ENHANCING REALISM AND SCALABILITY IN NETWORK TESTBEDS

by

Robert Preston Riekenberg Ricci

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

School of Computing

The University of Utah

May 2010

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Robert Preston Riekenberg Ricci

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Sneha Kumar Kasera

John Regehr

Matthew Might

John Byers

David Andersen

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Robert Preston Riekenberg Ricci in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Sneha Kumar Kasera
Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

Charles A. Wight
Dean of The Graduate School

ABSTRACT

Network emulation has become an indispensable tool for the conduct of research in networking and distributed systems. It offers more realism than simulation and more control and repeatability than experimentation on a live network. However, emulation testbeds face a number of challenges, most prominently realism and scale.

Because emulation allows the creation of arbitrary networks exhibiting a wide range of conditions, there is no guarantee that emulated topologies reflect real networks; the burden of selecting parameters to create a realistic environment is on the experimenter. While there are a number of techniques for measuring the end-to-end properties of real networks, directly importing such properties into an emulation has been a challenge. Similarly, while there exist numerous models for creating realistic network topologies, the lack of addresses on these generated topologies has been a barrier to using them in emulators.

Once an experimenter obtains a suitable topology, that topology must be mapped onto the physical resources of the testbed so that it can be instantiated. A number of restrictions make this an interesting problem: testbeds typically have heterogeneous hardware, scarce resources which must be conserved, and bottlenecks that must not be overused. User requests for particular types of nodes or links must also be met. In light of these constraints, the network testbed mapping problem is NP-hard. Though the complexity of the problem increases rapidly with the size of the experimenter's topology and the size of the physical network, the runtime of the mapper must not; long mapping times can hinder the usability of the testbed.

This dissertation makes three contributions towards improving realism and scale in emulation testbeds. First, it meets the need for realistic network conditions by creating Flexlab, a hybrid environment that couples an emulation testbed with a live-network testbed, inheriting strengths from each. Second, it attends to the need for realistic topologies by presenting a set of algorithms for automatically annotating generated topologies with realistic IP addresses. Third, it presents a mapper, `assign`, that is capable of assigning experimenters' requested topologies to testbeds' physical resources in a manner that scales well enough to handle large environments.

For Jay.

CONTENTS

CHAPTERS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

Jay Lepreau was a close collaborator on all of the work presented here. Though he passed away before this dissertation was finished, he gave me many years of valuable mentoring, collaboration, and friendship, and for these, I will always be grateful.

I would like to thank Sneha Kasera for stepping up and helping me with the motivation and guidance to cross the finish line after Jay's passing.

I would also like to thank my coauthors on the paper efforts that have contributed to this dissertation. For the work presented in Chapter 3, which was published in the proceedings of NSDI 2007 [112], I worked with Jonathon Duerig, Pramod Sanaga, Daniel Gebhardt, Mike Hibler, Kevin Atkinson, Junxing Zhang, Sneha Kasera, and Jay Lepreau. The work presented in Chapter 4 was done with Jonathon Duerig, John Byers, and Jay Lepreau. Chris Alfeld and Jay Lepreau were my co-authors on the work in Chapter 5, which was published in SIGCOMM CCR in 2003 [111].

For nearly a decade, I have had the unique privilege of working with the talented faculty, staff, and students of the Flux Research Group at the University of Utah. During that time, many faces have changed, but one thing has been constant: it remains a stimulating and encouraging environment in which to conduct research that makes a difference. Thanks to the members of the Flux group, I have grown immensely, both professional and personally, during my time here.

Finally, I would like to thank my wife, Doni, for her support through my long years as a student. I am grateful for her encouragement and her patience over the long evenings and weekends while I finished this dissertation. I thank her for her invaluable editing skills, particularly her ruthless excising of dozens, of, extraneous, commas and duplicated words words.

Chapter 3: I am grateful to the co-workers who gave assistance with implementation, evaluation, operations, discussion, design and writing on the Flexlab work: Leigh Stoller, Sachin Goyal, David Johnson, Tim Stack, Kirk Webb, Eric Eide, Vaibhave Agarwal, Russ Fish, and Venkat Chakravarthy. Many people provided comments and feedback: the anonymous reviewers for HotNets and NSDI, Srini Seshan, Ken Yocum, Dave Andersen, and David Eisenstat. Dave Andersen and Nick Feamster provided access to and assistance with the Datapository, and Vivek Pai and KyoungSoo Park offered access to sources of measurements.

Chapter 4: Shang-Hua Teng provided helpful discussions regarding spectral orderings and advice on working with Laplacian matrices. Sneha Kasera, Mike Hibler, and Eric Eide provided valuable feedback on drafts of this work.

Chapter 5: Dave Andersen did early work on formulating the testbed mapping problem and on early versions of `assign`. Chad Barb implemented the genetic algorithm mapper used for comparison with `assign`. Mac Newbold worked on an initial framework to transfer topologies from ModelNet's input format into `assign`'s, and ran utilization tests. Shashi Guruprasad worked on modifications to PDNS for evaluation purposes. Mike Hibler provided valuable feedback. This work is based on an earlier work: "A Solver For the Network Testbed Mapping Problem," in SIGCOMM Computer Communication Review, Volume 33, Issue 2, April 2003 © ACM, 2003. http://doi.acm.org/10.1145/956981.956988

This work was supported by the National Science Foundation under grants ANI-0082493, ANI-0205702, CNS-0335296, CNS-0205702, and CNS-0338785. It was also supported by DARPA/Air Force grant F30602-99-1-0503 and by Cisco Systems.

CHAPTER 1

INTRODUCTION

Network emulation testbeds have become indispensable tools for conducting research in networking and distributed systems. An emulation testbed creates an environment in which experimenters can run *real* applications and protocols (called “systems under test”) on *real* hosts that are connected by an *artificial* network. This artificial network is constructed inside of a laboratory by configuring infrastructure such as switches and routers to realize a topology specified by the experimenter. Typically, traffic from the system under test is subjected to *traffic shaping* in order to reproduce conditions that would be seen on a deployed network. Such traffic shaping may include inducing delay, limiting bandwidth, and causing packet loss. An experimental network created in this way is isolated: the effects seen on it are products solely of the hardware and software used to build the network, the parameters used to configure the emulator, and the system under test itself.

Emulation testbeds occupy an important position in the spectrum between two other popular experimentation techniques, *simulation* and *live-network experimentation*. By using models of applications, networks, and protocols, simulators can be completely repeatable and highly controllable; these desirable features, however, come at the expense of realism, because such models are necessarily only approximations of reality. In contrast, by running experiments over a live network such as the Internet, experimenters can get realistic network conditions, but lose repeatability and control over the network.

Emulation testbeds offer more realism than simulators such as *ns-2* [100] and *ns-3* [101] by using real hosts and software: commodity or custom operating systems, full network stacks, and real applications. This means that effects of real software and hardware behavior, the sort that are easily missed by simulated models, are captured. However, it also means that emulation does not offer the *perfect* repeatability of simulation, as the complex combination of large hardware and software systems is less predictable than the pure models of simulators.

Live-network testbeds such as PlanetLab [106] and the RON testbed [6] offer a variety of vantage points on the edges of real networks. Experiments are subject to conditions that vary unpredictably over time. This has both advantages and disadvantages: while such variation is unquestionably realistic, it makes repeatable experimentation and comparison of different systems problematic at best. Experimenters also have very little visibility into the interior of real networks, and no ability to directly observe the failures, topology changes, and competing traffic that cause these varying conditions. Thus, it is difficult to understand and explain the behavior of systems run on these networks. Experimenters cannot be guaranteed that particular conditions of interest will occur during their experiments, and because they have no control over the network, these conditions often cannot be intentionally induced. Emulation improves on live-network experimentation by offering an isolated environment where conditions and cross traffic can be precisely controlled. The downside of this is that certain aspects of emulated networks are modeled, and as with simulation, some characteristics of the real network are lost.

Thus, many of the key challenges in network emulation revolve around issues of network realism. These challenges broadly fall into three categories: constructing realistic network topologies; introducing appropriate network conditions on those topologies; and constructing emulated networks that are of sufficient scale to capture the effects of large systems. This dissertation makes key contributions to each of these categories. The remainder of this chapter introduces the specific problems that it addresses.

1.1 Realistic Topologies and Network Conditions

One of the key strengths of emulation testbeds is also one of their greatest weaknesses: they allow the creation of arbitrary (within some limits) virtual topologies, exhibiting a wide range of network conditions. This flexibility means that the topologies created do not necessarily reflect real networks, and thus the burden of selecting parameters to create a network that approximates a realistic deployment environment is on the experimenter. Many experimenters construct topologies in an ad-hoc manner, using basic structures such as LANs, dumbbells, and trees. While this results in networks that are easy to understand and exhibit specific desired properties (such as known bottleneck links) topologies created in this manner are not necessarily representative of real networks.

An obvious strategy for increasing realism is to use known properties of an existing network, such as the Internet, a campus network, or a backbone, to configure the emulator. Such topologies, however, are often unobtainable. In many networks, particularly commercial ISPs, network topologies are considered proprietary information and are not available to researchers. Some amount of topological information can be inferred from the edges of the network, using simple tools such as `traceroute` or more complicated network tomography [121]. Simply knowing the topology, however, is not sufficient to create an emulation that faithfully re-creates the end-to-end conditions seen on that network. In a traditional emulator, additional details such as IP addresses and routing tables are necessary to correctly route packets, and in order to create realistic emulation of bandwidth, delay, and packet loss, some estimate of the competing traffic at every link is necessary [131, 132].

In order to re-create conditions seen on a real network inside of an emulator, we present a novel approach called Flexlab. Flexlab takes advantage of the fact that it is possible for researchers to take end-to-end measurements from a variety of vantage points on the Internet using publicly available facilities. Unlike traditional emulation, which models the interior of the network at the router level, Flexlab abstracts over the core of the network, using only properties measurable from end hosts to drive the emulation. Flexlab recognizes that emulation and live experimentation represent extremes on a spectrum, and there are several interesting points between the two; we present models that lie at three different points on this spectrum, but the Flexlab framework is general enough to support others. On one end of the spectrum, Flexlab uses coarse-grained historical measurements taken on a real network. These measurements are used to statically configure a predictable and repeatable emulation, but one that is insensitive to the foreground flows. On the other end of the spectrum, Flexlab creates a model of the application under test's flows, re-creates them in real time on the live testbed, measures the reaction of background flows, and re-creates the measurements inside of the emulator, resulting in a closed-loop emulation.

Flexlab-style emulation is appropriate for experimenting on applications and protocols deployed at the edges of the Internet, but it is not suitable for studying behavior that occurs in the core of the network. Experiments for which the network interior is important include those that change the network layer of the protocol stack (including forwarding behavior, routing protocols, and congestion avoidance) and middleboxes (such as firewalls, intrusion detection systems, and NATs). For these, many experimenters turn to another style of creating realistic virtual topologies: topology generators.

Topology generators have parameterized models of the Internet's topology, created through measurements of the Internet or from first principles. Using appropriate parameters, they can produce topologies that are more realistic than ad-hoc creations and have more internal detail than is measurable from end hosts. A key challenge is that most topology generators are designed for use with network simulators, such as *ns-2* [100], and such simulators typically use an abstract view of the network which does not include IP addresses. Thus, in order to use the topologies created by popular generators [91, 143, 139] in an emulator, they must be annotated with IP addresses.

Real networks, especially within a single domain or autonomous system, tend to be constructed hierarchically. Therefore, a realistic IP address assignment is one that reflects the natural hierarchy of the network. Real networks and those created by topology generators are not strictly hierarchical, and extracting the hierarchy that they contain is a challenging problem. We present a set of algorithms that examine the structure of a virtual network in order to assign appropriate IP addresses. We judge success by two criteria: compactness of the resulting routing tables and the runtime of the assignment algorithm. Because IP routing is itself hierarchical, annotations reflecting the network's hierarchy will tend to produce smaller routing tables. Scaling is a concern because the annotation process must complete in reasonable time to be of practical value.

1.2 Scalable Topology Embedding

Once an experimenter has a suitable virtual topology, that topology must be embedded, or mapped, onto the physical resources of the testbed. This is a challenging problem: testbeds typically have heterogeneous hardware, scarce resources which must be preserved, and bottlenecks that must not be over-used. User requests for particular types of nodes or links must also be met. In light of these constraints, the testbed mapping problem is NP-hard [5], and the complexity of the mapping problem increases rapidly with the sizes of the virtual and physical topologies. Because mapping is on the critical path for instantiating an experiment on a network testbed, it must complete quickly; instantiation speed contributes directly to the efficiency of resource use on the testbed [54].

We present a solver, `assign`, which is is concerned with two areas of the network testbed mapping problem:

- Scalability: ensuring that solutions to large problems can be found in reasonable time
- Expressivity: providing the appropriate primitives for describing the testbed environment in a way that is powerful, yet efficient to map

`assign` employs a simple but flexible set of primitives for describing networks, and as a result, it can be used for a wide range of testbed mappings; in addition to emulation, it can also be used to map parallel distributed simulations [114] and virtualized networks [107]. `assign` achieves scaling in two primary ways: by using a very carefully tuned randomized heuristic which returns good solutions in most cases, and by exploiting structure in virtual and physical networks. Using this structure, `assign` is able to reduce the effective size of the mapping problem, making it more tractable.

1.3 Contributions

The contributions of this dissertation can be divided into three categories:

- **Realistic Network Conditions**

  We offer a novel method for introducing realistic network conditions into an emulation testbed. We describe a flexible framework for importing these conditions and develop a set of models that trade off realism with repeatability. These models demonstrate that live-network and emulated experimentation can be combined in useful ways to create hybrids that inherit strengths from each environment.

- **Scalable, Realistic IP Address Assignment**

  We present a set of algorithms for automatically annotating virtual topologies with IP addresses. These algorithms mimic some aspects of allocation policies in real networks and some of them scale well to large topologies. Our key contribution to this area is to propose a novel metric for quantifying aggregatability in prefix-routed networks. This metric, Routing Equivalence Sets (RES), directly quantifies the savings in routing table size that come from aggregating candidate sets and is efficiently computable. Using this metric, addresses can be assigned in a way that maximizes their aggregatability. With this automatic annotation, topologies generated for simulation can be imported into an emulation environment. While developed in the context of emulation testbeds, these algorithms also have applications in the assignment of addresses to real networks.

- **Scalable Resource Mapping**

  We design and implement a mapper for scalably assigning virtual topologies to physical ones, taking into account the unique features of emulation testbeds. With its foundations in established techniques such as simulated annealing [129] and graph partitioning [71], `assign` makes use of a number of domain features in order to scale to large networks. It takes advantage of the fact that emulation testbeds typically have sizeable sets of homogeneous nodes and uses this to avoid exploring redundant parts of the solution space. `assign` also includes a pre-pass which takes advantage of the hierarchy in virtual networks, using a graph partitioner to reduce the size of the input topology.

1.4 Organization

This dissertation is organized as follows. The next chapter provides more background on emulation testbeds: the needs that motivate them, important aspects of their design, and some of the key challenges in building them. It motivates our work on emulation testbeds, focusing on Emulab, the testbed environment that provided the context for this work. The following three chapters are organized around the three areas of contribution: Chapter 3 addresses the need for realistic network conditions by coupling an emulation testbed with a live-network testbed to create Flexlab, a hybrid environment. Chapter 4 attends to the need for realistic topologies by devising a set of algorithms that solve one of the key challenges in using topology generators for emulation, that of automatically annotating them with IP addresses. Chapter 5 presents a mapper, `assign`, that is capable of assigning virtual resources to physical ones in a manner that scales well enough to handle large testbeds and is expressive enough to allow a wide range of topologies to be mapped. Chapter 6 concludes by summarizing our results and indicating directions for future research.

CHAPTER 2

BACKGROUND AND MOTIVATION: NETWORK EMULATION TESTBEDS

While emulation testbeds are well-established as experimentation platforms [34], they are not without their problems, as we have seen in the previous chapter. This chapter gives an overview of the major environments and methodologies used by the networking and distributed systems research communities. It identifies the key strengths and weaknesses of each. In doing so, it highlights the role that emulation testbeds fill in the larger scope of network experimentation. This motivates the work of this dissertation, which seeks both to improve the ability of emulation testbeds to fill this role and to expand it by incorporating elements from other environments.

Five major environments are used for the evaluation of networked systems:

- *Ad-hoc Experimentation*, in which experiments are conducted in an unsystematic manner on a network not designed as an experimental facility

- *Purpose-built Testbeds*, which are built to evaluate a specific system under test or a particular network environment

- *Live-network Testbeds*, which provide experimenters with a systematic way to run applications on hosts connected by a network that is in production use

- *Simulators*, which use software models in place of real hardware, protocols, and applications

- *Emulation Testbeds*, which provide experimenters with a general-purpose, controlled environment that contains real hosts connected by an artificial network

2.1 Ad-hoc Experimentation

Ad-hoc experimentation is not so much an experimentation methodology as it is a lack of methodology. An experimenter simply runs an application on a collection of hosts which are generally chosen because it is convenient for the experimenter to gain access to them. These hosts are often in a LAN or in a campus or corporate environment, and are often desktop machines, laptops, servers, or other parts of an institution or individual's computing environment. The hosts and network may be shared by other users or production traffic and are often not under the complete control of the experimenter. Deployment and control of the system under test is typically done manually or using simple ad-hoc scripts.

The benefits of this method are scant. By taking advantage of existing computing and network resources, ad-hoc experimentation can have low monetary and administrative costs. This benefits usually only applies to small-scale experiments, since production computing environments are usually not set up to support or automate large-scale experimentation.

The problems with this method are clear. The network environment is unlikely to be representative of the deployment environment targeted by the application, especially if that deployment target is the Internet. The fact that the environment is often shared with production traffic means that the experimenter has no control over network conditions, and often low visibility into the workings of the core of the network. Experiments are limited to the network topology and host configuration of the production network, giving the experimenter very limited ability to experiment with different topologies, network conditions, or host environments.

Ad-hoc experimentation is typically only valuable for initial development and debugging. During this phase, limited scale and lack of automated tools for controlling the experiment are usually not a significant impediment. Ad-hoc experimentation is rarely suitable for the scientific study of networked systems. For such studies, researchers typically turn to facilities that have been designed for experimentation.

2.2 Purpose-built Testbeds

One strategy for improving on ad-hoc experimentation is to construct a testbed specifically designed to evaluate a single system under test or a set of related systems. By using hardware dedicated to experimentation, a purpose-built testbed avoids many of the problems associated with running on production infrastructure: the experimenter has control over the network topology and configuration, and the testbed can be a closed environment,[1] removing interference from background traffic and other activity. This can make a purpose-built testbed significantly more controllable and repeatable than an ad-hoc one.

Purpose-built testbeds do have a number of drawbacks, however. Building a facility specifically for a small set of experiments is typically not cost-effective and requires substantial effort; this often limits such testbeds to evaluating high-value systems, such as in commercial settings. It also limits the scale of such testbeds. Creating a realistic network environment in a purpose-built testbed can still be problematic; if the target deployment environment involves background traffic (eg. cross-traffic on the Internet) or a geographically distributed network, these effects can be difficult to reproduce in a closed testbed environment.

Purpose-built testbeds can be invaluable for studying specialized hardware platforms or unusual deployment scenarios [137]. In many cases, however, it is more effective to build a more general-purpose environment for network experimentation, amortizing hardware, deployment, and tool development costs among a larger set of experiments and users. General-purpose testbeds fall into two categories: those that offer experimentation on live networks and those that offer experimentation in a closed, controllable environment.

2.3 Live-network Testbeds

Live-network testbeds, such as PlanetLab [106], the RON testbed [117], and RoofNet [20], give experimenters systematic access to a large collection of hosts.

[1] The extent to which a testbed can be made a closed environment is dependent on the networking technology in use: wired and fiber-optic networks can easily be isolated, but removing all sources of interference in wireless testbeds is much more problematic and expensive.

These hosts are usually on the edge of a real network, whose topology and conditions are not under the control of the testbed or the experimenter. Experiments are run across the live network itself, giving a high degree of realism.[2] This frees experimenters from having to worry about the realism of their networks; they are "real by default." Such testbeds often come with tools [19, 3] to aid in the deployment, control, and monitoring of experiments.

While the hosts are generally under the control of the testbed, the network is not. This is both the main advantage and main disadvantage of such testbeds. The test environment's internal topology, cross traffic, specific hardware, etc. generally cannot be known by the experimenter, and may change over time. In the case of testbeds on the edges of the Internet, this is because the paths connecting hosts cross through a number of administrative domains and carry traffic from many users. In the case of wireless testbeds, it is often because the wireless channel is affected by external influences that change over time, such as sources of interference or objects that absorb or reflect radio waves. Such properties make these networks interesting targets for study, and live-network testbeds can provide good platforms for studying them. The downside is it can be difficult to establish "ground truth" [112, 31]—while experimenters can report results obtained on such testbeds, they cannot report in detail on the conditions that contributed to those results, and it can be difficult to distinguish behavior caused by the environment from behavior caused by the system under test itself. Similarly, two experiments run at different times, even minutes apart [147], cannot be fairly compared, as conditions may have changed in the meantime.

Live-network testbeds have a valuable role as deployment platforms. Because of their locations within real networks, they can be used to deploy services that attract real end-users [134, 110, 45]. This can enable experimenters to gather valuable data not just about real networks but real application deployments as well. It can also

[2] It should be noted, however, that the realism of a live-network testbed is limited by the extent to which the network over which it runs can be said to be representative of a real deployment environment [122].

enable experimenters to gather data about the network connectivity of these users, giving them a much larger set of network perspectives than can be offered by the testbed itself [90, 29].

Because of the difficulty and expense involved in deploying and maintaining large collections of hosts, often in a geographically distributed fashion, there are practical limits on live-network testbeds. In particular, the number of deployed hosts is typically much smaller than the number of simultaneous experiments. Due to the popularity of such testbeds, it becomes necessary to run multiple simultaneous experiments on each host. This means that experiments can be affected, sometimes quite dramatically, by other experiments [112]. It also means that the privileges that can be given to each experimenter are limited; experimenters cannot be allowed to change the host environment in ways that would hinder other experimenters, such as changing shared network stacks or replacing operating systems.

Live-network testbeds are valuable for deployment studies, for measuring networks, and for offering services to end users. Their primary strengths are the realism of their network conditions, diversity of their vantage points into networks, and their potential as deployment platforms. As environments for the scientific study of systems under test, their value is limited: it is difficult to explain the behavior of results gathered on them and to compare different systems. In practice, the shared nature of such testbeds means that they tend to be useful primarily for experiments in the upper layers of the network stack. This makes live-network testbeds particularly valuable for the evaluation of overlay networks. For study of low-layer protocols, or for a more predictable and controllable environment, many experimenters turn to simulation and emulation.

2.4 Simulation

Simulators generally fall into two categories: purpose-built simulators, used to evaluate specific applications or protocols [13] and general-purpose simulators [100, 101, 144], which can be used and extended to evaluate a wider range of systems and networks. Both types of simulators rely heavily on *models* of network behavior: rather than sending real packets through real networks, they simulate hosts, routers, and links in software. The generally do not include real operating system network stacks, protocol implementations, etc. Often, simulators do not run real application code, using models of application behavior as well.

A primary advantage of simulators is that they can be completely deterministic and thus repeatable. This enables careful comparisons between different systems and scientific studies in which variables (such as network topology, bandwidth, queuing disciples, etc.) are changed one at a time. Another advantage is that simulators can be used to model arbitrary networks: the experimenter does not need to gain access to a real network that fits his or her needs. The simulated network need not even be buildable using current technology; this allows for forward-looking experiments that explore possible future networking technologies. The experimenter may examine all aspects of the network being tested, giving excellent opportunities for understanding the emergent properties of complex systems. Because simulations are generally not constrained to run in real-time, the scale of simulated networks is limited by available computational power rather than the physical size of available networks. Depending on the simulator and the system being simulated, simulations of networks reaching tens or hundreds of thousands of nodes may be possible. The network topologies used with simulators commonly come from *topology generators*, which use models of the Internet to create realistic topologies of arbitrary scale. Current popular topology generators include GT-ITM [143], BRITE [91], and Orbis [87],

The key weakness of simulators is lack of realism. Because they operate on models, they may miss important details of network or application behavior. By abstracting over details of hardware, operating systems, protocols, and applications, their behavior may differ from actual implementations and behavior “in the wild” [37, 40, 41, 56]. Because every aspect of a simulated network is modeled, simulators are highly dependent on the realism of those models. Models of the Internet [41] and wireless networks [75] are notoriously lacking in realism, though improvements are constantly made. This leaves simulators open to criticism: they are generally considered to be less realistic than the other experimentation environments.

Simulators are valuable for experiments that require a high degree of flexibility, control, observability, or repeatability. They are also useful for studies early in the lifecycle of a protocol or application; constructing a model of the system's behavior may be simpler than implementing the entire system. They are generally not appropriate when a high degree of realism is required, and are not used to evaluate real implementations.

2.5 Network Emulation

In network emulation [138, 127, 61, 92, 66], real software is run on real hosts. The hosts are typically under the complete control of the experimenter and are not shared by more than one experiment at a time. The network connecting these hosts is artificial, in the sense that it is manipulated to create specific conditions. Though a network emulation testbed uses actual network hardware such as interface cards, switches, and routers, the network is often configured or intentionally degraded in order to emulate a specific topology or set of network conditions. For example, a densely-connected Ethernet network may be configured using VLANs to resemble a sparser network. Similarly, though the hosts in an emulation testbeds are often located in a single lab and connected by a high-speed network, traffic shaping [116, 127, 98] may be used to limit the bandwidth on the network, introduce latency, or induce packet loss; thus, paths can be constructed within the emulator that exhibit characteristics of wide-area links or slower network technologies.

Emulation testbeds enable experimentation on a wide range of network conditions, applications, network stacks, and operating systems. They enable repeatable results, parameter space exploration, "what if" experiments, sensitivity analysis, and other forms of systematic experimentation on real applications and protocols. The contained environment offered by emulation testbeds makes them ideal for security experiments, and they are also well suited to developing and debugging applications and systems software. Their two primary weaknesses are with the realism of the network environment and the scale of experiments that they can support.

Unlike live-network experimentation, the network in an emulator is under the control of the experimenter and is isolated from effects caused by sources outside of the experiment. This makes emulation more flexible than live-network experimentation: rather than simply making use of the deployment environment offered by a particular network, an emulation testbed can be configured to resemble a wide range of potential deployment environments. It also enables controlled and repeatable experiments. As with simulation, however, the emulation is only as realistic as the model used to drive the emulated network. Unlike simulation, emulation testbeds provide realistic hosts, applications, and protocols, and provide a valuable platform for working with real implementations of these systems. Emulation testbeds typically offer a set of tools for configuring the network, deploying software, and controlling experiments, representing a significant improvement over ad-hoc experimentation [138, 32]. While purpose-built testbeds may offer an excellent environment or tools for testing a particular system, emulation testbeds represent a good value by supporting a wider variety of experiments.

Scale is a major challenge for emulation testbeds. As systems that manage a large set of hosts and network hardware, they face many of the same challenges as traditional network management: they must be able to boot, configure, and control a large set of real systems in a scalable manner. The nature of network experimentation also presents some unique challenges. Entire networks are created and torn down with relatively high frequency. The problem of network embedding—that is, of finding a subset of the physical topology that matches some requested topology—has much more prominence in testbeds than in traditional network settings. Many services that are part of the established infrastructure in production systems must be instantiated for each experiment, and some network basics such as IP addresses and routing tables must be re-computed frequently. All of these tasks have runtimes or failure rates that can increase dramatically with the scale of the testbed and individual experiments. The largest emulation testbeds currently consist of several hundred physical nodes [42]; solutions must scale to at least this order of magnitude, and higher if they are to support future testbeds.

2.5.1 Emulab

The main context for the work presented in this dissertation is a specific emulation testbed, Emulab [138]. The name “Emulab” is used to refer to both a facility run at the University of Utah and the software written to manage it, which is now in use at dozens of similar facilities [33].

The Emulab software is a state-of-the-art management system for network testbeds. Emulab provides an integrated “full-service” interface for experimentation; experiments are set up rapidly (on the order of a few minutes) and reliably, with the setup and subsequent control provided through web, XML-RPC, or script-driven interfaces. Emulab experiments may be interactive or completely scripted, and Emulab provides a distributed event system through which both the testbed software and users can control and monitor experiments. Emulab also provides efficient mechanisms for distributing experimental applications to hosts, automatic packet trace collection, and gathering of logfiles and other results. It is used to manage dozens of testbeds at a diverse set of educational institutions, research facilities, and corporations [33]. In this dissertation, we focus on its capabilities as a testbed for emulation of wired network experiments, though it also transparently integrates other experimental environments such as live-network experimentation, simulation, wireless experimentation, and sensor networks. As part of ongoing work for the GENI [48] project, the Emulab software supports wide-area network environments which use shared or dedicated network resources [108]. Other important features include the ability to run experiments using virtualization technology [53] and the ability to federate multiple facilities to run experiments across them. Versions of the Emulab software have been in production use since April 2000.

The Emulab facility at the University of Utah includes hundreds of PCs, over a dozen Ethernet switches, and thousands of Ethernet ports. This cluster is designed to provide artifact-free emulation through a configurable network. It also includes a building-scale wireless testbed incorporating 802.11 and software radio [36] devices.

It is used to manage the RON wide-area testbed [117] and has a portal that enables it to create slices on another live-network testbed, PlanetLab [135]. Its primary mission is to support research and education in operating systems, networking, and distributed systems: it supports thousands of users at hundreds of institutions, mostly universities, worldwide. Experiments are created and torn down at a rate of dozens per day. The facility is space-shared: it can be arbitrarily partitioned for use by multiple experimenters simultaneously. Some resources in the system, such as nodes, can only be used in one experiment at a time, although an experiment can be "swapped out" to free resources while it is idle. In this sense, Emulab is also time-shared.

An *experiment* is Emulab's central operational entity. To run an experiment on Emulab, an experimenter submits a network topology. This virtual topology can include links and LANs with associated characteristics such as bandwidth, latency, and packet loss. The network topology is specified using an extended version of the *ns-2* [100] language. Traffic shaping on links, if requested, is done by interposing "delay nodes" between the endpoints. Delay nodes are inserted as transparent Ethernet bridges, and use Dummynet [116] to induce delay, limit bandwidth, and cause packet loss on the paths. Delay nodes can be used to emulate asymmetric point-to-point *links*, *LANs* in which each connected node has all outgoing traffic shaped, and *clouds*, which are LANs in which each node's paths to other nodes may be individually shaped. Specifications for node hardware and software resources can also be included in the virtual topology.

Once an experimenter submits a virtual topology, Emulab must select a set of physical resources on which it can be instantiated. The space-shared nature of Emulab means that this set is constantly changing, so this selection is done on each "swap in." Once a suitable set of nodes has been selected, Emulab realizes the topology by providing automated setup of hosts, switches, and path emulators. Emulab is capable of loading operating system images on PCs [54] and sensor network nodes [63], and creating VLANs on Ethernet switches from several different vendors. Depending on the size and complexity of the experiment, this process typically takes minutes or tens of minutes. Emulab is designed to provide “zero penalty for remote access”: experimenters are given access to serial consoles and power control for nodes in their experiments, providing a level of control similar to that which an experimenter would have over nodes located at his or her own site. When an experiment is “swapped out” or terminated, the nodes it used are returned to a clean state before allocation to another experiment.

2.6 Conclusion

Emulation fills an important role in network experimentation: it is used to evaluate real implementations of systems in a controlled and repeatable environment. Its two primary weaknesses are realism and scale. The remainder of this dissertation presents improvements to emulation, tackling key problems relating to these weaknesses.

CHAPTER 3

REALISTIC NETWORK CONDITIONS: THE FLEXLAB APPROACH

3.1 Overview

This chapter describes the motivation, design, and implementation of Flexlab, a testbed environment that combines strengths of both live-network and emulation testbeds. It enhances an emulation testbed by providing the ability to integrate a wide variety of network models, including those obtained from an overlay network. We present three models that demonstrate its usefulness, including Application-Centric Internet Modeling (ACIM), which we specifically developed for Flexlab. Its key idea is to run the application within the emulation testbed and use the application's own offered load to measure the overlay network. These measurements are used to shape the emulated network. Our results indicate that for evaluation of applications running over Internet paths, Flexlab with the ACIM model can yield far more realistic results than either PlanetLab without resource reservations, or Emulab without topological information.

3.2 Introduction

As we saw in Chapter 2, two of the major classes of networking testbeds, *emulation testbeds* and *live-network* testbeds, have complementary properties. *Emulation testbeds*, such as the emulation component of Emulab [138], create artificial network conditions that match an experimenter's specification and offer control and repeatability. *Live-network testbeds* such as PlanetLab [106], send an experimenter's traffic over a live network, sacrificing control and repeatability for realism. These two types of testbeds have been considered to be separate types of environments and their strengths to be mutually exclusive. In this chapter, we argue that it is possible to create a testbed that gives users some benefits of each of these two environments, bringing more realistic network conditions to emulators and more controllability and repeatability to live-network testbeds. We present Flexlab, which bridges an emulation testbed with an *overlay* testbed. An overlay testbed is a special case of live-network testbed in which the live network is the Internet and the testbed environment is "overlaid" on the network by virtue of being run "on top" of a common protocol (in this case, IP).

In Flexlab, experimenters obtain networks that exhibit real Internet conditions *and* full, exclusive control over hosts. At the same time, Flexlab provides more control and repeatability than the Internet. We created this new environment by closely coupling an emulation testbed with an overlay testbed, using the overlay to provide network conditions for the emulator. Flexlab's modular framework qualitatively increases the range of network models that can be emulated. In this chapter, we describe this framework and three models derived from the overlay testbed. These models are by no means the only models that can be built in the Flexlab framework, but they represent interesting points in the design space, and demonstrate the framework's flexibility. The first two use traditional network measurements in a straightforward fashion. The third, "Application-Centric Internet Modeling" (ACIM), is itself a novel contribution.

ACIM stems directly from our desire to combine the strengths of emulation and live-Internet experimentation. We provide machines in an emulation testbed and "import" network conditions from an overlay testbed. Our approach is application-centric in that it confines itself to the network conditions relevant to a particular application, using a simplified model of that application's own traffic to make its measurements on the overlay testbed. By doing this in near real-time, we create the illusion that network interfaces in the emulator are distributed across the Internet.

Flexlab is built atop the most popular and advanced testbeds of each type, PlanetLab and Emulab, and exploits a public federated network data repository, the Datapository [7]. Flexlab is driven by Emulab testbed management software which has been enhanced to extend most of Emulab's experimentation tools to

PlanetLab slivers [135]. These include automatic link tracing and distributed data collection. Because Flexlab allows different network models to be "plugged in" without changing the experimenter's code or scripts, this testbed also makes it easy to compare and validate different network models.

This chapter presents the following contributions:

- A software framework for incorporating a variety of highly-dynamic network models into Emulab
- The ACIM emulation technique, which provides high-fidelity emulation of live Internet paths
- Techniques that infer available bandwidth from the TCP or UDP throughput of applications that do not continually saturate the network
- An experimental evaluation of Flexlab and ACIM
- A flexible network measurement system for PlanetLab. We demonstrate its use to drive emulations and construct simple models

We also present measurement data from PlanetLab that show the significance of non-stationary network conditions, shared bottlenecks, and CPU scheduling delays. Flexlab is currently deployed on Emulab and is part of the Emulab open source software release.

3.3 Flexlab Architecture

The architecture of the Flexlab framework is shown in Figure 3.1. The application under test runs on emulator hosts, where the *application monitor* instruments its network operations. The application's traffic passes through the *path emulator*, which shapes it to introduce latency, limit bandwidth, and cause packet loss. The parameters for the path emulator are controlled by the *network model*, which may optionally take input from the monitor, from the *network measurement repository*, and from other sources. Flexlab's framework provides the ability to incorporate new network models, including highly dynamic ones, into Emulab. All parts of Flexlab except for the underlying emulation testbed are user-replaceable.

