One explanation is that GOBM + NBP performs adaptation such that more applications can enter the system. However, our comparison is based on metrics that measure the performance of only a single application in isolation.

8.6.3 Application trace representative of an enterprise wide-area file transfer

The application trace constructed here models wide-area file transfer of large files. Intel Corporation, currently, does not implement a wide-area file system such as AFS [82] between its different geographic locations. Intel Corporation's engineering and research groups are often spread across geographic sites thus creating virtual teams. Hence when files have to be accessed, wide-area file transfers are often used to create copies which can then be accessed locally. We are, at this point in time, not quite sure of how wide-spread this practice is in other large enterprises. However, we constructed a trace modeling this class as these file transfers are quite prevalent inside Intel Corporation. The trace is very simple to construct, it consists of large data transfers between a set of VMs.

Figure 8.17 illustrates the effectiveness of different adaptation schemes measured in terms of application completion times. Lower the completion times, better it is for the application. We note that all the bandwidth centric adaptation schemes (GOBM + OHR, GOBM + WBP + S, GOBM + WBP + I, GTBM + WBP + S and GTBM + WBP + I) perform well and reduce the completion time of the transfers. Latency centric adaptation schemes do not have as significant an effect in decreasing the completion times. The links being chosen by these algorithms (GOELM + STLP and GOPLM + STLP) are probably low latency but also low bandwidth. It is very interesting to note that wide-area file transfers are very similar to VM migration. These applications can not only be speeded up by ensuring that bottleneck links are bypassed, but can also be significantly improved by leveraging some of the VM migration schemes. In particular, exploiting the block level commonality in data to reduce the amount of data sent over the wire would be particularly helpful [137].

8.6.4 Application trace representative of an enterprise simulation

Intel Corporation primarily operates in the semiconductor industry. Before chip designs are cast in silicon, extensive simulations are carried out to better understand the expected performance and bottlenecks. Such simulations are typically stand-alone, running on a single machine. We were not able to hold discussions with people closely associated with

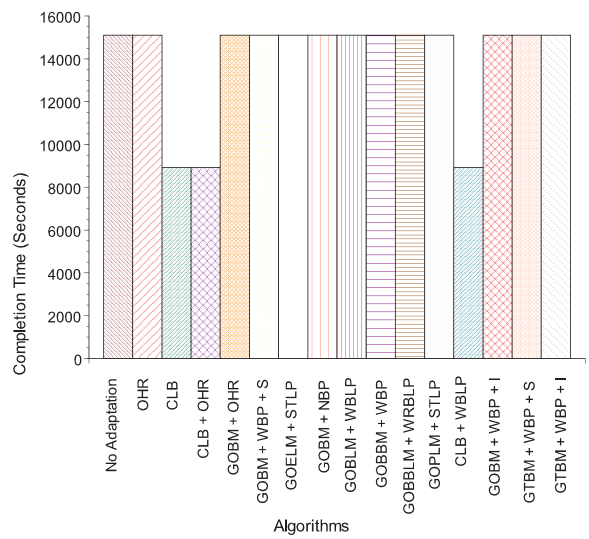


Figure 8.18: Performance of adaptation schemes for an application trace modeling compute intensive enterprise simulations.

such simulations. We were able to obtain only an indirect input on the nature of these applications. However, these applications are very important in the context of large engineering companies such as Intel Corporation and hence we constructed a trace for the same and carried out adaptation studies. We used our experience with high performance computing applications to construct this trace. In particular we likened this application to an embarrassingly parallel (EP) application executing on a single node. Such an application has no network demands, only CPU based demands.

Figure 8.18 presents the results of the adaptation studies comparing the fifteen different adaptation schemes in the context of simulation completion times. The only three adaptation schemes that have any effect are CLB, CLB + OHR and CLB + WBLP. The common algorithm component in all these adaptation schemes is CLB. CLB is the classic load balancing algorithm wherein, we migrate a VM to a new host, if we estimate a shorter execution time on the new node. This objective very closely ties in with what the appli-

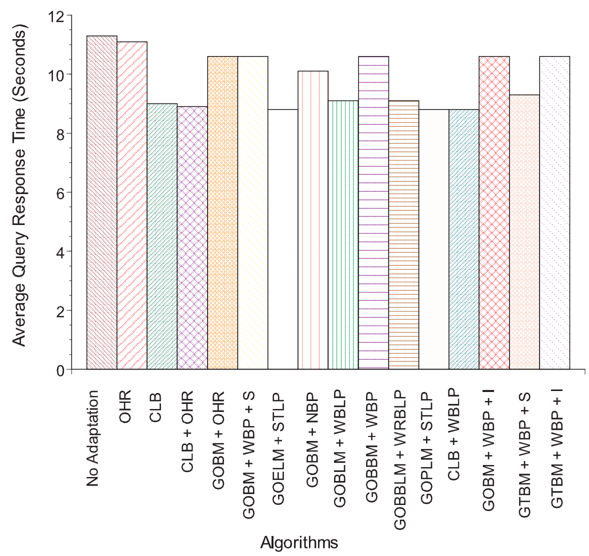


Figure 8.19: Performance of adaptation schemes for an application trace modeling a database application with high bandwidth requirements.

cation wants. All the other algorithms are optimizing the network aspects of the recourse demands, but in this case, the application makes no network based resource demands and hence executing these adaptation schemes has no effect. The key point here is prediction of how long the application will continue executing on the current mapping. This parallels load balancing in process migration systems [78] for which there exists an accepted model to predict the process lifetimes for UNIX process. Currently, no such model exists for virtualized execution systems.

8.6.5 Application trace representative of an enterprise database application with high bandwidth requirements

Database applications are all pervasive. Hence, it should come as no surprise that these constitute an important class of applications in enterprises. We have divided our study of such applications into two classes. First, in response to a query, significant amount of computation is performed and the data returned is very large. Second, in response to a query, a small amount of computation is performed and the data returned is small. We study the former in this section and the latter in Section 8.6.6. These database applications are also similar to the web transactional e-commerce applications studied and discussed earlier in this chapter (Section 8.3.2). However, they differ in how they are deployed. Enterprise database applications (as opposed to applications deployed by web commerce companies) are mostly for housekeeping purposes. We have constructed a trace that models enterprise database applications that perform significant amounts of computation and returns large amounts of data. In enterprises such as Intel Corporation, such queries are especially common at the end of each month and at the end of financial quarters. We model a three tier architecture, with a set of four front-ends, three application logic servers and two back-ends. Each query transmits a small amount of data, which flows through different paths ultimately resulting in a significant amount of computation at the back end. The result of the query is then ferried back to the origination point of the query.

Figure 8.19 illustrates the effectiveness of the fifteen adaptation schemes in terms of improving application performance. The application specific metric of interest in this case is average query response time. The algorithm CLB + WBLP performs the best. This application trace has small amounts of communication, large data transfers and significant amounts of computation. Hence we notice that the algorithm that covers all three (CLB + WBLP) performs the best. We also notice that latency centric applications (GOELP + STLP and GOPLM + STLP) also performed reasonably well. However, it should be noted that even for adaptation schemes that perform the best, the improvement over the scenario, where no adaptation is performed, is not significant. This is due to a strong initial mapping of VMs to host.

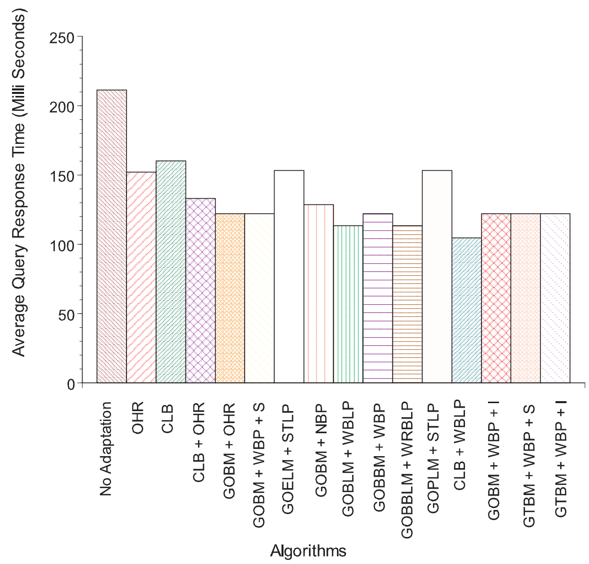


Figure 8.20: Performance of adaptation schemes for an application trace modeling a database application with low latency requirements. Average query response time is measured in milli seconds.

8.6.6 Application trace representative of an enterprise database application with low latency requirements

This is the second half of the studies on enterprise database applications. Here we focus on applications that comprise of short queries and short responses. The architecture represented is similar to the one described in Section 8.6.5. Figure 8.20 represents the performance of the adaptation schemes when measured using average query response time. We see that CLB + WBLP performs the best. The average query response time is reduced from 220 milli seconds to about 110 milli seconds. Bandwidth centric adaptation schemes such as GOBM + OHR + S and GTBM + OHR + I also improve the application performance. The application, however, does not make large bandwidth demands. Hence the benefit seen in that case is due to the choosing of links with high bandwidth and low

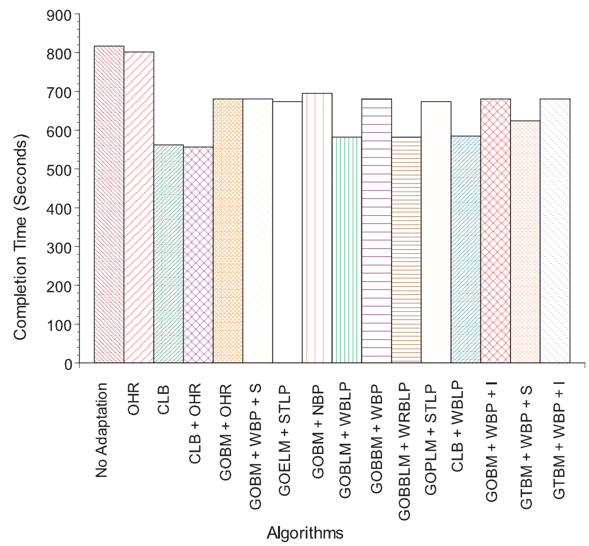


Figure 8.21: Performance of adaptation schemes for an application trace modeling an engineering computation application.

latency, and hosts with better utilization. However, what is extremely surprising is that, latency centric adaptation schemes such as GOELP + STLP and GOPLM + STLP do not perform that well as compared to the other algorithms. This seems a little counter-intuitive, as latency seems to be the property that is most important to the application. We believe that this behavior is to do with the computation component of the trace. The latency centric algorithms have found hosts with low latencies between them but, the utilization available to the VMs is low, resulting in longer compute times and lower average response times.

8.6.7 Application trace representative of an enterprise engineering computing application

It should come as no surprise that engineering computation applications are an important class of applications at engineering companies such as Intel Corporation. We have constructed a trace that models a typical engineering computation application. The trace mimics an application that executes inside of sixteen virtual machines, hosted across the wide area on four different sites. It consists of computation and communication between these VMs. Though disk accesses are a big part of such an application, our simulator does not models disks and hence we ignore this aspect. To a certain degree these traces look similar to the high performance computing application traces generated using XPVM. The difference is that in the former, there is no clear communication pattern between the VMs, while the latter typically exhibits well known communication patterns such as two-dimensional bus and two-dimensional ring topologies.

Figure 8.21 shows the benefits of the different adaptation schemes in such a scenario. We see that algorithms CLB, CLB + OHR and CLB + WBLP perform the best. These algorithms optimize the computation time and also attempt to locate paths that find maximum bandwidth between communicating source and destination VM pairs. However, we notice that the benefits obtained by these adaptation schemes is not significant as compared to the case with no adaptation. We explain this as follows, by the time the adaptation engine decides to perform migration of certain VMs and completes the same, a significant lifetime of the application has completed. After this though the application is executing on an improved mapping, its lifetime is much shorter than its lifetime while executing on the previous mapping and routing. Latency centric algorithms such as GOELM + STLP and GOPLM + STLP are significantly effective.

8.6.8 Real application trace from the patterns application

Patterns belongs to the high performance computing application class, which was used to devise the objective functions and then to design the adaptation algorithms. Hence, the adaptation schemes are expected to perform well in general, designating this scenario as a biased scenario. However, since we have already presented evaluation of the adaptation

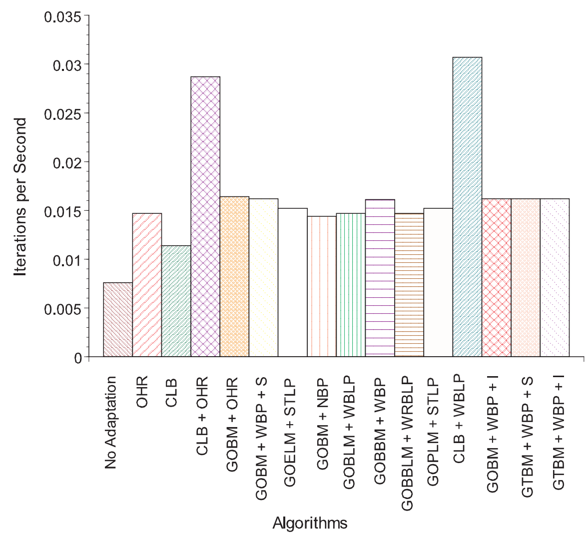


Figure 8.22: Performance of adaptation schemes for a real patterns application trace.

schemes for seven different application classes that were not used in the design of the algorithms, we argue that we have effectively closed the evaluation loop. Patterns is an application for which we have been able to collect a real trace (as opposed to synthetic traces for the previously evaluated classes of application), we present our evaluation of the adaptation schemes in the context of patterns. In other words, we expect the adaptation schemes to perform well for patterns, here we check to see if that is indeed the case. In this particular scenario, patterns is configured with an all-to-all communication topology and is executing among 18 VMs distributed geographically across six sites. The compute to communicate ratio is one. The application specific metric used was the number of iterations completed per second.

Figure 8.22 shows the performance of the different adaptation schemes in this context. At the outset we see that almost all the algorithms improve the performance of patterns as opposed to the case with no adaptation by at least a factor of two. Algorithms CLB + OHR and CLB + WBLP improve performance by more than a factor of three. The adaptation scheme CLB improves performance by the least amount. CLB does not optimize for communication and since the compute to communication ratio is one, its effects on performance improvement are somewhat limited. Overall, as expected, all the adaptation schemes significantly improve application performance.

8.7 Summary, recommendations and taxonomies

This dissertation answers the question, is there a single adaptation scheme that is effective for a wide range of distributed applications? In this section we provide the answer to this question based on all the analysis, evaluations and simulations conducted. We also present two application taxonomies and recommendations for performing adaptation for distributed applications executing in virtual environments.

8.7.1 Answer to the question posed in this dissertation

The answer to the question, is there a single optimization scheme that is effective for a range of distributed applications, is provided here. Based on all the studies conducted and evidence presented in this dissertation, we state that the CLB + WBLP adaptation scheme has wide applicability among the ten different application classes studied. In addition to the empirical results, we also provide intuitive justification for the same.

Empirically, we have seen that in seven of the ten (hence 70%) application classes studied, the CLB + WBLP adaptation scheme was either the best or very close to the best. These seven classes of applications were high performance computing applications, DNS type applications, enterprise mail applications, enterprise compute intensive simulations, both the enterprise database categories and the engineering computation applications. CLB + WBLP does not perform well for applications that fall in the enterprise backup category. The reason for this is that the specific metric being optimized, reducing bottlenecks and leaving maximum scope for other applications to enter and thrive in the system. CLB + WBLP is an application centric adaptation scheme. GOBM + NBP, on the other hand is system centric adaptation scheme, whose aim is more in line with the objective function most relevant for enterprise backup applications. CLB + WBLP is also not suited for wide-area file transfer that are common in some enterprises. Pure bandwidth centric adaptation schemes such as GOBM + WBP + S, GOBM + WBP + I, GTBM + WBP + S and GTBM + WBP + I perform the best. The reason for this behavior is that though CLB + WBLP has a bandwidth focus, it also optimizes for other properties such as compute times and path latencies. These dimensions shift the eventual mapping from those that provide the best end-to-end throughput. For the web transactional e-commerce application class, CLB + WBLP is effective, but not as good as some of the other algorithms such as GOELM + STLP and GOPLM + STLP. The reasoning for this follows the same line of reasoning as for the enterprise wide-area file transfer class. However, even for this category, when we create a scenario with external load imbalance, physical experiments illustrate that algorithms that optimize for multiple properties are most effective.

We provide the following intuitive reasons for why CLB + WBLP has the widest applicability. CLB + WBLP optimizes for multiple properties. It attempts to move computation to hosts which are faster and make available higher utilization rates. Further, it tries in a greedy fashion to locate high bandwidth links that also have low latencies. We found that majority of the application classes have multiple dependencies on optimization of different compute and network properties. Intuitively, it seems logically that case, that CLB + WBLP would have the widest applicability. The empirical evaluation backs this intuition and illustrates that the specific algorithms driving CLB + WBLP, though sub-optimal, are indeed effective in moving computation to lightly loaded hosts and in locating high bandwidth and low latency paths. Finally, it should be noted again that CLB + WBLP comes with three tunable parameters which can be tweaked to better suit different scenarios.

Single scheme works	Require application interaction	Single scheme does not work
High performance scientific	Orchestrated web services	Transactional web ecommerce
DNS based	Orchestrated database	Enterprise backup
Enterprise mail	Application specific overlays	Enterprise file transfers
Enterprise simulations	Set of batch parallel applications	Security centric
Enterprise DB (small transfers)		
Enterprise DB (large transfers)		
Enterprise Engg. computing		

Table 8.2: Taxonomy of application classes based on our recommended single adaptation scheme.

Application classes	Computation	Bandwidth	Latency
High performance scientific applications	Yes	Yes	No
Transactional web e-commerce applications	No	Yes	Yes
Enterprise backup applications	Yes	Yes	No
DNS	No	No	Yes
Enterprise mail applications	No	Yes	Yes
Enterprise wide-area file transfers	No	Yes	No
Enterprise compute-intensive simulations	Yes	No	No
Enterprise database applications with small transfers	Yes	Yes	No
Enterprise database applications with large transfers	Yes	No	Yes
Enterprise engineering computing applications	Yes	Yes	Yes

Table 8.3: Taxonomy of application classes based on resource requirements.

8.7.2 Application taxonomies

Based on our studies and insights gained from our discussions with Intel IT we present two application taxonomies in Tables 8.2 and 8.3. Table gives a taxonomy of applications categorizing them in one of the following three categories.

1. Application classes for which my single optimization scheme is most effective. All the applications in this category have been extensively studied. It should be noted that this category contains the largest number of application classes.

2. Application classes for which automatic adaptation invisible to the application is not suitable unless the adaptation scheme is provided some application specific details up front and during execution. Due to the lack of automatic and application invisible adaptation schemes, many enterprise applications are built with specific orchestrators to perform dynamic adaptation and improve performance. One approach to these applications is to say that in the presence of my adaptation system, such applications no longer need to be constructed with orchestrators. However, keeping in mind the legacy applications already built with such orchestrators and which might execute inside of Virtuoso, there needs to be some means of a dialog between the application specific orchestrator and Virtuoso's adaptation engine. Adaptation at two levels can be mutually destructive and the dialog would be an to attempt to make this dialog constructive. This category also includes the class of applications that we studied in Chapter 6, namely, multiple batch parallel applications executing in tightly coupled computing environment. In such a setting, it may not always be possible to infer resource demands as they need to be supplied by humans and hence cannot be automatically inferred.

3. Application classes for which my suggested single adaptation scheme does not work. Though I have not empirically evaluated security centric applications, I have discussed the same with Intel IT Research in the context of my adaptation schemes. Applications that spend a considerable amount of time performing multiple authentications care about packet loss rates. The adaptation schemes discussed in this dissertation do not directly take into account packet loss rates along the different links and paths.

Table 8.3 provides a taxonomy of applications based on their demands for computational resources, network bandwidth and latency. This taxonomy was based on our discussions with Intel IT Research. A preliminary version of such analysis has been previously published by Intel Corporation [15]. It should be noted that applications in many of these application classes can operate in multiple modes, where each mode makes different resource demands. For example, DNS type applications have been classified as mostly caring about latency. However, as discussed previously, bandwidth becomes the bottleneck in the context of DNS zone transfers. Further, it should be noted that the CLB + WBLP adaptation scheme covers all these three aspects and hence has the widest applicability among distributed application classes.

8.7.3 Adaptation recommendations

Finally, we would like to make some recommendations which can be used as a rule of thumb when performing adaptation for distributed applications executing in virtual executing environments. In particular when faced with the challenging task of performing adaptation for applications whose resource demands are not supplied by the application, user or developer, we recommend the use of the CLB + WBLP adaptation scheme. This scheme optimizes for lower execution times, and high bandwidth, low latency links. Further, as described in Chapter 7, it comes with three tunable parameters, which can be tweaked to better suit individual scenarios. If the application in question belongs to an application class for which CLB + WBLP is known to be ineffective, then based on, taxonomy in Table 8.3 and some additional information about the application (obtained from previous dry runs), one of the other fifteen adaptation schemes can be used.

Chapter 9

Conclusions

This dissertation recommends building virtual execution environments consisting of operating system level virtual machines connected by virtual networks as a means to realize the full potential of wide-area distributed computing. Currently, wide-area distributed computing is plagued by unnecessary complexity, lack of isolation and security, provisioning issues and challenges in developing applications for such environments. Virtuoso is the prototype virtualized distributed adaptive system jointly developed as part of this dissertation that addresses the current issues obstructing the wide-spread adoption of this computing paradigm.

The core of this dissertation describes the design, implementation and evaluation of adaptation mechanisms, optimization objective functions and heuristic algorithms that provide for an automatic, run-time and dynamic adaptation scheme that leverages the powerful paradigm of virtualization and is effective for a range of unmodified distributed applications running on unmodified operating systems without requiring any developer or user help.

This dissertation answers the question, is there a single adaptation scheme that is effective for a range of distributed applications. In the context the ten application classes studied, we found a single adaptation scheme with tunable parameters that performs VM migrations based on compute requirements and demands, and modifies the overlay topology and routes to meet latency and bandwidth requirements works well for the majority of the application classes studied (seven out of ten).

In response to complexity and inapproximability results for the adaptation problem, we explored the space of greedy heuristic driven adaptation algorithms as solutions to it. We used three of the ten application classes to design fifteen adaptation schemes and the remainder seven to evaluate the adaptation schemes. Parts of this study were carried out in collaboration with Intel Corporation's Corporate Technology Group and IT Innovation and Research Group.

This work has the potential to fill an important gap in distributed systems research. Building virtualized adaptive distributed computing systems addresses the main issues with traditional wide-area distributed computing and will hopefully lead to faster and widespread adoption of this powerful computing paradigm.

In the remainder of this chapter, we summarize the steps and contributions of this dissertation, discuss related work and identify directions for future research.

9.1 Summary and contributions

In this section we describe the contributions made in this dissertation. We also describe the insights gained along the way. Among other things, the contributions include a physical working distributed system, optimization models, algorithms, system simulator, evaluations and artifacts.

- **VNET:** I co-designed and co-implemented VNET, an Ethernet layer virtual network tool that creates and maintains the networking illusion, that the user's VMs are on the user's local area network (Chapter 3).

- **Language for describing VNET topology:** I have designed a language for describing the VNET topology and forwarding rules. I defined a context free grammar for the language (Chapter 3, [167]).

- **Parser for the VNET language:** I implemented a parser to parse descriptions in the VNET language. I have also implemented a variant of the parser that takes in high-level user requirements and generates a VNET topology description and associated forwarding rules in the VNET language (Chapter 3).

- **VNET tools:** I designed and implemented a set of VNET tools that perform diverse functions such as setup, reconfigurations and create visualizations of the current state of the system (Chapter 3).

- **VNET Evaluation:** I have carried out a detailed evaluation of VNET. To the best of my knowledge, this is the only existing detailed performance evaluation of such virtual network.

- **VNET performance:** I have evaluated VNET's performance with respect to throughput and latency. I carried out this evaluation for two separate cases, one with SSL encryption and one without. I compared its performance to the raw physical hardware and to a commercially available virtualization software. This was done for both LAN and WAN settings [169]. I adapted VNET for high speed networks by supporting creation of overlay links using UDP and by improving the forwarding rule lookup mechanism through a forwarding rule cache that gives us constant time lookup on average. All these enhancements have lead to a factor of three performance improvement, though there is still room for improvement (Chapter 3).

- **VNET overheads:** I have also quantified the VNET overheads such as its setup and reaction times (Chapter 3).