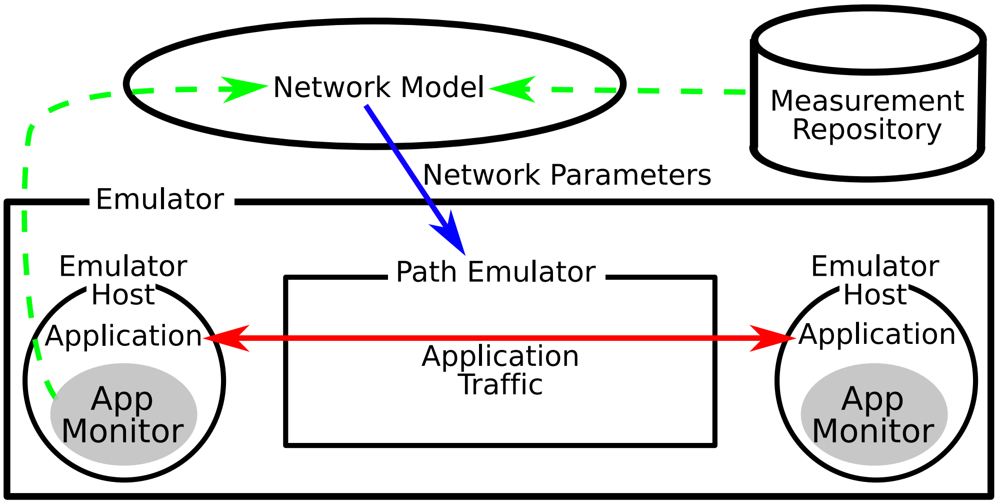


Figure 3.1. Architecture of the Flexlab framework. Any network model can be "plugged in," and can optionally use data from the application monitors or measurement repository.

3.3.1 Emulator

Flexlab runs on top of the Emulab testbed management system, which provides critical experiment management infrastructure. Flexlab makes use of Emulab's automated configuration of hosts, switches, and path emulators. It is built on Emulab's mechanisms for distributing experimental applications to nodes, controlling those applications, collecting packet traces, and gathering log files and other results. Emulab's portal [135] extends these management benefits to PlanetLab nodes as well. Experimenters can therefore easily move back and forth between emulation, live experimentation, and Flexlab experimentation. Also integrated into Emulab is a full experiment and data management system [32], which was used to gather and manage many of the results in this chapter.

3.3.2 Application Monitor

The application monitor reports on the network operations performed by the application, such as the connections it makes, its packet sends and receives, and the socket options it sets. This information can be sent to the network model, which can use it to track which paths the application uses and discover the application's offered network load. Knowing the paths in use aids the network model by limiting the set of paths it must measure or compute; most applications will use only a small subset of the n^2 paths between n hosts. The monitor is described in detail in Section 3.6.

3.3.3 Path Emulator

The path emulator shapes traffic from the application. It can, for example, queue packets to emulate delay, dequeue packets at a specific rate to control bandwidth, and drop packets from the end of the queue to emulate saturated router queues. Our path emulator is an enhanced version of FreeBSD's Dummynet [116]. We have made extensive improvements [118] to Dummynet to add support for the features discussed in Section 3.6.2, as well as adding support for jitter and for several distributions: uniform, Poisson, and arbitrary distributions determined by user-supplied tables. Dummynet runs on separate hosts from the application, both to reduce contention for host resources, and so that applications can be run on any operating system.

For Flexlab we typically configure Dummynet so that it emulates a "cloud," abstracting the Internet as a set of per-flow pairwise network characteristics. This is a significant departure from Emulab's typical use: it is typically used with router-level topologies, although the topologies may be somewhat abstracted. The cloud model is necessary for us because Flexlab deals with end-to-end conditions rather than trying to reverse engineer the Internet's router-level topology.

A second important piece of our path emulator is its control system. The path emulator can be controlled with Emulab's event system, which is built on a publish/subscribe model. A "delay agent" on each emulator node subscribes to events for the path it is emulating and updates characteristics based on the events it receives. Any node can publish new characteristics for any path, which makes it easy to support both centralized and distributed implementations of network models. For example, control is equally easy by a single process that computes all model parameters or by a distributed system in which measurement agents independently compute the parameters for individual paths. The Emulab event system is lightweight, making it feasible to implement highly dynamic network models that send many events per second, and is secure: event senders can affect only their own experiments.

3.3.4 Network Model

The network model supplies network conditions and parameters to the path emulator. The network model is the least-constrained component of the Flexlab architecture; the only constraint on a model implementation is that it must configure the path emulator through the event system. Thus, a wide variety of models can be created. A model may be static, setting network characteristics once at the beginning of an experiment, or dynamic, keeping them updated as the experiment proceeds. Dynamic network settings may be sent in real-time as the experiment proceeds, or the settings may be pre-computed and scheduled for delivery by Emulab's event scheduler.

We have implemented three distinct network models, discussed in Sections 3.5 and 3.6. All of our models pair each emulator node with a node in the overlay network, attempting to give the emulator node the same view of network characteristics as its peer in the overlay. The architecture, however, does not require that models come directly from overlay measurements. Flexlab may also be used with network models from other sources, such as analytic models.

3.3.5 Measurement Repository

Flexlab's measurements are stored in Andersen and Feamster's Datapository [7]. Information in the Datapository is available for use in constructing or parameterizing network models, and the networking community is encouraged to contribute their own measurements. We describe Flexlab's measurement system in the next section.

3.4 Wide-area Network Monitoring

Good measurements of Internet conditions are important in a testbed context for two reasons. First, they can be used as input for network models. Second, they can be used to select Internet paths that tend to exhibit a chosen set of properties. To collect such measurements, we developed and deployed a wide area network monitor, Flexmon [62]. It has been running since February 2006, and has placed to date over 1.2 billion measurements of connectivity, latency, and bandwidth between PlanetLab hosts into the Datapository. Flexmon's design provides a measurement infrastructure that is shared, reliable, safe, adaptive, controllable, and accommodates high-performance data retrieval. Flexmon has some features in common with other measurement systems such as S^3 [141] and Scriptroute [123], but is designed for shared control over measurements and the specific integration needs of Flexlab.

Flexmon, shown in Figure 3.2, consists of five components: *path probers*, the *data collector*, the *manager*, *manager clients*, and the *auto-manager client*. A path prober runs on each PlanetLab node, receiving control commands from a central source, the manager. A command may change the measurement destination nodes, the type of measurement, and the frequency of measurement. Commands

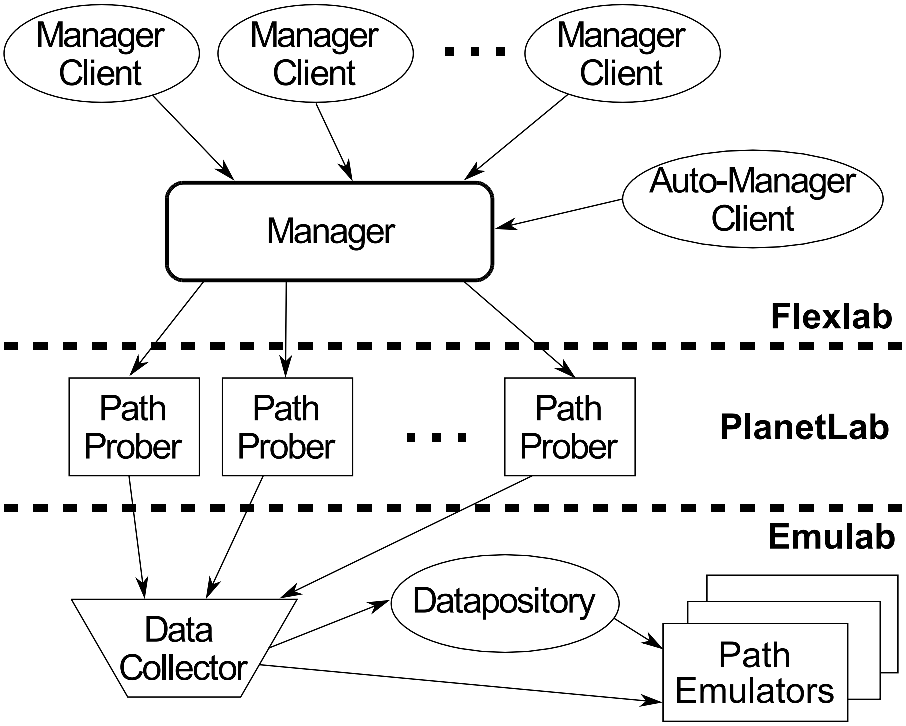


Figure 3.2. The components of Flexmon and their communication.

are sent by experimenters, using a manager client, or by the auto-manager client. The purpose of the auto-manager client is to maintain measurements between all PlanetLab sites. The auto-manager client chooses the least loaded node at each site to include in its measurement set, and makes needed changes as nodes and sites go up and down. The data collector runs on a server in Emulab, collecting measurement results from each path prober and storing them in the Datapository. To speed up both queries and updates, it contains a write-back cache in the form of a small database instance.

Due to the large number of paths between PlanetLab nodes, Flexmon measures each path at a fairly low frequency—approximately every 2.5 hours for bandwidth, and 10 minutes for latency. To get more detail, experimenters can control the frequency of Flexmon's measurement of any path. Flexmon maintains a global picture of the network resources it uses, and caps and adjusts the measurement rates to maintain safety to PlanetLab.

Flexmon currently uses simple tools to collect measurements: `iperf` for bandwidth, and `fping` for latency and connectivity. We had poor results from initial experiments with packet-pair and packet-train tools, including `pathload` and `pathchirp`. Our guiding principles thus far have been that the simpler the tool, the more reliable it typically is, and that the most accurate way of measuring the bandwidth available to a TCP stream is to use a TCP stream. Flexmon has been designed, however, so that it is relatively simple to plug in other measurement tools. For example, tools that trade accuracy for reduced network load or increased scalability [27, 43, 86, 97] could be used, or we could take opportunistic measurements of large file transfers by the content distribution networks running on PlanetLab [45, 134].

3.5 Simple Measurement-Driven Models

We have used measurements taken by Flexmon to build two simple, straightforward network models. These models represent incremental improvements over the way emulators are typically used today. Experimenters typically choose network parameters on an ad hoc basis and keep them constant throughout an experiment. Our *simple-static* model improves on this by using actual measured Internet conditions. The *simple-dynamic* model goes a step further by updating conditions as the experiment proceeds. Because the measurements used by these models are stored permanently in the Datapository, it is trivial to "replay" network conditions starting at any point in the past. Another benefit is that the simple models run entirely outside of the emulated environment itself, meaning that no restrictions are placed on the protocols, applications, or operating systems that run on the emulator hosts. The simple models do have some weaknesses, which we discuss in this section. These weaknesses are addressed by our more sophisticated model, ACIM, presented in Section 3.6.

3.5.1 Simple-static and Simple-dynamic

In both the simple-static and simple-dynamic models, each PlanetLab node in an experiment is associated with a corresponding emulation node in Emulab. A manager client called `dbmonitor` runs on an Emulab server, collecting path characteristics for each relevant Internet path from the Datapository. It applies the characteristics to the emulated network by sending events to the path emulator.

In simple-static mode, `dbmonitor` starts at the beginning of an experiment, reads the path characteristics from the database, issues the appropriate events to the emulation agents, and exits. This model places minimal load on the path emulators and the emulated network, at the expense of fidelity. If the real path characteristics change during an experiment, the emulated network becomes inaccurate.

In simple-dynamic mode the experimenter controls the frequencies of measurement and emulator update. Before the experiment starts, `dbmonitor` commands Flexmon to increase the frequency of probing for the set of PlanetLab nodes involved in the experiment. Similarly, `dbmonitor` queries the DB and issues events to the emulator at the specified frequency, typically on the order of seconds. The dynamic model addresses some of the fidelity issues of the simple-static model, but it is still constrained by practical limits on measurement frequency.

3.5.2 Stationarity of Network Conditions

The simple models presented in this section are limited in the detail they can capture, due to a fundamental tension. We would like to take frequent measurements, to maximize the models' accuracy. However, if they are too frequent, measurements of overlapping paths (such as from a single source to several destinations) will necessarily complete, causing interference that may perturb the network conditions. Thus, we must limit the measurement rate.

To estimate the effect that low measurement rates have on accuracy, we performed an experiment. We measured latency between pairs of nodes every 2 seconds for 30 minutes. We analyzed the latency distribution to find "change points" [124], which are times when the mean value of the latency samples changes. This statistical technique was used in a classic paper on Internet stationarity [146]; our method is similar to their "CP/Bootstrap" test. The analysis provides insight into the required measurement frequency—the more significant events missed, the poorer the accuracy of a measurement.

Table 3.1 shows some of the results from this test. We used representative nodes in Asia, Europe, and North America. One set of North American nodes was connected to the commercial Internet, and the other set to the Internet2 research and education network [57]. The first column shows the number of change points seen in this half hour. In the second column, we have simulated measurement at lower frequencies by sampling our high-rate data; we used only one of every ten

Table 3.1. Change point analysis for latency.

Path	High	Low	Change
Asia to Asia	2	1	0.13%
Asia to Commercial	2	0	2.9%
Asia to Europe	4	0	0.5%
Asia to Internet2	6	0	0.59%
Commercial to Commercial	**20**	2	**39%**
Commercial to Europe	4	0	3.4%
Commercial to Internet2	**13**	1	**15%**
Internet2 to Internet2	4	0	0.02%
Internet2 to Europe	0	0	–
Europe to Europe	**9**	1	**12%**

measurements, yielding an effective sampling interval of 20 seconds. Finally, the third column shows the magnitude of the median change, in terms of the median latency for the path.

Several of the paths are largely stable with respect to latency, exhibiting few change points even with high-rate measurements, and the magnitude of the few changes is low. However, three of the paths (in bold) have a large number of change points, and those changes are of significant magnitude. In all cases, the low-frequency data misses almost all change points. In addition, we cannot be sure that our high-frequency measurements have found all change points that would be found in even higher-frequency data. The lesson is that there are enough significant changes at small time scales to justify, and perhaps even necessitate, high-frequency measurements.

In Section 3.6, we describe ACIM, which addresses this accuracy problem by using the application's own traffic patterns to take measurements. As a result, the only load on the network and the only self-interference induced, is that which would be caused by the application itself.

3.5.3 Modeling Shared Bottlenecks

Network emulation based on path measurements is complicated by the presence of bottlenecks that are shared by multiple paths. Because Flexmon obtains pairwise available bandwidth measurements using independent `iperf` runs, it does not reveal these shared bottlenecks. Thus, modeling flows that originate at the same host but terminate at different hosts as independent can cause inaccuracies. This is mitigated by the fact that if there is a high degree of statistical multiplexing on the shared bottleneck, interference by other flows dominates interference by the application's own flows [59]. In that case, modeling the application's flows as independent remains a reasonable approximation.

In the "cloud" configuration of Dummynet we model flows originating at the same host as being noninterfering. To understand how well this assumption holds, we measured multiple simultaneous flows on PlanetLab paths, shown in Table 3.2. For each path we ran three tests in sequence for 30 seconds each: a single TCP

Table 3.2. Available bandwidth estimated by multiple iperf flows, in bits per second. The PCH to IRO path is administratively limited to 10 Mbps, and the IRP to UCB-DSL path is administratively limited to 1 Mbps.

	Sum of multiple TCP flows		
Path	1 flow	5 flows	10 flows
Commodity Internet Paths			
PCH to IRO	485 K	585 K	797 K
IRP to UCB-DSL	372 K	507 K	589 K
PBS to Arch. Tech.	348 K	909 K	952 K
Internet2 Paths			
Illinois to Columbia	3.95 M	9.05 M	9.46 M
Maryland to Calgary	3.09 M	15.4 M	30.4 M
Colorado St. to Ohio St.	225 K	1.20 M	1.96 M

iperf, five TCP iperfs in parallel, and finally ten TCP iperfs in parallel. The reverse direction of each path, not shown, produced similar results.

Our experiment reveals a clear distinction between paths on the commodity Internet and those on Internet2. On the commodity Internet, running more TCP flows achieves only marginally higher aggregate throughput. On Internet2, however, five flows always achieve much higher throughput than one flow. In all but one case, ten flows also achieve significantly higher throughput than five. Thus, our previous assumption of noninterference holds true for the Internet2 paths tested, but not for the commodity Internet paths.

This difference may be a consequence of several possible factors. It could be due to the fundamental properties of these networks, including proximity of bottlenecks to the end hosts and differing degrees of statistical multiplexing. It could also be induced by peculiarities of PlanetLab. Some sites impose administrative limits on the amount of bandwidth PlanetLab hosts may use, PlanetLab attempts to enforce fair-share network usage between slices, and the TCP stack in the PlanetLab kernel is not tuned for high performance on links with high bandwidth-delay products (in particular, TCP window scaling [58] is disabled).

To model this behavior, we developed additional simple Dummynet configurations. In the “shared” configuration, a node is assumed to have a single bottleneck that is shared by all of its outgoing paths, likely its last-mile link. In the “hybrid” configuration, some paths use the cloud model and others the shared model. The rules for hybrid nodes are: If a node is an Internet2 node, it uses the cloud model for Internet2 destination nodes, and the shared model for all non-Internet2 destination nodes. Otherwise, it uses the shared model for all destinations. The bandwidth for shared pipes is set to the maximum found for any destination in the experiment. Flexlab users can select which Dummynet configuration to use.

Clearly, more sophisticated shared-bottleneck models are possible for the simple models, and we have explored some in follow-on work [118]. Our ACIM model, discussed next, sidesteps this issue by taking a completely different approach to the shared-bottleneck problem.

3.6 Application-Centric Internet Modeling

The limitations of our simple models led us to develop a more complex technique, *application-centric Internet modeling.* The difficulties in simulating or emulating the Internet are well known [41, 84], though progress is continually made. Likewise, creating good *general-purpose* models of the Internet is still an open problem [40]. While progress has been made on measuring and modeling aspects of the Internet sufficient for certain uses, such as improving overlay routing or particular applications [86, 95], the key difficulty we face is that a general-purpose emulator, in theory, has a stringent accuracy criterion: it must yield accurate results for *any* measurement of *any* workload.

ACIM approaches the problem by *modeling the Internet as perceived by the application*—as viewed through its limited lens. We do this by running the application and Internet measurements simultaneously, using the application's behavior *running inside Emulab* to generate traffic *on PlanetLab* and collect network measurements. The network conditions experienced by this replicated traffic are then applied in near real-time to the application's emulated network environment.

ACIM has five primary benefits. The first is in terms of node and path scaling. A particular instance of any application will use a tiny fraction of all of the Internet's paths. By confining measurement and modeling only to those paths that the application actually uses, the task becomes more tractable. Second, we avoid numerous measurement and modeling problems by assessing end-to-end behavior rather than trying to model the intricacies of the network core. For example, we do not need precise information on routes and types of outages—we need only measure their effects, such as packet loss and high latency, on the application. Third, rare or transient network effects are immediately visible to the application. Fourth, it yields accurate information on how the network will react to the offered load, automatically taking into account factors that are difficult or impossible to measure without direct access to the bottleneck router. These factors include the degree of statistical multiplexing, differences in TCP implementations and RTTs of the cross traffic, the router's queuing discipline, and unresponsive flows. Fifth, it tracks conditions quickly, by creating a feedback loop which continually adjusts offered loads and emulator settings in near real-time.

ACIM is *precise* because it assesses only relevant parts of the network, and it is *complete* because it automatically accounts for all potential network-related behavior. Its concrete approach to modeling and its level of fidelity should provide an environment that experimenters can trust when they do not know their application's dependencies.

Our technique makes two common assumptions about the Internet: that the location of the bottleneck link does not change rapidly (though its characteristics may), and that most packet loss is caused by congestion. In the next section, we first concentrate on TCP flows, then explain how we have extended the concepts to UDP.

3.6.1 Architecture

We pair each node in the emulated network with a peer in the live network as shown in Figure 3.3. The portion of this figure that runs on PlanetLab corresponds with the "network model" element of the Flexlab architecture shown in Figure 3.1. The ACIM architecture consists of three basic parts: an application monitor which runs on Emulab nodes, a measurement agent which runs on PlanetLab nodes, and a path emulator connecting the Emulab nodes. The agent receives characteristics of

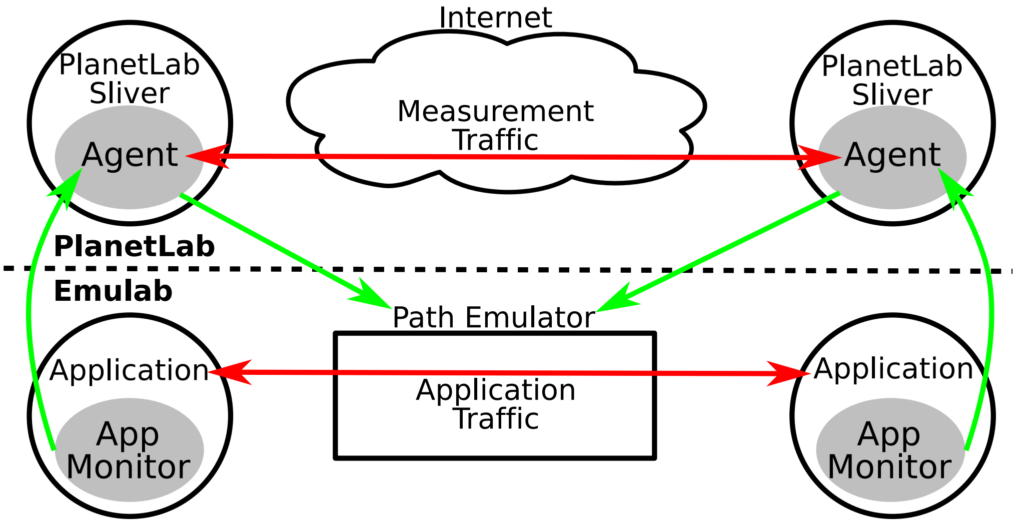


Figure 3.3. The architecture and data flow of application-centric Internet modeling.

the application's offered load from the monitor, replicates that load on PlanetLab, determines path characteristics through analysis of the resulting TCP stream, and sends the results back into the path emulator as traffic shaping parameters. We now detail each of these parts.

3.6.1.1 Application Monitor on Emulab

The application monitor runs on each node in the emulator and tracks the network calls made by the application under test. It tracks the application's network activity, such as connections made and data sent on those connections. The monitor uses this information to create a simple model of the offered network load and sends this model to the measurement agent on the corresponding PlanetLab node. The monitor supports both TCP and UDP sockets. It also reports on important socket options, such as socket buffer sizes and the state of TCP's `TCP_NODELAY` flag.

We instrument the application under test by linking it with a library we created called `libnetmon`. This library's purpose is to provide the model with information about the application's network behavior. It wraps network system calls such as `connect()`, `accept()`, `send()`, `sendto()`, and `setsockopt()`, and informs the application monitor of these calls. In many cases, it summarizes: for example, we do not track the full contents of `send()` calls, simply their sizes and times. `libnetmon` can be dynamically linked into a program using the `LD_PRELOAD` feature of modern operating systems, meaning that most applications can be run without modification. We have tested `libnetmon` with a variety of applications, ranging from `iperf` to Mozilla Firefox to Sun's JVM.

By instrumenting the application directly, rather than snooping on network packets it puts on the wire, we are able to measure the application's *offered load* rather than simply the *throughput achieved*. This distinction is important, because the throughput achieved is, at least in part, a function of the parameters the model has given to the path emulator. Thus, we cannot assume that what an application is *able* to do is the same as what it is *attempting* to do. If, for example, the available bandwidth on an Internet path increases so that it becomes greater than the bandwidth setting of the corresponding path emulator, offering only the achieved throughput on this path would fail to find the additional available bandwidth.

3.6.1.2 Measurement Agent on PlanetLab

The measurement agent runs on PlanetLab nodes, and receives information from the application monitor about the application's network operations. Whenever the application running on Emulab connects to one of its peers (also running inside Emulab), the measurement agent likewise connects to the agent representing the peer. The agent uses the simple model obtained by the monitor to generate similar network load; the monitor keeps the agent informed of the `send()` and `sendto()` calls made by the application, including the amount of data written and the time between calls. The agent uses this information to recreate the application's network behavior by making analogous `send()` calls. Note that the offered load model does not include the packets' payloads, making it relatively lightweight to send from the monitor to the agent.

The agent uses `libpcap` for fine-grained inspection of the resulting packet stream, from which it derives network conditions. For every ACK it receives from the remote agent, it calculates instantaneous throughput and round trip time. For TCP streams, we use TCP's own ACKs, and for UDP, we add our own application-layer ACKs. The agent uses these measurements to generate parameters for the path emulator, as discussed below.

3.6.2 Inference and Emulation of Path Conditions

Our path emulator is an enhanced version of the Dummynet traffic shaper. We emulate the behavior of the bottleneck router's queue within this shaper as shown in Figure 3.4. Dummynet uses two queues: a bandwidth queue, which emulates queuing delay, and a delay queue, which models all other sources of delay, such as propagation, processing, and transmission delays. Thus, there are three important parameters: the size of the bandwidth queue, the rate at which it drains, and the length of time spent in the delay queue. Since we assume that most packet loss is caused by congestion, we induce loss only by limiting the size of the bandwidth queue and the rate it drains.

Because the techniques in this section require that there be application traffic to measure, we use the simple-static model to set initial conditions for each path. They will only be experienced by the first few packets; after that, ACIM provides higher-quality measurements.

3.6.2.1 Bandwidth Queue Size

The bandwidth queue has a finite size, and when it is full, packets arriving at the queue are dropped. The bottleneck router on a real path has a queue whose

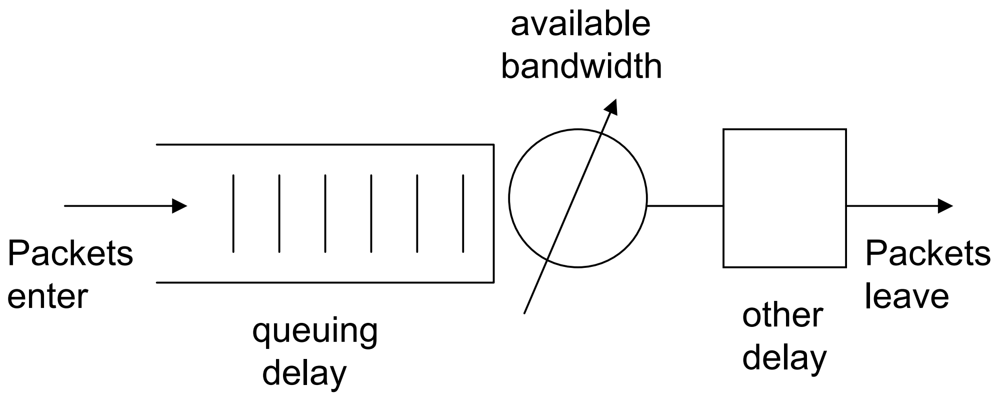


Figure 3.4. Path emulation.

maximum capacity is measured in terms of bytes and/or packets, but it is difficult to directly measure either of these capacities. Sommers et al. [120] proposed using the maximum one-way delay as an approximation of the size of the bottleneck queue. This approach is problematic on PlanetLab because of the difficulty of clock synchronization, which is required to calculate one-way delay. Instead, we approximate the size of the queue in terms of *time*—that is, the longest period one of our packets has spent in the queue without being dropped. We assume that congestion will happen mostly along the forward edge of a network path, and thus can approximate the maximum queuing delay by subtracting the minimum RTT from the maximum RTT. We refine this number by finding the maximum queuing delay just before a loss event.

3.6.2.2 Available Bandwidth

TCP's fairness (the fraction of the capacity each flow receives) is affected by differences in the RTTs of flows sharing the link [80]. Measuring the RTTs of flows we cannot directly observe is difficult or impossible. Thus, the most accurate way to determine how the network will react to the load offered by a new flow is to offer that load and observe the resulting path properties.

We observe the inter-send times of acknowledgment packets and the number of bytes acknowledged by each packet to determine the instantaneous goodput of a connection:

$$goodput = \frac{bytes\ acked}{time\ since\ last\ ack}$$

We then estimate the throughput of a TCP connection between PlanetLab nodes by computing a moving average of the instantaneous goodput measurements for the preceding half-second. This averages out any outliers, allowing for a more consistent metric.

This measurement takes into account the reactivity of other flows in the network. While calculating this goodput is straightforward, there are subtleties in using it to set available bandwidth. The traffic generated by the measurement agent may not fully utilize the available bandwidth. For instance, if the load generated by the application is lower than the available bandwidth or TCP fills the receive window, the throughput does not represent available bandwidth. When this situation is detected, we should not cap the emulator bandwidth to that artificially slow rate. Thus, we *lower* the bandwidth used by the emulator only if we detect that we are fully loading the PlanetLab path. If we see a goodput that is higher than the goodput when we last saturated the link, then the available bandwidth must have increased, and we *raise* the emulator bandwidth.

Queuing theory shows that when a buffered link is overutilized, the time each packet spends in the queue, and thus the observed RTT, increases for each successive packet. Additionally, `send()` calls tend to block when the application is sending at a rate sufficient to saturate the bottleneck link. In practice, since each of these signals is noisy, we use a combination of them to determine when the bottleneck link is saturated. To determine whether RTT is increasing or decreasing, we find the slope of RTT vs. sample number using least squares linear regression.

3.6.2.3 Other Delay

The measurement agent takes fine-grained latency measurements. It records the time each packet is sent, and when it receives an ACK for that packet, calculates the RTT seen by the most recent acknowledged packet. For the purposes of emulation, we calculate the "Base RTT" the same way as TCP Vegas [16]: that is, the minimum RTT recently seen. This minimum delay accounts for the propagation, processing, and transmission delays along the path, factoring out the influence of queuing delay.

We use the base RTT to set the time spent in the delay queue; this avoids double-counting queuing latency, which is modeled in the bandwidth queue. We assume that the base RTT is distributed evenly in the forward and reverse directions and set the delay queue value in each direction to half of the observed base RTT.

3.6.2.4 Outages and Rare Events

There are many sources of outages and other anomalies in network characteristics. These include routing anomalies, link failures, and router failures. Work such as PlanetSeer [145] and numerous BGP studies seeks to explain the causes of these anomalies. Our application-centric model has an easier task: to faithfully reproduce the effect of these rare events, rather than finding the underlying cause. Thus, we observe the features of these rare events that are *relevant* to the application. Outages can affect Flexlab's control plane, however, by cutting off Emulab from one or more PlanetLab nodes. We believe that we can improve robustness by using an overlay network such as RON [6] to distribute control traffic.

3.6.2.5 Per-Flow Emulation

In our application-centric model, the path emulator is used to shape traffic on a per-flow rather than a per-path basis. If there is more than one flow using a path, the bandwidth seen by each flow depends on many variables, including the degree of statistical multiplexing on the bottleneck link, when the flows begin, and the queuing policy on the bottleneck router. We let this contention for resources occur in the overlay network, and reflect the results into the emulator by per-flow shaping.

3.6.3 UDP Sockets

ACIM for UDP differs in some respects from ACIM for TCP. The chief difference is that there are no protocol-level ACKs in UDP. We have implemented a custom application-layer protocol on top of UDP that adds the ACKs needed for measuring RTT and throughput. This change affects only the replication and measurement of UDP flows; path emulation remains unchanged.

3.6.3.1 Application Layer Protocol

Whereas the TCP ACIM sends random payloads in its measurement packets, UDP ACIM runs an application-layer protocol on top of them. The protocol embeds sequence numbers in the packets on the forward path, and on the reverse path, sequence numbers and timestamps acknowledge received packets. Our protocol requires packets to be at least 57 bytes long; if the application sends packets smaller than this, the measurement traffic uses 57-byte packets.

Unlike TCP, our UDP acknowledgements are selective, not cumulative, and we also do not retransmit lost packets. We do not need *all* measurement traffic to get through, we simply measure how much does. An ACK packet is sent for every data packet received, but each ACK packet contains ACKs for several recent data packets. This redundancy allows us to get accurate bandwidth numbers without re-sending lost packets, and works in the face of moderate ACK packet loss.

3.6.3.2 Available Bandwidth

Whenever an ACK packet is received at the sender, goodput is calculated as $g = s/(t_n - t_{n-1})$, where g is goodput, s is the size of the data being acknowledged, t_n is the receiver timestamp for the current ACK, and t_{n-1} is the last receiver ACK timestamp received. By using inter-packet timings from the receiver, we avoid including jitter on the ACK path in our calculations, and the clocks at the sender and receiver need not be synchronized. Throughput is calculated as a moving average over the last 100 acknowledged packets or half second, whichever is less. If any packet loss has been detected, this throughput value is fed to the application monitor as the available bandwidth on the forward path.

3.6.3.3 Delay Measurements

Base RTT and queuing delay are computed the same way for UDP as they are for TCP.

3.6.3.4 Reordering and Packet Loss

Because TCP acknowledgements are cumulative, reordering of packets on the forward path is implicitly accounted for. We must handle it explicitly in the case of UDP. Our UDP measurement protocol can detect packet reordering in both directions. Because each ACK packet carries redundant ACKs, reordering on the reverse path is not a concern. A data packet is considered to be lost if 10 packets sent after it have been acknowledged. It is also considered lost if the difference between the receipt time of the latest ACK and the send time of the data packet is greater than:

$$10 \cdot (average\ RTT + 4 \cdot standard\ deviation\ of\ recent\ RTTs)$$

3.6.4 Challenges

Although the design of ACIM is straightforward when viewed at a high level, a host of complications limit the accuracy of the system. We now describe three of these challenges and how Flexlab addresses them.

3.6.4.1 Libpcap Loss

We monitor the connections on the measurement agent with `libpcap`. The `libpcap` library copies a part of each packet as it arrives or leaves the (virtual) interface and stores them in a buffer pending a query by the application. If packets are added to this buffer faster than they are removed by the application, some of them may be dropped. The scheduling behavior described in Section 3.6.4.3 is a common cause of this occurrence, as processes on PlanetLab can be starved of CPU for hundreds of milliseconds. These dropped packets are still seen by the TCP stack in the kernel, but they are not seen by the application.

This poses two problems. First, we found it not uncommon for all packets over a long period of time (up to a second) to be dropped by the `libpcap` buffer. In this case it is impossible to know what has occurred during that period. The connection may have been fully utilizing its available bandwidth or it may have been idle during part of that time, and there is no way to reliably distinguish between these cases. Second, if only one or a few packets are dropped by the `libpcap` buffer, the false nature of the drops may not be detectable and may skew the calculations.

Our approach is to reset our measurements after periods of detected `libpcap` loss, no matter how small. This avoids the potential hazards of averaging measurements over a period of time when the activity of the connection is unknown. The downside is that in such a situation, a change in bandwidth would not be detected as quickly and we may average measurements over noncontiguous periods of time. We know of no way to reliably detect which stream(s) a `libpcap` loss has affected in all cases, so we must accept that there are inevitable limits to our accuracy.

3.6.4.2 ACK Bursts

Some paths between PlanetLab hosts have anomalous behaviors. The most severe example of this is a path that delivers bursts of acknowledgments over small timescales. In one case, ACKs that were sent over a period of 12 milliseconds arrived over a period of less than a millisecond, an order of magnitude difference. This caused some over-estimation of delay (by up to 20%), and an order of magnitude over-estimation of throughput. We cope with this phenomenon in two ways. First, we use the timestamps that TCP includes in each packet to obtain the ACK *inter-departure times* rather than the ACK *inter-arrival times.* This technique corrects for congestion and other anomalies on the reverse path. Second, we lengthened the period over which we average to 0.5 seconds, which helps to dampen excessive jitter.

3.6.4.3 Scheduling Accuracy

Our experience shows that there is noticeable jitter and delay in process scheduling on PlanetLab nodes; this can have negative effects on own own measurements, as well as "native" PlanetLab experiments. To quantify these properties, we implemented a test program that schedules a sleep with the `nanosleep()` system call, and measures the actual sleep time using `gettimeofday()`. We ran this test on three separate PlanetLab nodes with load averages of roughly 6, 15, and 27 (representative of loads typically seen on PlanetLab), plus an unloaded Emulab node running a PlanetLab-equivalent OS. 250,000 sleep events were continuously performed on each node with a target latency of 8 ms, for a total of approximately 40 minutes.

Figure 3.5 shows the CDF of the undesired additional delay, up to the 90th percentile; Figure 3.6 displays the tail in log-log format. Ninety percent of the events are within -1–5 scheduler quanta (msecs) of the target time. However, a significant tail extends to several hundred milliseconds. We also ran a one week survey of 330 nodes that showed these samples to be representative.