- **Scaling:** I have evaluated VNET to study how it scales with the increase in the number of VNET daemons and VMs serviced by them in terms of its overhead and the benefits of adaptation (Chapter 8). My important findings were, VNET scales in terms of the number of VNET links, forwarding rules and overheads with the number of VNET daemons and VMs supported by them. The cost of migration does not scale with the number of VMs in the system. The benefit of adapting the topology to the adaptation grows as the number of VMs grow.

- **Vision for adaptive environment:** We detailed the steps towards an adaptive virtual overlay network wherein un-modified applications running on top of un-modified operating systems could be adapted to available computational and network resources without the need for any user intervention [169].

- **Application independent adaptation mechanisms:** I have designed and implemented a second generation VNET, which includes support for arbitrary topologies and routing, and provides adaptive control of the overlay (Chapter 3).

- **Formalized generic adaptation problem:** I have designed a framework for formalization of the generic incarnation of the adaptation problem occurring in virtual execution environments (Chapter 4, [172]).

- **Defined a specific case of the generic adaptation problem:** I have a defined a specific case of the generic adaptation problem where some of the constraints and requirements have been relaxed and a particular objective function used as an aid to gain better understanding of the problem (Chapter 4).

- **Analyzed its computational complexity:** I have carried out an analysis of the computational complexity of the above mentioned specific problem and found it to be NP-hard (Chapter 4, [172]).

- **Analyzed the hardness of approximation:** I have carried out an analysis of the hardness of approximation for the above mentioned specific problem and have presented inapproximability results for the same (Chapter 4, [173]).

- **Adaptation metrics:** I have proposed and evaluated six different metrics for the adaptation problem in adaptive virtual environments (Chapter 7).

- **Adaptation heuristics:** I have proposed and evaluated fifteen greedy heuristic driven adaptation algorithms as solutions to the adaptation problem (Chapter 7).

- **Adaptation Evaluation:** I studied benefits of the adaptation mechanisms made available by virtual machines connected via virtual networks. I found that for different applications the effectiveness of the adaptation mechanisms varied, but overall, the combined effect was significantly enhanced application performance (Chapter 8).

- **Evaluation using applications:** I have studied the effectiveness of the adaptation mechanisms in a physical system in the context of the following two classes of application:

  - **BSP applications:** I found that for BSP applications the performance could be enhanced to up to a factor of two (Chapter 8).

  - **Multi-tier web sites:** I studied the benefits of adaptation in the context of non-parallel applications, specifically an industry benchmark for multi-tier web sites. The results indicate that considerable performance gains are possible (Chapter 8).

- **Automatic dynamic run-time optical network reservations:** To date very little work has been done on automatic network reservations based entirely on the applications needs at run time. I have shown that it is both feasible and relatively straightforward to automatically determine the necessary paths and reserve them appropriately on behalf of un-modified applications running in virtual execution environments. We showed that this considerably improved application performance (Chapter 5, [109]).

- **Integrated a passive network measurement tool with VNET:** I have shown that it is possible and feasible to monitor the performance of the underlying physical network by using the application's own traffic to automatically and cheaply probe it (Chapter 3 and Appendix B, [74]).

- **Virtuoso system simulator:** I have designed and implemented a simulator that simulates the entire Virtuoso system. It comprises of an application execution component, an application inference component, a system resource characterization component and an adaptation engine that supports plug and play adaptation algorithms. The simulator is also being used by others to study different variations of the adaptation problem (Chapter 8).

- **Answer to the question posed in the thesis statement** The most important contribution of my work is an answer to the question posed in the thesis statement in my thesis proposal [168]. I found that a single adaptation scheme with tunable parameters that performs VM migrations based on compute requirements and demands and modifies the overlay topology and routes to meet latency and bandwidth requirements works well for the majority of the application classes studied (seven out of ten) (Chapter 8).

- **Taxonomy of distributed applications:** I studied a range of distributed applications and have presented two taxonomies for the same (Chapter 8).

- **A working adaptive system:** I have built a working adaptive virtual execution environment consisting of virtual machines connected via virtual networks. In addition to executing applications, it also provides myself and others with a working setup to experiment and evaluate diverse research ideas related to adaptive virtual environments. I have also played a part in integrating the different Virtuoso components into one single working system.

9.2 Related work

In this section we discuss related work in wide-area distributed computing, virtual machine technology, virtual networks, adaptive overlay networks, virtual machine migration, measurement and inference, adaptation mechanisms and control, feedback based gang scheduling schemes, network reservations and network optimization problems.

9.2.1 Wide-area distributed computing

Recently there has been a great deal of interest in wide area distributed computing, primarily due to the substantial increase in commodity computer and network performance. This has allowed computational resources geographically distributed under different administrative domains and connected via wide area networks to be harnessed thus providing the illusion of a single unified computing resource. This is most commonly known as Grid computing [55]. Globus provides software infrastructure and services required to construct a computational grid [54, 56].

Legion is an object-based meta-system developed at the University of Virginia [69]. It provides the software infrastructure for a system of heterogeneous, geographically distributed high-performance computers to interact seamlessly. WebFlow is a computational extension of the “web” model that can act as a framework for wide-area distributed computing [2]. Its main design goal was to build a seamless framework for publishing and reusing computational modules on the web. NetSolve is a client/server application designed to solve computational science problems in a distributed environment [20]. It searches for computational resources on the network, chooses the best one available, solves the problem and returns the answer to the user. Condor is a high throughput computing environment that can deliver large amounts of processing capacity over long periods of time by harnessing large collections of distributively owned heterogeneous computing resources.

Much grid middleware and application software is quite complex. Recently, interest in using OS-level virtual machines as the abstraction for distributed computing has been growing [52, 62, 76, 97]. Our group made the first detailed case for grid computing on virtual machines [52]. Being able to package a working virtual machine image that contains the correct operating system, libraries, middleware, and application can make it much easier to deploy something new, using relatively simple middleware that knows only about virtual machines. We have been developing a middleware system, Virtuoso, for virtual machine grid computing [153]. Others have shown how to incorporate virtual machines into the emerging grid standards environment [102].

In-VIGO is a companion project to Virtuoso run by Prof. José A.B. Fortes and Prof. Renato Figueiredo at the University of Florida, Gainesville [53]. In-VIGO is a virtualization based grid computing system. It uses virtualization technologies, grid computing standards and other Internet middleware to create dynamic pools of virtual resources that can be aggregated on-demand for application-specific user-specific grid-computing [1].

9.2.2 Virtual machine technology

My work builds on operating-system level virtual machines, of which there are essentially two kinds. Virtual machine monitors, such as VMware [182], IBM's VM [86], and Microsoft's Virtual Server [127] present an abstraction that is identical to a physical machine. For example, VMWare, which we use, provides the abstraction of an Intel IA32-based PC (including one or more processors, memory, IDE or SCSI disk controllers, disks, network interface cards, video card, BIOS, etc.) On top of this abstraction, almost any existing PC operating system and its applications can be installed and run. The overhead of this emulation can be made to be quite low [52, 164]. Our work is also applicable to virtual server technology such as UML [32], Ensim [46], Denali [184], and Virtuozzo [181]. Here, existing operating systems are extended to provide a notion of server id (or protection domain) along with process id. Each OS call is then evaluated in the context of the server id of the calling process, giving the illusion that the processes associated with a particular server id are the only processes in the OS and providing root privileges that are effective only within that protection domain. In both cases, the virtual machine has the illusion of having network adaptors that it can use as it sees fit, which is the essential requirement of our work. VNET, without modification, has been successfully used with User Mode Linux [32] and the VServer extension to Linux [112].

9.2.3 Virtual networks

The Stanford Collective is seeking to create a compute utility in which "virtual appliances" (virtual machines with task-specialized operating systems and applications that are intended to be easy to maintain) can be run in a trusted environment [62, 145]. Part of the Collective middleware is able to create "virtual appliance networks" (VANs), which essentially tie a group of virtual appliances to an Ethernet VLAN. My work is similar in that I also, in effect, tie a group of virtual machines together as a LAN. However, my work differs in that the collective middleware attempts also to solve IP address and routing, while we remain completely at the link layer and push this administration problem back to the user's site. Another difference is that I target the wide area environment in which remote sites are not under our administrative control. Hence, we make the administrative requirements at the remote site extremely simple and focused almost entirely on the machine that will host the virtual machine. Finally, because the nature of the applications and networking hardware in grid computing tend to be different (parallel scientific applications running on clusters with very high speed wide area networks) from virtual appliances, the nature of the adaptation problems and the exploitation of resource reservations made possible by VNET are also different. However, I should point out that one adaptation mechanism that I have used, migration, has been extensively studied by the Collective group [147].

Perhaps closest to VNET is that of Purdue's SODA project, which aims to build a service-on-demand grid infrastructure based on virtual server technology [97] and virtual networking [98]. Similar to VANs in the Collective, the SODA virtual network, VIOLIN, allows for the dynamic setup of an arbitrary private link layer and network layer virtual network among virtual servers. In contrast, VNET works entirely at the link layer and with the more general virtual machine monitor model. Furthermore, our model has been much more strongly motivated by the need to deal with unfriendly administrative policies at remote sites and to perform adaptation and exploit resource reservations. I have conducted a detailed performance analysis for VNET such results currently do not exists, to the best of our knowledge, for VAN or VIOLIN.

IPOP [59] is a system that leverages P2P technology in a different way than its traditional use. In the past, P2P networks have been built for specific applications. IPOP allows for the creation of virtual IP networks on top of a P2P network. ViNE [179] on the other hand, builds IP overlays on top of the Internet. Its operation is similar to that of traditional VPNs, but solves some of the issues with traditional VPNs such as the VPN firewall requiring a static publicly visible IP address. VNET differs fundamentally from IPOP and ViNE in that it operates at the Ethernet level. VNET provides the abstraction of a virtual LAN, while IPOP and ViNE both build virtual IPs on top of P2P networks and IP Internet respectively.

VNET is a virtual private network (VPN [48, 66, 92]) that implements a virtual local area network (VLAN [87]) spread over a wide area using link layer tunneling [177].

9.2.4 Adaptive overlay networks

I have extended VNET to act as an adaptive overlay network [3, 18, 83, 94] for virtual machines as opposed to for specific applications. The adaptation problems introduced are in some ways generalizations (because we have control over machine location as well as the overlay topology and routing) of the problems encountered in the design of and routing on overlays [152]. Further, VNET allows us to modify the overlay topology among a user's VMs at will. A key difference between it and overlay work in the application layer multicast community [8, 12, 84] is that the VNET provides global control of the topology, which our adaptation algorithms currently (but not necessarily) assume.

WOW is a distributed system that combines virtual machine, overlay networking and peer-to-peer techniques to create scalable wide-area networks of virtual workstations for high-throughput computing [60]. WOW's approach to adaptation is fundamentally different than VNET's in that nodes join and leave the overlay in a decentralized, self-organizing fashion. Virtuoso's control system is centralized and adaptation instructions are issued from this centralized location.

9.2.5 Virtual machine migration

Virtuoso allows us to migrate a VM from one physical host to another. Much work exists that demonstrates that fast migration of VMs running commodity applications and operating systems is possible [107, 134, 137, 146], including live migration schemes with downtime on the order of a few seconds [25]. As migration times decrease, the rate of adaptation we can support and our work's relevance increases. Note that while process migration and remote execution has a long history [41, 128, 161, 176, 193], to use these facilities, we must modify or relink the application and/or use a particular OS. Neither is the case with VM migration. Although Virtuoso supports plug-in migration schemes, of which we have implemented copy using SSH, synchronization using rsync [178], and migration by transferring redo logs in a versioning file system [29]. In my work up till now, I have used rsync. I have still not decided as to the migration mechanism that I will use in my thesis.

9.2.6 Measurement and inference

At a very high level we adapt the application to the network and in the case of network reservations adapt the network to the application. In either case we need to have some means of inferring the application demands and measuring the underlying network.

I use the Virtual Topology and Traffic Inference Framework (VTTIF), developed by my fellow graduate student, Ashish Gupta [71]. VTTIF integrates with VNET to automatically infer the dynamic topology and traffic load of applications running inside the VMs in the Virtuoso system. There are many well known mechanisms to measure available compute rate of the hosts [34, 124, 189].

There is abundant work that suggests that underlying network measurements can be accomplished within or without the virtual network using both active [148, 190] and passive techniques [118, 150, 194]. In joint work, I have shown that the naturally occurring traffic of an existing, unmodified application running in VMs can be used to measure the underlying physical network [73].

9.2.7 Adaptation mechanisms and control

An application running in some distributed computing environment must adapt to the (dynamically changing) available computational and networking resources to achieve stable high performance. Over the years there have been numerous attempts at adaptation in different settings such as load balancing in networks of shared processors [30, 78, 188], solutions to workflow problems, component placement problems and support for heavy-weight applications in computational grids [13, 103, 116], adaptation, load balancing and fault tolerance in message passing and parallel processing systems spread over heterogeneous resources [5, 68, 117, 154], distributed mobile applications [133], automated runtime tuning systems [174] and extensions to commercial standards such as CORBA [196].

Despite these efforts adaptation and control mechanisms are not common in today's distributed computing environments as most of the approaches are very application-specific and require considerable user or developer effort. We have shown that adaptation using the low-level, *application-independent* adaptation mechanisms made possible by virtual machines interconnected with a virtual network is highly effective [109, 167, 171]. Furthermore, our adaptation mechanisms can be controlled automatically *without developer or user help.*

9.2.8 Feedback-based gang scheduling

Our work in Chapter 6 ties to gang scheduling, implicit co-scheduling, real-time schedulers, and feedback control real-time scheduling. As far as we aware, we are the first to develop real-time techniques for scheduling parallel applications that provide performance isolation and control. We also differ from these areas in that we show how external control of resource use (by a cluster administrator, for example) can be achieved while maintaining commensurate application execution rates. That is, we can reconcile administrator and user concerns.

The goal of gang scheduling [95, 135] is to "fix" the blocking problems produced by blindly using time-sharing local node schedulers. The core idea is to make fine-grain scheduling decisions collectively over the whole cluster. For example, one might have all of an application's threads be scheduled at identical times on the different nodes, thus giving many of the benefits of space-sharing, while still permitting multiple applications to execute together to drive up utilization, and thus allowing jobs into the system faster. In essence, this provides the performance isolation we seek, while performance control depends on scheduler model. However, gang scheduling has significant costs in terms of the communication necessary to keep the node schedulers synchronized, a problem that is exacerbated by finer grain parallelism and higher latency communication [100]. In addition, the code to simultaneously schedule all tasks of each gang can be quite complex, requiring elaborate bookkeeping and global system knowledge [162].

Implicit co-scheduling [6] attempts to achieve many of the benefits of gang scheduling without scheduler-specific communication. The basic idea is to use communication irregularities, such as blocked sends or receives, to infer the likely state of the remote, uncoupled scheduler, and then adjust the local scheduler's policies to compensate. This is quite a powerful idea, but it does have weaknesses. In addition to the complexity inherent in inference and adapting the local communication schedule, the approach also doesn't really provide a straightforward way to control effective application execution rate, response time, or resource usage.

The feedback control real-time scheduling project at the University of Virginia [23, 119, 120, 160] had a direct influence on our thinking in Chapter 6. In that work, concepts from feedback control theory were used to develop resource scheduling algorithms to give quality of service guarantees in unpredictable environments to applications such as online trading, agile manufacturing, and web servers. In contrast, we are using concepts from feedback control theory to manage a tightly controlled environment, targeting parallel applications with collective communication.

There are a wide range of implementations of periodic real-time schedulers [24, 43, 99, 132, 138], including numerous kernel extensions for Linux [42, 81, 88, 149, 192]. The theory of periodic real-time scheduling dates to the 70s [115].

9.2.9 Network reservations

Much work has been done on simulating distributed applications and their communication behavior. Tools such as GridSim [165], SimGrid [21], and Prophesy [175] were developed by the grid community to model an application's behavior with the goal of understanding its computational and communication requirements. Using these models network reservations can be made before the application starts, using simulation results as predictors for network traffic requirements. Our system of Chapter 5 provides a true *run-time* reservation service that does not require any application simulations. Our system also alleviates the requirement that the user explicitly request advance reservations on behalf of the application.

Run-time adaptation of optical networks to ISP level traffic has been previously demonstrated [64]. Our work takes place at the opposite end of the spectrum; we measure and adapt for individual applications. The other work also treats the optical network as a closed topology, most often seen in the backbone infrastructure of large ISPs. The network topology was modified to reach an optimized state by measuring flow characteristics over the entire ISP. Our project complements this work because we simply make reservations on an optical network and do not care about the physical topology, so long as our bandwidth and latency requirements are met, while their work demonstrates a method of optimizing the physical topology to better meet collective demands.

Advance reservations [158] can be incorporated into optical and other kinds of networks to enhance application and network performance. VRESERVE can easily coexist with advance reservations because on-demand reservation requests are a special case of advance reservations [50]. An early version of VRESERVE specifically targeted operation with deferred reservation requests. While VRESERVE is able to accept deferred reservations, it is unable to make advance reservation decisions as it would have to predict, not just measure, application demands, a service that we have not yet developed

To the best of our knowledge no previous work exists that demonstrates on-demand run-time reservations for unmodified applications. Our system alleviates the need for application developers and/or users to directly interface with the reservation system. Reservation requests are made at run time and are dependent on the applications current communication requirements.

9.2.10 Network optimization problems

I have formulated the generic adaptation problem in virtual environments that aims to maximize the sum of the residual bottleneck bandwidths over all the mapped paths. A lot of related work exists in optimizing network flows. The closest work to mine is the unsplittable-flow problem (UFP) [104]. One of the motivations for formulating UFP is to address the problem of allocating bandwidth for traffic with different bandwidth requirements in heterogeneous networks. The UFP is MAXSNP-hard [75]. Approximation algorithms for the UFP and related problems have been presented in several prior works [10, 75, 104, 106]. Kleinberg [104] provides a comprehensive background on these problems. Baveja and Srinivasan [10] provided a $O(\sqrt{m})$, where m is the number of edges in the graph, approximate algorithm by an LP-based algorithm. Azar and Regev [7] gave a simpler, combinatorial algorithm with the same approximation guarantee. Kolliopoulos and Stein [106] presented the first nontrivial approximation, $O(logm\sqrt{m})$, for a more general version of the UFP in which each request has an associated profit p_i and the goal is to maximize the total profit of accepted requests (UFP with profits). Further, my adaptation problem also has a strong connection to parallel task graph mapping problems [14, 108]. To the best of my knowledge no prior theoretical works exists that includes both the mapping and network flow components.

9.3 Future work

This dissertation describes the design, implementation and evaluation of automatic, run-time and dynamic adaptation of un-modified applications executing in virtualized distributed computing environments. At a high level we intend to build upon this dissertation by generalizing Virtuoso to be applicable to an even larger class of applications than it already is.

9.3.1 Making VNET faster

As detailed in this dissertation, VNET has no virtualization overhead in a 100Mbit LAN environment and that it only slightly trails the leading commercial virtual network software. However, both VNET and commercial softwares significantly lag behind the performance of the raw hardware couple with standard software such as TCP/IP. There are a number of applications which demand throughput in Gigabytes as opposed to Megabytes, for Virtuoso to be applicable in such environments, it is essential that we further increase VNET's performance. It should be noted that currently VNET operates completely at user level. We can significantly speed up VNET by moving the forwarding core of VNET into the Linux kernel on the host to avoid context switches in their entirety. Another direction to explore would be writing a device driver for use inside the VM that will more efficiently deliver data to VNET.

9.3.2 Uncoupling application component schedules

Another direction of future work is to generalize the global controller in our feedback based control system. In our current design, we make the assumption that each component of an application has the same CPU utilization in the absence of any scheduling. However, there are many applications such as the NAS benchmarks, wherein, the different application components have different CPU utilizations. We are currently designing a system in which the global controller is given the freedom to set a different schedule on each node thus making our control system more flexible.

9.3.3 Widening the scope of adaptation

This dissertation studies automatic, run-time and dynamic adaptation. But as we reported this scheme, though applicable to a range of application classes, is not applicable to all application classes. It would be interesting to study if involving the user in the adaptation loop will help solve problems that are suitably targeted by automatic adaptations. In such a scenario the adaptation would be driven by the user. In this model it is the user who gives feedback to the application and directs the control system. Lin et al. have done some preliminary work to study the effectiveness of user (human) driven adaptation [36, 72, 110, 121].

There already exists a large class of distributed applications which have an adaptation engine built into it. For the wide-spread adoption of automatic adaptation, there has to be some means of reconciling the adaptive system with the orchestrated applications. This is a very challenging problem, but one that has to be faced and solved. One possible approach to this problem is to open up a dialog between the two, in particular to provide some means for the adaptive system to provide monitoring data to the application. Further, the application will also need to be provided with some means of telling the adaptive system to not perform adaptation on its behalf.

9.3.4 Combinatorially approximate solutions

Another direction is to explore combinatorially approximate algorithms with a view to improving the adaptation quality while reducing the algorithm computational time. What is missing in the greedy heuristic solutions proposed in this chapter, is some form of performance guarantees. It would be interesting to explore efficient approximate solutions for which we can provide some performance guarantees.

9.3.5 Constraint-based approach to adaptation

Another approach to adaptation is to create a constraint based hierarchical system wherein the application provides all the constraint and the system attempts to either meet the same or notify the application upon failure. Intel Corporation is currently designing such a system [15, 90]. Such an approach creates a trade-off between the ease of feasibility of adaptation versus the benefits obtained from performing the adaptations.

Bibliography

[1] Sumalatha Adabala, Vineet Chadha, Puneet Chawla, Renato Figueiredo, Jose A. B. Fortes, Ivan Krsul, Andrea Matsunaga, Mauricio Tsugawa, Jian Zhang, Ming Zhao, Liping Zhu, and Xiaomin Zhu. From virtualized resources to virtual computing grids: the In-VIGO system. *Future Gener. Comput. Syst.*, 21(6):896–909, 2005.

[2] Erol Akarsu, Geoffrey C. Fox, Wojtek Furmanski, and Tomasz Haupt. Webflow: high-level programming environment and visual authoring toolkit for high performance distributed computing. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–7, Washington, DC, USA, 1998. IEEE Computer Society.

[3] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, 2001.

[4] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Symposium on Operating Systems Principles*, pages 131–145, 2001.

[5] J. Arabe, A. Beguelin, B. Lowekamp, M. Starkey E. Seligman, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CMU-CS-95-137, Carnegie Mellon University, School of Computer Science, April 1995.

[6] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan Mainwaring. Scheduling with implicit information in distributed systems. In *ACM Sigmetrics*, 1998.

[7] Yossi Azar and Oded Regev. Strongly polynomial algorithms for the unsplittable flow problem. In *Proceedings of the 8th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 15–29, London, UK, 2001. Springer-Verlag.

[8] Suman Banerjee, Seungjoon Lee, Bobby Bhattacharjee, and Aravind Srinivasan. Resilient multicast using overlays. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2003.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[10] Alok Baveja and Aravind Srinivasan. Approximation algorithms for disjoint paths and related routing and packing problems. *Mathematics of Operations Research*, 25(2):255–280, 2000.

[11] Peter A. Dinda Bin Lin, Ananth I. Sundararaj. Time-sharing parallel applications with performance isolation and control. under submission.

[12] Stefan Birrer and Fabian Bustamante. Nemo: Resilient peer-to-peer multicast without the cost. In *Proceedings of the 12th Annual Multimedia Computing and Networking Conference*, January 2005.

[13] Jim Blythe, Ewa Deelman, Yolanda Gil, Carl Kesselman, Amit Agarwal, Gaurang Mehta, and Karan Vahi. The role of planning in grid computing. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

[14] S. Bollinger and S. Midkiff. Heuristic techniques for processor and link assignment in multicomputers. *IEEE Transactions on Computers*, 40(3), March 1991.

[15] Mic Bowman, Paul Brett, Patrick Fabian, Cheng-Chee Koh, Rob Knauerhase, Julia Palmer, Justin Richardson, Sanjay Rungta, Jeff Sedayao, and John Vicente. Virtualization in the enterprise. *Intel Technology Journal*, August 2006.

[16] A. R. Butt, S. Adabala, N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes. Fine-grain access control for securing shared resources in computational grids. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.

[17] Ali Raza Butt, Rongmei Zhang, and Y. Charlie Hu. A self-organizing flock of condors. In *Proceedings of ACM/IEEE SC 2003 (Supercomputing)*.

[18] A. Campbell, M. Kounavis, D. Villela, J. Vicente, H. De Meer, K. Miki, and K. Kalaichelvan. Spawning networks. *IEEE Network*, pages 16–29, July/August 1999.

[19] Canarie Inc. Lighting the Path to Innovation, Annual Technical Report, 2003 - 2004.

[20] Henri Casanova and Jack Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[21] Henri Casanova, Arnaud Legrand, and Loris Marchal. Scheduling distributed applications: the SimGrid simulation framework. In *Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, May 2003.