This scheduling tail poses problems for the fidelity of programs that are time-sensitive. Many programs may still be able to obtain accurate results, but it is

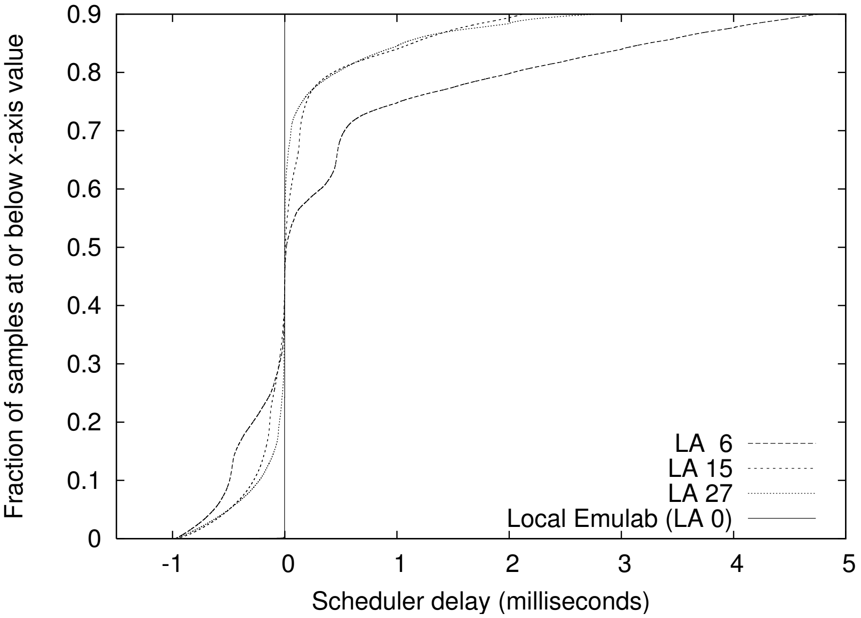


Figure 3.5. 90th percentile scheduling time difference CDF. The vertical line is "Local Emulab."

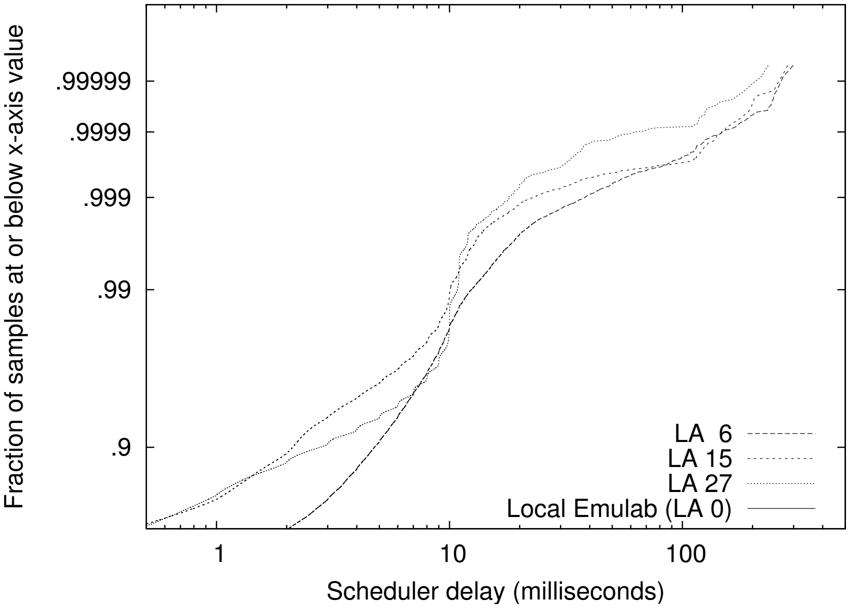


Figure 3.6. Log-log scale scheduling time difference CDF showing distribution tail. The "Local Emulab" line is vertical at $x = 0$.

difficult to determine in advance which those are.

Spring et al. [122] also studied availability of CPU on PlanetLab, but measured it in aggregate instead of our timeliness-oriented measurement. That difference caused them to conclude that "PlanetLab has sufficient CPU capacity." They did document significant scheduling jitter in packet sends, but were concerned only with its impact on network measurment techniques. Our results in Section 3.7.2 strongly suggest that PlanetLab scheduling latency can greatly impact normal applications.

3.7 Evaluation

We evaluate Flexlab by presenting experimental results from three microbenchmarks and a real application. Our results show that Flexlab is more faithful than simple emulation, and can remove artifacts of PlanetLab host conditions. Doing a rigorous *validation* of Flexlab is extremely difficult, because it is impossible to establish ground truth: each environment being compared can introduce its own artifacts. Shared PlanetLab hosts can hurt performance, experiments on the live Internet are fundamentally unrepeatable, and Flexlab might introduce artifacts through its measurement or path emulation. With this caveat, our results show that for at least some complex applications running over the Internet, Flexlab with ACIM produces more accurate and realistic results than running with the host resources typically available on PlanetLab, or in Emulab without network topology information.

3.7.1 Microbenchmarks

We evaluate ACIM's detailed fidelity using `iperf`, a standard measurement tool that simulates bulk data transfers. `iperf`'s simplicity makes it ideal for microbenchmarks, as its behavior is consistent between runs. With TCP, it simply sends data at the fastest possible rate, while with UDP it sends at a specified constant rate. The TCP version is, of course, highly reactive to network changes.

As in all of our experiments, each application tested on PlanetLab and each major Flexlab component are run in separate slices.

3.7.1.1 TCP iperf and Cross-Traffic

Figure 3.7 shows the throughput of a representative two minute run in Flexlab of `iperf` using TCP. The top graph shows throughput achieved by the measurement agent, which replicated `iperf`'s offered load on the Internet between AT&T Labs and the University of Texas at Arlington. The bottom graph shows the throughput of `iperf` itself, running on an emulated path and dedicated hosts inside Flexlab.

To induce a change in available bandwidth, between times 35 and 95 we sent cross-traffic on the Internet path, in the form of ten `iperf` streams between other PlanetLab nodes at the same sites. Flexlab closely tracks the changed bandwidth, bringing the throughput of the path emulator down to the new level of available bandwidth on the real path. It also tracks network changes that we did not induce, such as the one at time 23. However, brief but large drops in throughput occasionally occur in the PlanetLab graph but not the Flexlab graph, such as those

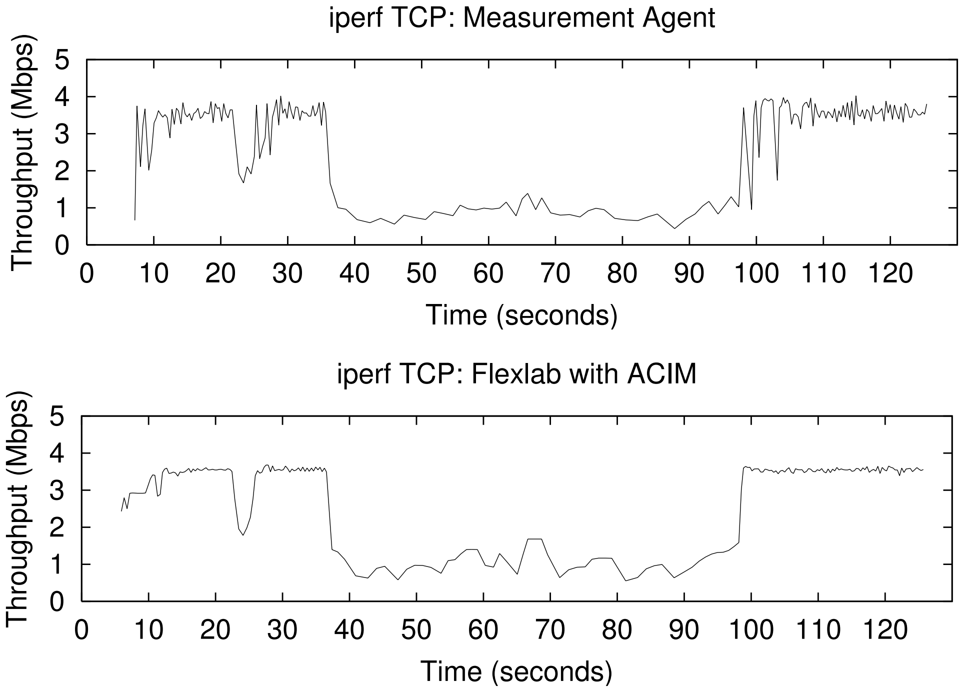


Figure 3.7. Application-centric Internet modeling, comparing agent throughput on PlanetLab (top) with the throughput of the application running in Emulab and interacting with the model (bottom).

starting at time 100. Through log file analysis we were able to determine that these drops are due to temporary CPU starvation on PlanetLab, preventing even the lightweight measurement agent from sustaining the sending rate of the real application. These throughput drops demonstrate the impact of the PlanetLab scheduling delays documented in Section 3.6.4.3. The agent correctly determines that these reductions in throughput are not due to available bandwidth changes, and deliberately avoids mirroring these PlanetLab host artifacts on the emulated path. Finally, the measurement agent's throughput exhibits more jitter than the application's, showing that we could probably further improve ACIM by adding a jitter model.

3.7.1.2 Simultaneous TCP iperf Runs

ACIM is designed to subject an application in the emulator to the same network conditions that application would see on the Internet. To evaluate how well ACIM meets this goal, we compared two instances of `iperf`: one on PlanetLab, and one in Flexlab. Because we cannot expect runs done on the Internet at different times to show the same results, we ran these two instances simultanously. The top graph in Figure 3.8 shows the throughput of `iperf` run directly on PlanetLab between NEC Labs and Intel Research Seattle. The bottom graph shows the throughput of another `iperf` run at the same time in Flexlab using the same pair of hosts. As network characteristics vary over the connection's lifetime, the throughput graphs correspond impressively. The average throughputs are close: PlanetLab was 2.30 Mbps, while Flexlab was 2.41 Mbps (4.8% higher). These results strongly suggest that ACIM has high fidelity. The small difference may be due to CPU load on PlanetLab; we speculate that difference is small because `iperf` consumes few host resources, unlike a real application on which we report shortly.

3.7.1.3 UDP iperf

We have made an initial evaluation of the UDP ACIM support, which is less mature than our TCP support. We used a single `iperf` to generate a 900 Kbps UDP stream. As in Section 3.7.1.1, we measured the throughput achieved by both

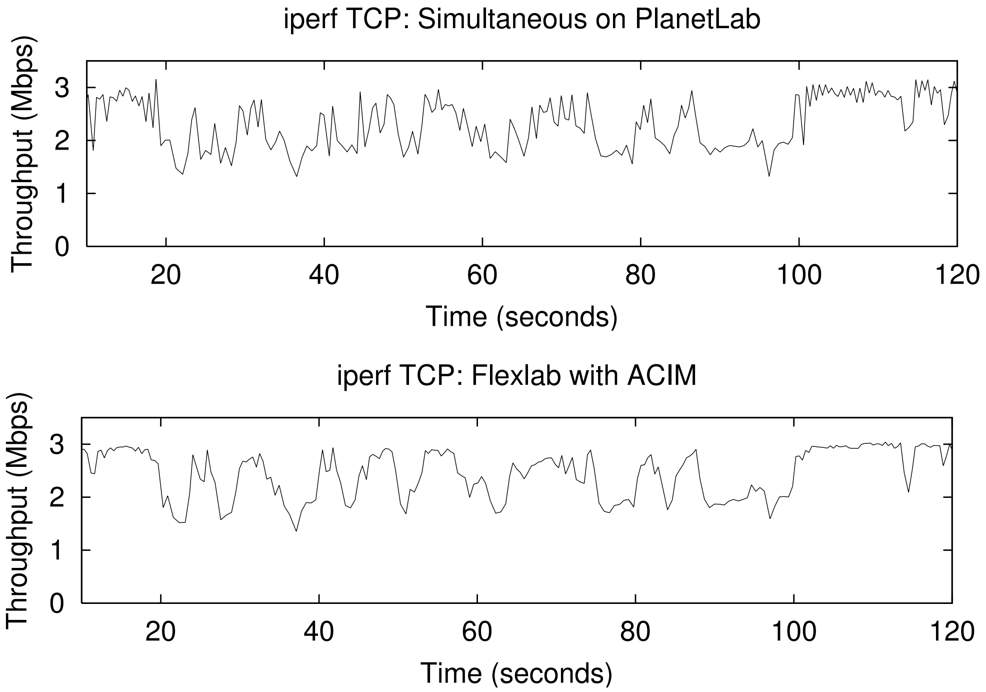


Figure 3.8. Comparison of the throughput of a TCP `iperf` running on PlanetLab (top) with a TCP `iperf` simultaneously running under Flexlab with ACIM (bottom).

the measurement agent on PlanetLab and the `iperf` stream running on Flexlab. The graphs in Figure 3.9 closely track each other. The mean throughputs are close: 746 Kbps for `iperf` and 736 Kbps for the measurement agent, 1.3% lower. We made three similar runs between these nodes, at target rates varying from 800–1200 Kbps. The differences in mean throughput were similar: -2.5%, 0.4%, and 4.4%. ACIM's UDP accuracy appears very good in this range. We leave improvements to Flexlab's UDP model and a more thorough evaluation to future work.

3.7.2 Macrobenchmark: BitTorrent

The next set of experiments demonstrates several properties: first, that Flexlab is able to handle a real, complex, distributed system that is of interest to researchers; second, that PlanetLab host conditions can have an enormous impact on the network performance of real applications; third, that both Flexlab and PlanetLab with host CPU reservations give similar and likely accurate results; and fourth,

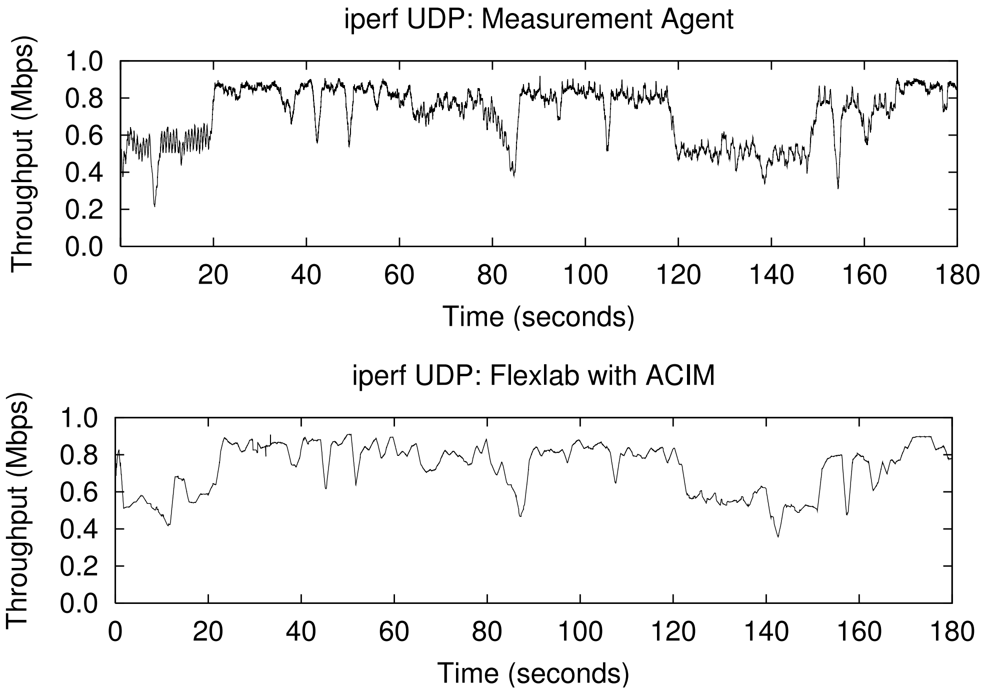


Figure 3.9. The UDP throughput of `iperf` (below) compared with the actual throughput successfully sent by the measurement agent (above) when using the ACIM model in Flexlab.

preliminary results indicate that our simple static models of the Internet don't (yet) provide high-fidelity emulation.

The application that we use for these experiments is BitTorrent, a popular peer-to-peer program for cooperatively downloading large files. Peers act as both clients and servers: once a peer has downloaded part of a file, it serves that to other peers. We modified BitTorrent to use a static tracker, which removes some, but not all, sources of non-determinism from repeated BitTorrent runs.

3.7.2.1 Methodology

Each experiment consisted of a seeder and seven BitTorrent clients, each located at a different site on Internet2 or GÉANT, the US and European research networks, respectively. The sites used are show in Table 3.3. Five- and 15-minute load averages for all nodes except the seeder were typically 1.5 (range 0.5–5); the seed (Stanford) had a load average of 14–29. Runs with a more lightly-loaded seeder gave

Table 3.3. Sites used for BitTorrent macrobenchmarks.

Site	Network	Bandwidth Cap
Stanford	Internet2	10 Mbps
University of Oregon	Internet2	10 Mbps
Carnegie Mellon University	Internet2	5 Mbps
University of South Florida	Internet2	none
University of Texas El Paso	Internet2	none
Internet2 Colocation Facility, Kansas City	Internet2	none
Universität Klagenfurt, Austria	GÉANT	none
TSSG, Waterford Institute of Technology, Ireland	GÉANT	none

similar results. Flexlab/Emulab hosts were all “pc3000”s [35], with 3.0 Ghz Xeon processors, 2GB RAM, and 10K RPM SCSI disks. We used the official BitTorrent program, version 4.4.0, which is written in Python.

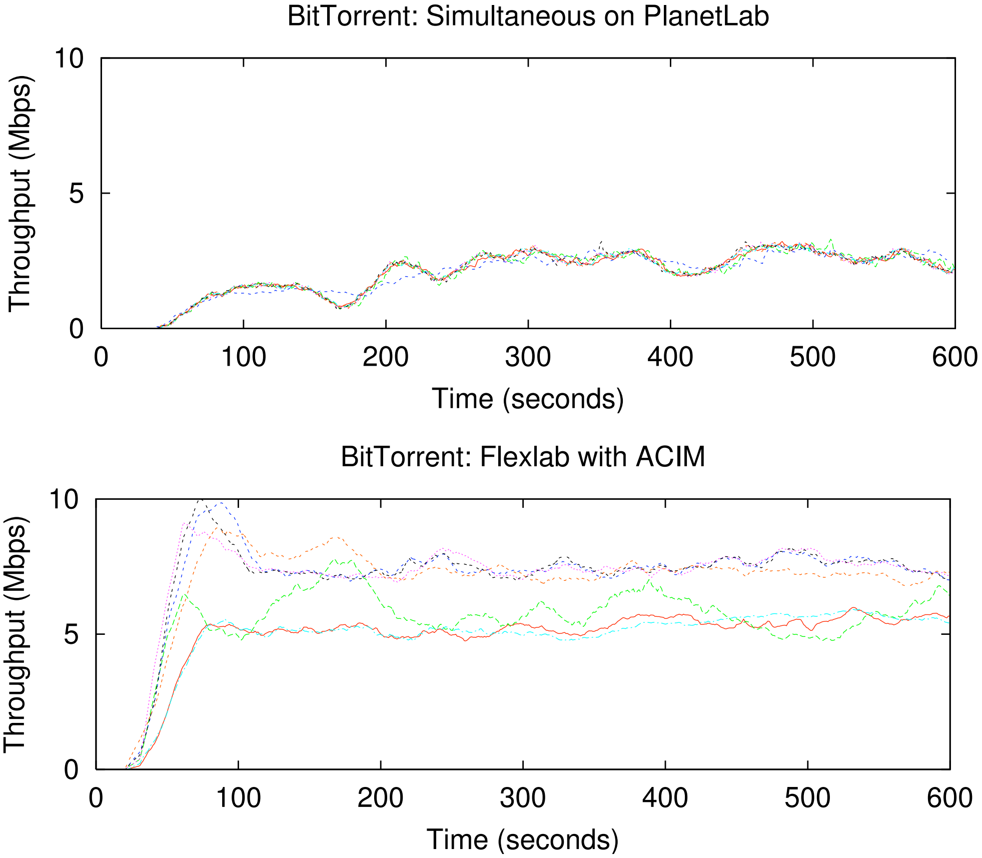
All sites ran PlanetLab version 3.3, and bandwidth caps, for those sites at which they existed, were enforced to all other sites. [1] We ran the experiments for ten minutes, using a file that was large enough that no client could finish downloading it in that period; this enabled us to focus primarily on the steady-state behavior of BitTorrent.

3.7.2.2 ACIM vs. PlanetLab

We began by running BitTorrent in a manner similar to the simultaneous `iperf` microbenchmark described in Section 3.7.1.2. We ran two instances of BitTorrent simultaneously: one on PlanetLab and one using ACIM on Flexlab. These two sets of clients did not communicate directly, but they did compete for bandwidth on the same paths: the PlanetLab BitTorrent directly sends traffic on the paths, while the Flexlab BitTorrent causes the measurement agent to send traffic on those same paths.

[1] PlanetLab 3.3 contained a bug that affected the enforcement of bandwidth caps: the stated policy was that bandwidth limits were not enforced between Internet2 sites due to the over-provisioned nature of that network. This bug caused bandwidth caps to be applied regardless of the site’s network.

Figure 3.10 shows the download rates of the BitTorrent clients, with the PlanetLab clients in the top graph, and the Flexlab clients in the bottom. Each line represents the download rate of a single client, averaged over a moving window of 30 seconds. The PlanetLab clients were only able to sustain an average download rate of 2.08 Mbps, whereas those on Flexlab averaged triple that rate, 6.33 Mbps. The download rates of the PlanetLab clients also clustered much more tightly than in Flexlab. A series of runs showed that the clustering was consistent behavior. The first row of Table 3.4 summarizes those runs, and shows that the throughput differences were also repeatable, with Flexlab receiving higher bandwidth by a factor of 2.5 on average.


BitTorrent: Simultaneous on PlanetLab

BitTorrent: Flexlab with ACIM

Figure 3.10. A comparison of download rates of BitTorrent running simultaneously on PlanetLab (top) and Flexlab using ACIM (bottom). The seven clients in the PlanetLab graph are tightly clustered.

Table 3.4. Mean BitTorrent download rate in Mbps and standard deviation (in parentheses) of multiple Flexlab and PlanetLab runs, as in Section 3.7.2. Since each run was made at a different time, network conditions may have changed between runs.

Experiment	Flexlab	PlanetLab	Ratio
No Sirius (6 runs)	5.78 (0.072)	2.27 (0.074)	2.55 (0.088)
Sirius (5 runs)	5.44 (0.29)	5.24 (0.34)	1.04 (0.045)

These results, combined with the accuracy of the microbenchmarks, suggest that BitTorrent's throughput on PlanetLab is constrained by host overload not found in Flexlab. Our next experiment attempts to test this hypothesis.

3.7.2.3 ACIM vs. PlanetLab With Sirius

Sirius is a CPU and bandwidth reservation system for PlanetLab. It ensures that a sliver receives at least 25% of its host's CPU, but does not give priority access to other host resources such as disk I/O or RAM. Normally, Sirius also includes a bandwidth reservation feature, but to isolate the effects of CPU sharing, we asked PlanetLab operations to disable this feature in our Sirius slice. Only one slice, PlanetLab-wide, can have a Sirius reservation at a time. By using Sirius, we reduce the potential for PlanetLab host artifacts and get a better sense of Flexlab's accuracy.

We repeated the previous experiment fifteen minutes later, with the sole difference that the PlanetLab BitTorrent slice used Sirius. We ran BitTorrent on Flexlab at the same time; its measurement agent on PlanetLab did not have the benefit of Sirius. Figure 3.11 shows the download rates of these simultaneous runs. Sirius more than doubled the PlanetLab download rate of our previous PlanetLab experiment, from 2.08 to 5.80 Mbps. This demonstrates that BitTorrent's download rate is highly sensitive to CPU availability, and that the CPU typically available on PlanetLab is insufficient to produce accurate results for some complex applications. It also highlights the need for sufficient reserved host resources on current and future network testbeds. In this run, the Flexlab and PlanetLab download rates are within 4% of each other, at 5.56 Mbps and 5.80 Mbps, respectively. These results

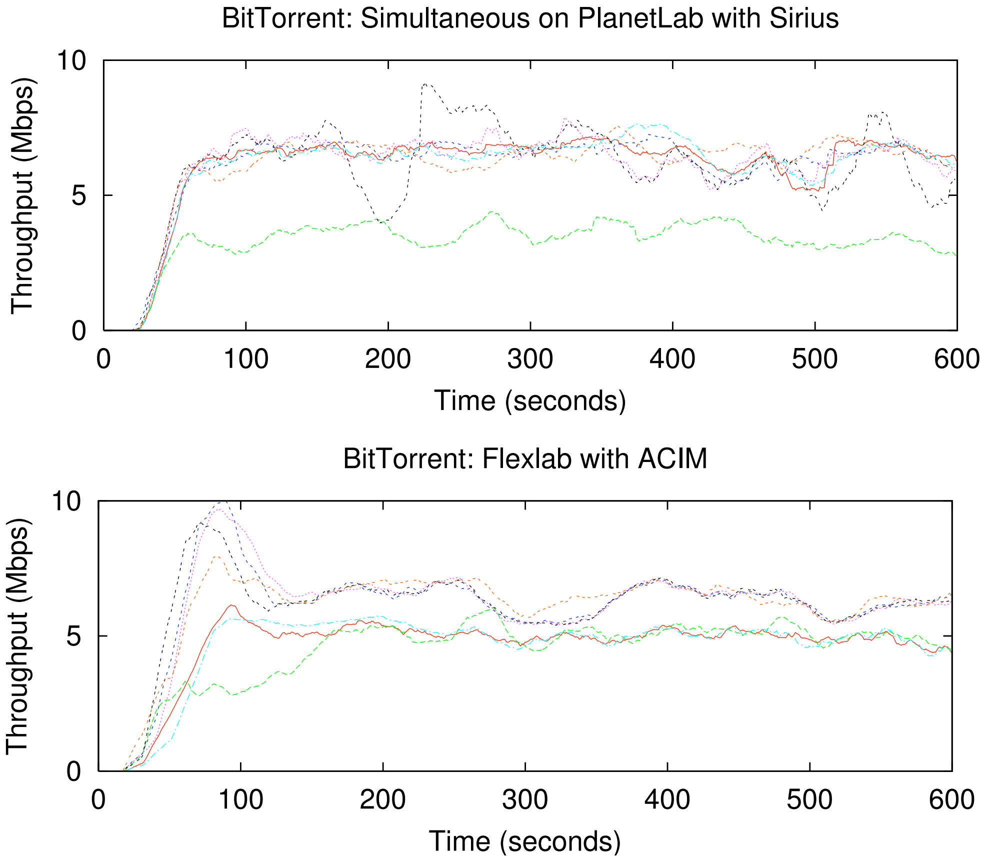


Figure 3.11. Download rates of BitTorrent simultaneously running on PlanetLab with Sirius (top), compared to Flexlab ACIM (bottom).

are consistent, as shown by repeated experiments in the second row of Table 3.4. This indicates that Flexlab with ACIM provides a good environment for running experiments that need PlanetLab-like network conditions without host artifacts.

3.7.2.4 Resource Use

To estimate the host resources consumed by BitTorrent and the measurement agent, we ran Flexlab in a special configuration in which the “PlanetLab side” was run on an emulated network inside of Emulab instead of on a live network using PlanetLab hosts. This allowed us to measure consumption when ample CPU and memory resources are available. The agent took only 2.6% of the CPU, while BitTorrent took 37–76%, a factor of 14–28 higher. The agent’s resident memory use was about 2.0MB, while BitTorrent used 8.4MB, a factor of 4 greater. Because the resource needs of our agent are so much smaller than the original application under test, the agent is much less likely to encounter artifacts in resource-poor environments (such as PlanetLab) than the application itself.

3.7.2.5 Simple Static Model

We ran BitTorrent again, this time using the simple-static model outlined in Section 3.5.1. Network conditions were those collected by Flexmon five minutes before running the BitTorrent experiment described in Section 3.7.2.2, so we would hope to see a mean download rate similar to ACIM's: 6.3 Mbps.[2] We did three runs using the "cloud," "shared," and "hybrid" Dummynet configurations. We were surprised to find that the shared configuration gave the best approximation of BitTorrent's behavior on PlanetLab. The cloud configuration resulted in very high download rates (12.5 Mbps average), and the rates showed virtually no variation over time. Because six of the eight nodes used for our BitTorrent experiments are on Internet2, the hybrid configuration made little difference; all nodes were treated by the hybrid model as cloud nodes. The two GÉANT nodes now had realistic (lower) download rates, but the overall mean was still 10.7 Mbps. The shared configuration produced download rates that varied on timescales similar to those we have seen on PlanetLab and with ACIM. While the mean download rate was more accurate than the other configurations, it was 25% lower than what we would expect, at 5.1 Mbps.

This shows that the shared bottleneck models we developed for the simple models are not yet sophisticated enough to provide high fidelity emulation. The cloud configuration seems to under-estimate the effects of shared bottlenecks, while

[2] The experiment run in Section 3.7.2.2 differed from this one in that the former generated traffic on PlanetLab from two simultaneous BitTorrent runs, while this experiment ran only one BitTorrent at a time. This methodological difference could explain much of the difference between ACIM and the simple cloud model, but only if the simultaneous BitTorrent's in Section 3.7.2.2 significantly affected each other. This seems unlikely due to the high degree of statistical multiplexing we expect on Internet2 and GÉANT paths, both from our knowledge of those networks and from the results in Section 3.5.3. However, this assumption needs further study.

the shared configuration seems to over-estimate them, though to a lesser degree. Some of our follow-on work [118] has made progress in improving these models.

3.8 Related Work

Network measurement to understand and model network behavior is a popular research area. There is an enormous amount of related work on measuring and modeling Internet characteristics including bottleneck-link capacity, available bandwidth, packet delay and loss, topology, and more recently, network anomalies. Examples include [23, 27, 121, 79, 120, 140]. In addition to their use for evaluating protocols and applications, network measurements and models are used for maintaining overlays [6] and even for offering “underlay” services [95]. PlanetLab has attracted many measurement studies specific to it [122, 82, 145, 104]. Zhang et al. [146] showed that there is significant stationarity of Internet path properties, but argued that this alone does not mean that the latency characteristics important to a particular application can be modeled sufficiently with a stationary model.

Monkey [22] collects live TCP traces near servers to faithfully replay client workload. It infers some network characteristics. However, Monkey is tied to a webserver environment, and does not easily generalize to arbitrary TCP applications. Jaisal et al. did passive inference of TCP connection characteristics [60], but focused on other goals, including distinguishing between TCP implementations.

Trace-Based Mobile Network Emulation [99] has similarities to our work, in that it used traces from mobile wireless devices to develop models to control a synthetic networking environment. However, it emphasizes production of a parameterized model, and was intended to collect application-independent data for specific paths taken by mobile wireless nodes. In contrast, we concentrate on measuring ongoing Internet conditions, and our key model is application-centric.

SatelliteLab [29] uses an idea similar to Flexlab: it runs an application on one set of hosts while using network conditions from another. In SatelliteLab, applications are run on PlanetLab hosts and agents are run on “satellite” hosts nearby. Traffic from the application is sent from a PlanetLab host to a nearby satellite, transmitted to another satellite, then delivered to a PlanetLab node near the second satellite. This fills a different role than Flexlab: it allows an application running on PlanetLab to see the network from the perspective of the satellites. The goal of SatelliteLab is to increase the heterogeneity of network viewpoints; PlanetLab is known to be heavily biased towards educational institutions, and SatelliteLab seeks to allow other sites to participate by running lightweight satellites rather than full PlanetLab hosts. SatelliteLab does not attempt to affect the controllability or repeatability of experiments, and applications must still run on resource-starved PlanetLab nodes.

3.8.1 Overlay Networks

Our ACIM approach can be viewed as a highly unusual sort of overlay network. In contrast to typical overlays designed to provide resilient or optimized services, our goal is to provide realism—to *expose* rather than mitigate the effects of the Internet. A significant practical goal of our project is to provide an experimentation platform for the development and evaluation of "traditional" overlay networks and services. By providing an environment that emulates real-world conditions, we enable the study of new overlay technologies designed to deal with the challenges of production networks.

Although our aims differ from those of typical overlay networks, we share a common need for measurement. Recent projects have explored the provisioning of common measurement and other services to support overlay networks [86, 95, 77, 105]. These are exactly the types of models and measurement services that our new testbed is designed to accept.

Finally, both VINI [11] and Flexlab claim "realism" and "control" as primary goals, but provide different types of realism and control. In VINI, realism means the ability to carry real end-user traffic by peering with real ISPs. Control is experimenter-controlled routing, forwarding, and fault injection, and provisioning of some dedicated links. In contrast, Flexlab's realism is the inclusion of real, variable Internet conditions and dedicated hosts. Experimenters' control in Flexlab is over pluggable network models, the complete hardware and software environment of the hosts, and rich experiment management.

3.9 Conclusion

Flexlab combines two popular network testbed environments: PlanetLab, a live-network testbed, and Emulab, an emulation testbed. In doing so, it creates a new environment that allows experimenters to make tradeoffs between the strong points of each: the control and repeatability of emulation, and the realism of an overlay testbed. Our results show that ACIM is able to achieve high fidelity, producing network conditions that closely track those seen on PlanetLab, but without many of the artifacts that come from running in a highly shared environment.

Flexlab is not, however, suitable for all types of network experiments. It focuses exclusively on the end-to-end properties of paths, abstracting over the details of the interior of the network. For many types of experiments, this is sufficient: the system under test is one that is meant to be run at the edges of a network, and the experimenter wishes to examine the sensitivity of the system to high-level properties such as path latency and available bandwidth. When testing a system that is deployed within the network, such as a routing protocol, packet forwarding scheme, or middlebox, it is not enough to treat the interior of the network as a "black box." For such experiments, what is needed is a different kind of realism: a realistic topology, rather than realistic end-to-end conditions. Because it can be difficult to obtain topologies from real networks, many experimenters turn to topology generators. Topologies generated this way typically do not, however, include the IP addresses necessary to use them inside of an emulator. Realistically assigning such addresses is the subject of our next chapter.

CHAPTER 4

REALISTIC AND SCALABLE IP ADDRESS ASSIGNMENT

4.1 Overview

Some types of emulated experiments require realistic topologies for the interior of the network. In order to get such topologies, experimenters commonly turn to topology generators or network tomography. These sources of topologies, however, typically do not come annotated with IP addresses. This presents a problem for their use in emulation, which, because it uses full IP stacks, requires the use of appropriate addresses.

In this chapter, we consider the problem of leveraging topological information to automate the assignment of IP addresses to the nodes in a network. Because addresses in real IP networks are typically assigned with the natural hierarchy of the network as a key consideration, our method for automatically assigning them is built around this idea as well. We formalize the problem and point to several practical considerations that distinguish it from related theoretical work. We then describe several of the algorithmic directions and metrics we have explored. Some are based on previous graph partitioning work and others are based on our own methods.

Because IP routing is hierarchical by nature, an assignment that exploits the hierarchy of the topology naturally minimizes the sizes of the routing tables on the nodes. We use this metric to gauge the effectiveness of our methods. The other metric that we use for evaluation is runtime: emulations and simulations can reach sizes of thousands of nodes, so it is important that this automated assignment scale well. We compare our algorithms on a variety of real and automatically generated router-level Internet topologies. Our two best algorithms, yielding the highest quality namings, can assign addresses to networks of 5000 routers, comparable to today's largest single-owner networks, in 2.4 and 58 seconds.

4.2 Introduction

Assigning names to the nodes of a network for the purposes of addressing and routing is a fundamental aspect of networking. In today's Internet, IP addresses are typically allocated manually by administrators. In several important experimental contexts, however, manual allocation is cumbersome or entirely impractical. In particular, network emulators and simulators use ever-larger generated topologies, which do not come annotated with IP addresses. Automated IP address assignment is required for large scale network emulation and some realistic simulation [101].

Though the main goal of our work is to support emulation and simulation, there are potential applications beyond these domains. Some overlay networks choose to use a virtual IP address space to name their members and there are increasingly more enabling technologies [106, 126] and reasons [8] for deploying such virtualized networks. Occasionally, even operators of real networks redesign their address assignment schemes, and that process can be aided by automation. For example, the University of Utah has completed a project to re-assign addresses to its entire 20,000+ node network because the old assignment had led to unacceptably large routing tables. Many enterprises use memory-constrained legacy routers, which may be overburdened by routing tables due to poor address assignment.

Fundamentally, a desirable address assignment is one that reflects the underlying hierarchy of the network. A significant caveat is that real topologies are not strictly hierarchical, and thus the challenges of identifying a suitable hierarchical embedding of the topology come to the fore. This challenge—inferring hierarchy in this practical setting—is the focus of this chapter. We explore the problem from several directions and produce two particularly successful address assignment algorithms with entirely different approaches: an algorithm that makes use of the unique graph-theoretic properties of the domain and a heuristic that produces assignments that are nearly as good, but at much lower computational cost.