[22] Vincent W. S. Chan, Katherine L. Hall, Eytan Modiano, and Kristin A. Rauschenbach. Architectures and technologies for high-speed optical data networks. *Journal of Lightwave Technology*, 16(12):2146–2168, December 1998.

[23] Tarek F. Abdelzaher Gang Tao Sang H. Son Michael Marley Chenyang Lu, John A. Stankovic. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of 21st IEEE Real-Time Systems Symposium*, 2000.

[24] Hao-Hua Chu and Klara Narhstedt. CPU service classes for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, June 1999.

[25] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005.

[26] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of NSDI*, 2005.

[27] D. Clark. Industry trends: Face-to-face with peer-to-peer networking. *COMPUTER*, 34(1):18–21, 2001.

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, Cambridge, Massachusetts, 2001.

[29] Brian Cornell, Peter Dinda, and Fabian Bustamante. Wayback: A user-level versioning file system for linux. In *Proceedings of USENIX 2004 (Freenix Track)*, July 2004.

[30] G. Cybenko. Dynamic load balancing for distributed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, October 1989.

[31] DELL Inc. http://www.dell.com.

[32] J. Dike. A user-mode port of the linux kernel. In *Proceedings of the USENIX Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[33] Andreas Dilger, Jakob Flierl, Lee Begg, Matthew Grove, and Francois Dispot. The PVM Patch for POV-Ray. Available at http://pvmpov.sourceforge.net.

[34] Peter A. Dinda. Online prediction of the running time of tasks. *Cluster Computing*, 5(3), 2002. Earlier version appears in HPDC 2001. Summary in SIGMETRICS 2001.

[35] Peter A. Dinda. Design, implementation, and evaluation of an extensible toolkit for resource prediction in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(2), February 2006.

[36] Peter A. Dinda and Bin Lin. Towards scheduling virtual machines based on direct user input. In *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing (VTDC 06), in conjunction with Supercomputing*, November 2006.

[37] Peter A. Dinda and Dong Lu. Nondeterministic queries in a relational grid information service. In *Proceedings of ACM/IEEE Supercomputing 2003 (SC 2003)*, November 2003.

[38] Peter A. Dinda and David R. O'Hallaron. An extensible toolkit for resource prediction in distributed systems. Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.

[39] Peter A. Dinda and David R. O'Hallaron. Realistic CPU workloads through host load trace playback. In *Proc. of 5th Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers (LCR2000)*, May 2000. To appear.

[40] Distributed Optical Testbed. http://www.dotresearch.org.

[41] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems (ICDCS)*, September 1987.

[42] L. Dozio and P. Mantegazza. Real-time distributed control systems using rtai. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2003.

[43] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *ACM SOSP*, pages 261–276. ACM Press, 1999.

[44] K. Egevang and P. Francis. The ip network address translator (nat). Technical Report RFC 1631, Internet Engineering Task Force, May 1994.

[45] D. W. Embley and G. Nagy. Behavioral aspects of text editors. *ACM Computing Surveys*, 13(1):33–70, January 1981.

[46] Ensim Corporation. http://www.ensim.com.

[47] Anja Feldmann, Thomas Stricker, and Thomas Warfel. Supporting sets of arbitrary connections on iWarp through comminication context switches. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, 1993.

[48] P. Ferguson and G. Huston. What is a vpn? Technical report, Cisco Systems, March 1998.

[49] D. Ferrari, A. Banerjea, and H. Zhang. Network support for multimedia - a discussion of the tenet approach. *Computer Networks and ISDN Systems*, 26(10):1167–1180, July 1994.

[50] S. Figueira, N. Kaushik, S. Naiksatam, S. A. Chiapparic, and N. Bhatnagar. Advanced reservation of lightpaths in optical-network based grids. In *Proceedings of ICST/IEEE Gridnets*, October 2004.

[51] Renato Figueiredo, Peter Dinda, and Jose Fortes. Resource virtualization renaissance. *IEEE Computer Special Issue On Resource Virtualization*, 38(5):28–31, 2005.

[52] Renato Figueiredo, Peter A. Dinda, and Jose Fortes. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, May 2003.

[53] Renato Figueiredo and Jose A. B. Fortes. In-VIGO system. http://invigo.acis.ufl.edu.

[54] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[55] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15, 2001.

[56] Ian Foster. Globus web page. http://www.mcs.anl.gov/globus.

[57] Ian Foster and Carl Kesselman, editors. *The Grid2, Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2nd edition, 2004.

[58] Editor G. Vigna. Mobile agents and security. *Springer-Verlag Lecture Notes in Computer Science*, 1419, June 1998.

[59] Arijit Ganguly, Abhishek Agrawal, P. O. Boykin, and Renato Figueiredo. IP over P2P: Enabling self-configuring virtual ip networks for grid computing. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2006.

[60] Arijit Ganguly, Abhishek Agrawal, P. Oscar Boykin, and Renato Figueiredo. WOW: Self-organizing wide area overlay networks of virtual workstations. In *Proceedings of the Fourteenth International Symposium on High Performance Distributed Computing (HPDC)*, 2006.

[61] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database Systems: The Complete Book*. Prentice Hall, October 2001.

[62] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, October 2003.

[63] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.

[64] Aysegul Gencata and Biswanath Mukherjee. Virtual-topology adaptation for WDM mesh networks under dynamic traffic. *IEEE/ACM Transactions on Networking*, 11(2):236–247, 2003.

[65] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[66] B. Gleeson, A. Lin, J Heinanen, G. Armitage, and A. Malis. A framework for IP-based virtual private networks. Technical Report RFC 2764, Internet Engineering Taskforce, February 2000.

[67] R. Goldberg. Survey of virtual machine research. *IEEE Computer*, pages 34–45, June 1974.

[68] A. S. Grimshaw, W. T. Strayer, and P.Narayan. Dynamic object-oriented parallel processing. *IEEE Parallel and Distributed Technology: Systems and Applications*, (5):33–47, May 1993.

[69] Andrew Grimshaw, William Wulf, and the Legion Team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), 1997.

[70] Ashish Gupta and Peter A. Dinda. Infering the topology and traffic load of parallel programs running in a virtual machine environment. In *Proceedings of the 10th Workshop on Job Scheduling Policies for Parallel Program Processing(JSPPP)*, June 2004.

[71] Ashish Gupta and Peter A. Dinda. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, June 2004.

[72] Ashish Gupta, Bin Lin, and Peter A. Dinda. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC 2004)*, June 2004. To Appear.

[73] Ashish Gupta, Marcia Zangrilli, Ananth I. Sundararaj, Peter Dinda, and Bruce Lowekamp. Free network measurement for adaptive virtualized distributed computing. Technical Report NWU-CS-05-13, Northwestern University, June 2005.

[74] Ashish Gupta, Marcia Zangrilli, Ananth I. Sundararaj, Anne Huang, Peter A. Dinda, and Bruce B. Lowekamp. Free network measurement for adaptive virtualized distributed computing. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[75] Venkatesan Guruswami, Sanjeev Khanna, Rajmohan Rajaraman, Bruce Shepherd, and Mihalis Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. *Journal of Computer and System Sciences*, 67(3):473–496, 2003.

[76] Steven Hand, Tim Harris, Evangelos Kotsovinos, and Ian Pratt. Controlling the xenoserver open platform. In *Proceedings of OPENARCH 2003*, April 2003.

[77] Andreas Hansson, Kees Goossens, and Andrei Radulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *Proceedings of the 3rd IEEE/ACM/IFIP CODES+ISSS*, 2005.

[78] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of ACM SIGMETRICS '96*, pages 13–24, May 1996.

[79] Morris Marden Harold W. Cain, Ravi Rajwar and Mikko H. Lipasti. An architectural evaluation of Java TPC-W. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

[80] G. Hoo, W. Johnston, I. Foster, and A. Roy. Qos as middleware: Bandwidth reservation system. In *Proceedings of the 8th IEEE Symposium on High Performance Distri buted Computing*, pages 345–346, 1999.

[81] Sean House and Douglas Niehaus. Kurt-linux support for synchronous fine-grain distributed computations. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS)*, 2000.

[82] John Howard. An overview of the andrew file system. In *Proceedings of the Usenix Winter Conference*, 1988.

[83] Yang hua Chu, Sanjay Rao, Srinivasan Sheshan, and Hui Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of ACM SIGCOMM*, 2001.

[84] Yang hua Chu, Sanjay Rao, and Hui Zhang. A case for end-system multicast. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 2000.

[85] IBM Corporation. http://www.ibm.com.

[86] IBM Corporation. White paper: S/390 virtual image facility for linux, guide and reference. GC24-5930-03, Feb 2001.

[87] IEEE 802.1Q Working Group. 802.1q: Virtual lans. Technical report, IEEE, 2001.

[88] David Ingram and Stephen Childs. The linux-srt integrated multimedia operating system: bringing qos to the desktop. In *Proceedings of the IEEE Real-time Technologies and Applications Symposium (RTAS)*, 2001.

[89] Intel Corporation. http://www.intel.com.

[90] Intel Corporation, IT Innovation and Research. http://www.intel.com/technology/techresearch/itresearch/.

[91] International Center for Advanced Internet Research. http://www.icair.org/omninet/.

[92] G. Italiano, R. Rastogi, and B. Yener. Restoration algorithms for virtual private networks in the hose model. In *Procedings of IEEE INFOCOM*, June 2002.

[93] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991.

[94] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. OToole Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of OSDI 2000*, October 2000.

[95] Morris Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–12, 1997.

[96] Hao Jiang and Constantinos Dovrolis. Why is the internet traffic bursty in short time scales? In *SIGMETRICS 2005*. ACM, August 2005.

[97] Xuxian Jiang and Dongyan Xu. Soda: A service-on-demand architecture for application service hosting platforms. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC 2003)*, pages 174–183, June 2003.

[98] Xuxian Jiang and Dongyan Xu. Violin: Virtual internetworking on overlay infrastructure. Technical Report CSD TR 03-027, Department of Computer Sciences, Purdue University, July 2003.

[99] Michael Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *ACM SOSP*, 1997.

[100] Y. Kozakai K. Hyoudou and Y. Nakayama. An implementation of concurrent gang scheduler for pc cluster systems. In *Parallel and Distributed Computing and Networks*, 2004.

[101] R.M. Karp. *Compexity of Computer Computations*, chapter Reducibility among combinatorial problems, pages 85–103. Miller, R.E. and Thatcher, J.W. (Eds.). Plenum Press, New York, 1972.

[102] K. Keahey, K. Doering, and I. Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *Proceedings of the 5th International Workshop on Grid Computing*, November 2004.

[103] Tatiana Kichkaylo and Vijay Karamcheti. Optimal resource-aware deployment planning for component-based distributed applications. In *Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, pages 150–159, June 2004.

[104] Jon Kleinberg. *Approximation Algorithms for Disjoint Paths Problems*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1996.

[105] James Arthur Kohl and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. In *Proceedings of the 29th Annual International Conference on System Sciences*, 1996.

[106] Stavros G. Kolliopoulos and Clifford Stein. Approximating disjoint-path problems using greedy algorithms and packing integer programs. In *Proceedings of the 6th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 153–168, London, UK, 1998. Springer-Verlag.

[107] M. Kozuch, M. Satyanarayanan, T. Bressoud, and Y. Ke. Efficient state transfer for Internet suspend/resume. Technical Report IRP-TR-02-03, Intel Research Laboratory at Pittburgh, May 2002.

[108] K. Kwong and A. Ishfaq. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[109] Jack R. Lange, Ananth I. Sundararaj, and Peter A. Dinda. Automatic dynamic runtime optical network reservations. In *Proceedings of the Fourteenth International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[110] Bin Lin. *Human-directed Adaptation, Proposal*. PhD thesis, Northwestern University, Department of Electrical Engineering and Computer Science, March 2006.

[111] Bin Lin and Peter Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of ACM/IEEE SC 2005 (Supercomputing)*, November 2005.

[112] Linux Vserver Project. http://www.linux-vserver.org.

[113] M. Litzkow, M. Livny, and M. W. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS '88)*, pages 104–111, June 1988.

[114] Mike Litzkow and Miron Livny. Experience with the condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, October 1990.

[115] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[116] Julio Lopez and David O'Hallaron. Support for interactive heavyweight services. In *Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing HPDC*, 2001.

[117] Bruce Lowekamp and Adam Beguelin. Eco: Efficient collective operations for communication on heterogeneous networks. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 399–406, 1996.