We generate network addresses for use with Classless Inter-Domain Routing (CIDR), the dominant routing scheme used in the Internet. In CIDR, a route for an entire IP prefix can be specified with a single table entry, making it inherently hierarchical; an address assignment that exploits hierarchy in the topology leads to small routing tables. Thus, we use total routing table size as a metric to evaluate the quality of our assignments. This has the valuable side effect of producing assignments that lead to efficient routing table storage.

While there are many factors that influence address assignments in real networks, such as the policies and organic growth [4] of the organizations that own them, hierarchy is natural in large-scale IP networks and required for scaling. Thus, by assigning addresses in a way that matches the natural hierarchy of the network, we produce automatic assignments that account for the primary factor in real assignments, though there are secondary factors that we do not model.

This chapter makes the following contributions:

- We build upon a theoretical formulation of interval routing to formulate the IP address assignment problem and help to open a new area of study by bringing theoretical work to bear on this practical problem (Section 4.3)

- We define a new concept, "Routing Equivalence Sets," and use it as a metric to quantify the extent to which routes to sets of destinations can be aggregated (Section 4.6.1.1)

- We develop three classes of algorithms to optimize IP naming, each using a fundamentally different approach to attack the problem (Section 4.6)

- We devise a pre-processing step that improves the running times of several of our algorithms by orders of magnitude without sacrificing solution quality (Section 4.5)

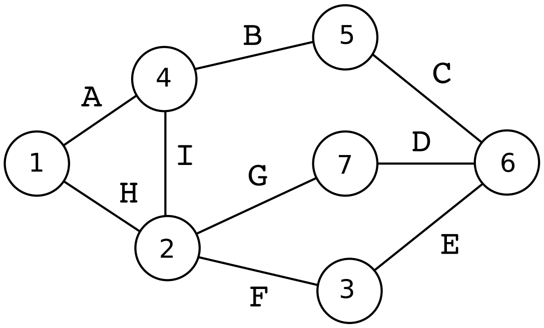
- We devise methods for compacting the optimized namings to fit within practical limits required by IPv4 (Section 4.7)

- We implement the algorithms and evaluate them on a number of topologies. We find two of them, recursive partitioning and tournament RES, to be particularly effective and efficient enough to run on topologies as large as today's largest single-owner networks (Section 4.9)

4.3 Problem Statement

This chapter seeks to produce a global address assignment automatically, i.e., an assignment in which IP addresses are assigned to each network interface in a network. In practice, IP address assignment directly impacts the sizes of routing tables, since a set of destinations with contiguous IP addresses that share the same first hop can be captured as a single routing table entry. Thus, by leveraging properties of the topology, we can produce a naming that seeks to minimize total routing table size. It is also essential to name hosts from a compact namespace, as the available address space is limited. Naturally, it is also important to consider the running time of an assignment algorithm in evaluating its effectiveness. We formulate our assignment problem first using the clean conceptual notion of *interval routing*, and then describe the additional constraints that CIDR prefixes and CIDR aggregation impose on the problem.

As an example of interval routing, consider the network represented by the graph in Figure 4.1, in which nodes are assigned addresses from $\{1, \ldots, 7\}$. Interval



Node 1		Node 7	
Range	First Hop	Range	First Hop
2–3	H	1–4	G
4–6	A	5–6	D
7	H		

Figure 4.1. A 7-node network with two interval routing tables. Vertices are labeled with numbers and edges are labeled with letters. The first hops in the routing tables are designated by the label of the first edge to traverse.

routing table entries are shown by the table in Figure 4.1 for the interfaces of nodes 1 and 7. Node 7 can express its shortest-path routes with two disjoint intervals, one per interface, and therefore has a routing table of size two. With the given address assignment, Node 1 must use three disjoint intervals to exactly specify the routes on its outbound interfaces. Note that in this example, ties between shortest-path routes can be exploited to minimize routing table size. For example, the routing table at Node 7 elected to group Node 3 on the same interface as nodes 1, 2, and 4 to save two table entries.

For a formal definition of interval routing, consider an n-node undirected graph $G = (V, E)$, where we will refer to vertices as hosts, and an edge (u, v) as a pair of interfaces (one at vertex u and one at vertex v). An address assignment $\mathcal{A}$ assigns each vertex in V a unique label from the namespace of integers $\{1, \dots n\}$. The interval routing table of vertex u associates every vertex x with one edge (u, v) (the next hop towards x). In this manner, a subset of labels in $\mathcal{A}$ is associated with each edge. Interval routing compacts the routing table by expressing each of these subsets of labels in $\mathcal{A}$ as a set of intervals of integers.

On a node using interval routing, the size of the minimal set of intervals is the routing table size, or the *compactness* of the routing table. We denote the number of entries in the routing table of vertex u by k_u. The theory literature has considered questions such as determining the minimum value of k for which an assignment results in routing tables all of size smaller than k [130, 47, 39]. For a given graph, this value of k is defined to be the *compactness* of the graph. We are primarily concerned with the *total* routing table size, so we work with the following objective function:

Definition 1 *For a graph G, generate an address assignment $\mathcal{A}$ that minimizes*

$$\sum_{u \in V} k_u$$

It is well known that search and decision problems of this form are NP-complete, and several heuristics and approximation algorithms are known [47]. Our focus is on the practical considerations that cause CIDR routing to be a significantly different problem than interval routing.

4.3.1 Practical Considerations

There are three main differences between the theoretical approach to compact addressing that we have described so far and the actual addressing problem that must be solved in emulation and simulation environments. First, although interval routing is intuitively appealing and elegant, routing table aggregation in practice is performed using the set of classless interdomain routing (CIDR) rules [46], adding significant complexity. Second, in IP addressing, each individual interface (outbound edge) of a node is assigned an address (label), not each vertex, adding subtleties to the naming process. Finally, widely used local-area network technologies such as Ethernet provide all-to-all connectivity, and these networks are best described by *hypergraphs* [12], not ordinary graphs.

4.3.1.1 CIDR

CIDR specifies aggregation rules that change the problem in the following ways. A CIDR address is an aggregate that is specified as a prefix of address bits of arbitrary length. It encompasses all the IP addresses that match that prefix. This implies that a CIDR address can express only those intervals of IP addresses that are a power of two in size and that start on an address that is a multiple of that same power of two. In other words the interval must be of the form $[c * 2^y, (c+1) * 2^y)$ for integers c and y. This more restrictive aggregation scheme means that an IP assignment must be aligned in order to fully take advantage of aggregation. In practice, dealing with this alignment challenge consumes many bits of the namespace, and *address space exhaustion* becomes an issue even when the number of interfaces is much smaller than the set of available names. We explore this restriction further in Section 4.7. Note that interval routing runs into no such difficulty. A second difference between interval routing and CIDR aggregation arises because CIDR routing tables accommodate a *longest matching prefix* rule. With longest matching prefix, the intervals delimited by CIDR routing table entries may overlap, but the longest (and consequently most specific) matching interval is used for routing. The ability to use overlapping intervals is advantageous for CIDR, as it admits more flexibility than basic interval routing.

4.3.1.2 Labeling Interfaces

When IP addresses are assigned, they are assigned to network interfaces, not hosts. For single-homed hosts this is immaterial, but for hosts with multiple interfaces, such as network routers, this distinction can impact address assignment, as these multihomed hosts are associated with multiple addresses. Within a single autonomous system (AS) using shortest-path routing, when a packet is sent to any one of a host's addresses, it is typically expected to take the shortest path to *any* interface on the host. As a result, it is valuable to be able to aggregate all addresses assigned to a host. This means that we must not only be concerned with how links aggregate with each other, but also with how the interfaces on a host aggregate as well.

4.3.1.3 Hypergraphs

The networks we consider in simulation and emulation environments are best represented as *hypergraphs*, since they often contain local-area networks such as Ethernet, which enable all-pairs connectivity among a set of nodes rather than connectivity between a single pair of nodes. A hypergraph captures this, since it is a generalized graph in which each edge is associated with a set of vertices rather than a pair of vertices. As before, when assigning addresses to a hypergraph, we must assign addresses to individual network interfaces. With the hypergraph representation, this becomes more difficult to reason about, since each network edge may be associated with a set of vertices of arbitrary size.

To address this, we work instead with the dual hypergraph [12]; to find the dual hypergraph of a given topology, we create a hypergraph with vertices that correspond to links in the original topology and hyperedges that correspond to hosts in the original topology. Each vertex in the dual hypergraph is incident on the edges that represent the corresponding hosts in the original graph. For example, Figure 4.2 shows the dual of the topology in Figure 4.1. By labeling vertices of the dual hypergraph, we are labeling the network LANs and links in the original topology. We label the vertices with IP subnets, and then assign an address to each interface from the subnet of its adjacent LAN. By operating on whole LANs, rather than their constituent hosts, we also gain scaling benefits: our algorithms' runtimes scale in relation to the number of LANs and links in the topology, rather than the number of hosts. In many edge networks, single-homed hosts in large LANs constitute the majority of nodes in the network, resulting in dramatic improvements in the runtimes of our algorithms.

In the remainder of this chapter, when we discuss graphs, we refer to the dual hypergraph of a topology unless otherwise noted.

4.4 Algorithmic Contributions

We decompose solutions to the IP address assignment problem into three steps:

1. Graph Preprocessing: Because the running times of our algorithms are dependent upon the size of the graph, we provide methods to reduce the size of the input topology by identifying and removing subgraphs whose addresses can be assigned optimally using only local information

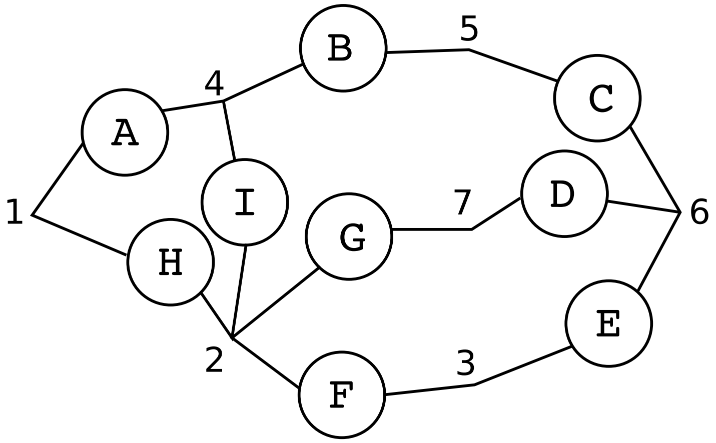


Figure 4.2. The dual hypergraph of Figure 4.1. Each host in the original graph becomes a hyperedge and each link becomes a vertex.

2. Trie Embedding: We then embed the vertices of the graph into the leaves of a binary trie, where each internal node represents a logical subnet of its associated leaves and encompasses the interval of IP addresses of its children. This step is the linchpin of our approach, and we compare several different methods with which we have experimented

3. Address Compaction: To minimize the impact of address space exhaustion, we devise a postprocessing step that reorients the tree to minimize its height

The methods for the steps above constitute the main technical contributions of this chapter. We describe these algorithms in the following sections and evaluate their practical effectiveness in Section 4.9.

4.5 Graph Preprocessing

Most of the algorithms that we propose for the key step of Trie Embedding have superlinear time complexity in the size of the graph, which limits their scalability on large topologies. To achieve scaling we have devised a pre-pass phase which meets two goals: (1) identify and remove subgraphs for which locally optimal address assignment is possible and (2) decompose the remaining input topology into subgraphs to which addresses can be independently assigned.

To achieve the first goal, we use the fact that there are some structures for which there are simple optimal algorithms for address assignment, like trees. Such structures are relatively common in some types of networks, such as at the periphery of enterprise and campus networks. They are also seen frequently in the synthetic topologies used in simulation and emulation, and thus it is worth optimizing these common cases. To achieve the second goal, we take advantage of the fact that any singly connected component, i.e., a subgraph where removal of a single edge called a *bridge* breaks the component in two, is also amenable to preprocessing. Here, address labelings for the subgraphs on either side of the bridge can be generated independently with a minimal impact on the overall quality of the approximation. The property we exploit is this: if each biconnected component [25] is assigned a unique prefix, the internal assignment of addresses within a component does not change the number of routes of any host outside of that component. By identifying trees and bridge edges (both of which can be done in linear time), the pre-pass phase naturally decomposes the graph into a set of smaller biconnected components and trees, as shown in Figure 4.3.

While the preprocessing step has obvious benefits, there are also some less obvious costs. First, there are some technicalities introduced by our need to work with the dual hypergraph. Second, the partitioning performed in the pre-pass typically leads to a small increase in routing table sizes.

4.5.1 Hypergraph Biconnectivity and Hypertrees

To perform the pre-pass, we must extend the definitions of biconnectivity and trees into the domain of hypergraphs. A number of alternative definitions potentially apply; we use the following one which best fits our purposes.

For every path p, the function edges(p) is the set of edges or hyperedges along that path. (We will use the term 'edge' in a general sense to denote either an edge

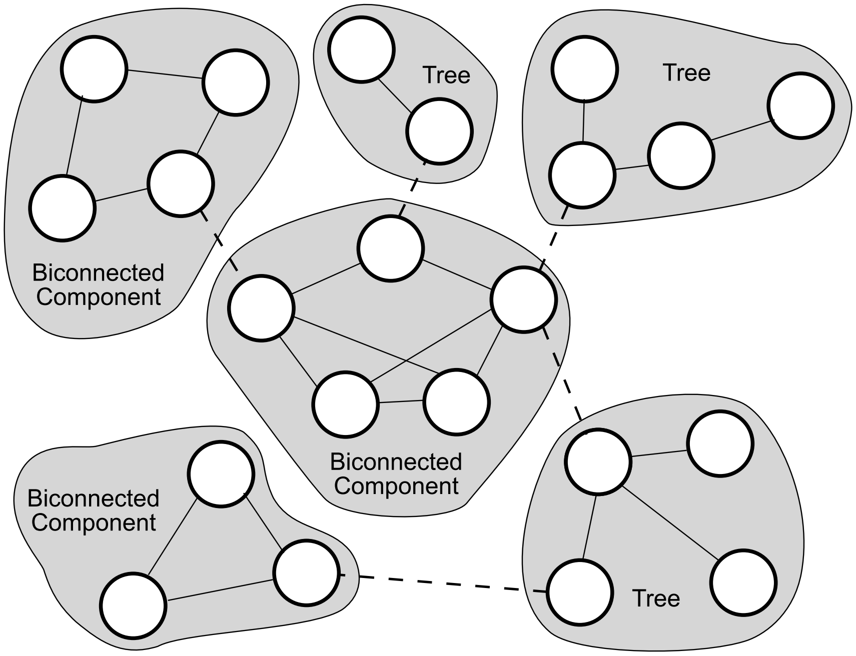


Figure 4.3. The pre-pass partitions the graph into trees and biconnected components. Bridges are shown as dashed lines.

or a hyperedge.) A pair of vertices u and v is said to be *edge-biconnected* if and only if there exist two paths p and q between u and v such that:

$$\text{edges}\,(p) \cap \text{edges}\,(q) = \emptyset$$

Similarly, an *edge-biconnected component* is a set of vertices V such that for all u, v in V, u and v are edge-biconnected. An *edge-biconnected partitioning* of a graph G is a partitioning of the vertices of G into partitions $G_1, G_2, \ldots G_n$ such that for all i, G_i is a maximal edge-biconnected component.

Using similar notions, we define a *hypertree* to be a connected subgraph of a hypergraph that contains no cycles. As with trees on regular graphs, it is straightforward to optimally assign IP addresses to a hypertree of a hypergraph.

Using these definitions, our pre-processing step partitions the hypergraph into edge-biconnected components and hypertrees. Fast algorithms for computing such a decomposition on regular graphs are known; by maintaining some additional information about vertices incident to each hyperedge, these methods can be extended to apply to hypergraphs. Once the decomposition is complete, addresses on the hypertrees are assigned optimally by a special tree-assignment procedure; addresses are assigned on the edge-biconnected components by the procedures described in Section 4.6 The super-graph of partitions is created and can be used to label the partitions themselves; each node is labeled by a concatenation of its partition's label and its label within the partition.

4.5.2 Increase in Routing Table Size

Suppose the partitioning elects to separate biconnected components A and B by cutting a bridge edge (a, b) for some vertices $a \in A$ and $b \in B$. Our methods will then (naturally) assign a an address in the space assigned to subnet A, allowing all vertices in B to use a single routing table entry to reach all of A. But consider the nonintuitive assignment of giving a an address in subnet B instead. This has the likely effect of complicating routing tables of hosts within B. However, all tables in A stand to gain, as the hosts can route to all of $B \bigcup \{a\}$ with a single entry. The best choice of address assignment on the cut boundary depends on the relative sizes of A and B and their internal topologies. When the pre-pass is used, it prevents us from taking advantage of this opportunity, and thus the pre-pass comes at some cost. In Section 4.9.3 we evaluate the trade-offs that the pre-pass imposes, and find that the decrease in runtime is well worth this small cost.

4.6 Trie Embedding

We explore several methods for embedding the vertices of the graph into a binary trie. Some are of our own devising, and some leverage work from the graph partitioning community.

The goal of each of these algorithms is to build a binary trie, with nodes in the trie corresponding to IP subnets. A binary trie is a special case of a binary tree in which left branches are labeled with 0, right branches are labeled with 1, and nodes are labeled with the concatenation of the labels on edges along the path from the root to the node. Using the trie we build, each leaf, representing a vertex, is given an IP subnet corresponding to its trie label, appended with zeroes to fill out the address in the event the label length is smaller than the desired address length.

The choice of embedding determines the smallest routing table size that can be achieved at a given host; different embeddings can clearly have significant impact.

Our three algorithms for trie embedding each approach the problem differently. Our first algorithm uses a bottom-up greedy tournament to create a binary trie. The second is a top-down approximation using graph partitioning methods. The third approach decomposes trie embedding into two simpler subproblems: (1) identifying an appropriate ordering of the vertices, and (2) embedding that ordering into a trie.

4.6.1 Bottom-Up Tree Building

Our first approach leverages the intuition that a natural way to assign addresses is bottom-up, i.e., to identify groups of vertices that can be combined into a logical subnet to decrease the routing tables on other hosts in the network. In terms of the trie that is constructed, the grouping operation corresponds to producing a rooted trie for the entire subnet, where the children are the groups of vertices that were present prior to the coalescence operation. Indeed, if we have an appropriate function benefit(S, T) that quantifies the benefit of merging arbitrary sets of vertices S and T, then we can construct a binary trie via the following greedy tournament:

Greedy Tournament

```
$X = \{\{v_1\}, \{v_2\}, \{v_3\}, \dots\}$
**repeat until** $|X| = 1$
    For all $S, T \in X$, compute benefit$(S, T)$
    For maximizing benefit$(S, T)$, create $U = S \bigcup T$
    Delete $S$ and $T$ from $X$.
    Add $U$ to $X$.
**end repeat**
```

There are two key challenges to realizing this approach: defining an appropriate benefit function and avoiding the time complexity embodied in naïve direct implementation of this approach, which involves $\mathrm{O}(n^2)$ computations of benefit(S, T) in each of $n - 1$ rounds.

4.6.1.1 Routing Equivalence Sets

To motivate the derivation of an appropriate benefit function for the tournament above, we consider two sets of vertices that constitute logical subnets S_1 and S_2, and perform the thought experiment: is S_1 a good candidate for aggregation with S_2? To quantify the benefit, consider that of the vertices in $V \setminus (S_1 \bigcup S_2)$, there will be some set of vertices that use the same first hop to all vertices in $(S_1 \bigcup S_2)$, and thus could express them in a single routing table entry if we give all vertices in $(S_1 \bigcup S_2)$ addresses that allow aggregation. Some vertices will require different first hops to reach the vertices in $(S_1 \bigcup S_2)$, and thus cannot aggregate routes to them. Therefore, the benefit of aggregating S_1 and S_2 is proportional to the size of the first set, or the external vertices that can save a routing table entry.

Following this intuition, we have devised Routing Equivalence Sets (RES) as a way to characterize the benefit of aggregating the addresses of sets of vertices. For a set of vertices D, those vertices whose first-hop routes to *all* vertices in D are identical are said to be in the Routing Equivalence Set of D, res(D). Equivalently, if vertices of D are assigned IP addresses in the same subnet, then a routing table in any member of res(D) can store all routes to D with a single routing table entry. Formally, let V be the set of vertices in a graph. Let D be a set of destination vertices, a subset of V. Let $H_x[y]$ be the first hop from source vertex x to destination vertex y. Then we define res(D) as:

$$\mathrm{res}\,(D) = \{v \in V : \forall d, e \in D, H_v[d] = H_v[e]\}.$$

Figures 4.4 and 4.5 show a concrete example of RES, using the example graph from Figure 4.1. In each, the first-hop routes from each of the vertices not in D to each vertex in D are shown. Vertices that have a single outbound arrow, such as vertex 4 in Figure 4.4, use the same first hop to every vertex in D, and are thus members of res(D). Vertexes with multiple outbound arrows, such as vertex 5, must store multiple first hops to reach all of D along shortest paths, and are thus not members of res(D).

We use RES to measure the impact on routing table sizes of aggregating sets of vertices. As shown in Figures 4.4 and 4.5, since $|\,\mathrm{res}(\{2,3,7\})| > |\,\mathrm{res}(\{5,6,7\})|$, it is more advantageous to aggregate the former set than the latter. We can then use the size of the RES set as the measure in performing pairwise comparisons: the maximum benefit merger in the *Greedy Tournament* is the one where the RES of the aggregated subnet is maximized.

One potential issue is that finding a RES set directly from this definition is costly: computing res(D) has time complexity $O(n|D|^2)$. However, we can prove that res(D) has a recursive decomposition that is amenable to much more efficient computation by the following lemma:

Lemma 1 *For any sets D and E, and given* res(D) *and* res(E), res($D \cup E$) *can be computed in time O(n).*

Proof: First note that by transitivity, for any v and for all $a, b, c \in V$:

$$(H_v[a] = H_v[b] \wedge H_v[b] = H_v[c]) \rightarrow (H_v[a] = H_v[c])$$

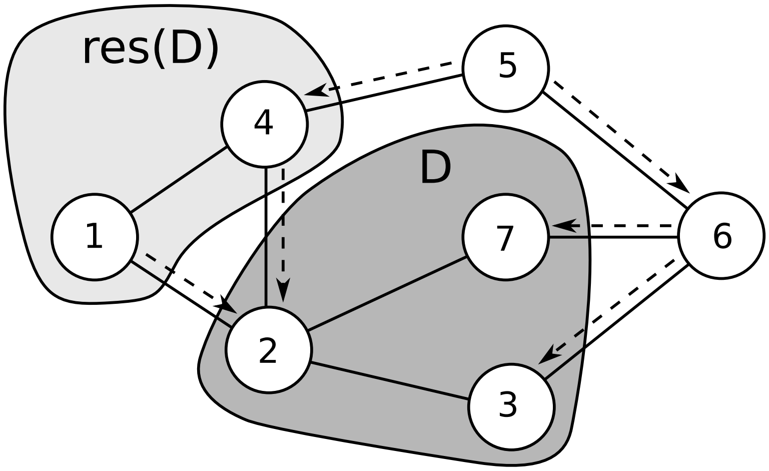


Figure 4.4. res(D) for set $D = \{2, 3, 7\}$. First-hop routes from vertices outside D to the members of D are shown as arrows.

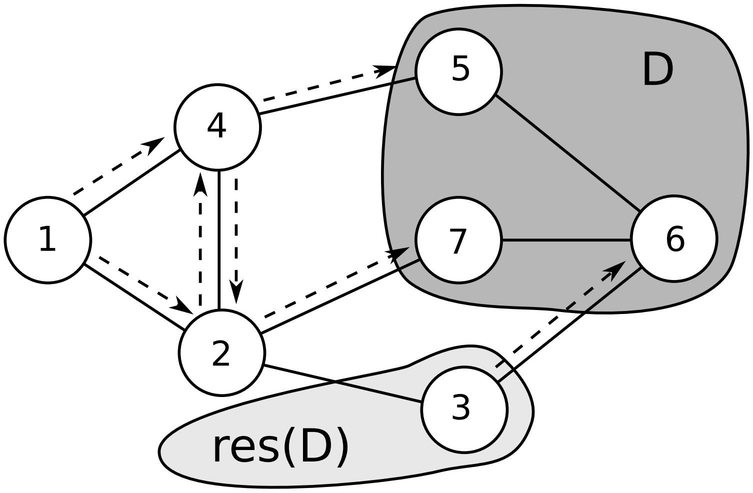


Figure 4.5. res(D) for set $D = \{5, 6, 7\}$.

Further, the definition of RES and transitivity imply that for destination set D, and specializing to $v \in \operatorname{res}(D)$ and $d, e \in D$,

$$H_v[a] = H_v[d] \rightarrow H_v[a] = H_v[e]$$

which means that $\forall v \in V, \forall d \in D$:

$$\operatorname{res}(D \cup \{v\}) = \operatorname{res}(D) \cap \operatorname{res}(\{v, d\})$$

Therefore, given two destination sets D and E, we can select *any* $d \in D$ and $e \in E$ to give the recurrence:

$$\operatorname{res}(D \cup E) = \operatorname{res}(D) \cap \operatorname{res}(E) \cap \operatorname{res}(\{d, e\})$$

Since res($\{d, e\}$) can be computed from the definition in O(n) time, and assuming use of standard set representations that allow intersection in linear time, the lemma follows. ■

4.6.1.2 Efficient Tournament Design

Using the RES metric, we now use the greedy tournament algorithm to build a binary address assignment trie from the vertices in the graph. We determine the cost of merging and the order of the coalescence operations by setting benefit(S, T) to res(S, T). Next we demonstrate how to improve upon the running time of the tournament. Let n be the number of vertices in the graph. A straightforward implementation of the RES tournament takes $n - 1$ rounds, and must find the best of $O(n^2)$ RES sets, each of which takes $O(n)$ time to construct in the worst case. This leads to a running time of $O(n^4)$. Although n^2 pairwise combinations must be considered for each of $n - 1$ rounds, there is an optimization available that cuts the running time by a factor of n. In the first round, we must compute the RES metric for all n singletons, and all n^2 possible combinations of singletons. In subsequent rounds, however, the RES values of most of the n^2 combinations have not changed—in fact, the only ones that have changed are those pairs in which S or T were one of the combined vertices. There are at most $2n$ such combinations. So, in fact, we can store all possible combinations in a priority queue, and only update those entries that change based upon the winners of each round. Scoring a pairwise combination of two sets with RES can be done in $O(n)$ time, as proved in Lemma 1. Thus, all rounds after the first take $O(n^2)$ time, and there are $n - 1$ of them, leading to an overall time complexity of $O(n^3)$ for the tournament. Storing the values of combinations under consideration in a priority queue requires $O(n^2)$ space.

4.6.2 Recursive Graph Partitioning

An alternative approach to trie-building is to perform a top-down decomposition of the graph. Recursive graph partitioning is a widely-studied decomposition method. At each step, an input topology is partitioned into a set of subtopologies. Typically, a partition is constrained by requiring each of the subtopologies to have some minimum size, and feasible partitions are scored by a metric, such as the size of the edge cut set induced by the partition. While optimal graph partitioning is NP-hard in general, the problem is well studied and a number of algorithms provide good heuristic approximations. We chose the widely-used METIS package [71] because of its maturity and its high performance.

We use METIS to perform a full recursive decomposition of the graph, using the minimum-cut metric, down to the vertex level. Such a decomposition naturally creates a tree (not necessarily binary) in which an internal node represents a subgraph and its children reflect the one-round recursive decomposition of that graph. Since METIS performs each round of partitioning in linear time, a full decomposition takes time $O(n \log n)$ (provided that each round reduces the size of the largest subgraph by a constant factor).

By using minimum edge cut for decomposition, we cut the topology where it is "narrow." Most nodes in each partition will have a small number of first hops to nodes in the other partitions, since there are few cross-partition edges over which traffic can be routed. By placing each partition in its own IP subnet, we make it likely that the members of each partition will be able to aggregate the members of the other into a small number of routes.

One characteristic of METIS and graph partitioners in general is that they require the user to specify the number of partitions to create at each round. Since our trie-building algorithm recursively calls METIS on each partition, we can bypass the problem of selecting the number of partitions by using METIS exclusively for bisection. We tested this intuition by experimenting with search-based approaches to choosing the number of partitions, at each round partitioning the graph into 2–20 subgraphs, scoring each different partitioning, and selecting the partitioning with the best score. We tried several scoring metrics for evaluating the number of partitions and found two promising ones: conductance [70] and ratio cut [136]. Comparing these two search-based approaches with one another and with bisection, we empirically determined that the number of partitions per level had relatively little impact on the final routing table size. For simplicity and runtime performance, we settled on bisection, as it conveniently produces a binary trie. A final component of a top-down algorithm is an appropriate termination condition. The obvious termination condition is when the partition is trivial (size 1). However, in some cases, this can lead to a trie that is too deep for the limits of IP addressing; we address this in more detail in Section 4.7.

4.6.3 Spectral Orderings

The final set of approaches we consider are two-phase methods that first produce an order on the vertices in the graph and then embed this ordering into a trie.

Our main motivation is twofold: first, we speculated that the use of spectral methods, and in particular, use of the Laplacian, might well provide an ordering of the vertices that could be leveraged to produce a good binary tree embedding. Second, we are often given an ordering of the vertices when the test topology is specified, and we noticed that this (nonrandom) default ordering often captures some interesting locality in the graph. We were curious as to the quality that an embedding of this default ordering would provide.

4.6.3.1 The Laplacian Ordering

Our starting point is a standard technique from graph theory, that of obtaining an ordering using the Laplacian matrix [38] of the graph and the eigenvector associated with the second-smallest eigenvalue of the matrix [93, 68]. We refer to the second-smallest eigenvalue as λ_2 and the associated eigenvector by $\vec{v_2}$. The Laplacian matrix is essentially a representation of the adjacency of vertices in the graph, and thus it contains only *local* information about each node. The vector $\vec{v_2}$ contains a value for each vertex in the graph. These values can be used to generate an approximation for a minimum-cut bipartitioning of the graph. The characteristic value for each vertex relates to its distance from the cut, with vertices in the first partition having negative values and those in the second partition having positive values. By sorting the vertices by their characteristic values, we obtain a spectral ordering.

4.6.3.2 DRE Ordering

A limitation of using only the second-smallest eigenvector of the Laplacian is that this captures notions of adjacency, but does not necessarily capture the notions of similarity between vertices from the perspective of routing. We therefore considered an alternative Laplacian-like graph that goes beyond 0/1 adjacency values and instead incorporates real-valued coefficients that reflect the degree of similarity between a pair of vertices. To do so, we use RES to create a new metric, called Degree of Routing Equivalence (DRE). DRE is defined for a pair of vertices $i, j \in V$, as:

$$\mathrm{dre}(i, j) = |\,\mathrm{res}(\{i, j\})|$$

We then construct an n by n matrix containing the DRE for every pair of vertices. In essence, what we have created is similar to the Laplacian of a fully-connected graph, with weights on the edges such that the higher the edge weight, the more benefit is derived from placing two vertices together. This more directly captures the routing properties of vertices than the standard Laplacian. As with the Laplacian, we then take a characteristic valuation of the matrix to obtain an ordering.

4.6.3.3 From Ordering to Trie Embedding

After an ordering has been generated, constructing an appropriate trie embedding is relatively straightforward. A tournament similar to the *Greedy Tournament* can be run, except that not all pairs of vertices need be considered in each round; only those remaining vertices that are adjacent in the original ordering are considered. This reduces the tournament running time by another factor of n, since there are now only $\mathrm{O}(n)$ such pairs in a given round, not $\mathrm{O}(n^2)$. Finally, by the same trick used in the original tournament, only two new combinations need be scored in rounds subsequent to the first, so the total time complexity of the tournament is $O(n^2)$. In the event that the ordering has done an effective job in grouping vertices that are similar from a routing perspective, then the intuition is that the tournament algorithm will produce a good assignment tree.

4.7 Address Compaction

A key practical limitation is that the IP address space has a fixed size: in IPv4, each address is limited to 32 bits. Since each level in a trie represents a successively longer prefix, if the trie is too deep, the resulting addresses will require more bits than are available. As we saw in Section 4.3.1, the nature of IP requires that addresses be aligned, and a good assignment may "waste" parts of the address space in order to produce small routing tables. Thus, the trie-building algorithms must gracefully deal with bitspace exhaustion. We have developed and implemented bitspace compaction algorithms for our two best algorithms, tournament RES and recursive partitioning.

When there is sufficient bitspace, both algorithms produce a full-depth trie. When they detect limited bitspace, they proceed until there is just enough bitspace for the remainder of the hierarchy. At that point, their sole objective becomes minimizing bitspace consumption. Therefore, the bottom-up algorithm produces poorer quality assignments at the top of the trie, while top-down is poorer at the bottom.

4.7.1 Bottom-Up Compaction

The tournament trie building algorithm operates on a forest of subtrees, combining pairs of subtrees to build a single trie bottom-up. Combining two trees of depths p and q results in a tree of depth $\max(p, q) + 1$. When $p \neq q$, inefficient bitspace usage occurs, as the resulting tree is not a full tree. Such inefficiencies are common, as the goal of the tournament is to minimize routing tables, not to create the minimum-height tree, and these two goals are often at odds.

In order find a minimal tree given a set of subtrees, we simply combine the two trees with the smallest depth and repeat until only a single tree remains. This depth can be calculated in time linear in the number of subtrees to be combined. When the tournament combines two subtrees, the new minimal tree can be calculated incrementally in constant time. After a round of the tournament, if the height of the minimum-depth tree is equal to the total number of address bits, the algorithm halts and yields the minimum-depth tree.

4.7.2 Top-Down Compaction

The recursive partitioning algorithm operates from the top down. At each stage, we check to see if the partition would result in a subgraph too large to fit into the IP subnet available to it. If such a situation occurs, we do not continue to recursively partition; we simply assign sequential addresses to all LANs in the subgraph. At each level of recursion, this invariant is checked before recursing further. Thus, we proceed with recursive partitioning until running out of address space, then fall back to simple sequential assignment. Sequential assignment makes compact use of the address space, but can be inefficient with respect to routing table size.

4.8 Putting It All Together

Figure 4.6 shows how the methods detailed in the previous sections fit together. All of the trie embedding schemes go through the pre-pass except for the recursive partitioner, which is sufficiently fast that it does not require this step. After the pre-pass, each component of the graph is processed separately to obtain a local naming. We then recombine the components and run them through the address compaction algorithm to produce a global naming. Finally, we compute the routing tables for all nodes and compress them with ORTC [30], which, for a given naming, produces routing tables that are provably optimal.

4.9 Experimental Results

We now evaluate the algorithms presented in the previous sections by using them on a variety of topologies, and comparing their resulting routing table sizes and runtimes.

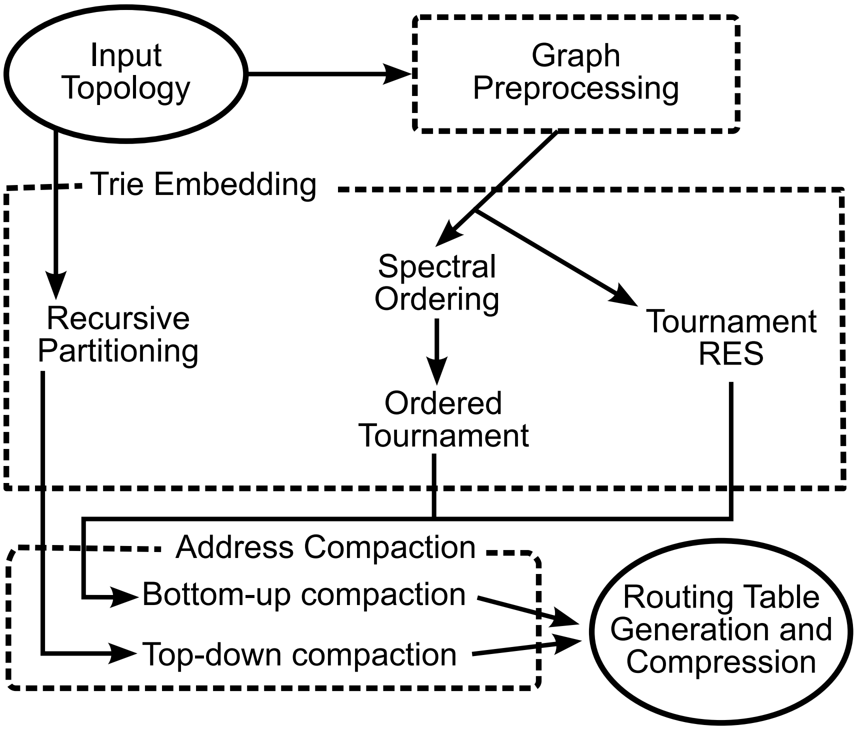


Figure 4.6. A flowchart showing how the different algorithms presented in this chapter are combined.