[118] Bruce Lowekamp, David O'Hallaron, and Thomas Gross. Direct queries for discovering network resource properties in a distributed environment. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC99)*, pages 38–46, August 1999.

[119] Chenyang Lu, John A. Stankovic, Gang Tao, and Sang H. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Special issue of Real-Time Systems Journal on Control-Theoretic Approaches to Real-Time Computing*, 23(12):85–126, September 2002.

[120] Chenyang Lu, Xiaorui Wang, and Xenofon Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):550–561, 2005.

[121] Arindam. Mallik, Bin Lin, Peter A.Dinda, Gokhan Memik, and Robert Dick. User-driven frequency scaling. *IEEE Computer Society Computer Architecture Letters*, 2006.

[122] Joe Mambretti, Jeremy Weinberger, Jim Chen, Elizabeth Bacon, Fei Yeh, David Lillethun, Bob Grossman, Yunhong Gu, and Marco Mazzuco. The photonic terastream: Enabling next generation applications through intelligent optical networking at iGRID2002. *Future Generation Computer Systems*, 19(6):897–908, August 2003.

[123] Cao Le Thanh Man, Go Hasegawa, and Masayuki Murata. A merged inline measurement method for capacity and available bandwidth. In *Passive and Active Measurement Workshop (PAM2005)*, pages 341–344, 2005.

[124] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7), July 2004.

[125] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Prodeedings of USENIX Annual Technical Conference*, pages 259–270, 1993.

[126] Microsoft Corporation. http://www.microsoft.com.

[127] Microsoft Corporation. Virtual server beta release.

[128] Dejan Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[129] R. J. O. Figueiredo N. H. Kapadia and J. A. B. Fortes. Enhancing the scalability and usability of computational grids via logical user accounts and virtual file systems. In *Proceedings of the Heterogeneous Computing Workshop (HCW) at the International Parallel and Distributed Processing Symposium (IPDPS)*, April 2001.

[130] National LambdaRail. http://www.nlr.net.

[131] NetherLight. http://www.netherlight.nl.

[132] Jason Nieh and Monica Lam. The design, implementation, and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[133] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[134] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[135] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of ICDCS*, 1982.

[136] Noam Palatin, Arie Leizarowitz, Assaf Schuster, and Ran Wolff. Mining for misconfigured machines in grid systems. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 687–692, New York, NY, USA, 2006. ACM Press.

[137] KyoungSoo Park and Vivek S. Pai. Scale and performance in the coblitz large-file distribution service. In *Proceedings of the Third Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[138] Andreas Polze, Gerhard Fohler, and Matthias Werner. Predictable network computing. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, pages 423–431, May 1997.

[139] R.S. Prasad, M. Murray, C. Dovrolis, and K.C. Claffy. Bandwidth estimation: Metrics, measurement techniques, and tools. In *IEEE Network*, June 2003.

[140] Rajiv Ramaswami. Optical fiber communication: From transmission to networking. *IEEE Communications Magazine*, 40(6):138–147, May 2002.

[141] S. Ranjan, J. Rolia, E. Knightly, and H. Fu. Qos-driven server migration for internet data centers. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS)*, 2002.

[142] John Reumann, Ashish Mehra, Kang G. Shin, and Dilip Kandlur. Virtual services: A new abstraction for server consolidation. pages 117–130.

[143] V. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil, and L. Cottrell. pathChirp:Efficient Available Bandwidth Estimation for Network Paths. In *Passive and Active Measurement Workshop (PAM2003)*, 2003.

[144] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1), January/February 1998.

[145] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA 2003)*, October 2003.

[146] Constantine Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monic Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[147] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum:. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.

[148] Stefan Savage. Sting: A TCP-based network measurement tool. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1999.

[149] Caixue Lin Scott A. Brandt, Scott Banachowski and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of IEEE Real-Time Systems Symposium*, 2003.

[150] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared passive network performance discovery. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and System (USITS)*, 97.

[151] Srinivas Shakkottai, Nevil Brownlee, and K.C. Claffy. A study of burstiness in tcp flows. In *Passive and Active Measurement Workshop (PAM2005)*, pages 13–26, 2005.

[152] S. Shi and J. Turner. Routing in Overlay Multicast Networks. In *Proceedings of IEEE INFOCOM 2002*, June 2002.

[153] Alex Shoykhet, Jack Lange, and Peter Dinda. Virtuoso: A system for virtual machine marketplaces. Technical Report NWU-CS-04-39, Department of Computer Science, Northwestern University, July 2004.

[154] Bruce Siegell and Peter Steenkiste. Automatic generation of parallel programs with dynamic load balancing. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing (HPDC)*, pages 166–175, August 1994.

[155] D. Simeonidou, R. Nejabati, M. J. O'Mahony, A. Tzanakaki, and I. Tomkos. An optical network infrastructure suitable for global grid computing. In *Proceedings of TERENA Networking Conference*, 2004.

[156] Larry L. Smarr, Andrew A. Chien, Tom DeFanti, Jason Leigh, and Philip M. Papadopoulos. The OptIPuter. *Commun. ACM*, 46(11):58–67, 2003.

[157] Wayne D. Smith. TPC-W: benchmarking an ecommerce solution. Technical report, Intel Corporation.

[158] Quin Snell, Mark Clement, David Jackson, and Chad Gregory. The performance impact of advance reservation meta-scheduling. In *Proceedings of the 6th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 137–153, 2000.

[159] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public internet measurement facility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2001.

[160] John A. Stankovic, Tian He, Tarek F. Abdelzaher, Michael Marley, Gang Tao, Sang H. Son, and Chenyang Lu. Feedback control scheduling in distributed real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, 2001.

[161] Bjarne Steensgaard and Eric Jul. Object and native code process mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM, December 1995.

[162] Peter Strazdins and John Uhlmann. A comparison of local and gang scheduling on a beowulf cluster. In *Proceedings of the 2004 IEEE International Conference of Cluster Computing*, pages 55–62, 2004.

[163] J. Subhlok, T. Gross, and T. Suzuoka. Impact of job mix on optimizations for space sharing schedulers. In *Proceedings of Supercomputing '96*, November 1996.

[164] Jeremy Sugerman, Ganesh Venkitachalan, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.

[165] Anthony Sulistio, Gokul Poduvaly, Rajkumar Buyya, and Chen-Khong Tham. Constructing a grid simulation with differentiated network servi ce using GridSim. Technical Report GRIDS-TR-2004-13, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, December 2004.

[166] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major seti project based on project serendip data and 100,000 personal computers. In C.B. Cosmovici, S. Bowyer, and D. Werthimer, editors, *Proceedings of the Fifth International Conference on Bioastronomy*, number 161 in IAU Colloquim. Editrice Compositori, Bologna, Italy, 1997.

[167] Ananth Sundararaj, Ashish Gupta, and Peter Dinda. Dynamic topology adaptation of virtual networks of virtual machines. In *Proceedings of the Seventh Workshop on Languages, Compilers and Run-time Support for Scalable Systems (LCR)*, November 2004.

[168] Ananth I. Sundararaj. *Automatic, Run-time and Dynamic Adaptation of Distributed Application Executing in Virtual Environments, Proposal*. PhD thesis, Northwestern University, Department of Electrical Engineering and Computer Science, November 2005.

[169] Ananth I. Sundararaj and Peter A. Dinda. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research and Technology Symposium (VM 04)*, May 2004.

[170] Ananth I. Sundararaj and Dan Duchamp. Analytical characterization of the throughput of a split tcp connection. Technical report, Department of Computer Science, Stevens Institute of Technology, 2003.

[171] Ananth I. Sundararaj, Ashish Gupta, and Peter A. Dinda. Increasing application performance in virtual environments through run-time inference and adaptation. In *Proceedings of the Fourteenth International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[172] Ananth I. Sundararaj, Manan Sanghi, Jack Lange, and Peter A. Dinda. An optimization problem in adaptive virtual environments. *ACM SIGMETRICS Performance Evaluation Review*, 33(2), 2005.

[173] Ananth I. Sundararaj, Manan Sanghi, John R. Lange, and Peter A. Dinda. Hardness of approximation and greedy algorithms for the adaptation problem in virtual environments. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC)*, June 2006.

[174] Cristian Tapus, I-Hsin Chung, and Jeffrey Hollingsworth. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2002.

[175] V. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, R. S tevens, M. Hereld, and I. Judson. Prophesy: An infrastructure for analyzing and modeling the perfo rmance of parallel and distributed applications. In *Proceedings of the 9th International Symposium on High Performan ce Distributed Computing (HPDC)*, August 2000.

[176] Douglas Thain and Miron Livny. Bypass: A tool for building split execution systems. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburgh, PA, August 2000.

[177] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, and B. Palter. Layer two tunneling protocol "l2tp". Technical Report RFC 2661, Internet Engineering Task Force, August 1999.

[178] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

[179] M. Tsugawa and Jose A.B. Fortes. A virtual network (ViNe) architecture for grid computing. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[180] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), August 1990.

[181] Virtuozzo Corporation. http://www.swsoft.com.

[182] VMware Corporation. http://www.vmware.com.

[183] John Y. Wei. Advances in the management and control of optical internet. *IEEE Journal on Selected Areas in Communications*, 20(4):768–785, May 2002.

[184] Andrew Whitaker, Marianne Shaw, and Steven Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)*, December 2002.

[185] S. White, A. Alund, and V. S. Sunderam. Performance of the NAS parallel benchmarks on PVM-Based networks. *Journal of Parallel and Distributed Computing*, 26(1):61–71, 1995.

[186] S. White, A. Alund, and V. S. Sunderam. Performance of the NAS parallel benchmarks on PVM-Based networks. *Journal of Parallel and Distributed Computing*, 26(1):61–71, 1995.

[187] S. White, A. lund, and V. S. Sunderam. Performance of the nas parallel benchmarks on pvm-based networks. *J. Parallel Distrib. Comput.*, 26(1):61–71, 1995.

[188] M. Willebeek-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[189] Rich Wolski, Neil Spring, and Jim Hayes. Predicting the CPU availability of time-shared unix systems. In *Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing HPDC99*, pages 105–112. IEEE, August 1999. Earlier version available as UCSD Technical Report Number CS98-602.

[190] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: A distributed resource performance forecasting system. *Journal of Future Generation Computing Systems*, 1999.

[191] Dean Yao, Tisson Mathew, Mazin Yousif, and Sharad Garg. A framework for platform-based dynamic resource provisioning. Technical report, Technical report, Research and Development at Intel Corporation, 2004.

[192] Victor Yodaiken and Michael Barabanov. A real-time linux, 1997.

[193] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. In *Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing (HPDC)*, Redondo Beach, CA, August 1999.

[194] Marcia Zangrilli and Bruce B. Lowekamp. Using passive traces of application traffic in a network monitoring system. In *Proceedings of the Thirteenth IEEE International Symposium on High Performance Distributed Computing (HPDC 13)*. IEEE, June 2004.

[195] Marcia Zangrilli and Bruce B. Lowekamp. Applying principles of active available bandwidth algorithms to passive tcp traces. In *Passive and Active Measurement Workshop (PAM 2005)*, pages 333–336. LNCS, March 2005.

[196] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, April 1997.

Appendix A

VTTIF: Topology and Traffic Inference

The Virtual Topology and Traffic Inference Framework integrates with VNET to automatically infer the dynamic topology and traffic load of applications running inside the VMs in the Virtuoso system. In earlier work [70], Gupta et al. have demonstrated that it is possible to successfully infer the behavior of a BSP application by observing the low level traffic sent and received by each VM in which it is running. Here we describe how VTTIF's reactions can be smoothed such that adaptation decisions made on its output cannot lead to oscillations.

A.1 Operation

VTTIF works by examining each Ethernet packet that a VNET daemon receives from a local VM. VNET daemons collectively aggregate this information producing a global traffic matrix for all the VMs in the system. The application topology is then recovered from this matrix by applying normalization and pruning techniques [70]. Since the monitoring is done below the VM, it does not depend on the application or the operating system in any manner. VTTIF automatically reacts to interesting changes in traffic patterns and reports them, driving the adaptation process. Figure A.1 illustrates VTTIF.

VTTIF can accurately recover common topologies from both synthetic and application

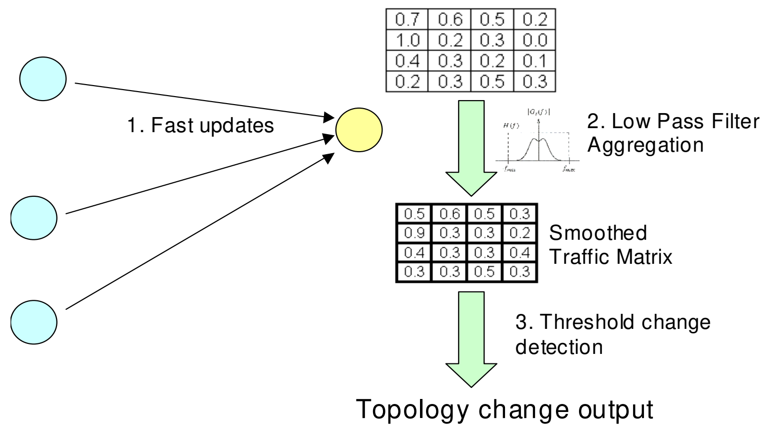


Figure A.1: An overview of the dynamic topology inference mechanism in VTTIF.

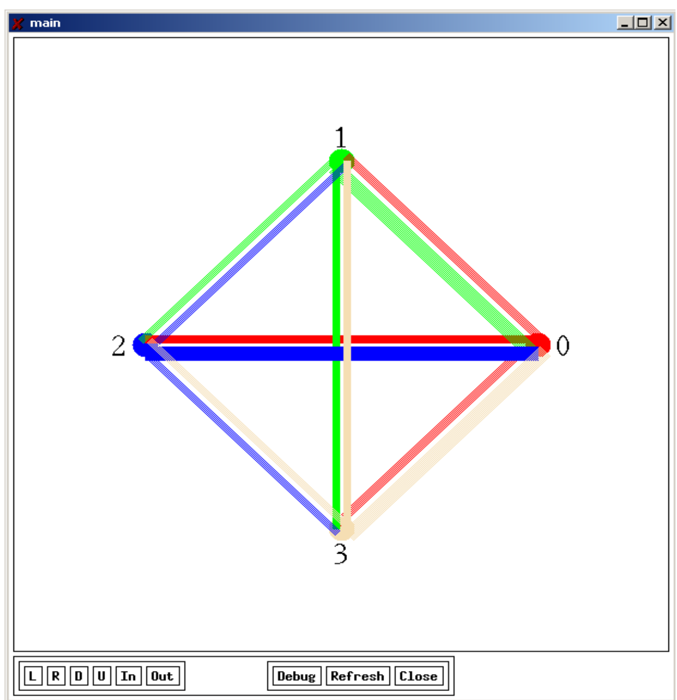


Figure A.2: The NAS IS benchmark running on 4 VM hosts as inferred by VTTIF.

benchmarks like the PVM-NAS benchmarks. For example, Figure A.2 shows the topology inferred by VTTIF from the NAS benchmark Integer Sort [186] running on VMs. The thickness of each link reflects the intensity of communication along it. VTTIF adds little overhead to VNET. Latency is indistinguishable while throughput is affected by $\sim$1%.

A.2 Performance

VTTIF runs continuously, updating its view of the topology and traffic load matrix among a collection of Ethernet addresses being supported by VNET. However, in the face of dynamic changes, natural questions arise: How fast can VTTIF react to topology change? If the topology is changing faster than what VTTIF can react to, will it oscillate or provide a damped view of the different topologies? VTTIF also depends on certain configuration parameters which affect its decision whether the topology has changed. How sensitive is VTTIF to the choice of configuration parameters in its inference algorithm?

The reaction time of VTTIF depends on the rate of updates from the individual VNET daemons. A fast *update rate* imposes network overhead but allows a finer time granularity over which topology changes can be detected. In the current VTTIF implementation, at the fastest, these updates arrive at a rate of 20 Hz. At the Proxy, VTTIF then aggregates the updates into a global traffic matrix. To provide a stable view of dynamic changes, it applies a low pass filter to the updates, aggregating the updates over a sliding window and basing its decisions upon this aggregated view.

Whether VTTIF reacts to an update by declaring that the topology has changed depends on the *smoothing interval* and the *detection threshold*. The smoothing interval is the sliding window duration over which the updates are aggregated. This parameter depends on the adaptation time of VNET, which is measured at startup, and determines how long a change must persist before VTTIF notices. The detection threshold determines if the change in

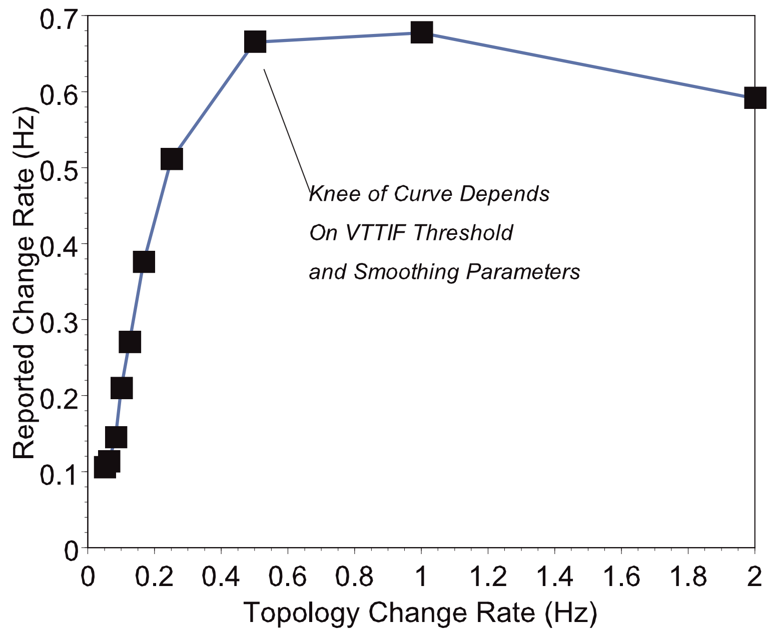


Figure A.3: VTTIF is well damped.

the aggregated global traffic matrix is large enough to declare a change in topology. After VTTIF determines that a topology has changed, it will take some time for it to settle, showing no further topology changes. The best case settle time measured is one second, on par with the adaptation mechanisms.

Given an update rate, smoothing interval, and detection threshold, there is a maximum rate of topology change that VTTIF can keep up with. Beyond this rate, VTTIF has been designed to stop reacting, settling into a topology that is a union of all the topologies that are unfolding in the network. Figure A.3 shows the reaction rate of VTTIF as a function of the topology change rate and shows that it is indeed well damped. Here, the scenario is that that application's communication rapidly keeps switching between two seperate topologies. When this topology change rate exceeds VTTIF's configured rate, the reported change rate settles and declines. The knee of the curve depends on the choice of smoothing interval and update rate, with the best case being $\sim$1 second. Up to this limit, the rate and

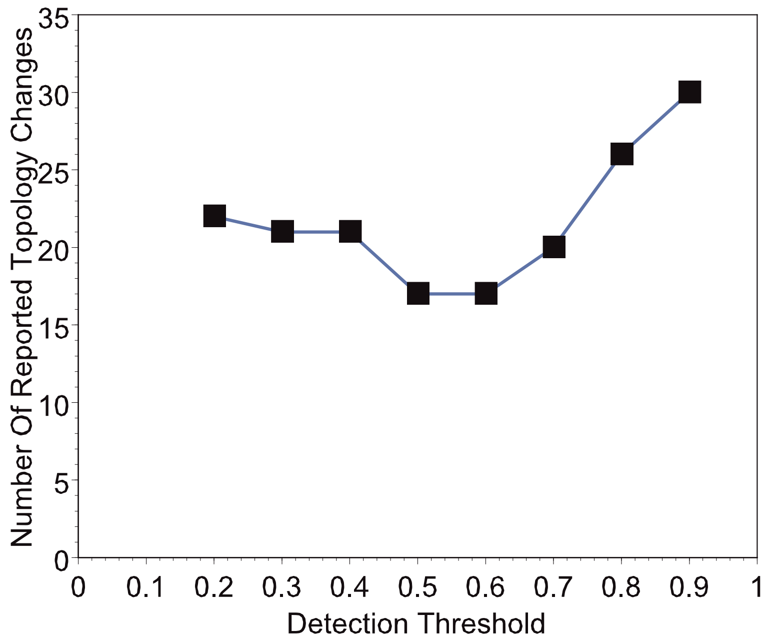


Figure A.4: VTTIF is largely insensitive to the detection threshold.

interval set the knee according to the Nyquist criterion.

VTTIF is largely insensitive to the choice of detection threshold, as shown in Figure A.4. However, this parameter does determine the extent to which similar topologies can be distinguished. Note that appropriate settings of the VTTIF parameters are determined by the *adaptation mechanisms*, not the *application*.

Appendix B

Wren online: Network resource measurment

In this chapter we describe Wren, watching resources from edge of the network, a passive network measurement tool. Wren was developed by Zangrilli et al. at the College of William and Mary. Wren has been successfully integrated with Virtuoso [73].

Many applications adapt to network performance simply by observing the throughput of their own network connections. VNET's natural abstraction of the underlying network makes such application-level adaptation more difficult because the application cannot accurately determine which network resources are in use. However, VNET is in a good position to observe an application's traffic itself. Because VNET does not alter that traffic, however, it can only observe the amount of traffic naturally generated by the application. Since there are many applications with potentially bursty and irregular communication patterns, these applications will not generate enough traffic to saturate the network and provide useful information on the current bandwidth achievable on the network.

Watching Resources from the Edge of the Network (Wren) is designed to passively monitor applications' network traffic and use those observations to determine the available bandwidth along the network paths used by the application. The key observation beind Wren is that even when the application is not saturating the network it is sending bursts

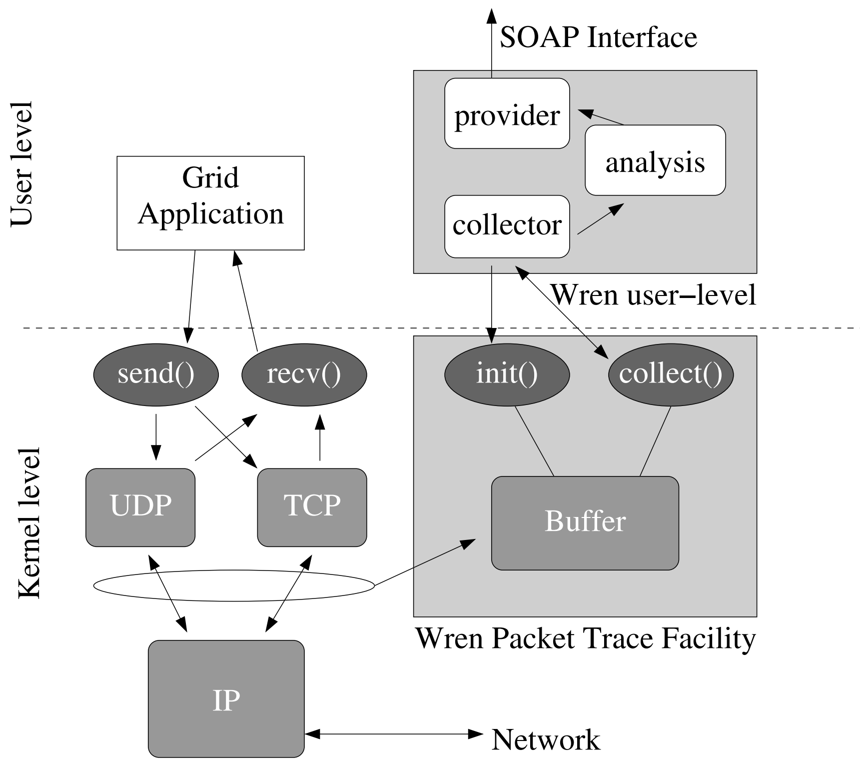


Figure B.1: Wren architecture.

of traffic that can be used to measure the available bandwidth of the network. A good example of such an application is a typical scientific computing BSP-style application that sends short messages at regular intervals. Even though the application is not using all of the available bandwidth, we can determine the available bandwidth along that path and use that information to guide adaptation.

The Wren architecture is shown in Figure B.1. The key feature Wren uses is kernel-level packet trace collection. These traces allow precise timestamps of the arrival and departure of packets on the machines. The precision of the timestamps is crucial because the passive available bandwidth algorithm relies on observing the behavior of small groups of packets on the network. A user-level component collects the traces from the kernel. Run-time analysis determines available bandwidth and the measurements are reported to other applications through a SOAP interface. Alternatively, the packet traces can be filtered for useful observations and transmitted to a remote repository for analysis.