4.9.1 Methodology

We ran experiments on topologies from three sources: two popular router-level topology generators and topologies gathered using Internet mapping techniques. Our primary interest lies with the generated topologies because such topologies are prevalent in simulation and emulation. The real Internet topologies serve two purposes. First, they give us insight into the applicability of our methods on ISP and enterprise networks. Second, new research in topology models and generators [4, 88] is improving the degree to which they are representative of the real Internet—thus, these topologies give us a sense of how well our methods will operate on future generations of topology generators.

The first set of topologies are generated by the BRITE [91] topology generator, using the GLP [18] model proposed by Bu and Towsley. These topologies are intended to model the topology within a single organization, ISP, or AS. The second set of topologies are generated by the GT-ITM [143] topology generator. They model topologies that include several different administrative domains. Thus, they contain at least two levels of inherent hierarchy. Finally, we use real-world topologies gathered by the Rocketfuel topology mapping engine [121]. These are maps of real ISPs, created from traceroute data of packets passing through the ISP. All Rocketfuel graphs are of individual transit ASes. Although the Rocketfuel topologies are annotated with some IP addresses, there are not enough to reconstruct routing tables or interpolate the missing addresses. This has two consequences. First, it means that these topologies cannot be directly used in an emulator without techniques like ours for address assignment. Second, it means that we cannot make fair comparisons between our assignments and the assignments on the real-world networks that Rocketfuel has mapped.

We compare against two different baseline results. The first is a complete binary tree with the vertices of the graph placed randomly as leaves. This provides an upper bound: such an assignment does not take the network topology into account at all, and a topology-aware assignment should be able to produce smaller routing tables. In our results, we call this method "Random." Second, we create another complete trie and order the vertices according to the input order (that is, the order output by the topology generator). When topology generators output the graphs they have created, they serialize these graphs into an output file. We have found that, in some cases, this serialization process places nodes that are in similar parts of the topology close to each other in the output. Thus, assigning addresses based on this ordering can yield reasonable results, and it yields an estimate of how much extra information is provided by the topological generation or discovery process. We refer to this method as "Default Order."

For each topology we report the number of interfaces, rather than the number of nodes. This gives a more accurate view of the complexity of the assignment problem, since it is interfaces that must be named. All topologies are router-level topologies—they contain no end hosts. End hosts do not significantly impact the complexity of the assignment problem, because they tend to be organized into relatively large LANs, which can be assigned as a single subnet; each LAN appears a single node in the dual hypergraph.

All of our experiments were run on PCs with Pentium IV processors running at 3.0 GHz, 1 MB of L2 cache, and 2 GB of main memory.

4.9.2 Full-Graph Algorithms

We begin by comparing results for our assignment algorithms without using the pre-pass stage, as this isolates the performance of the algorithms themselves.

4.9.2.1 BRITE Topologies

Figure 4.7 shows the global aggregate routing table size produced by each method for the BRITE topology set. We see that all of our algorithms do significantly better than random assignment, with the best (recursive partitioning) saving 42% of routes over Random in a graph containing 2500 interfaces. For these topologies, the assignment derived from the default ordering is indistinguishable from a random assignment. Recursive partitioning and tournament RES perform similarly, producing the smallest routing tables among the methods we studied. The two spectral ordering methods also give similar performance, falling between the random assignment and the best methods.

4.9.2.2 GT-ITM Topologies

Figure 4.8 shows results from the GT-ITM topology set. For this set, the route savings are more pronounced—the best improvement we see over random assignment is 70% fewer routes. However, the various assignment algorithms are far more clustered: most result in similar routing table sizes. It is interesting to note that the default ordering is competitive with the more sophisticated orderings on this set, indicating that, unlike in the BRITE graphs, the order in which nodes are emitted from the generator is correlated with their routing similarity.

4.9.2.3 Rocketfuel Topologies

Rocketfuel graphs provide some idea of how our methods compare on real topologies. The Rocketfuel results are shown in Table 4.1 for two European networks,

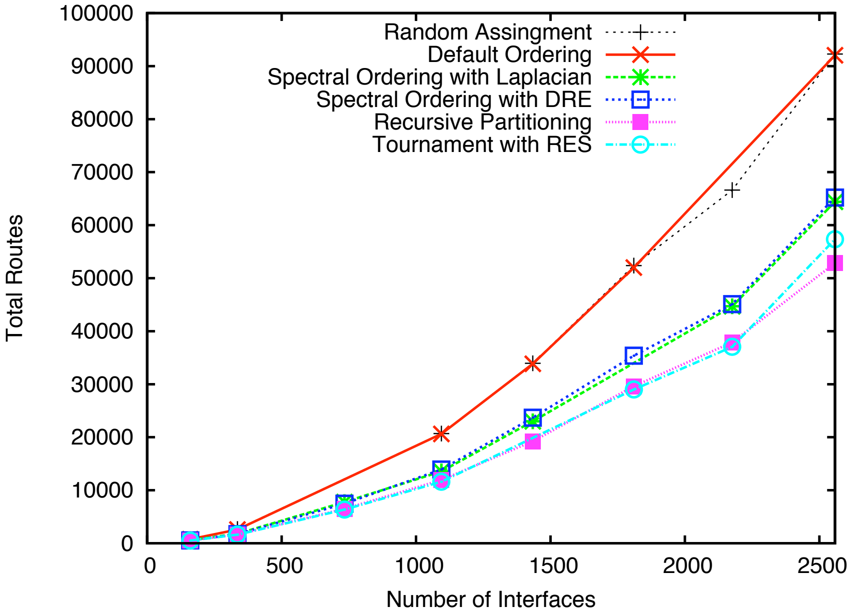


Figure 4.7. Global number of routes for a variety of assignment algorithms for the BRITE topology set.

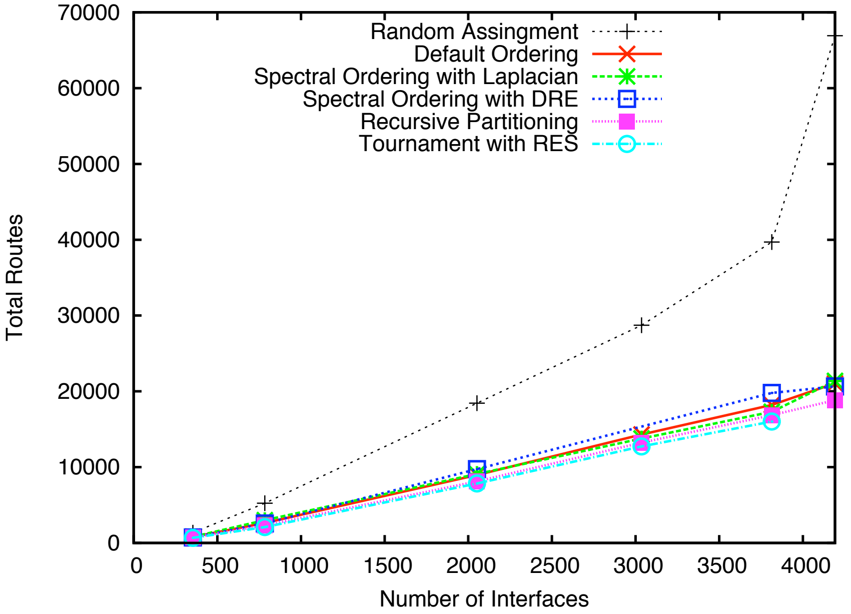


Figure 4.8. Global number of routes for a variety of assignment algorithms for the GT-ITM topology set.

Table 4.1. Number of routes generated for the Rocketfuel topologies. Improvement over random assignment is shown in parentheses.

Algorithm	EBONE	Tiscali
Random	28955	35109
Default Ordering	18364 (37%)	21697 (38%)
Spectral Laplacian	18128 (37%)	22761 (35%)
Spectral DRE	15581 (46%)	20579 (41%)
Recursive Partitioning	11630 (60%)	16802 (52%)
Tournament RES	11427 (61%)	16354 (53%)

EBONE and Tiscali. Like the BRITE topology set, the tournament RES and recursive partitioning algorithms perform similarly, with a slight advantage going to tournament RES. The default ordering is again far better than random, but this time, the two best algorithms provide a large improvement over the default ordering. The spectral methods give results similar to the default ordering.

4.9.2.4 Runtime Comparison

Figure 4.9 shows the runtimes for the BRITE topology set for our full-graph assignment algorithms. Here, recursive partitioning is the clear winner: its runtime appears nearly linear, while the other methods show quadratic behavior in their runtime. Tournament RES showing the poorest scaling, while the two spectral ordering methods have nearly identical runtimes.

4.9.3 Pre-pass Effects

We now evaluate the pre-pass, its effects on address assignment and runtime, and a give a characterization of the subgraphs it generates. We evaluate tournament RES and the spectral ordering with DRE on the GT-ITM topologies; recursive partitioning is fast enough on its own that it does not require the pre-pass, and because the two spectral methods perform similarly, we omit the results for the Laplacian method.

We expect to see three effects. First, the pre-pass finds tree-like structures and assigns addresses to them optimally, which will tend to improve solution quality.

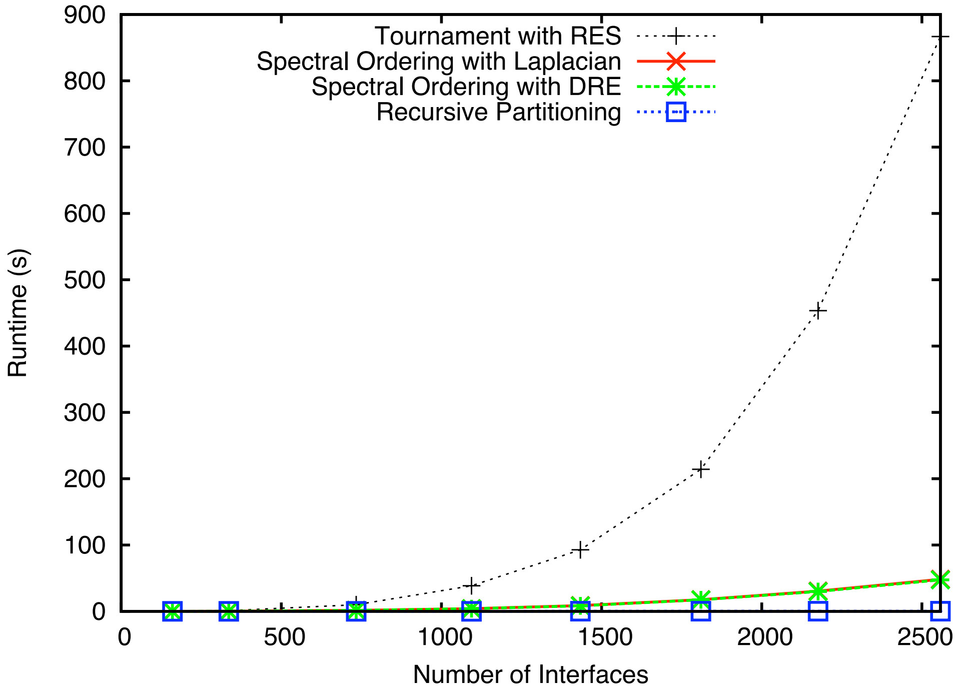


Figure 4.9. Runtimes in seconds for a variety of assignment algorithms, on the BRITE topology set. The recursive partitioning line is nearly coincident with the X axis.

Second, because of the effects discussed in Section 4.5, the pre-pass may force the use of suboptimal decisions at the edges of biconnected components, tending to decrease solution quality. Third, we expect to see a dramatic reduction in overall runtime by reducing the effective input sizes of the subgraphs.

4.9.3.1 Routing Table Size

Figure 4.10 shows that for this set of topologies, the positive and negative effects of the pre-pass on the solution quality largely balance each other out. The routing table sizes seen from tournament RES are almost identical with and without the pre-pass, while spectral ordering obtains 10–15% improvement from the pre-pass.

4.9.3.2 Runtime Benefit

Figure 4.11 shows the improvement in runtime due to the pre-pass. The improvement is dramatic; at 3800 interfaces, the runtime for tournament RES drops

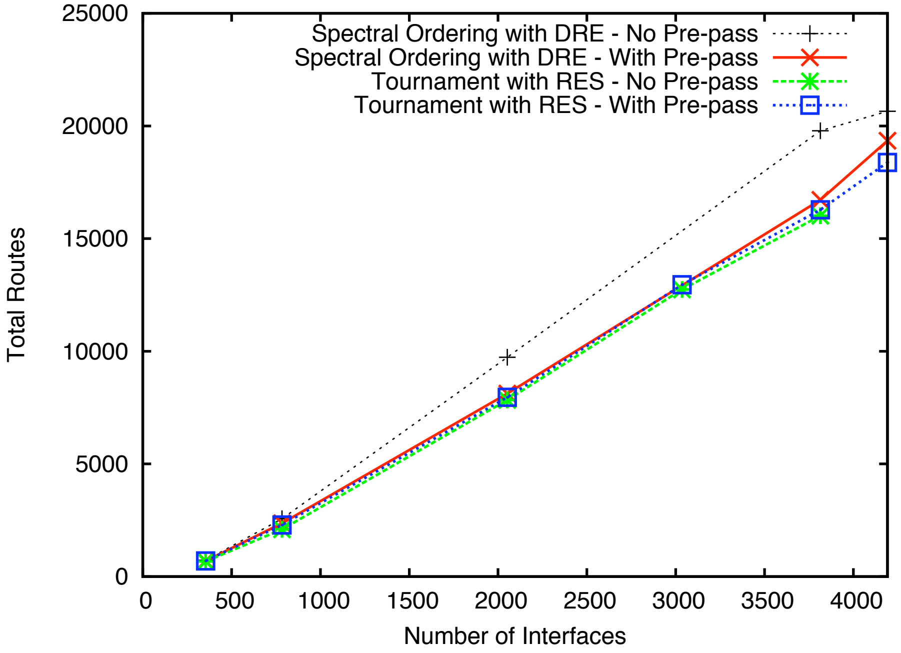


Figure 4.10. Routing table sizes with and without the pre-pass.

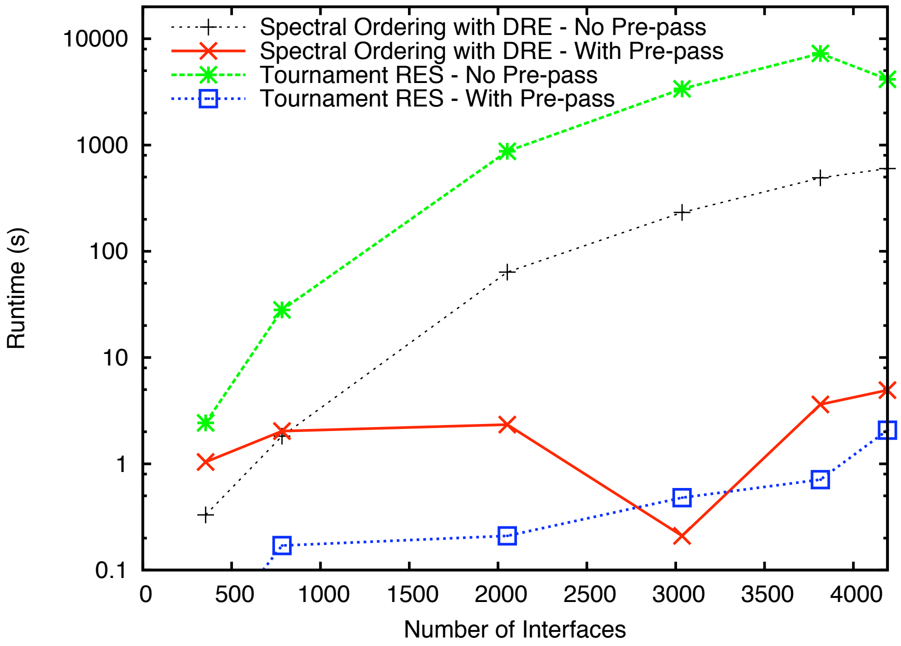


Figure 4.11. Runtimes in seconds with and without the pre-pass. The y-axis has a logarithmic scale.

Table 4.2. Histogram of pre-pass component sizes for three graphs.

Component size	GT-ITM	BRITE	Rocketfuel
1	4	15	29
2–10	49	22	23
11–20	1	4	1
21–30	1	4	
91-100	1		
321-330		1	
420-430			1
# of components	56	46	54
# of links	392	547	543

by three orders of magnitude, from over 2 hours to less than a second, while spectral ordering improves by over 2 orders of magnitude, to about 2 seconds. In order to show this change, the graph uses a logarithmic scale for the y-axis. These results demonstrate clear value for the pre-pass: it brings the runtimes of tournament RES and spectral ordering down to a level where they are competitive with recursive partitioning, and does so without sacrificing solution quality.

4.9.3.3 Component Characterization

To understand the source of the runtime improvement offered by the pre-pass, we obtained a quantitative and qualitative feel for the subgraph components themselves. For this part of the study, we chose a representative topology from each of the three topology sets (GT-ITM, BRITE, Rocketfuel); we chose topologies with similar link counts. Table 4.2 shows a histogram of the sizes of the components into which the pre-pass divides these input topologies. The smaller the largest component, the better the runtime of the quadratic algorithms will be. We can see from this table that the pre-pass has varying levels of effectiveness for the different topology types. For the GT-ITM topology, the largest component is roughly one-fourth of the topology size, while on the other topologies, the largest components are three-fifths and four-fifths of the size of the original topology. For all topologies, the majority of the components are smaller than 10 nodes. Manual examination reveals that for all three topologies, the largest components are biconnected; the smaller components are almost always trees.

4.9.4 Address Compaction

We compared the routing tables resulting from the bottom-up (tournament RES) and top-down (recursive partitioning) compaction algorithms for the EBONE topology. We started by limiting the bitspace to 10 bits. This is the smallest address space the topology will fit into: since there are 543 links, the $2^9 = 512$ subnets provided by 9 bits of address space are insufficient. We then relaxed the constraints until both methods converged to their minimum size routing tables. Figure 4.12 shows the results of this test.

In both cases, limiting the bitspace results in more routes. This is expected, because dense use of the address space packs together sets of nodes that aggregate poorly at the expense of sets that aggregate well. The top-down compaction handles small bitspaces more gracefully than bottom-up does. When nodes are more tightly

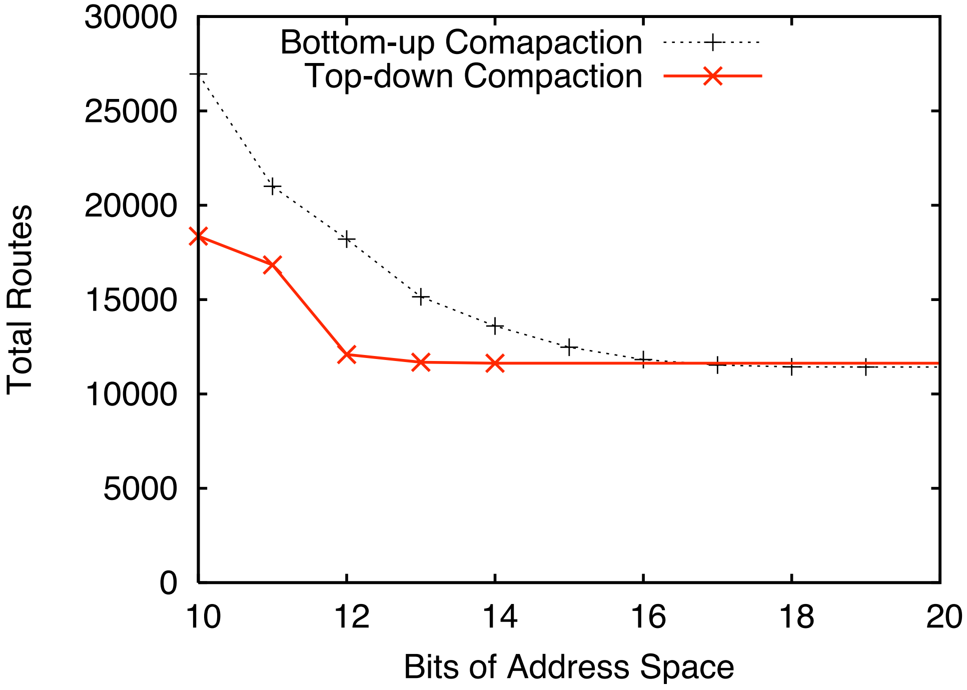


Figure 4.12. Total routes resulting from limiting the bitspace available to the compaction algorithms.

packed into the address space, it produces a routing table only 57% larger than when it has unlimited bit space. In comparison, the bottom-up method produces routing tables 135% larger. We conclude that top-down compaction is more suitable in the face of limited address space.

The fact that the two converge at different points illustrates a difference in the bitspace used by the tournament RES algorithm compared to recursive partitioning. When bitspace is not constrained, the former uses 19 bits of network address space, which is why its total number of routes remains constant after this point in the graph. Recursive partitioning converges on its minimum routing table size at only 14 bits. In general, the RES tournament makes sparser use of the address space than recursive partitioning.

4.9.5 Large Graphs

For our final experiments, we compare recursive partitioning with tournament RES on very large graphs. The pre-pass is used with tournament RES. Figure 4.13 shows the number of routes for these experiments, and Figure 4.14 shows the runtimes. The two algorithms produce a nearly equal number of routes, with the slight advantage going to tournament RES. For graphs under 12,500 interfaces, the runtimes are comparable and very low, but for the largest graphs, recursive partitioning shows much better scaling. Recursive partitioning is preferable for time-sensitive applications, but tournament RES provides slightly better results and still completes in under a minute on even the largest topologies. The largest graph in this test set has 5,000 router nodes.

4.9.6 Summary of Experimental Results

Figure 4.15 summarizes the results across all algorithms, showing the number of routes for one large, representative topology from each generator and for both Rocketfuel graphs. The number of routes for each topology is normalized to the number of routes produced by the random assignment. There are two clear patterns in these results: First, spectral ordering with DRE often does not perform better than the default ordering. Second, the recursive partitioning and RES tournament

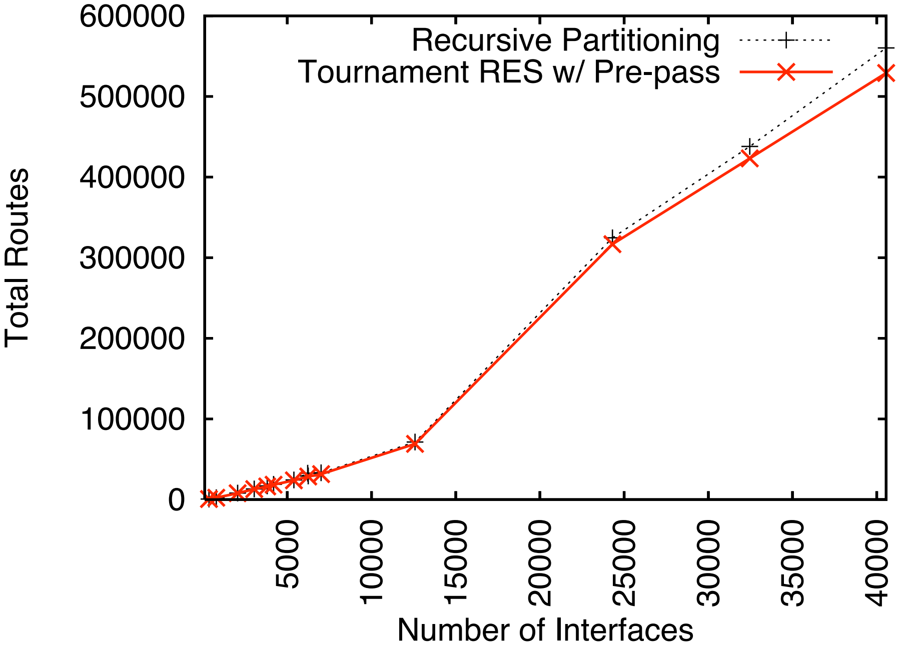


Figure 4.13. Total number of routes on large graphs.

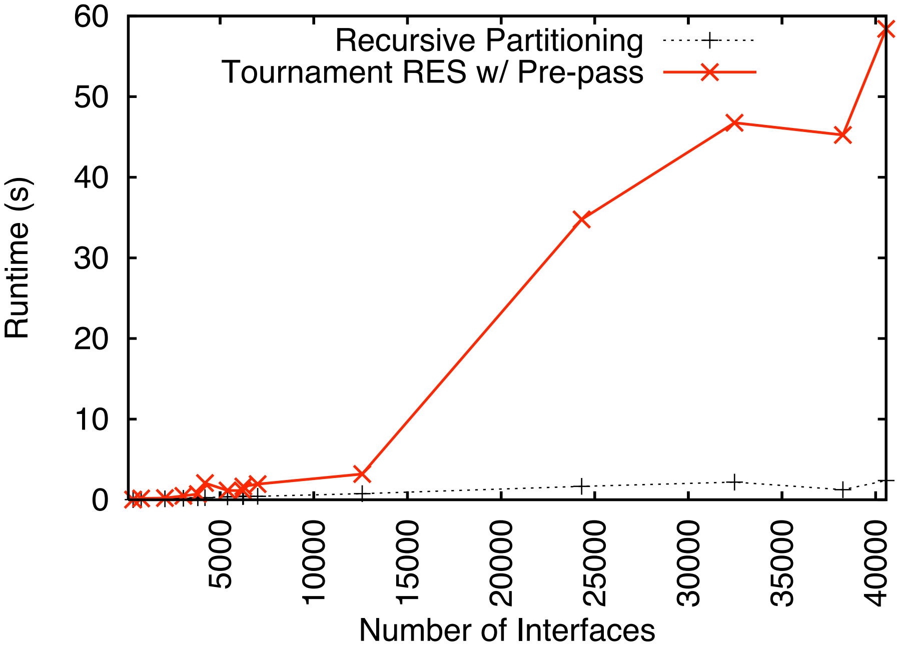


Figure 4.14. Runtimes on large graphs, in seconds.

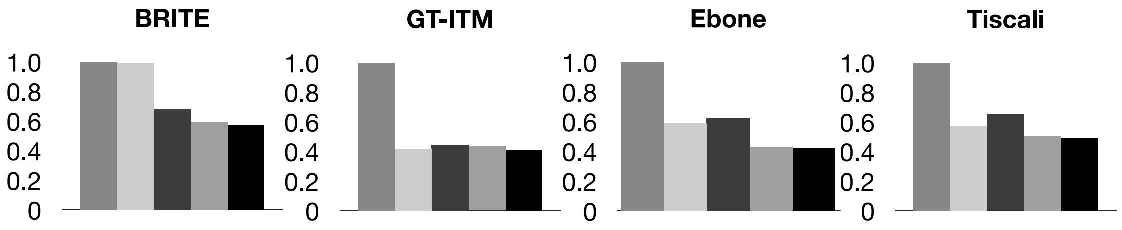


Figure 4.15. Summary comparison of global routing table sizes resulting from different algorithms. The results are normalized to the largest table size, which results from *random assignment*. From left to right, the bars illustrate the results of *random assignment*, *default ordering*, *spectral ordering with DRE*, *recursive partitioning* and *tournament RES*.

methods consistently yield the fewest routes, with RES usually having a slight advantage. We conclude that the latter two methods are superior.

It is important to keep in mind that all topologies are router-level and do not contain end hosts. Thus, they are most representative of ISP networks. In an enterprise or campus network, if we estimate 5 to 10 end hosts per router, an extremely conservative estimate, the largest topologies in our results represent networks of 25,000 to 50,000 nodes. Our best algorithms are clearly efficient enough to scale to very large networks.

4.10 Related Work

Methods for optimizing assignment of names to hosts in a network to minimize routing table size date back to the mid-1980s [130, 44]. In 1987, van Leeuwen and Tan formulated the notion of interval routing [130]; their work and subsequent work studied the problem of computing bounds on the *compactness* of graphs, the space complexity of a graph's shortest-path routing tables using interval routing [47, 39]. Their work is similar in direction to our problem; however, their work emphasizes worst-case routing table size for specific families of graphs and uses the idealized interval routing approach, not CIDR.

A more recent direction, pursued in the theoretical literature, is compact routing [9, 26, 125]. By relaxing the requirement of obtaining true shortest paths, compact routing enables much smaller routing tables at the expense of routes with *stretch*: the ratio between the routed source-destination path and the shortest source-destination path. Although these methods appear promising for realistic Internet topologies [76], true shortest-path routes are still the norm in simulated, emulated, and real-world environments.

A different direction related to our work is that of designing routing table compression schemes for network simulators and emulators, to avoid the $O(n^2)$ memory required for precomputing all-pairs shortest-path routes. For example, NIx-Vector-related designs [113, 115] replace static tables with on-demand dynamic route computation and caching. Each packet contains a compact representation of the remaining hops to its destination. This source routing means that routing at each node is simple and fast. Depending on the traffic pattern, the size of this global cache can be much smaller than the memory required to pre-calculate all of the routes.

Another practical alternative uses spanning trees [21]. Several spanning trees are calculated, covering most of the edges in the topology. These spanning trees cover most of the shortest-path routes in the topology and a small cache is kept for the remainder. The spanning trees and cache can be stored in a linear amount of memory. While this is a novel routing method, it assumes global information, since routing requires access to the global spanning trees and cache, a potential bottleneck for distributed simulations and emulations.

Finally, there has been work on optimizing Internet routing tables. First, a number of guidelines for CIDR addressing have been proposed to facilitate manual assignment of IP addresses [49, 55] to take advantage of CIDR routing. Second, the Optimal Routing Table Construction (ORTC) [30] technique produces a routing table that uses the minimum the number of CIDR table entries possible to represent a *given* set of desired routes and IP addresses. We employ ORTC as a post-processing step in our work.

4.11 Future Work

The tournament algorithms that we use are greedy, so there is almost certainly room to improve them. This improvement may take the form of pruning the solution space by not considering combinations that we know cannot or are not likely to be part of the optimal solution. It may also take the form of introducing some lookahead, so that the algorithms have the ability to trade lower scores in one round for higher scores in a later round.

One of the potential causes of the spectral orderings' relatively poor performance is that ordering based on a single eigenvector is in essence a single partitioning of the graph. Recursively ordering each partition based on the eigenvector may improve the spectral orderings greatly; the success of the recursive partitioning algorithm suggests that this may be the case. We can also try to adapt well-known algorithms which approximate Minimum Linear Arrangement [28, 109] to the ordering problem. Because transforming the ordering to a tree is the cheapest part (in time) of the ordering algorithms, there may be ways of improving the results while keeping the total runtime low.

Our work has focused on a single factor influencing IP naming, network topology. For our target domain, this is sufficient. However, there is also certainly value in considering the other factors, such as growth and policy. We do not yet have a metric to evaluate all aspects of the realism of our labellings; this remains a challenging open problem.

4.12 Discussion And Conclusion

We have investigated challenges associated with annotating an Internet-like topology with IP addresses in such a way as to minimize the sizes of the routing tables on hosts and routers. While there is considerable related work, especially for interval routing, none of it adequately handles the complexities of CIDR aggregation: longest prefix matching, the need to name network interfaces instead of hosts, and the nuances of addressing hosts on LANs. These factors must be considered in realistic simulation and emulation environments, and they impose a challenging set of constraints beyond those imposed by a simpler interval routing problem.

We attacked the address assignment problem from many angles. All of our methods produce routing tables that are far smaller than those that result from naïve, randomly chosen assignments, but two consistently show the best results. Recursive partitioning runs the fastest and produces small routing tables. The RES metric leads to the best solutions and makes a useful theoretical contribution by providing a clean way of quantifying "routing similarity." We believe that further refinements to the tournament tree-builder and tuning of the implementation can further improve performance.

In this chapter and the previous one, we have dealt with two of the key challenges in large scale, realistic emulation: creating realistic end-to-end conditions and generating realistic addresses for router-level topologies. Once an experimenter has obtained a realistic topology and an appropriate set of addresses have been assigned to it, the next step in running an emulated experiment is to select a set of physical hardware resources on which to run it. In the next chapter, we develop techniques for solving this mapping problem in a scalable fashion.

CHAPTER 5

SCALABLE NETWORK TESTBED RESOURCE MAPPING

5.1 Overview

Network experimentation environments of many types, especially emulation, require the ability to map virtual resources requested by an experimenter onto available physical resources. These resources include hosts, routers, switches, and the links that connect them. Experimenter requests, such as nodes with special hardware or software, must be satisfied, and bottleneck links and other scarce resources in the physical topology should be conserved when physical resources are shared. In the face of these constraints, this mapping becomes an NP-hard problem. Yet, in order to prevent mapping time from becoming a serious hindrance to experimentation, this process cannot consume an excessive amount of time.

In this chapter, we explore this problem, which we call the *network testbed mapping problem*. We describe the interesting challenges that characterize it and explore its application to emulation and other spaces, such as distributed simulation. We present the design, implementation, and evaluation of a solver for this problem. Our solver builds on simulated annealing to find very good solutions in a few seconds for Emulab's historical workload and scales gracefully on large well-connected synthetic topologies.

5.2 Introduction

To conduct a network experiment, the experimenter typically designs the environment in which it will be performed, then instantiates that environment by configuring some set of hardware to match it. The primitives that describe this environment are nodes and links. For nodes, such as hosts and routers, the experimenter may need specific hardware or software. On links, parameters such as bandwidth and latency are important. For anything larger than a trivial experiment, the process of selecting and configuring hardware to instantiate the desired topology can be tedious and error-prone.

Emulab [138] automates this instantiation by taking the experimenter's topology specification as input and configuring it in real hardware. As part of this automation, Emulab must select appropriate physical resources from those available. This mapping from an experimenter's virtual topology to a physical topology, however, is difficult; it must take into account both the experimenter's requirements and the physical layout of the testbed. It must give the experimenter appropriate nodes and links while conserving scarce physical resources, such as bandwidth on network bottlenecks, for other experimenters. Poor mapping can degrade performance of the emulator or introduce artifacts into an experiment.

We call this problem of selecting hardware on which to instantiate network experiments the *network testbed mapping problem*. It shares some characteristics with graph partitioning [71] and graph embedding [94], but has domain-specific goals and constraints that make it a different problem and interesting unto itself; these aspects are the major focus of this chapter. We first encountered this mapping problem in our emulation testbed, but it also appears in similar forms in other network experimentation environments.

In formulating and solving this problem, we aim to:

- Make the problem specification broad enough to be applicable to a wide range of network experimentation environments
- Develop abstractions that through their description of virtual and physical resources yield power and flexibility
- Produce a solver that is able to find near-optimal solutions in a modest amount of time

In pursuit of these goals, this chapter makes the following contributions: First, in Sections 5.3 and 5.4, it defines the network testbed mapping problem, and examines the challenges that make it interesting. Second, in Section 5.5, it describes our solver for this problem, `assign`, and presents an evaluation of its performance in Section 5.6. Third, throughout, it presents lessons from our solver's implementation and its use in Emulab [138], a production network testbed. Fourth, it identifies open issues for future work in Section 5.8.

5.3 Environment and Motivation

In order to motivate the network testbed mapping problem, we begin by describing some of the environments to which it is relevant and identify the characteristics of these environments that make good mapping necessary, but difficult.

5.3.1 Emulab

An experimenter submits a "virtual topology" to Emulab, describing the nodes, links, and LANs on which they would like to run their experiment; this topology may be manually constructed or it may come from measurements of real networks or topology generators, as detailed in the preceding two chapters. When it receives this specification, Emulab must select the hardware that will be used to create the emulation. Since Emulab is space-shared, hardware resources are constantly changing; only those resources that have not already been allocated are available for use. The infrastructure switches used to build emulators have practical limitations on the number of ports on each switch. To build a large scale emulator, then, it is necessary to use multiple switches. Emulab's switches are connected via interswitch links; these links are typically an order of magnitude faster than the node ports (for example, switches with 1 Gbps node ports will have multiple 10 Gbps links as interconnects.) Since multiple experimenters, or even many links from a single experiment, may share these interswitch links, they become a bottleneck, and overcommitting them could lead to artifacts in experimental results. Because Emulab aims to avoid introducing artifacts, conservative resource allocation is our guiding principle.

In this environment, the mapping algorithm has a number of simultaneous goals. First, it must economize interswitch bandwidth by minimizing the total bandwidth of virtual links mapped across physical interswitch links. This is similar to a graph partitioning problem. Second, since not all nodes are identical, the mapping algorithm must take into account the experimenter's requirements regarding the nodes they are assigned. Furthermore, the mapping must be done in such a way as to maximize the possibility for future mappings; this means not using scarce resources, such as special hardware, that have not been explicitly requested by the experimenter. Finally, this mapping must be done quickly. Experiment creation times in Emulab typically take on the order of minutes to tens of minutes. Our goal is to keep the time used by the mapping process much lower than experiment creation time, so that it does not hamper interactive use.

5.3.2 Simulation: Integrated and Distributed

In addition to emulation, Emulab also integrates simulation capabilities. It uses *nse* [37] to allow the popular *ns-2* [17] network simulator to generate and interact with live traffic. This also allows packets generated in the simulator to cross between machines to effect transparent distributed simulation. When simulated traffic interacts with real traffic, however, it must keep up with real time. For large simulations, this makes it necessary to distribute the simulation across many nodes. In order to do this effectively, the mapping must avoid overloading any node in the system and must minimize the links in the simulated topology that cross real physical links.

"Pure" distributed simulation also requires a similar mapping. In this case, rather than keeping up with real time, the goal is to speed up long-running simulations by distributing the computation across multiple machines [15]. However, communication between the machines can become a bottleneck, so a "good" mapping of simulated nodes onto physical hosts is important to overall performance. PDNS [114], a parallelized and distributed version of *ns-2*, is an example of such a distributed simulator. However, except for certain restricted tree topologies, PDNS requires manual partitioning onto physical machines.

5.3.3 ModelNet

Mapping issues also arise in ModelNet [127], a large-scale network emulator which aims at accurate emulation of the Internet core through emulating a large number of router queues on a small number of physical machines. Thus, virtual router queues must be mapped onto physical emulation nodes, known as "core" nodes. In order to minimize artifacts in the emulation, ModelNet's mapping phase, known as "assignment," must spread queues between the core nodes to avoid overloading any one node by giving it a disproportionate share of the traffic. At the same time, it must minimize the bandwidth passing between the core nodes, to avoid overloading their links.

Some aspects of ModelNet mapping are different from those outlined above for Emulab. A major difference is that ModelNet's mapping is not conservative. To reach its goal of supporting large emulated topologies, ModelNet takes advantage of the fact that not all links will be used to capacity, and allows them to be over-allocated. The goal of ModelNet mapping, then, is minimization of the potential for artifacts, rather than constraint satisfaction. Artifacts introduced by over-taxed CPUs or over-used links can be detected by ModelNet, and the emulation topology can be modified to reduce these artifacts in exchange for less accurate emulation of the core.

ModelNet, as currently designed, is not space-shared, meaning that all available resources are used for a single experiment. The goal is to load-balance among these resources, rather than use the least number. ModelNet also has a second phase that includes mapping challenges, called "binding," in which virtual edges nodes are assigned to physical ones. If the mapping portions of the ModelNet assignment and binding phases are done in a single pass, as may be necessary in an integrated ModelNet/Emulab environment, there are additional constraints on acceptable solutions introduced by IP routing semantics.

5.3.4 Similarities

Emulab was the first environment that presented us with the testbed mapping problem. Over several years we developed and improved our solver, targeted exclusively at the Emulab domain. More recently, as we have integrated other network experimentation mechanisms such as distributed nodes and simulated nodes to form the general Emulab platform, we immediately faced the mapping issue in each of them.

In the geographically distributed wide-area case, we chose to develop a separate solver [138], based on a genetic algorithm; this solver is outlined in Section 5.8. This was mostly due to the degree to which the wide-area problem differs from the emulation mapping problem.

However, the simulated and ModelNet environments are more similar in their mapping needs to Emulab. For example, minimizing interswitch bandwidth in Emulab is similar to minimizing communication between simulator nodes in distributed simulation and to minimizing communication between cores in ModelNet. All three environments share a need for mapping that completes quickly. In Emulab and ModelNet, lengthy mapping times discourage experimenters from trying experiments on a variety of configurations, nullifying one of the major strengths of these platforms. In distributed simulation, little benefit is gained from distribution of work if the mapping time is a significant fraction of the simulation runtime.

Therefore, we have extended our solver to handle simulation and ModelNet. The algorithms and program proved general enough that the extension was not difficult. As reported later in this chapter, our initial experience with simulation and ModelNet is promising, although not yet tuned to the degree we have achieved for Emulab. It appears that more environments could be accommodated. Indeed, as outlined in Section 5.8, with modest work, our general solver might handle the wide-area case as well.

5.4 Mapping Challenges

In the context of the environments outlined in the last section, the network testbed mapping problem becomes the following:

- As input, take a virtual topology and a description of physical resources

- Map the virtual nodes to physical nodes, ensuring that the hardware requirements of the virtual nodes are met
- Map virtual links to physical links, minimizing the use of bottlenecks in the physical topology
- In shared environments, maximize the chance of future mappings by avoiding the use of scarce resources when possible

Flexibility in specifying these resources is essential, both for describing available physical resources and requesting desired virtual topologies.

In this section, we describe the interesting mapping challenges in more detail. While doing so, we also discuss the abstractions we have designed into our solver, `assign`, to deal with them, and the ways in which they relate to Emulab and our other target environments. These challenges can be divided into two classes: link mapping and node mapping. We begin by describing link mapping, which is applicable across all three target environments. We then address interesting aspects of node mapping, which are of greater specific interest when mapping for Emulab.

5.4.1 Network Links

One of the key parts of the the network testbed mapping problem is the task of mapping nodes in such a way that a minimal amount of traffic passes through bottleneck links in the physical topology.

The problem can be seen to be NP-hard by reducing the traveling salesman problem to it. Given cities and distances forming an undirected graph $G(V, E)$ with positive integral edge costs, we can create a physical testbed topology T that corresponds to G by replacing each edge of cost $c > 1$ with c edges through chains of switches. We also create a virtual network topology that is a loop of $|V|$ nodes. A solution to the assignment problem will map the virtual loop into T, minimizing the number of switches. This would then be a solution to the traveling salesman problem. Andersen has also shown the testbed mapping problem to be NP-hard [5], by reducing the multiway separator problem.

Figure 5.1 shows a trivial example of the mapping problem. The virtual topology on the left is to be mapped onto the physical topology shown to its right. The bandwidths of all virtual and physical links in this example are 100 Mbps. To avoid over-burdening the link between the two switches, the sets of nodes {A,B,C} and {D,E,F} should be assigned to physical nodes that are connected to the same switch. This way, the only virtual link that crosses between switches is the one between C and E.

In the virtual topology, assign accepts two types of network connections: links and LANs. A link is simply a point-to-point connection between two virtual nodes, and includes information such as the bandwidth that it requires. A LAN is specified by creating a virtual "LAN node" in the topology, and connecting all members of the LAN to the LAN node using standard links. The LAN node can be thought of as a virtual switch.

assign recognizes four different types of physical links onto which these virtual links can be mapped. Direct links connect two nodes without an intermediary switch. Intraswitch links are those that can be satisfied on a single switch. Interswitch links must cross between switches. Intranode links connect nodes run on the same physical node; these links do not need to traverse any network hardware at all, and are used to represent links in distributed simulation or ModelNet that remain on one machine.

When mapping topologies to physical resources, the key limitation is that switch

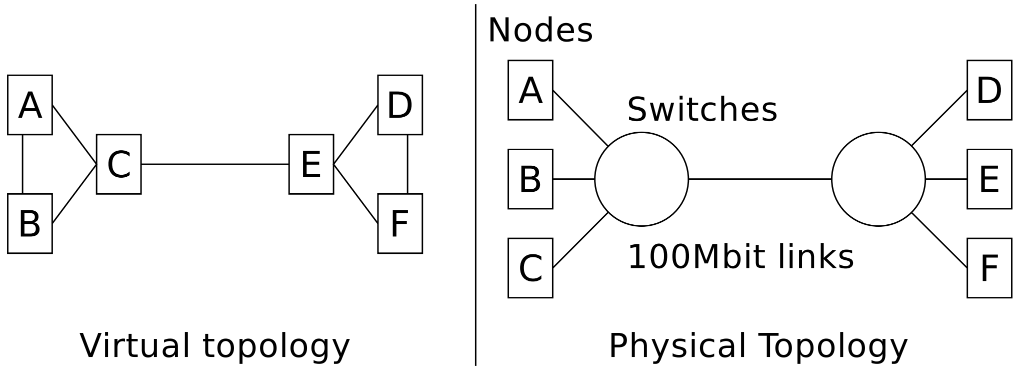


Figure 5.1. A trivial six-node mapping problem.

nodes are of finite degree; only a finite number of physical nodes can be attached to a given switch. Neighboring virtual nodes that are attached to the same switch can connect via intraswitch links that traverse only that switch's backplane.[1]

To allow topologies that cannot be fulfilled using the nodes of a single switch, Emulab employs several switches, connected together by high-bandwidth links. These interswitch links, however, do not have sufficient bandwidth to carry all traffic that could be put on them by an inefficient mapping. A goal, then, is to minimize the amount of traffic sent across interswitch links, and use intraswitch links instead, wherever possible. As Emulab is a space-shared facility it is important that interswitch traffic be minimized, rather than simply not oversubscribed. By minimizing such traffic, maximum capacity for future experiments is preserved.

This problem of minimizing interswitch connections is similar to sparse cuts in multicommodity flow graph problems—the goal is to separate the graph of the virtual topology into disjoint sets by cutting the minimum number of edges in the graph.

5.4.2 Node Types

A facility like Emulab will generally have distinct sets of nodes with identical hardware. Emulab, for example, has several distinct classes of PCs representing different phases of hardware expansion. Facilities like this will tend to grow incrementally as demand increases and to achieve the greatest possible number of nodes, old nodes will continue to be used alongside newly-added hardware. In addition, nodes with specialized hardware may be added. As network testbeds become larger, their hardware will therefore tend to become more heterogeneous. With varying node hardware, it becomes important for experimenters to be able to request specific types, for example, if they have run experiments on a specific type in the past, and need consistent hardware to ensure consistent results. Experimenters who do not have such requirements should not be burdened with this specification.

[1] This backplane, by design in Emulab, has sufficient bandwidth to handle all nodes connected to it, and can thus be considered to have infinite capacity.

In order to meet this challenge, we have designed a simple type system for `assign`. Each node in the virtual topology is given a type and each node in the physical topology is given a list of types that it is able to satisfy. The fact that a physical node can satisfy more than one type allows for differing levels of detail in specification, as we will see below. In addition, each type on a physical node is associated with a number indicating how many nodes of that type it can accommodate. This enables multiple virtual nodes to share a physical node, as required for distributed simulation and ModelNet. One restriction is observed: all virtual nodes mapped to the same physical node must be of the same type.

To illustrate the type system, consider the fragments of a virtual topology in Figure 5.2 and a physical topology in Figure 5.3. These samples are typical of nodes that are found in Emulab. In this example, virtual node `node1` can be mapped to any physical node, as all physical nodes are allowed to satisfy a single `pc` node. `node2`, on the other hand, specifically requests a `pc850`, which can be satisfied only by `pc1` or `pc2`. This allows an experimenter to specify a general class of physical node, such as `pc`, or request a specific type of PC, such as `pc850` or `pc600`. Virtual nodes `delay1` and `delay2` can be placed on the same physical node, since all nodes in the physical topology can accommodate two virtual nodes of type `delay`. [2]

```
node node1   pc
node node2   pc850
node delay1  delay
node delay2  delay
```

Figure 5.2. Sample nodes in a virtual topology.

```
node pc1 pc:1 pc850:1 delay:2
node pc2 pc:1 pc850:1 delay:2
node pc3 pc:1 pc600:1 delay:2
node pc4 pc:1 pc600:1 delay:2
```

Figure 5.3. Sample nodes in a physical topology.

[2] In Emulab, the traffic-shaping nodes, called delay nodes, that are used to introduce latency, packet loss, etc. into a link, can be multiplexed onto a single physical node; this is possible since delaying a link requires two network interfaces, and four are available on Emulab nodes.

Most types are opaque to `assign`—there is only one type that is treated specially: `switch`, which is necessary to support interswitch links; it is assumed that only nodes of type “switch” are able to forward packets. Thus, `assign` is not tied to the hardware types available on Emulab; new types can be added simply by including them in the physical topology.

5.4.3 Virtual Equivalence Classes

We have found that a common pattern is for experimenters to care not about which node type they are allocated, but that all nodes be of the same type. To address this, `assign` allows the creation of equivalence classes in the virtual topology. Virtual equivalence classes (*vclasses*) increase the flexibility of the type system, by allowing the user to specify that a set of nodes should be all of the same type, without forcing the user to pick a specific type ahead of time.

vclasses are declarations of virtual equivalence classes in the virtual topology. This includes a list of types that can be used to fulfill the *vclass*, which could be automatically determined by Emulab. Virtual nodes are then declared to belong to the *vclass*, rather than a specific physical type. `assign` will then attempt to ensure that all nodes in the *vclass* are assigned to physical nodes of the same type. Multiple *vclasses* can be used in a virtual topology.

vclasses can be of two types, hard or soft. Hard *vclasses* must be satisfied, or the mapping will fail. Soft *vclasses* allow `assign` to break the *vclass*—that is, use nodes of differing types—if necessary, but homogeneity is still preserved if possible. For soft *vclasses*, the weight used to determine how much a solution is penalized for violating the *vclass* is included in the virtual topology specification.

5.4.4 Features and Desires

On a finer granularity than types, `assign` also supports “features” and “desires.” Features are associated with physical nodes, and indicate special qualities of a node, such as special hardware. Desires are associated with virtual nodes and are requests for features. Unfulfilled desires—that is, desires of a virtual node that are not satisfied by the corresponding features on the mapped physical node—are penalized in the scoring function. Likewise, wasted features—features that exist on a physical node, but were not requested by the virtual node mapped to it—are also penalized.

The chief use of features and desires is to put a premium on scarce hardware. If some nodes have, for example, extra RAM, extra drive space, or higher-speed links, the penalty against using these features if they are not requested will tend to leave them free for use by experimenters who require them.

Other uses are possible as well. For example, features and desires can be used to prefer nodes that already have a certain set of software loaded. In Emulab, for example, custom operating systems can be loaded, but features can be used to prefer nodes that already have the correct OS loaded, saving the substantial time it would take to load the OS. Or, if some subset of physical resources have been marked as only usable by a certain experimenter (for example, by some sort of advance reservation system), those nodes can be preferred.

Specifying features and desires is easy. Since they are represented as arbitrary strings in the input files, like types, they are not restricted to the Emulab environment. Penalties for wasted features can be intuitively derived. In general, it is sufficient to choose a penalty based on a feature's relative importance to other resources—for example, one may choose to penalize waste of a gigabit interface more than using an extra link (thus preferring to use another link rather than waste the feature), but less than the cost of using an extra node (thus preferring to waste a gigabit interface before choosing to use another node). Weights can be made infinite, to indicate that a solution failing to satisfy a desire or wasting a feature, should not be considered a feasible mapping. This is analogous to a hard *vclass*.

5.4.5 Partial Solutions

Also useful is the ability to take partial solutions and complete them. These partial solutions can come from the user or from a previous run of the mapping process. In the virtual topology, `assign` can be given a fixed mapping of a virtual node onto a physical node, which it is not allowed to change. The two ways in which this feature is used on Emulab are for replacement of nodes in existing topologies and incremental topology changes.

When using a large amount of commodity hardware, failures are not uncommon. When such a failure occurs during a running experiment, the instantiated topology can be repaired by replacing the failed node or nodes. The topology is run through `assign` again, with nodes that do not need to be replaced fixed to their existing mapping. This will allow the mapping algorithm to select good replacements for the failed nodes.

To add or remove nodes from a topology that has already been mapped, a similar strategy is employed. In this case, parts of the topology that have not changed are fixed onto their currently mapped nodes, and new nodes are chosen by the algorithm that fit as well as possible into the existing mapping. In Emulab, this allows for the modification of running experiments, simply by supplying a new virtual topology.

5.5 Design, Implementation, and Lessons

`assign`, our implementation of a solver for the testbed mapping problem, is written in 10,000 lines of C++ code. It uses the Boost Graph Library [14] for efficient graph data structures and for generic graph algorithms such as Dijkstra's shortest path algorithm.

Use of a randomized heuristic algorithm helps fulfill our design goal of creating a mapper that is able to find near-optimal solutions in a modest amount of time. For `assign`, we have chosen simulated annealing.

Simulated annealing [73] is a randomized heuristic search technique originally developed for use in VLSI design, and commonly used for combinatorial optimization problems. It requires a *cost function*, for determining how "good" a particular configuration is, and a *generation function*, which takes a configuration and perturbs it to create a new configuration. If this new configuration is better than the old one, as judged by the cost function, it is accepted. If worse, it is accepted with some probability, controlled by a "temperature." This allows the search to get out of local minima in the search space, which would not be possible if only "better" solutions were accepted. The algorithm begins by setting the temperature to a high value, so that nearly all configurations are accepted. Over a large number of applications of the generation function (typically, at least in the hundreds of thousands), the temperature is slowly lowered, controlled by a *cooling schedule*, until a final configuration, the solution, is converged upon. Clearly, there is no guarantee that this is the optimal solution, but the goal of the algorithm is to arrive at a solution near the optimal one.

In this section, we discuss how the functions key to simulated annealing are designed and implemented in `assign`. We also introduce two concepts that are key to the design of `assign`: *violations*, which are used to flag whether or not a configuration is acceptable and *pclasses*, which are equivalence classes used to dramatically reduce the search space.

5.5.1 Initial Configuration

Typically, simulated annealing is started with a randomly-generated configuration [73]. However, `assign` uses a different strategy. `assign`'s concept of violations, explained later, allows it to begin with an empty configuration—one in which no virtual nodes are assigned to physical nodes. In the generation function, mapping of unassigned nodes gets priority over other transitions. The algorithm must, therefore, spend some time arriving at a valid configuration, but that configuration is likely to be much better than a purely random one, since constraints such as node types are taken into account.

5.5.2 Cost Function

`assign`'s cost function scores a configuration and returns a number that indicates how "good," in terms of the goals laid out in Section 5.3, the configuration is. To compute this score, the mappings for all nodes and links must be considered. In `assign`, a lower score is preferable.

Computing the cost for an entire configuration is quite expensive, requiring $O(n+l)$ time, where n is the number of nodes that have been mapped, and l is the number of links between them. If, instead, the cost is computed incrementally, as mappings are added and removed, the time to score a new solution is $O(l_n)$, where l_n is the number of links connected to the node being re-assigned; this is because, in addition to scoring the mapping of the node itself, all links that it has to other nodes must be scored as well. Clearly, incremental scoring provides better scaling to large topologies, so this approach is used in `assign`. This fits well with simulated annealing, which calls for a generation function that makes small perturbations, naturally leading to incremental scoring.

`assign`'s scoring function is split into three parts: `init_score` initializes the cost for an empty configuration, and computes the violations that result from the fact that `assign` begins with no nodes mapped. `add_node` takes a configuration, a physical node p, and a virtual node v. It computes the changes in cost and violations that result from mapping v to p. `remove_node` performs the inverse function, calculating the cost and violations changes that result in unmapping a virtual node.

While incremental scoring greatly reduces the time taken to score large topologies, it does have a cost in the complexity of the scoring function. In particular, care must be taken to ensure that `add_node` and `remove_node` are completely symmetric; `remove_node` must correctly remove the cost added by the corresponding `add_node`. This is made more difficult by the fact that other mappings may have been added and removed in the time between when a virtual node was mapped and when the mapping is removed. In general, though, we feel that the added complexity is an acceptable tradeoff for better evaluation times on large virtual topologies.

Link resolution, the mapping of a virtual link to a physical link, is also done in `add_node`—any virtual links associated with v for which the other end of the link has already been mapped are resolved at this point. This means that links are not first-class objects, subject to annealing. This limits `assign`'s effectiveness in physical topologies that have multiple paths between nodes, such as nodes that have both direct links to each other and intraswitch links. Our experience, however, is that such topologies are not common in practice in emulation testbeds. So, while `assign` supports these topologies, it does not include the additional code and time complexity to treat links as first-class entities. Instead, if multiple link paths are present between a set of nodes, assign greedily chooses lower-cost links before moving on to higher-cost ones.

To resolve a link, assign finds all possible links between the nodes (direct, intraswitch, and interswitch) and chooses one. Direct links are used first, if they exist, followed by intraswitch and interswitch links. To find interswitch paths, Dijkstra's shortest path algorithm is run for all switches when assign starts. The shortest paths between all switches to which the nodes are connected are then considered possible candidates. If no resolution for a link can be found, a violation is flagged.

A configuration is penalized based on the number of nodes and links it uses. The default penalties, listed in Table 5.1, can be overridden by passing them to assign on the command line. Intranode links, entirely contained within a single node and used in mapping simulations, are not penalized at all. Direct node-to-node links, which do not go through a switch, have only a small penalty. Slightly higher is the penalty for intraswitch links. Interswitch links have a cost an order of magnitude higher, since they consume the main resource we wish to conserve. A configuration is also penalized on the number of equivalence classes (explained in further detail in Section 5.5.5) that the chosen physical nodes belong to. This encourages solutions that use homogeneous hardware, which is a quality desired by many experimenters. Penalties for unsatisfied desires and unused features are given in the input, and can

Table 5.1. Scores used in assign.

Physical Resource	Cost
Intranode Link	0.00
Direct Link	0.01
Intraswitch Link	0.02
Interswitch Link	0.20
Physical Node	0.20
Switch	0.50
pclass	0.50

be chosen based on their relative importance to the resources listed above.

LANs are more computationally costly to score than links, since links involve only two nodes, and their scoring time is thus constant, but LANs can contain many nodes, and their scoring time is linear in the number of nodes that are in the LAN. In `assign`, we represent a LAN by connecting its members to a "LAN node," shown in Figure 5.4, which is used solely for the purpose of assessing scoring penalties. LAN nodes only exist in the virtual topology—they do not correspond to a real resource. As needed, LAN nodes are dynamically bound to switches in the physical topology. Thus, any LAN member that is on another switch will be assessed an interswitch link penalty.

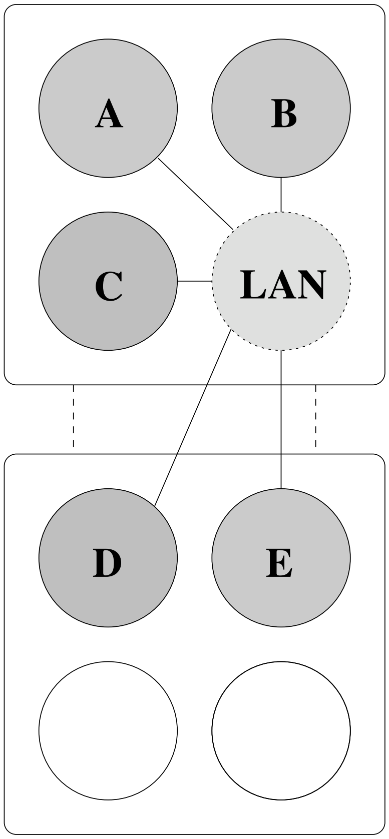


Figure 5.4. Scoring for LANs is done with a "LAN node," which LAN members have links to. This LAN uses 3 intraswitch links and 2 interswitch links.

5.5.3 Violations

One issue that must be decided when implementing simulated annealing is whether or not to allow the algorithm to consider infeasible solutions; that is, configurations that violate fundamental constraints. In the context of our problem, the primary constraint considered is over-use of bottleneck bandwidth between switches. The benefits to allowing infeasible solutions, as put forward in the simulated annealing literature [1], are twofold. First, this makes the generation function simpler, as it does not need to take feasibility into account. Second, it allows the search to more easily escape local minima, with the possibility that a lower minima will be found elsewhere. It does so by smoothing the cost function. A generation function that excludes infeasible solutions must either simply reject these configurations, or "warp" to a new area of the space, conceptually on the other side of the portion of the space that is infeasible. If infeasible solutions are simply rejected, the connectivity of the solution is reduced, possibly even leading to portions of the space that are isolated; these could leave the search trapped in a poor local minima. Figure 5.5 shows an example of this situation. If "warping" is used, the score from a configuration to its potential successor may be very high, resulting in a low probability of its acceptance, even at high temperatures.

A common approach to the search of infeasible configurations [1] is to give them a high cost penalty, thus making them possible to traverse at high temperatures, but unlikely to be reached at lower ones. This approach has some drawbacks, however. It is difficult to choose a penalty high enough such that an infeasible solution will never be considered to be better than a feasible one. If this can occur, the algorithm may abandon a feasible, but poor, solution and instead return an infeasible one. Thus, in `assign`, we have chosen to keep track of the violation of constraints separately from the cost function; this is implemented with "violations." Each possible configuration has a number of violations associated with it. If a configuration has one or more violations, then it is considered to be infeasible. If no solutions are found with zero violations, the algorithm has failed to find a mapping; frequently, this is because no mapping is possible.

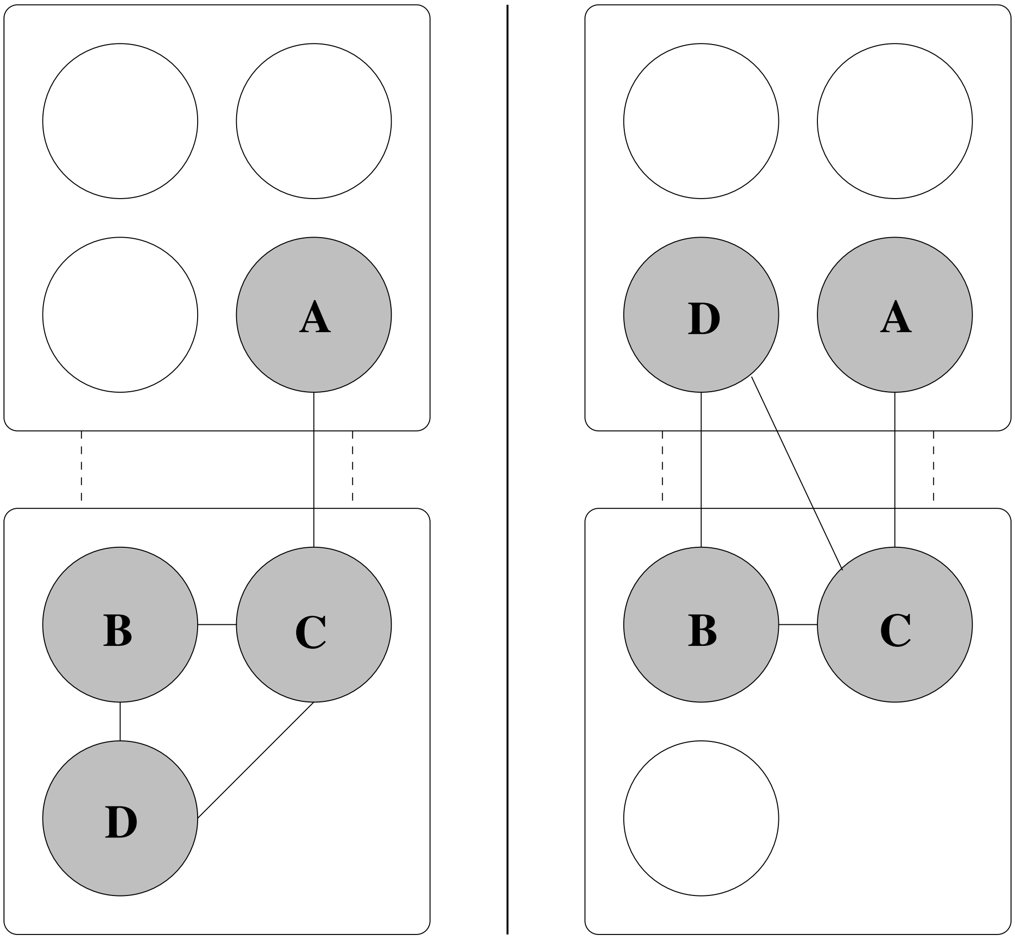


Figure 5.5. A situation in which allowing solutions with violations helps reach the optimal solution. If the bandwidth between switches is such that only one virtual link can cross between them, the mapping shown on the right is in violation of this constraint. However, it is a necessary intermediate step between the mapping on the left and the optimal mapping, which places all nodes on the upper switch.

When considering whether or not to accept a state transition, violations are considered before the configurations' costs. If the new configuration results in fewer violations than the old, it is accepted. If the number of violations in the new configuration is equal to or greater than the old violations, then the costs are compared normally. This allows the algorithm to leave feasible space for a time, guiding it back to feasible space fairly quickly so excessive time is not spent on infeasible solutions.

One important side effect of violations is that they provide the user of the program with feedback about why a mapping has failed. Twelve different types of violations are tracked, ranging from overuse of interswitch bandwidth to user desires that could not be met. These are summed together to produce the overall violations score. When `assign` fails to find a feasible solution, it prints out the individual violations for the best solution found. This helps the user to find the "most constraining constraint"; the one whose modification is most likely to allow the mapping to succeed. This gives the user the opportunity to modify and resubmit their virtual topology. It also gives the administrators of the testbed feedback about what factors are preventing experiments from mapping, so that they can work on remedying them. It may reveal, for example, that insufficient interswitch bandwidth is a limiting factor for mapping, or that experimenters need nodes that have more links or faster links.

5.5.4 Generation Function

`assign`'s generation function has the task of taking a potential configuration and generating a different, but similar, configuration for consideration. `assign` does this by taking a single virtual node and mapping it to a new physical node. First, `assign` maintains a list of virtual nodes that are currently unassigned to physical nodes. If this list is nonempty, it picks a member and randomly chooses a mapping for it. If there are no unassigned nodes, it picks a virtual node, removes its current mapping, and attempts to re-map it onto a different physical node. If there are no free nodes to which the virtual node can be mapped, `assign` frees one up by unmapping another virtual node. This is done to avoid getting stuck in certain exact-fit or resource-scarce conditions.

We have found that it is very important that `assign`'s generation function avoid *certain* classes of invalid solutions. Though certain violations are useful to explore, as covered in Section 5.5.3, others are not. In general, violations that cannot be removed by mapping changes to other virtual or physical nodes should be avoided. As an example, a virtual node with five links assigned to a physical node with only four links will always result in a violation, no matter what the rest of the virtual nodes' mappings are. This is in contrast to an overused interswitch link, where changes to other parts of the configuration may lower traffic on the link and remove the violation.

Exploring these invalid solutions can result in poor performance in some cases, particularly when there are scarce resources in the physical topology and only a few nodes in a large virtual topology that require them. `assign` can spend a long time exploring fruitless portions of the solution space in these circumstances. To help avoid certain invalid solutions, when it begins, `assign` precomputes a list of physical nodes that are acceptable assignments for each virtual node. An acceptable assignment is one that is capable of fulfilling the type of the virtual node, has at least enough physical links to satisfy the virtual node's links, and will not incur violations due to features and desires. A virtual node is assigned only to physical nodes from its list.

5.5.5 Physical Equivalence Classes

5.5.5.1 Reducing the Solution Space

One of the features of `assign` that has most improved its runtime and quality of solutions is the introduction of physical equivalence classes. This improvement comes from the observation that, in a typical network, many hosts are indistinguishable in terms of hardware and network links. For the purposes of the generation function, these nodes can be considered equivalent; mapping a virtual node to any of them will result in the same score. It does not matter which of these indistinguishable nodes is selected. The solution space to explore can be reduced by exploiting this equivalence.

The neighborhood structure, or branching factor, of a solution space in `assign` has a size on the order of $O(v \cdot p)$, where p is the number of nodes in the physical topology, and v is the set of nodes in the virtual topology. This number is an upper bound, because, as `assign` progresses, some physical nodes will be already assigned, reducing the number of choices to something less than p; once all virtual nodes have been assigned, it will be $O(v \cdot (p - v))$. Clearly, if we can safely reduce the size of v or p, `assign` will be able to explore a reasonable subset of the solution space in less time, resulting in lower runtimes.

Our first strategy is to reduce *p*. The Emulab facility consists of a large number of identical nodes connected to a small number of switches, and other emulation facilities are likely to have similar configurations. For example, in Emulab, depending on available resources, there are 168 PCs that can be in the physical topology input to `assign`. These reduce to only four physical equivalence classes, resulting in a branching factor two orders of magnitude smaller. Our work on reducing *v* is presented later in this chapter, in Section 5.5.8.1.

5.5.5.2 pclasses

In order to effect this reduction in the physical topology, `assign` defines an equivalence relation. Any equivalence relation on a set partitions that set into disjoint subsets in which all members of a subset are equivalent (meaning that they satisfy the relation); these subsets are called equivalence classes. When `assign` begins it calculates this partition. Each equivalence class is called a *pclass*.

The equivalence relation `assign` uses defines two nodes to be equivalent if they have identical types and features and there exists a bijection from the links of one node to the links of the other which preserves destination and bandwidth. It is easily verified that this relation is an equivalence relation.

When the generation function is invoked, rather than choosing a physical node directly, it instead selects a *pclass*, and a node is chosen from that *pclass*. This technique reduces the size of the search space dramatically, without adversely affecting quality of solutions found by `assign`. It reduces the search space by "collapsing" areas of the solution space that are equivalent. To gain a more intuitive feel for how *pclasses* reduce the search space, consider two physical nodes with identical hardware and an identical set of links to the same switch. When looking for a physical node to which to map a virtual node, it makes no difference which of these nodes `assign` chooses, since either choice will lead to the same score. By combining these two nodes into a *pclass* and selecting from *pclasses* rather than nodes, we have combined the two separate states that would result from choosing either of the physical nodes into a single state. Thus, the branching factor of the search space is reduced, but the set of unique states that `assign` visits is not.

pclasses have an interesting effect on the way that the solution space is explored; they tend to increase the probability with which physical nodes with scarce resources are selected by the generation function. Selecting from among all *pclasses* with equal probability results in a higher probability of selecting a node in a small *pclass* than selecting one in a large *pclass*. When selecting among nodes rather than among *pclasses*, it is more likely that a node from a large *pclasses* will be selected, simply because there are more of them. Thus, we have experimented with weighting the probability that each *pclass* will be selected by the number of nodes it contains to make the probability distribution similar to the case without *pclasses*. However, we have so far found that this is unnecessary, as it does not improve the solutions found for our test cases.

There are some circumstances in which *pclasses* are not appropriate. When mapping multiple virtual nodes onto each physical node, as is frequently the case with distributed simulations or ModelNet, the base assumption, equivalency of certain physical nodes, is violated. As a physical node becomes partially filled, it becomes no longer equivalent to other nodes. Mapping a new virtual node to different physical nodes in the same *pclass* can now result in different scores, as this affects whether some of their virtual links can be satisfied as intranode links or not. As a result, when mapping highly multiplexed topologies, we disable *pclasses*. In these cases, reducing the size of the virtual topology, as detailed in Section 5.5.8.1, is of critical importance.

5.5.6 Cooling Schedule

By default, `assign` uses the polynomial-time cooling schedule described by Aarts and Korst [1]. It uses a melting phase to determine the starting temperature, so that initially, nearly all configurations are accepted. It generates a number of new configurations equal to the branching factor (as defined in Section 5.5.5) before lowering the temperature. The temperature is decremented using a function that helps ensure that the stationary distribution of the cost function between successive temperature steps is similar. Finally, when the derivative of the average-cost function reaches a suitably low value, the algorithm is terminated. The parameters to this cooling schedule were chosen through empirical observation. However, we are exploring the idea of using another randomized heuristic algorithm, such as a genetic algorithm, to tune these constants for our typical workload, maximizing solution quality while keeping the runtime at acceptable levels.

The result of this cooling schedule is that `assign`'s runtime scales in relation to the number of virtual nodes and the number of *pclasses*. The temperature decrement function and termination condition, depends on how quickly `assign` is able to converge to a good solution, roughly reflecting the difficulty of mapping the supplied virtual and physical topologies.

`assign` also has two time-limited cooling schedules. The first simply takes a time limit, and, using the default cooling schedule, terminates annealing when the time limit is reached. The second mode attempts to run in a target time, even extending the runtime if necessary. It uses a much simpler cooling schedule in which the initial temperature is determined by melting, the final temperature is fixed, and the temperature is decreased multiplicatively, with a constant chosen such that annealing should finish at approximately the chosen time. Both of these cooling schemes are useful in limiting the runtime for large topologies, which otherwise could take many minutes or even hours to run. The latter is also useful for estimating the best solution to a given problem, as `assign` can be made to run much longer than normal in the hope that it will have a better chance of finding a solution near the optimal one.