The key observation behind Wren is that *even when the application is not saturating the network, it is sending bursts of traffic that can be used to measure the available bandwidth of the network.*

The analysis algorithm used by Wren is based on the self-induced congestion (SIC) algorithm [139, 143]. Active implementations of this algorithm generate trains of packets at progressively faster rates until increases in one-way delay are observed, indicating queues building along the path resulting from the available bandwidth being consumed. A similar analysis is applied to passively collected traces, but the key challenge is identifying appropriate trains from the stream of packets generated by the TCP sending algorithm. ImTCP integrates an active SIC algorithm into a TCP stack, waiting until the congestion window has opened large enough to send an appropriate length train and then delaying packet transmissions until enough packets are queued to generate a precisely spaced train [123]. Wren avoids modifying the TCP sending algorithm, and in particular delaying packet transmission.

The challenge Wren addresses compared to ImTCP and other active available bandwidth tools is that Wren must select from the data naturally available in the TCP flow. Although Wren has less control over the trains and selects shorter trains than would deliberately be generated by active probing, over time the burstiness of the TCP process produces many trains at a variety of rates [96, 151], thus allowing bandwidth measurements to be made. There are elements in common with TCP Vegas, Westwood, and FastTCP, but those approaches deliberately increase the congestion window until one-way delay increases, whereas we do not require the congestion window to expand until long-term congestion is observed and can detect congestion using bursts at slower average sending rates.

B.1 Online analysis

Wren's general approach, collection overhead, and available bandwidth algorithm have been presented and analyzed in previous papers [194, 195]. Wren has negligible effect on throughput, latency, or CPU consumption when collecting packet header traces. To support Virtuoso's adaptation, however, two changes are required. First, previous implementations of Wren relied on offline analysis. We describe here the online analysis algorithm used to report available bandwidth measurements using a SOAP interface. Second, Wren has previously used fixed-size bursts of network traffic. The new online tool scans for maximum-sized trains that can be formed using the collected traffic. This approach results in more measurements taken from less traffic.

The online Wren groups outgoing packets into trains by identifying sequences of packets with similar interdeparture times between successive pairs. The tool searches for maximal-length trains with consistently spaced packets and calculates the initial sending rate (ISR) for those trains. After identifying a train, the ACK return rate for the matching ACKs is calculated. The available bandwidth is determined by observing the ISR at which the ACKs show an increasing trend in the RTTs, indicating congestion on the path. This algorithm has been previously described in more detail [195].

The first step in the one-sided algorithm is to group packets into trains. The relationship between the interdeparture times of sequential data packets is then analyzed. If interdeparture times of successive pairs are similar, then the packets are departing the machine at approximately the same rate. Let Δ_0 be the interdeparture time between data packets 0 and 1. To form a train, the interdeparture times Δ_i between each successive pair of packets i and $i+1$ in the train must satisfy the requirement that $\min_i(\log(\Delta_i)) > \max_j(\log(\Delta_j)) - \alpha$, essentially requiring consistent spacing between the packets. For these experiments, we accepted trains where $\alpha = 1$. Because of the bursty transmission of packets within any TCP flow [151], interdeparture times typically vary by several orders of magnitude even during bulk data transfers, therefore this approach selects only the more consistently spaced bursts as valid trains. A minimum length of 7 packets is imposed for valid trains.

The first step in processing a train is to use a pairwise comparison test to determine the trend in the RTTs of the packets in that train. If $\forall i : RTT_i < RTT_{i+1}$ then the train has an increasing trend. Otherwise, the train trend is labeled as non-increasing. Next, the initial sending rate (ISR) is calculated by dividing the total number of bits in the train by the difference between the end time and the start time. If the train has a non increasing trend, it is known that the train ISR did not cause queuing on the path. Therefore, the ISR is reported as the lower bound available bandwidth measurement when the trend of the train is non increasing. Conversely, if a train presents an increasing trend, its ISR is an upper bound for the available bandwidth.

All available bandwidth observations are passed to the Wren observation thread. The observation thread provides a SOAP interface that clients can use to receive the stream of measurements produced using application traffic. Because the trains are short and represent only a singleton observation of an inherently bursty process, multiple observations are required to converge to an accurate measurement of available bandwidth.

B.2 Performance

The variable train-length algorithm was evaluated in a controlled-load/controlled latency testbed environment because validating measurements on real WANs is difficult due to the lack of access to router information in the WAN. For this experiment, iperf generated uniform CBR cross traffic to regulate the available bandwidth, changing at 20 seconds and stopping at 40 seconds, as shown by the dashed line of Figure B.2.

The application traffic monitored sent 20 200KB messages with 0.1 second inter-

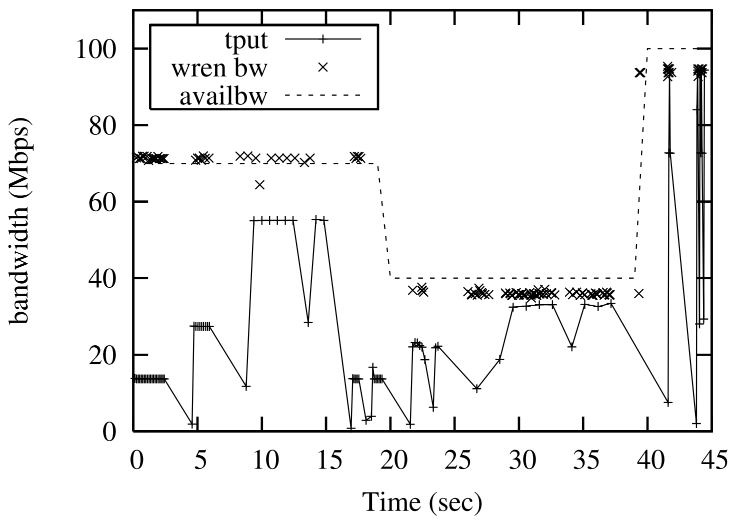


Figure B.2: Wren measurements reflect changes in available bandwidth even when the monitored application's throughput does not consume all of the available bandwidth.

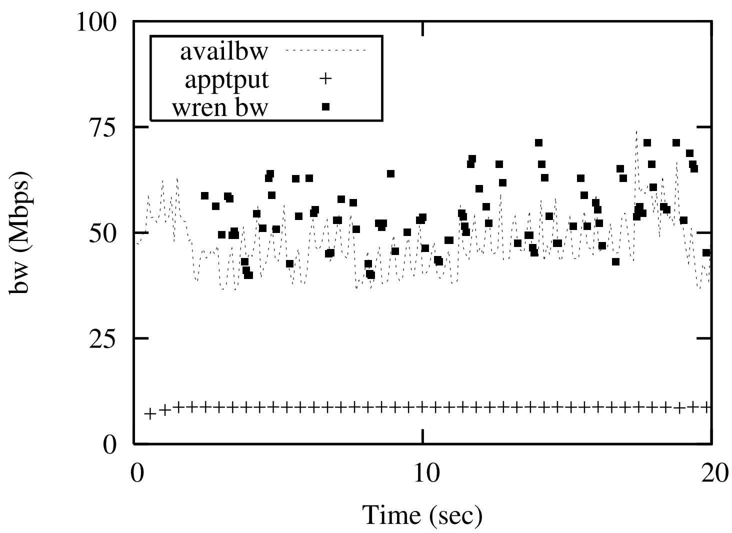


Figure B.3: Wren measurements from monitoring application on simulated WAN accurately detect changes in available bandwidth. The cross traffic in the testbed is created by on/off TCP generators.

message spacings, paused 2 seconds, 10 500KB messages with 0.1 second inter-message spacings, paused 2 seconds, and then sent 10 4MB messages with .1 second inter-message spacings. This pattern was repeated twice followed by 500KB messages sent with random inter-message spacings. The throughput achieved is shown by the solid line of Figure B.2.

In the first 40 seconds of Figure B.2, it can be seen that the throughput of the traffic generator varies according to the size of message being sent. The last 5 seconds of this graph show that the throughput of the generator also depends on the inter-message spacings. Figure B.2 shows that the algorithm produces accurate available bandwidth measurements even when the throughput of the application being monitored is not saturating the available bandwidth, as seen particularly well at the beginning and 20 seconds into the trace. The reported available bandwidth includes that consumed by the application traffic used for the measurement.

In the next experiment, a WAN environment was simulated using Nistnet to increase the latencies that the cross traffic and monitored application traffic experienced on the testbed. The on/off TCP traffic generators were used to create congestion on the path, with Nistnet emulating latencies ranging from 20 to 100ms and bandwidths from 3 to 25Mbps for the TCP traffic generators. The application traffic that was monitored sent 700K messages with 0.1 second inter-message spacing, with Nistnet adding a 50ms RTT to that path. SNMP was used to poll the congested link to measure the actual available bandwidth. Figure B.3 demonstrates how the Wren algorithm can measure the available bandwidth of larger latency paths with variable cross traffic.

It has been shown that online Wren can accurately measure available bandwidth by monitoring application traffic that does not consume all of the available bandwidth. Furthermore, Wren can be used to monitor available bandwidth on low latency LANs or high latency WANs [73].

A fixed-rate UDP stream is generated that shares the same link as the traffic between

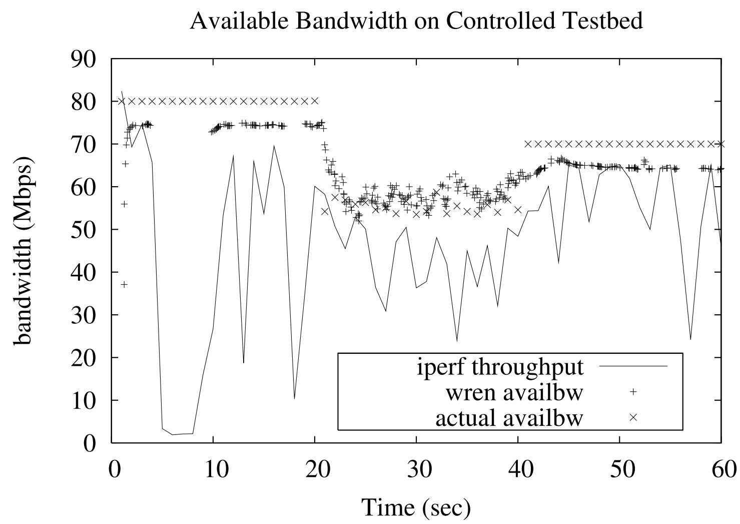


Figure B.4: The results of applying the Wren to bursty traffic generated on a testbed with a controlled level of available bandwidth.

the machines. iperf is used to generate TCP traffic on this link, reporting its throughput at 1 second intervals. Figure B.4 illustrates the resulting observations. Wren is able to measure the available bandwidth on the link throughout the experiment regardless of whether iperf is currently saturating the link.

B.3 Monitoring VNET application traffic

To validate the combination of Wren monitoring an application using VNET a simple BSP-style communication pattern generator was executed. Figure B.5 shows the results of this experiment, with the throughput achieved by the application during its bursty communication phase and Wren's available bandwidth observations. Although the application never achieved significant levels of throughput, Wren was able to measure the available bandwidth. Validating these results across a WAN is difficult, but iperf achieved approximately 24Mbps throughput when run following this experiment, which is in line with the expecta-

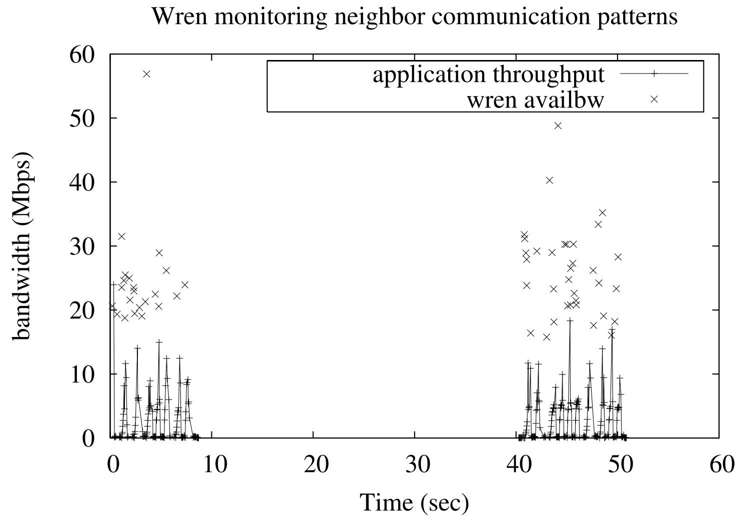


Figure B.5: Wren observing a neighbor communication pattern sending 200K messages within VNET.

tions based on Wren's observations and the large number of connections sharing W&M's 150Mbps Abilene connection.