5.5.7 Scaling to Large Multiplexed Experiments

One of the most important features that has been added to Emulab and other testbeds in recent years is the ability to conduct experiments using *virtualization technologies* [53] such as virtual machines [10, 133] and container-based operating systems [69, 102, 119]. This allows the *multiplexing* of multiple nodes from the experimenter's requested topology on to each physical node, allowing for experiments that are larger than the available physical topology. This introduces some new challenges to the testbed mapping problem.

For multiplexed experiments, a good mapping is one that “packs” virtual hosts, routers, and links on to a minimum number of physical nodes without overloading the physical nodes. This means placing, when possible, nodes that are adjacent in the virtual topology on the same physical node, so that the links between them need not use physical interfaces or switch capacity. This is particularly difficult because the virtual nodes may not have uniform resource needs, and physical nodes may not have identical capacities. In this process, all of `assign`’s other constraints on node types, link capacities, etc. must be met.

It was necessary to improve `assign` in two ways to meet the challenges of multiplexed virtual experiments. First, we needed flexibility in specifying how virtual nodes are to be “packed” onto physical nodes. To get efficient use of resources, we found it necessary to add fine-grained resource descriptions and to relax `assign`’s conservative resource allocation policies. Second, because virtualization allows for topologies that are an order of magnitude larger than one-to-one emulation, we ran into scaling limitations with `assign`. To combat these scaling problems, we made enhancements to `assign` that exploit the natural structure of the virtual topologies it is given to map.

5.5.7.1 Flexible Resource Specification

`assign` must use some criteria to determine how densely it can pack virtual nodes onto physical nodes. `assign`’s simplest packing mechanism is coarse-grained, in which each physical node has a specified number of “slots” and each virtual node is assumed to occupy a single slot. Thus, it can be specified that `assign` may pack up to, for example, 20 virtual nodes on each physical node. It became clear that this would not be sufficiently fine-grained for many applications because different virtual nodes will have different roles in the experiment and thus consume different amounts of resources.

To address this, we have added more packing schemes to `assign`. In the first, virtual nodes can fill more than one slot; experimenters can use this when they have knowledge that, for example, servers in their topology will require more resources than clients by an integer ratio: 2:1, 10:1, etc.

The second packing scheme models multiple independent resources such as CPU cycles and memory, and can be used when the experimenter has estimated or measured values for the resource needs of the virtual nodes. Each virtual node is tagged with the amount of each resource that it is estimated to consume and `assign` ensures that the sum of resource needs for all virtual nodes assigned to a particular physical node does not exceed the capacity of the physical node. This scheme builds on the system of "features and desires" described in Section 5.4.4, which we have enhanced to also express capacities. Desires may be associated with floating point-values that indicate the needs of each virtual node. Likewise, features may also be associated with a floating-point value, and `assign` assures that the mapping it selects does not over-use these capacities. Like regular features and desires, the names and capacities have no inherent meaning to `assign`, so this scheme can easily be extended to support new types of additive metrics. In current practice, we use this scheme for relatively low-level resources (CPU and memory), but it could also be used for higher-level metrics such as sustainable event rate for discrete event simulators such as *nse*.

The resource-modeling scheme is particularly useful for feedback-based auto-adaptation [53]. The values used for CPU and memory consumption of a virtual node can simply be obtained by taking measurements of an earlier run of the application. The maximum or steady-state usage can then be used as input to the mapping process. The coarse-grained and resource-based packing criteria can be used in any combination.

In addition to packing nodes, virtual links must be packed onto physical links. Though the two types of packing are conceptually similar, a different set of issues applies to link packing. Some of these issues exist for one-to-one emulation, but there are also some new challenges that come with virtual emulation.

When mapping multiplexed experiments, links between two virtual nodes that are mapped to the same physical node become "intranode" links that are carried over the node's "loopback" interface. It is advantageous to use intranode links, as they do not consume the limited physical interfaces of the physical node. Although the bandwidth on a loopback interface is high, packet processing and copying place practical limits on it, and for some experiments that use little CPU time but large amounts of bandwidth, loopback bandwidth can become the limiting factor. assign is able to to take this finite resource into account by associating a maximum value for loopback bandwidth with each node in the physical topology.

One of the guiding principles of assign has historically been conservative resource allocation; when assigning links, it ensures that the full bandwidth specified for the link will always be available. While this makes sense for artifact-free emulation, it is at odds with some of the goals of multiplexed emulation, which aims to provide best-effort, large-scale emulation. For example, an experimenter may have a topology containing a cluster of nodes connected in a LAN. Though the native speed of this LAN is 1 Gbps, the nodes in this LAN may never transmit data at the full line rate. Thus, if assign were to allocate the full 1 Gbps for the LAN, much of that bandwidth would be wasted. To make more efficient resource utilization possible, we have added a mechanism so that estimated or measured bandwidths can be passed to assign. As with node resources, this bandwidth can be estimated or measured from previous runs of similar experiments.

5.5.8 Improving Scaling on Multiplexed Topologies

By design, mutliplexed experimentation enables virtual topologies that are much larger than the physical topology. This presents new scaling challenges for assign, and we have developed several techniques to improve assign's scaling properties for large multiplexed topologies.

5.5.8.1 Searching the Solution Space

Our first techniques are aimed at improving the way in which assign searches through the solution space. As discussed earlier, assign's *pclass* strategy breaks down with the high degree of multiplexing that comes with virtual-node experiments. In order to continue using *pclasses* instead of disabling them altogether, we have made these equivalence classes adapt *dynamically* at runtime. assign starts by building *pclasses* normally. However, when a physical node is partially filled, the fact that it is no longer equivalent to other physical nodes is reflected by splitting it off into its own *pclass*; conversely, if it becomes empty, it is merged back into its original *pclass*. This helps accommodate the special issues of multiplexed nodes without the full performance impact of disabling *pclasses*. While this helps, it is not, by itself, sufficient. Very large virtual topologies tend to use most or all of the available physical topology, meaning that they tend to degenerate into a state where most physical nodes are in their own *pclasses*, resulting in performance similar to simply disabling *pclasses*.

Another improvement to the search strategy came from the observation that, in a good solution, nodes that are adjacent in the virtual topology will tend to be placed on the same physical node. So, we made an enhancement to `assign`'s generation function. In this alternate version, rather than selecting a random physical node, with some probability, `assign` selects a physical node that one of the virtual node's neighbors has already been mapped to. This improvement made a dramatic difference in solution quality, leading to much tighter packing and exhibiting much better behavior in clustering connected nodes together. This alternate generation function can be enabled or disabled at runtime with a command-line flag to `assign`.

5.5.8.2 Coarsening the Virtual Graph

Though these changes to the search strategy improved `assign`'s runtime and solution quality, running `assign` on very large topologies could still take much too long for our purposes. To make the problem more tractable, we exploit topological features of the virtual topology.

We expect that most large virtual topologies will be based on the structure of the Internet; these may come from actual Internet "maps" from tools like Rocketfuel [121] or from topology generators designed to create Internet-like networks, such as GT-ITM [143], Inet [139], and Orbis [87]. The key realization is that such networks tend to have subgraphs of well-connected nodes, such as ISPs, ASes, and enterprises. In addition, we expect that many topologies will have edge-LANs that represent clusters, groups of workstations, etc.

We exploit the structure of the input topology by applying a heuristic coarsening pre-pass to the virtual graph before running assign. By giving assign a smaller virtual topology, we reduce the solution space that it must search, in turn reducing the time required to find a good solution. The goal of this pre-pass is to find sets of virtual nodes that, in a good mapping, will likely be placed on a single physical node. A new virtual graph is then generated, with each of these sets combined into a single node. These "conglomerates" retain all properties of their constituent nodes; for example, the CPU needs of each constituent are summed together to produce the CPU required for the conglomerate.

We have implemented two coarsening algorithms. The first stems from the realization that many topologies contain LANs representing groups of clients or server farms. An optimal mapping will almost always place as many members of these LANs onto a single physical node as possible. So, we find leaf nodes in LANs (that is, nodes whose only network interface is in that LAN), and combine all leaf nodes from the same LAN into a conglomerate.

The second algorithm uses a graph partitioner, METIS [71], to partition the virtual graph. We choose a number of partitions such that the average partition will fit on the "smallest" available physical node. We then combine the virtual nodes in each partition into a single conglomerate node. The quality of the partitions returned by the partitioner is dependent on the extent to which separable clusters of nodes are present in the graph. Since we are focusing on Internet-like topologies with some inherent hierarchy, we expect good results from this method.

The coarsening algorithms (particularly METIS) do not know the intricacies of the network testbed mapping problem, such as constraints on node types, resource usage, and link bandwidths; this is one reason they are able to run faster than assign itself. As a result, they may return partitions that cannot be mapped onto any physical resources; for example, METIS may return partitions that require too much CPU power or have more bandwidth than a single node can handle. Once the coarsening algorithm has returned sets of nodes, we use a multidimensional version of the "first-fit decreasing" bin-packing approximation algorithm [64] to pack these sets into the minimum number of mappable conglomerates.

Both coarsening algorithms help `assign` to run faster by making heuristic decisions that limit `assign`'s search space, but could, in turn, make clustering decisions that result in suboptimal mapping. Note that this is in contrast to *pclasses*, which do not prevent assign from exploring any unique solutions. However, in our domain, obtaining a solution in reasonable time is of primary importance. The mappings obtained by `assign` will always be valid, but it is possible that some topologies are coarsened in such a way the mapping does not make the most efficient use of resources. The biggest potential problem is fragmentation, in which the coarsening pass makes conglomerates whose sizes do not pack well into the physical nodes. We take measures to try to avoid this circumstance, by carefully choosing our target conglomerate size. In practice, the worst fragmentation we have seen caused only a 13% increase in physical resources used.

5.5.9 Subnodes

Another mapping challenge arises from physical nodes that have a hierarchical physical dependency. For example, Emulab incorporates devices such as Intel IXP [65] network processors and NetFPGA cards [96]. These nodes are hosted inside a PC, but both the hosts and hosted devices can have their own distinct set of types, network links, features, etc. Thus, they need to appear as two separate nodes in the physical topology, but we must take care to assure that, when `assign` picks these two separate nodes, its selection reflects the actual relationship in the physical topology. Thus, we have introduced, in both the virtual and physical topologies, the notion of a *subnode*. A *subnode* declaration associates a child node with a parent node; a virtual parent-child pair must then be mapped to a pair of physical nodes that are likewise a parent-child pair, or a violation is flagged.

5.6 Evaluation

In this section, we evaluate the performance of `assign`. First, we consider the performance of `assign` on a real workload—a set of virtual and physical topology files collected on Emulab over a period of 17 months. Then, we use a synthetic workload to determine how `assign` will scale to larger virtual and physical topologies, and to examine the impact of some features and implementation decisions. Next, we examine assign's ability to map simulated, ModelNet, and multiplexed topologies. Finally, we compare assign to another mapper that we have implemented which uses a genetic algorithm instead of simulated annealing.

Evaluation is primarily done in two ways: through the runtime of assign, and through the quality of the solutions it produces. To compare the quality of solutions, we compute the average error for each test case. Ideally, the average error is defined as $\frac{median-opt}{opt}$, where *opt* is the optimal score, and *median* is the median of scores across all trials. However, since it is intractable to compute the true value of *opt*, we substitute $\frac{median-min}{min}$, where *min* is the minimum score found by assign for the test case. This standard metric gives a good feel for the differing scores found by assign over repeated runs on the same topology.

All tests were performed on a 2.0 GHz Pentium 4 with 512 MB of RAM, unless otherwise noted. Except for the experiments specifically designed to test them, the coarsening pre-pass and dynamic *pclasses* were not used.

5.6.1 Topologies From Emulab

Our first set of tests were done using historical data collected from Emulab. The 3,113 test cases are virtual topologies submitted by experimenters, paired with the physical topology available at the time the experiment was submitted. Since virtual topologies and available physical resources vary widely, the goal of these tests is not to show trends such as scaling to a large number of virtual nodes. Instead, the goal is to show that assign handles the typical workload on Emulab very well.

Figure 5.6 shows the runtimes for these tests. We see three important things. First, the majority of experiments run on Emulab, and thus the typical workload for assign, consist of experiments smaller than 20 virtual nodes. Second, the relatively flat runtimes up to 30 nodes are caused by lower bounds in assign—to prevent assign from exiting prematurely for small topologies, a lower limit is placed on the number of iterations assign executes before terminating. Finally, we can see that assign always completes in less than 2.5 seconds for its historical workload.

Figure 5.7 shows the amount of error for the same test cases, which were each

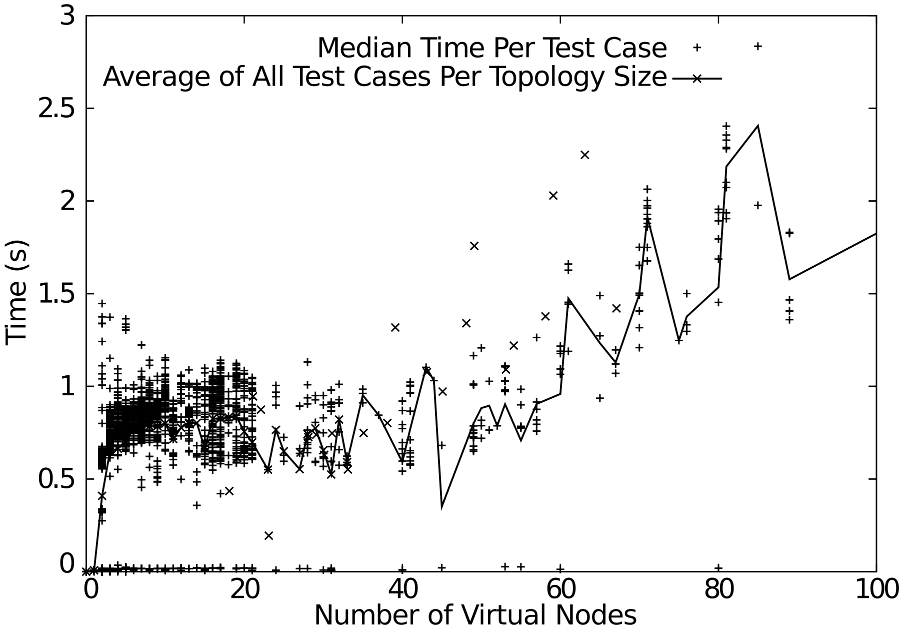


Figure 5.6. Runtimes for Emulab topologies. Each test case was run 10 times. The scatter-plot shows the median runtime for each test case. The line shows the average across all topologies of the same size.

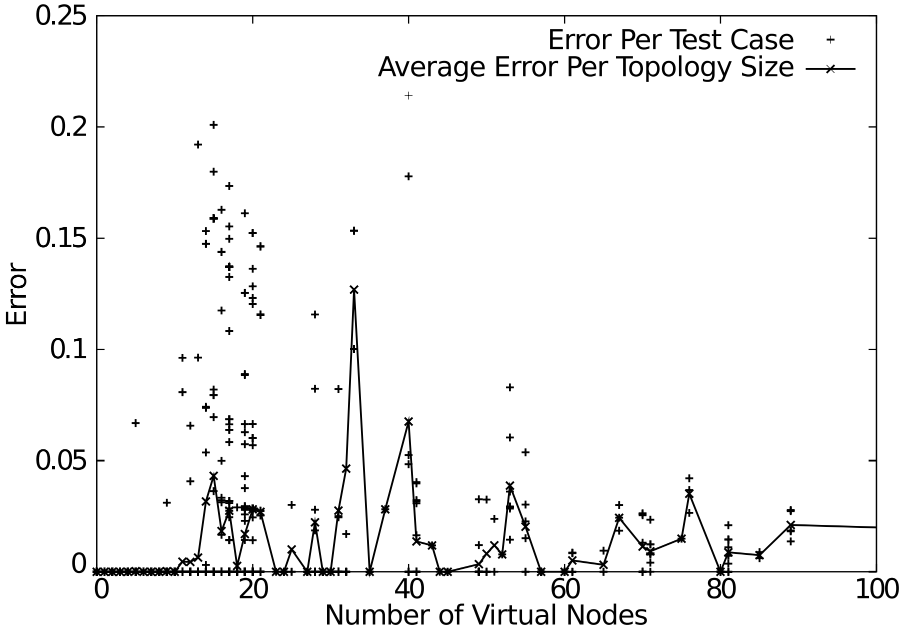


Figure 5.7. Error for Emulab topologies.

run 10 times. Here, we see that, for virtual topologies of up to 12 nodes, `assign` nearly always finds the same solution. Up to 20 nodes, covering most Emulab topologies, the error for most topologies remains below 0.05, or 5%. Even past this range, error stays low. More telling is the Cumulative Distribution Function (CDF) for these test cases, shown in Figure 5.8. Here, we see that approximately 93% of the test cases in this set showed an error of 0, 96% showed an error of less than .05, and over 99% showed an error of less than .17. From this, we can see that `assign` is more than adequate for handling the workload of the present-day Emulab. The tests in later subsections show that `assign` scales to larger Emulab-like facilities, in addition to being general enough for other environments.

5.6.1.1 Utilization

To evaluate the importance of good mapping to the utilization of Emulab's physical resources, we performed two tests. We used Emulab's actual physical topology,

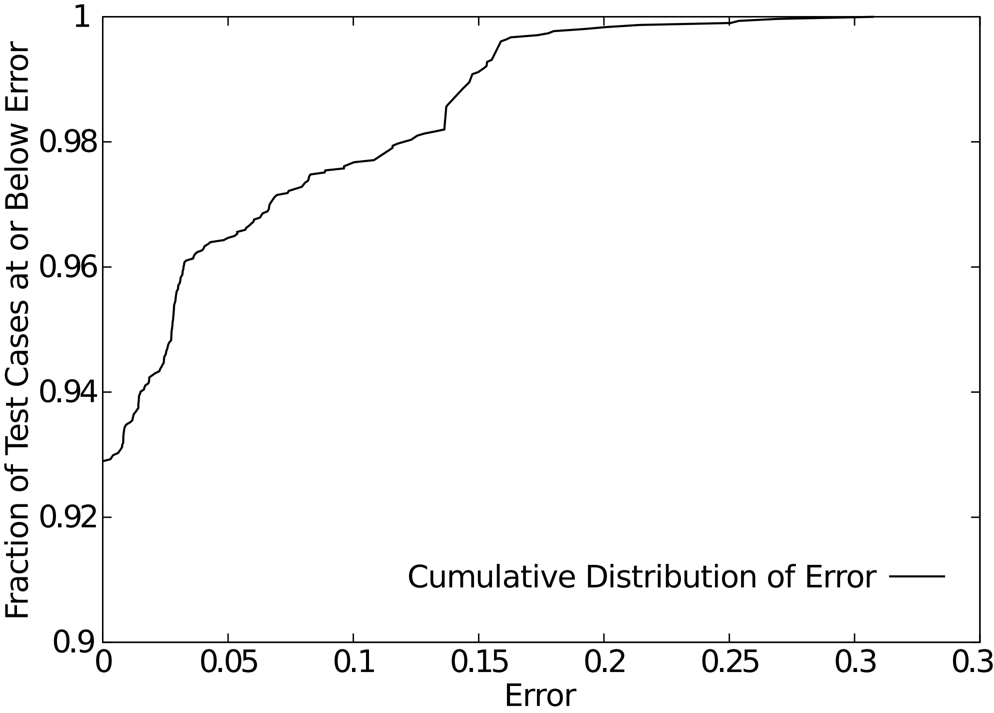


Figure 5.8. CDF of error on Emulab topologies. The line represents how many topologies had an error of a given value or smaller. Note that the y-axis for this graph begins at .90.

with the same historical virtual topologies from the last set of tests. In each test, we compared the benefit of using the normal `assign` with a version that randomly (instead of near-optimally) obtains a *valid* mapping of virtual to physical nodes. The random version still observes physical link limits, experimenters' constraints on node types, etc.; it simply returns the first solution that it finds with no violations.

For the first test, we measured throughput. We placed the virtual topologies into a randomly-ordered work queue. Experiments were removed from the queue and mapped until the mapper failed to find a solution due to overuse of interswitch bandwidth or lack of free nodes. At that point, the queue stalled until one or more experiments terminated, allowing the experiment at the head of the queue to be mapped. Each experiment was assumed to terminate 24 hours after beginning. Mapping using `assign` processed the queue in 194 virtual days, while random mapping took 604 days, a factor of 3.1 longer.[3] Limited by trunk link overuse, random mapping maintained an average of only 5.1 experiments on the testbed. Limited by available nodes, `assign` maintained an average of 16 experiments.

For the second test, we used consumption of interswitch bandwidth as our metric. First, we altered the physical topology to show infinite bandwidth between switches. As above, we generated a randomly-ordered work queue and mapped experiments until one failed to map by exceeding the number of available nodes. We recorded bandwidth consumption on the interswitch links. To prepare for the next iteration, we emptied the testbed and reshuffled the queue. The result, after 30 iterations, was that `assign`-based mapping used an average of 0.28 Gbps across the interswitch links, while random mapping used 7.4 Gpbs, a factor of 26 higher.[4]

[3] The random mapper timed out and could not map 98 large experiments due to overuse of the interswitch links, even on an empty testbed; we adjusted by assuming they mapped and took the entire testbed.

[4] The apparent disparity between the ratios in the throughput (3) and bandwidth consumption tests (26) is explained by observing that for bandwidth, the difference on the bottleneck link between bandwidth use (5.7 Gbps) and capacity (2 Gbps) is what governs job admission in the throughput test; the *use/capacity* ratio is 2.85.

5.6.2 Synthetic Topologies

For the remainder of our performance results, we use synthetically generated topologies, rather than those gathered from Emulab. One reason for this is that the Emulab topologies vary widely, making it difficult to discern whether trends are due to irregularities in the data, such as topologies with no links, or due to `assign` itself. Second, we wish to show that `assign` scales well past the resources currently available on Emulab.

Virtual topologies for these tests were generated using BRITE [91], a tool for generating realistic inter-AS topologies. A simple Waxman model with random placement was used. This results in topologies that are relatively well-connected, of average degree 4. This provides a good test of `assign`'s abilities, as such topologies are more difficult to map than ones that have tree-like structures, due to the lack of obvious "skinny" points in the topology.

The first test set, brite100, consists of 10 topologies ranging from 10 to 100 nodes. The physical topology is similar to Emulab's, with 120 nodes divided evenly among three switches. The majority of tests are run using this test set, as the randomized nature of `assign` makes it necessary to run a large number of tests to distinguish real overall trends from random effects, and the modest runtimes of this test set make this feasible; each topology in this test case was run 100 times.

The second test set, brite500, is similar to the brite100 test set, but has virtual topologies ranging from 50 to 500 nodes which are mapped onto a physical topology containing 525 nodes divided evenly across 7 switches.

5.6.2.1 Scaling

Figure 5.9 shows runtimes for the brite100 test set. Here, we can see that the mean runtime goes up in an approximately linear fashion, and that, for most test cases, the worst case performance is not much worse than the mean performance. While there is significant variation in the mean runtime, due, we believe, to the relative difficulty of mapping each topology, the best and worst case runtimes remain very linear.

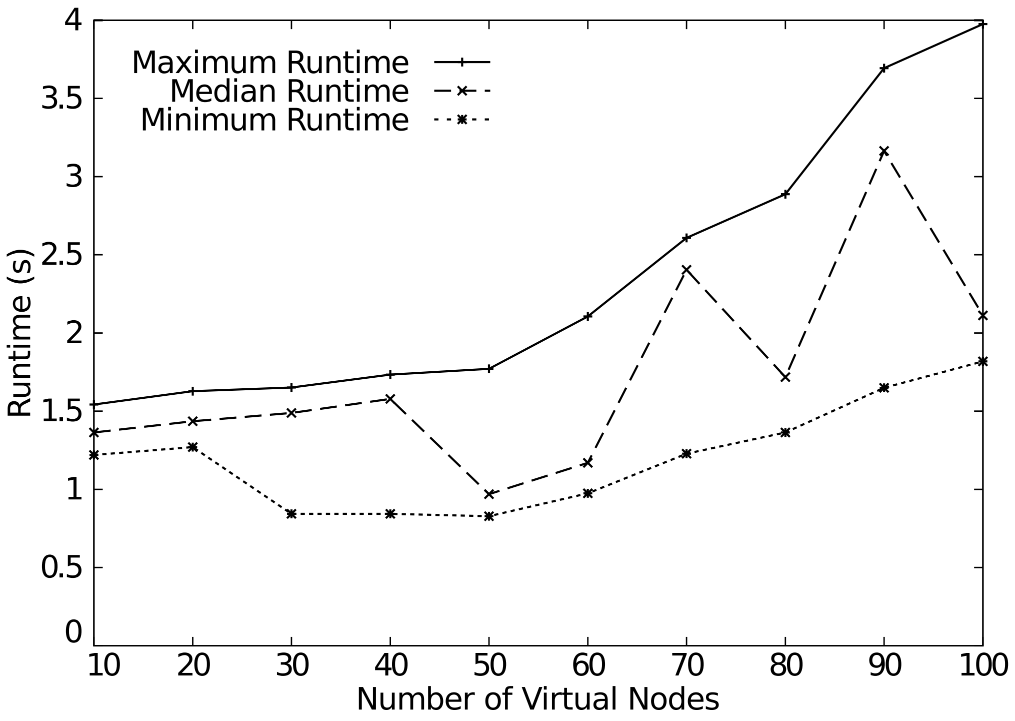


Figure 5.9. Runtimes for the brite100 test set.

Figure 5.10 shows error for the same test set. The low error up to 40 nodes reflects the fact that these topologies can be fit into the nodes on a single switch, and `assign` usually finds this optimal solution. For larger, more difficult, topologies, assign still performs well, with an average of only 5% error.

Figures 5.11 and 5.12 show, respectively, the runtimes and error for the brite500 test set. Again, we see linear scaling of runtimes. The slope of the line is somewhat steeper than that of the brite100 set. This is due to the larger physical topology onto which these test cases are mapped.

5.6.2.2 Physical Equivalence Classes

To evaluate the effect that *pclasses* have on `assign`, we ran it with *pclasses* disabled. Runtimes increased by two orders of magnitude, as shown in Figure 5.13, in which the runtime with *pclasses* enabled is barely visible at the bottom of the graph. This is primarily due to the fact that the physical topology used for this set of tests has 120 physical nodes that reduce to 6 *pclasses*, a 95% reduction.

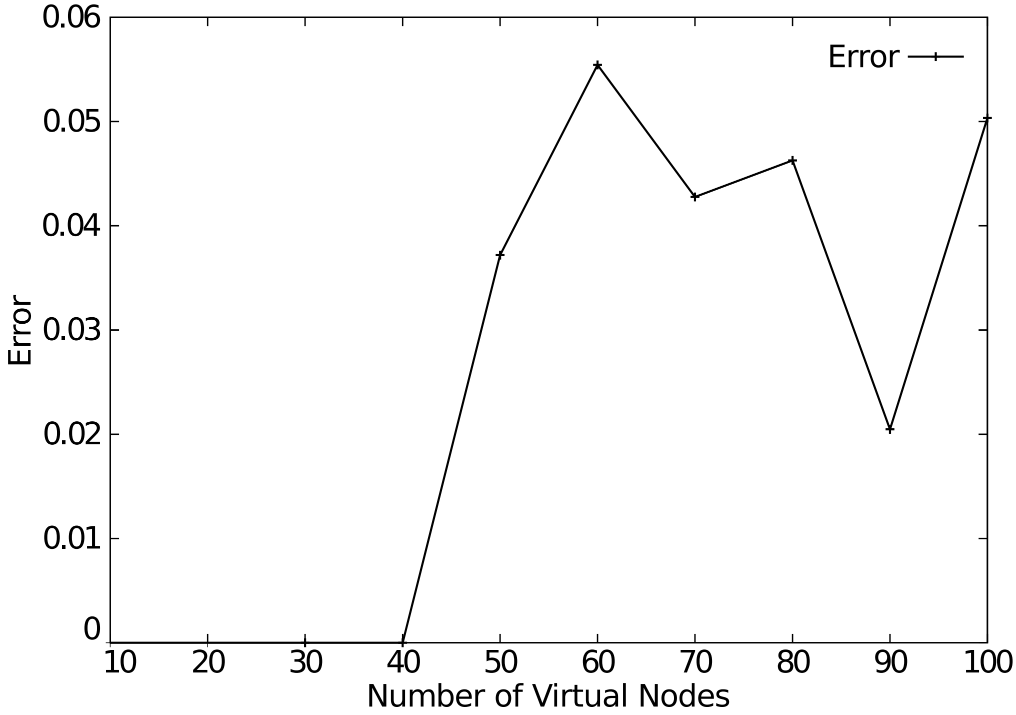


Figure 5.10. Solution quality for the brite100 test set.

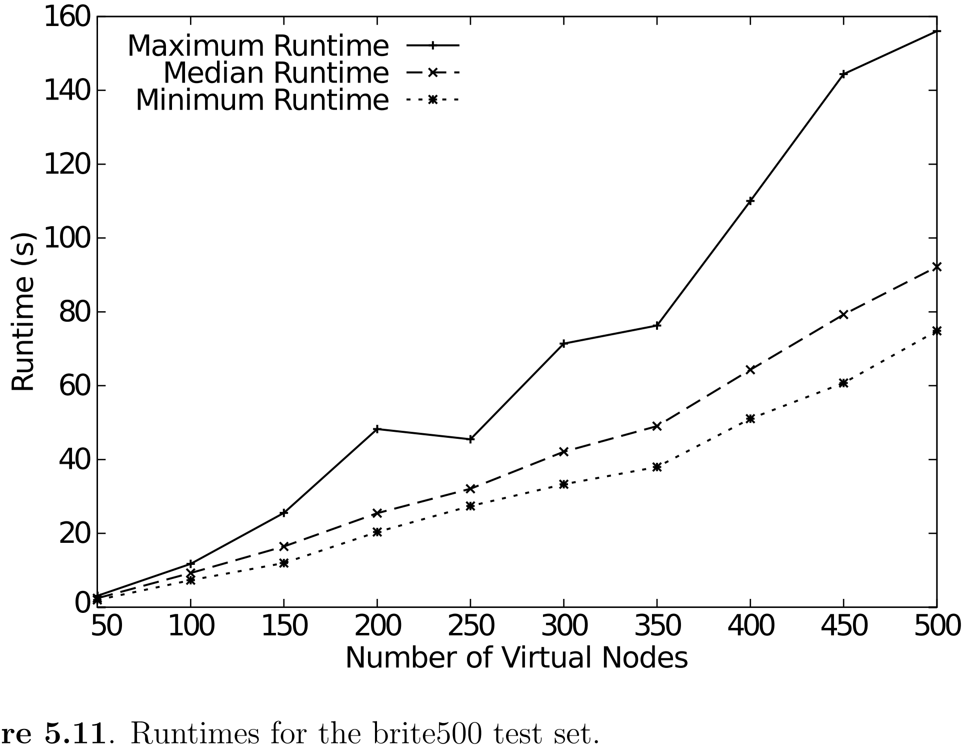


Figure 5.11. Runtimes for the brite500 test set.

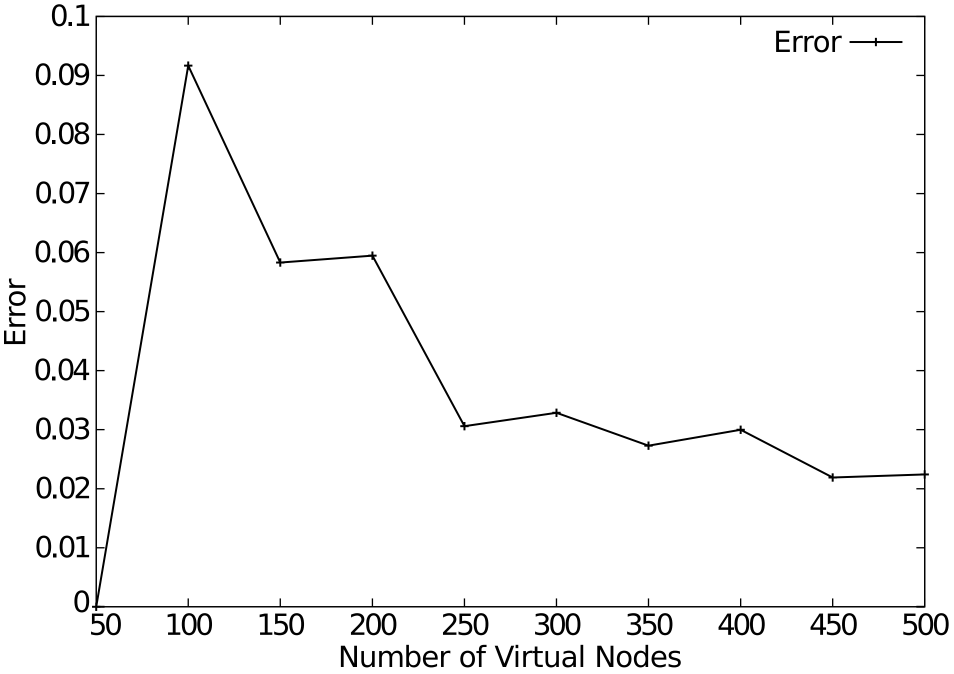


Figure 5.12. Solution quality for the brite500 test set.

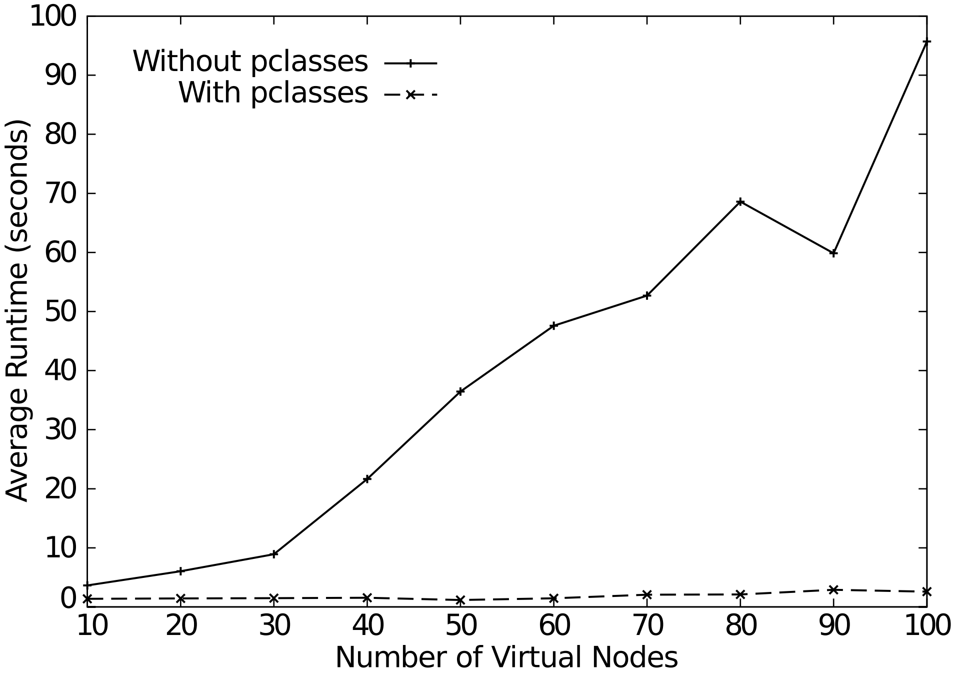


Figure 5.13. Runtimes for the brite100 test with and without *pclasses*.

Error in the solution found went down significantly due to the longer runtimes, as shown in Figure 5.14. The decrease suggests that some tuning may be possible to improve solution quality in the version of `assign` that has *pclasses*. However, the magnitude of the runtime increase clearly does not justify the extra reduction of error, which was already at an acceptable level. Though error is lower, the minimum-scored solution found both with and without *pclasses* is the same.

5.6.2.3 Features and Desires

For our first test of features and desires, we examined `assign`'s performance in avoiding nodes with undesired features. For this test, we gave 40, or one-third, of the physical nodes in the brite100 physical topology a feature, called `undesirable`, which was not desired by any nodes in the virtual topology. We gave this feature a weight that penalizes using an `undesirable` node more severely than using an extra interswitch link. This feature was given to all nodes on one of the three switches, so that it does not introduce additional *pclasses*, which would have lengthened the runtime.

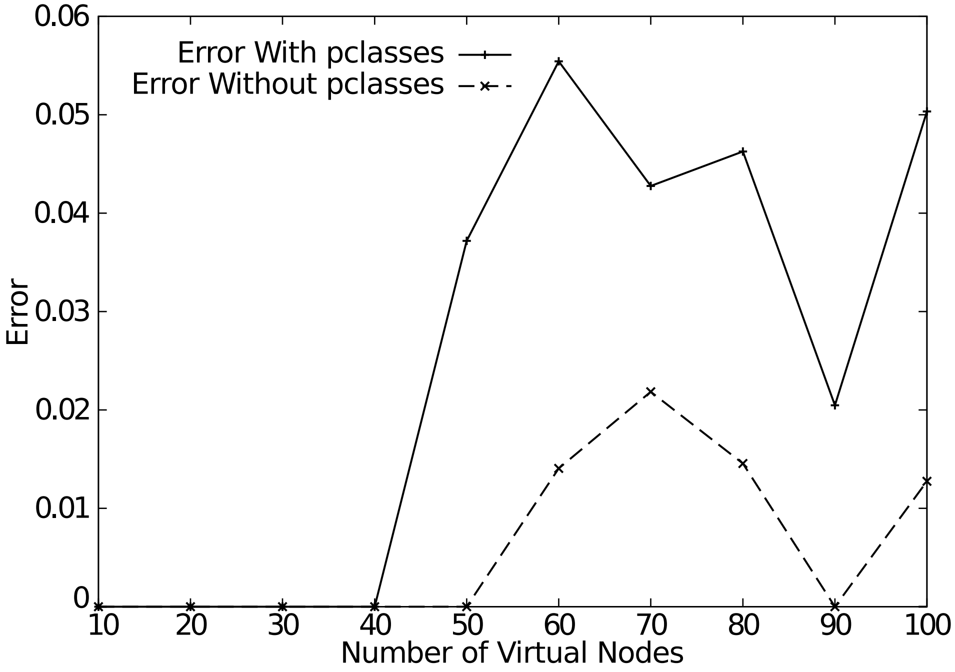


Figure 5.14. Solution quality for the brite100 test with and without *pclasses*.

We found that, in all runs, `assign` properly avoided using `undesirable` nodes. Up to 80, the number of nodes without the `undesirable` feature, `assign` avoided using `undesirable` nodes entirely. At 90 nodes, all solutions found used only the minimum of 10 `undesirable` nodes, and at 100 nodes, all solutions used only 20 `undesirable` nodes.

Figure 5.15 shows runtimes for this test. As we can see, features used in this manner do not adversely affect runtime. Figure 5.16 compares error for this test case to the cases without features, which is quite similar.

To examine how well `assign` does at finding desired features, we again modified the physical topology from the brite100 set, giving 10% of the nodes feature `A` and another 10% feature `B`. These nodes were spread evenly across all three switches in the physical topology. This results in a larger number of *pclasses* (specifically, three times as many) than the base brite100 physical topology, and thus longer runtimes. Then, 10% of nodes in the virtual topology were given the desire for feature `A`, and none given the desire for feature `B`. Thus, `assign` will attempt to map certain virtual nodes to the physical nodes with feature `A`, and will try to avoid

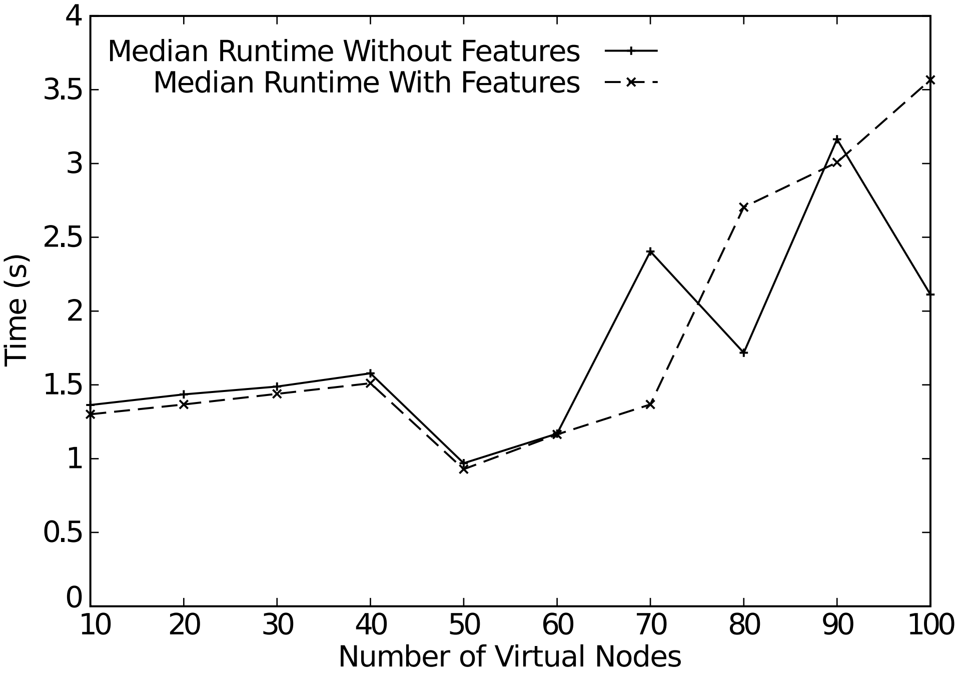


Figure 5.15. Runtimes for the brite100 test set when avoiding undesirable features.

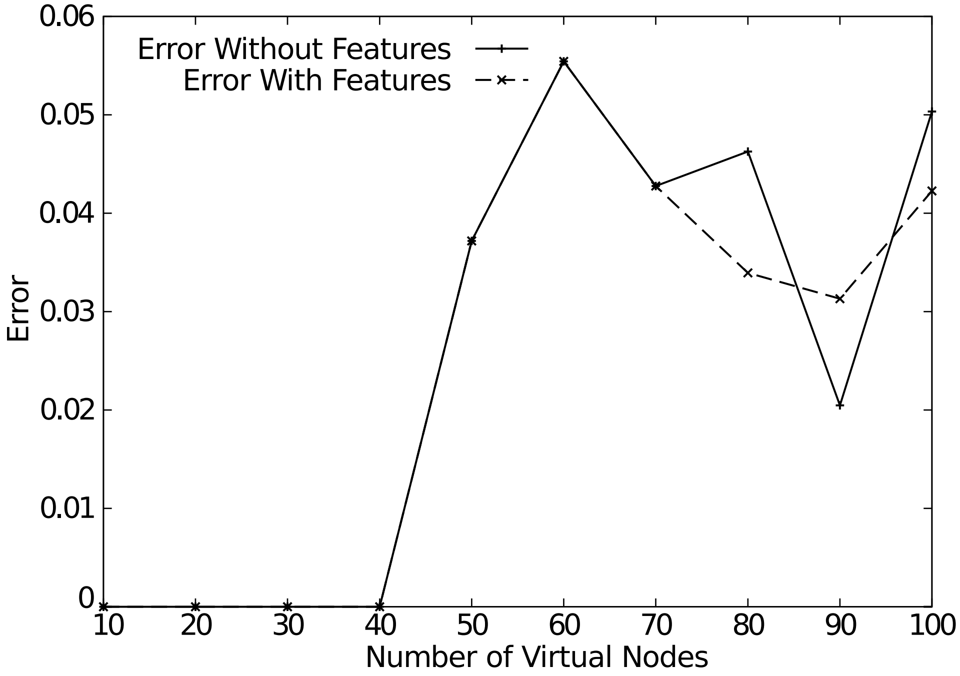


Figure 5.16. Solution quality for the brite100 test set when avoiding undesirable features.

the nodes with feature B.

Figures 5.17 and 5.18 show the results from this test. As expected, the slope of the runtime line is steeper with these features than without them, due to the fact that they introduce new *pclasses*. In nearly all tests runs, assign was able to satisfy all desires for feature A. In the 100-node test case, however, failure to satisfy the desire led to a 4% failure rate.

For topologies of 30 nodes or smaller, which allow a mapping that remains on a single switch without using nodes with feature B, avoiding these nodes is simple, and assign found such a solution in all of our test runs. For larger topologies, the weight that we gave to feature B, .5, plays an important role in the optimal solution. This weight scores the feature as being more valuable than two interswitch links, but less valuable than three. Thus, depending on the virtual topology, it may be desirable for assign to conserve interswitch links rather than nodes with this feature. Table 5.2 shows the number of nodes with feature B in the minimally-scored solution, along with the median number chosen. If we considered feature B to be

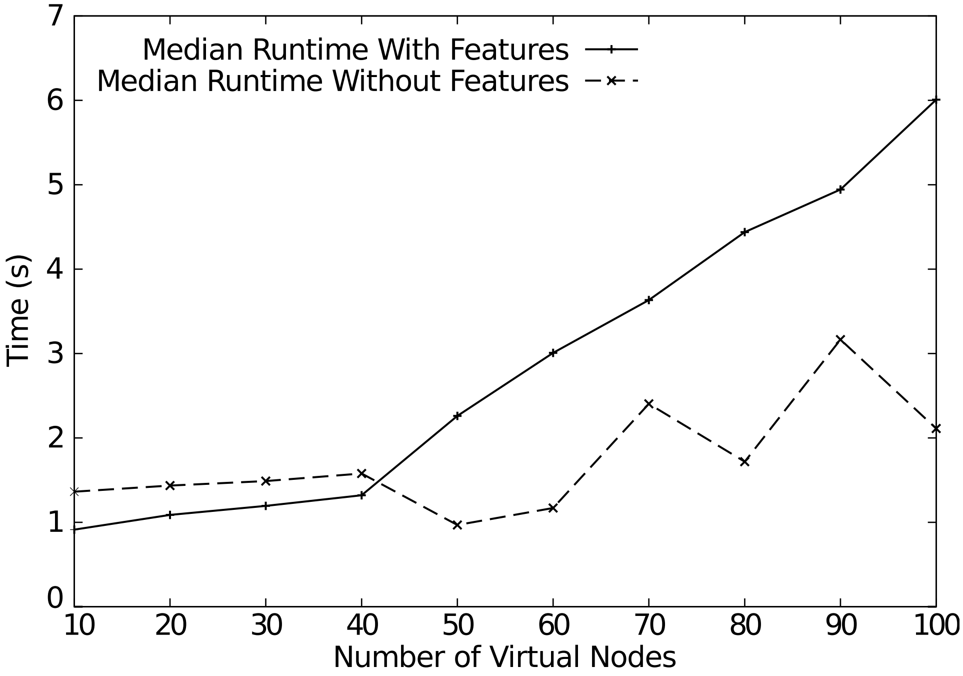


Figure 5.17. Runtimes for the brite100 test set when attempting to satisfy desires.

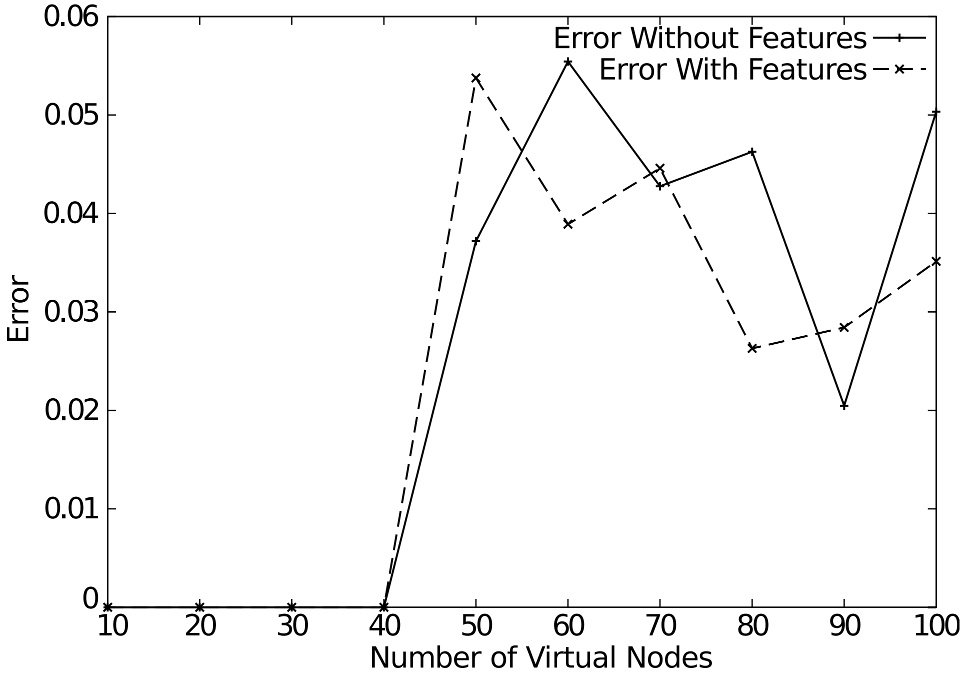


Figure 5.18. Solution quality for the brite100 test set when attempting to satisfy desires.

Table 5.2. assign's performance in avoiding feature B.

Test Case	Nodes selected with feature B	
	Minimum	Median
10	0	0
20	0	0
30	0	0
40	4	4
50	3	4
60	3	4
70	3	4
80	4	4
90	4	4
100	4	4

more valuable, we could give it a higher weight so that its cost is higher than a larger number of interswitch links.

5.6.3 Distributed Simulation

To test mapping of distributed simulation with assign, we first mapped the 500-node topology from the brite500 test set as a simulated topology. To do this, we multiplexed 50 virtual nodes on each of 10 physical nodes. The mapping typically took 46 seconds with an error of .023.

Second, we applied assign to a large topology generated by the specialized topology generator provided with the PDNS [114] simulator. This topology consists of 416 nodes divided into 8 trees of equal height, with the roots of all trees connected in a mesh. In total, this topology contains 436 links. Since the topology generated is of a very restricted nature, the script that generated it is able to optimally partition it, using only 56 cross-node links. Because of its generality, assign does not find the same solution. It does, however, typically find a very good solution: the median number of cross-node links found in our test runs was 60. For comparison, a random mapping of this topology typically results in 385 cross-node links.

The ideal test of the mappings found by assign for PDNS is to measure the runtime of the distributed simulation, both when mapped by assign, and when using the optimal mapping. However, limitations of PDNS at the time of writing make it unable to accept arbitrary network partitions, such as those generated by `assign`. Newer versions of PDNS, however, may remove these limitations and allow us to do this comparison.

Running these tests, we encountered unexpected behavior in `assign`; it performed very poorly when mapping these topologies as exact-fits. By slightly increasing the number of virtual nodes allowed on each physical node, we were able to dramatically increase `assign`'s solution quality. For example, with the PDNS topology, when each physical node was allowed to host exactly 52 virtual nodes (416/8), the error exceeded 0.4. By allowing each physical node to host 55 virtual nodes, we lowered this error to .05.

It remains an interesting problem for us, then, to analyze this phenomenon and improve `assign` accordingly. In the case of simulation, it appears we can easily adapt by providing excess "virtual capacity." For physical resources, we would need to improve exact-fit matches. Since simulated annealing has fundamental problems dealing with tightly constrained problems, this is likely best attacked by improving `assign`'s generation function.

5.6.4 ModelNet

In order to apply `assign` to mapping ModelNet, we developed tools to convert ModelNet's topology representation into `assign`'s. We then mapped the topology used by Yocum et al. [127] to evaluate ACDC, an application-layer overlay. This topology is a transit-stub network containing 576 nodes to be mapped onto the ModelNet core. Transit-transit links have a bandwidth of 155 Mbps, transit-stub links have a bandwidth of 45 Mbps, and stub-stub links are 100 Mbps. The results of mapping this topology to differing numbers of core nodes is shown in Table 5.3. Though the error is significantly higher than for the Emulab topologies that `assign` has been tuned for, the average bandwidth used by each core node stays near 1000 Mbps, which is the speed of the core nodes' links.

ModelNet's goal of balancing virtual nodes between core nodes can be met in two different ways with `assign`. First, the type system can be used to enforce

Table 5.3. Performance of assign when mapping a ModelNet topology. The bandwidth shown is the average bandwidth used by each core node to communicate with other cores.

Cores	Runtime (s)	Bandwidth (Mbps)	Error
1	0.184	0	0
2	4.81	1332	0.27
3	10.5	1183	0.36
4	16.61	947.5	0.28
5	26.0	807.6	0.24

limits on the number of virtual nodes that can be mapped onto a single ModelNet core. Second, we have implemented experimental load-balancing code in assign that attempts to spread virtual nodes evenly between physical nodes.

Because they use different scoring functions, direct comparison between the solutions from assign and ModelNet's mapper is problematic. The best test would be to run both mappers and the resulting emulations, and compare the details of their performance and behavior.

5.6.5 Multiplexed Virtual Topologies

Next, we examine assign's performance on large multiplexed virtual topologies such as those enabled by Emulab's virtual node support [53]. It is important to note the relationship between these experiments and those presented earlier in Section 5.6.2. The earlier set of experiments used only one-to-one physical mappings and thus assign got the full benefits of using *pclass*es. The multiplexed nature of the experiments presented in this section forces assign to use dynamic *pclass*es, described in Section 5.5.8.1. While this does not entirely remove the benefit of *pclass*es, it does greatly diminish their effects. As a result, even for topologies of the same size, assign is much slower on the multiplxed experiments when the pre-pass is not in use.

To understand the effects of the coarsening pre-pass, we compared runs of assign with and without the pre-pass. These runs mapped transit-stub topologies generated by GT-ITM onto Emulab's physical topology. These experiments were run on a 1.5 GHz Pentium 4, and each test was run ten times. In all cases, the runtime of the pre-pass itself was negligible compared to the runtime of `assign`.

Figure 5.19 presents the median runtimes for these tests, showing the significant time savings from the pre-pass. As we scale up the number of virtual nodes, the improvement goes from a factor of 15 at 100 nodes (12.0 vs. 0.78 seconds), to a factor of 32 at 1000 nodes (6560 vs. 200 seconds). The absolute result is also good: it takes just 200 seconds to map 1000 nodes.

The speedup from the pre-pass does not come without a cost. Figure 5.20 shows the decrease in solution quality, in terms of the quality of link mappings. Intranode links connect two virtual nodes mapped to the same physical node; they do not use shared switch resources, so having a large number of them is an indicator of a good mapping. Interswitch links, on the other hand, are an indicator of a poor mapping, because they consume the shared resource of bottleneck trunk links. Though the pre-pass does cause `assign` to find somewhat worse mappings, the differences are tolerable, and the speedup is a clear win. In over 70% of the test cases, the number of intranode links found when using the pre-pass was within 10% of the number found by `assign` by itself. The worst run was within 16%.

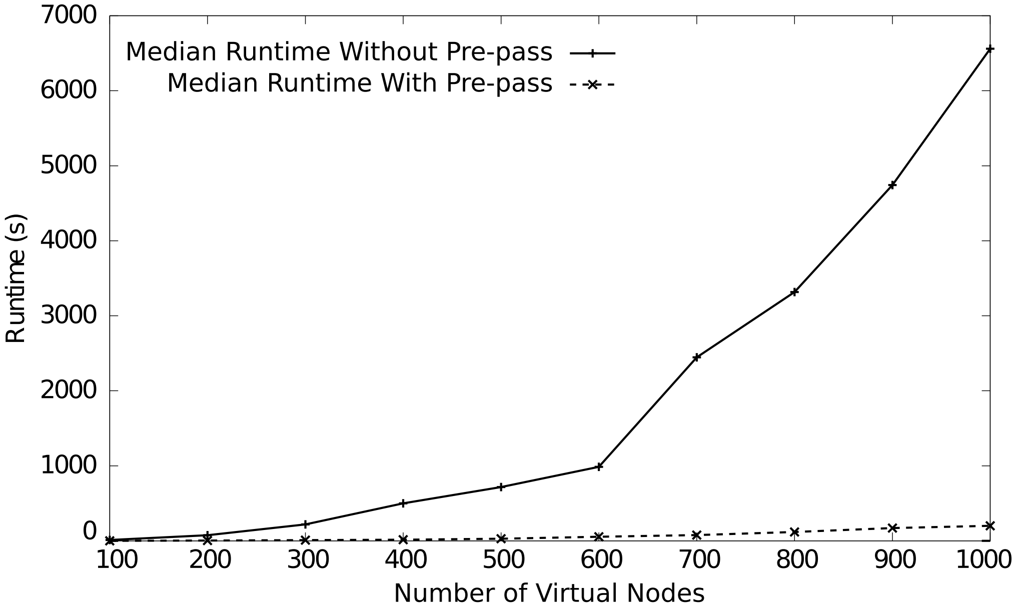


Figure 5.19. Median runtime of `assign` with and without a coarsening pre-pass.

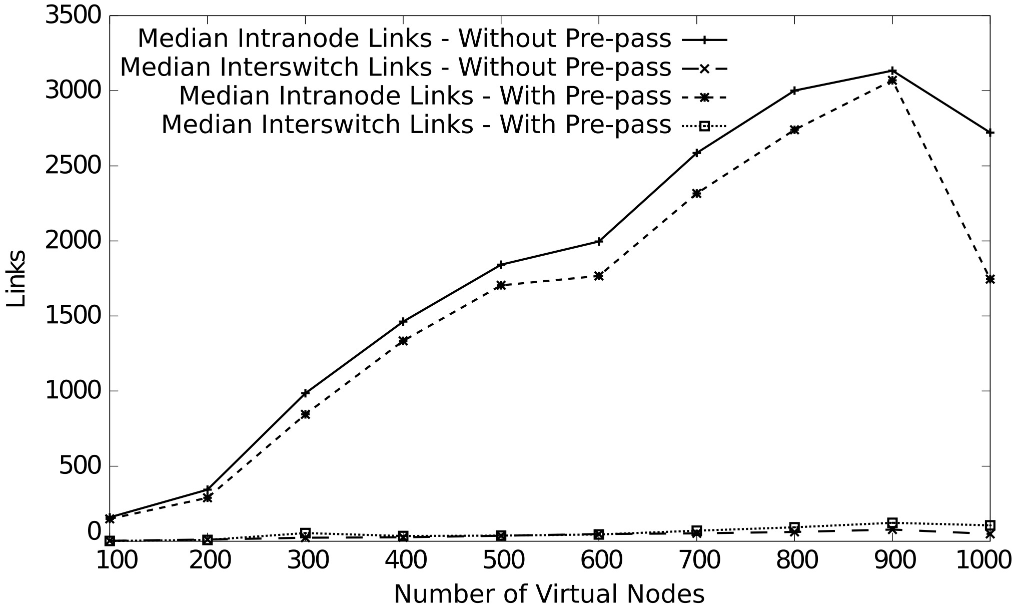


Figure 5.20. Number of intranode and interswitch links found by `assign`. Larger numbers of intranode links are better, and smaller numbers of interswitch links are better.

5.6.6 Comparison to Genetic Algorithm

Finally, we compared our simulated annealing approach to the testbed mapping problem to another general-purpose randomized heuristic approach, a genetic algorithm (GA) [51]. For this test, we independently implemented another mapper. This mapper uses a standard generational GA, with tournament selection and a specialized crossover operator. The population size is 32, the mutation rate 25%, and the crossover rate 50%. We took care to ensure that the cost functions of the two mappers are identical so that we can compare scores and errors of returned solutions.

Except for small topologies, where it was worse, the quality of solutions found by the GA mapper, shown in Figure 5.21, is close to `assign`'s. Performance, however, is quite different. For the brite100 topologies (not shown), the GA was faster when mapping 40 or fewer virtual nodes. However, as shown in Figure 5.22, the GA scaled much more poorly than simulated annealing; for all of the brite500 test cases, the GA was slower. At 500 virtual nodes, the GA mapper took nearly five times longer.

The key reason for this disparity in performance is incremental scoring, which

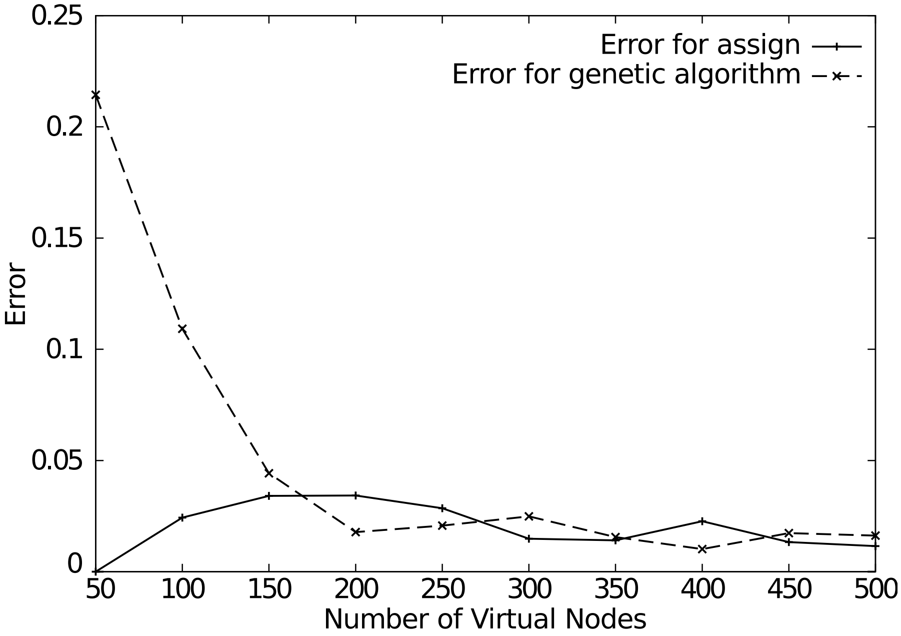


Figure 5.21. Solution quality for the brite500 test set for `assign` and our genetic algorithm.

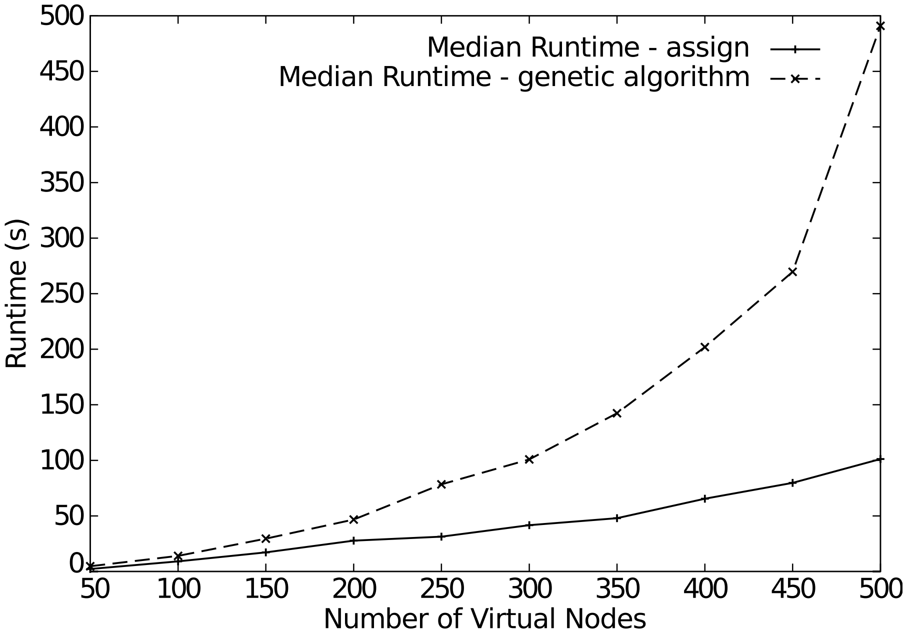


Figure 5.22. Runtimes for the brite500 test set for `assign` and our genetic algorithm.

cannot be done in GAs with crossover. When a new configuration is generated, assign incrementally alters the score. However, the GA relies on a crossover operator that blends two parents to produce two children. Here, incremental scoring is not feasible; childrens' scores must be entirely re-evaluated. The linearly increasing cost of evaluation is somewhat offset by the GA requiring fewer evaluations, on average, than simulated annealing; this accounts for its good performance on small topologies. However, the GA exhibits super-linear scaling as both the cost of evaluations and the number of evaluations required increase. This experiment indicates that simulated annealing and other search techniques that allow incremental scoring are, in general, likely to scale better on the network testbed mapping problem that those that do now allow it.

5.7 Related Work

Simulated annealing was first proposed for use in VLSI design [73], and has been studied extensively in the literature [1, 129, 128]. The key problem it was intended to solve was the placement of circuits, which are arranged in a connectivity graph, onto chips. The goal of the mapping is to minimize interchip dependencies, which require communication over expensive pins and busses. In this way, the problem is similar to ours, but does not have the unique challenges described in Section 5.4. Simulated annealing is also used in combinatorial optimization in various Operations Research fields.

Similar partitioning problems arise on parallel multiprocessor computers [52]. Some network mapping algorithms can also be found in the literature. For example, Boukerche and Trapper [15] discuss partitioning of distributed simulation using simulated annealing. Kumar et al. [78] discuss algorithms for network resources when providing bandwidth guarantees for VPNs. None of these, however, meet our goal of being generally applicable across a range of experimentation environments.

Since our work was originally presented [111], other approaches to the testbed mapping problem have been explored.

MacDonald [85] used tabu search [50] as a replacement for simulated annealing.

This work started from the `assign` source code, replacing the search mechanism. Tabu search is similar to simulated annealing in that it performs a random walk in the search space. The primary difference is that tabu search specifically avoids revisiting solutions that it has evaluated recently, and does not use the "temperature" method for deciding whether or not to accept new solutions. MacDonald found that, in general, tabu search outperformed simulated annealing on small topologies and underperformed on large topologies. In some experiments, however, tabu search was able to find solutions where simulated annealing was unable to, suggesting that tabu search may be a better choice when the fit is "tight."

Other related work considers a problem that is similar, but not identical, to ours: that in which the links are expressed as pairwise properties between nodes. In such a mapping, the problem becomes selecting a set of nodes such that the nodes and the links between them fall within parameters specified by the experimenter. These parameters are typically expressed in terms of latency and/or bandwidth, and may be expressed as a range or soft constraints. The SWORD mapper [103, 104] approaches this mapping as a combinatorial optimization problem and Considine et al. [24] consider it from a constraint-based perspective. Both approaches are complementary to our work, as they do not require the internal topology of the network between the nodes to be known; the are particularly valuable for live-network testbeds where the topology is not known, but end-to-end properties are measurable.

Yu et al. [142] proposed viewing the mapping problem as an instance of a multicommodity flow [2] problem (MCFP) and re-designing the network substrate in order to better accommodate mappings. MCFP is known to be NP-complete when the flows cannot be split, as is the case in our formulation; a virtual link must be satisfied by a single physical path. However, when the mapper is allowed to split a single flow across mutliple paths, MCFP becomes solvable in polynomial time. The authors argue that if the network substrate can be designed to allow for splitting of flows across multiple paths and for migration of flows to different paths over time, the mapping problem becomes simpler and better solutions can be found. Lischka and Karl [83] noted the relationship between the network testbed mapping problem and subgraph isomorphism detection, and applied a backtracking approached developed for that problem.

5.8 Future Work

5.8.1 Wide-Area Assignment

As network testbeds expand into the wide-area, such as Emulab's wide-area nodes [138] and PlanetLab [106], resource allocation faces a new challenge. When resources are distributed across the public Internet, an experimenter's desired topology must be chosen from the paths available, which are not controllable by the testbed's maintainers. Since the number of links between n nodes is $n(n-1)$, this problem has similar complexity characteristics to the one we have described in this chapter.

Emulab currently uses a separate program for mapping wide-area resources, which picks from among them using a genetic algorithm. Thus, two passes are used when mapping both wide-area and local resources. In general, we think that this two-phase strategy is appropriate, since doing both phases at once complicates the solution space and the choice of resources in each phase does not depend on choices made in the other phase. However, we plan to investigate whether it is appropriate to use the same program, or at least the same approach, for both phases.

5.8.2 Dynamic Delay Nodes

Emulab's delay nodes [138] present an interesting mapping challenge: whether or not a delay node is required is a function of the nodes and interfaces selected. For example, if the experimenter requests a 100 Mbps link and a node is selected that only has 1 Gbps interfaces, a delay node may be required to slow the link down to the requested speed. If a node with 100 Mbps interfaces is selected, however, the delay node will not be required. In general, it is not possible to tell ahead of time whether or not `assign` will be able to find a solution that requires a delay node. Emulab currently uses a set of heuristics to guess whether delay nodes will be required, and inserts them into the virtual topology passed to `assign` if it believes they are necessary. This state of affairs is not ideal, however, since it may insert delay nodes when they will not be needed. A more efficient solution would be for `assign` to insert delay nodes into the virtual topology itself. This dynamic addition of nodes to the virtual topology, however, presents challenges for the generation and cost functions. We have an initial implementation of dynamic delay nodes, but more work is needed.

5.8.3 Local Search

A possible way to improve `assign`'s performance would be to combine it with local search, another strategy for combinatorial optimization. One can combine simulated annealing with local search in such a way that simulated annealing is performed on local minima, rather than on all states [89]. The basic algorithm is to apply a "kick" to a potential solution, which, in contrast to the neighborhood structure typically used with simulated annealing, is designed to move to a very different area of the solution space. In `assign`, this would likely be best accomplished by reassigning a connected subset of the virtual topology, rather than a single virtual node. A local search is then done from the new configuration, attempting to find its local minima. Then, the same acceptance criteria for standard simulated annealing are applied to decide whether or not to move to the new minima.

5.9 Conclusion

We have presented the network testbed mapping problem, formulating it in such a way that it is applicable to a range of experimental environments. The distinguishing features of this problem include the necessity of giving the experimenter flexibility in specifying hardware requirements and taking into account the differences in network links in the physical topology, such as intranode, intraswitch, and interswitch links. We have presented our solver, `assign`, discussing its design, implementation, and lessons learned in the process. Through evaluation on real and synthetic workloads, we have shown its effectiveness for a range of experimental environments. A key focus of our work has been on scalability in the form of incremental scoring, *pclasses*, and a coarsening pre-pass. Finally, we have identified interesting problems for future work.

CHAPTER 6

CONCLUSION

6.1 Summary of the Dissertation

While emulation testbeds are widely used in the fields of networking and distributed systems, they have two key weaknesses. The first is with the network realism of such environments: results obtained in an emulated environment are only as realistic as the network configuration of the emulator. The second is that scaling them to sizes approximating realistic deployments is a serious challenge. This dissertation has made contributions to three key problems within these areas.

In Chapter 3, we showed that it is useful to think of emulation and live-network experimentation as being two points on a spectrum, rather than incompatible methodologies. We designed a general-purpose framework, Flexlab, for importing measurements from a live network into an emulation testbed and used it to couple PlanetLab with Emulab. Flexlab does not attempt to model the interior of the network in detail; rather, it concentrates on emulating *end-to-end* characteristics, whose effects dominate the behavior of applications deployed on end hosts, and which are easily measurable from the edges of a real network. Using this framework, we were able to produce experimentation environments that lay at several different points on the spectrum between live and emulated testbeds.

The first "Simple-static" model used measurements of a real network to set conditions within an emulation; these conditions are not changed while the experiment is running. It produces an environment that is quite predictable and repeatable, but that does not exhibit the variability over time or the reactivity to foreground traffic seen on production networks. The "Simple-dynamic" model is similar, but improves on the static model in two ways: it uses knowledge of which paths are actively used by an experiment to increase the rate of measurement on those paths, and it changes conditions in the emulator over time at fixed intervals. These conditions can be replayed for future experiments, keeping a degree of repeatability while capturing some time-varying aspects of network behavior. The final model, ACIM, is sophisticated enough to be a contribution in its own right. ACIM observes the system under test's traffic in real-time, replicates that traffic on the live network, and feeds the observed conditions back into the emulator. Doing so captures much finer-grained variability than the earlier models as well as reactive behaviors, which they miss entirely. As a result, however, it sacrifices repeatability.

Chapter 4 dealt with a different type of realism in emulation: realistic interior topologies for emulated networks. While a number of topology generators are available for simulators, these generators do not include IP addresses, which are required for emulated experimentation. The seemingly straightforward task of annotating these generated networks with addresses uncovered a wealth of interesting problems. Because we consider a "good" address assignment to be one that takes into account the hierarchy of the network, the work presented in this chapter is fundamentally about uncovering that hierarchy.

We identified a number of strategies for finding network hierarchy and assigning IP addresses based on it. The two most promising are *bottom-up tree building* and *recursive graph partitioning*. Bottom-up tree building uses a metric we devised called *routing equivalence sets* (RES), taking advantage of the graph-theoretic properties of the domain to quantify the extent to which sets of nodes can be aggregated for IP routing. The properties of RES enable an efficient greedy tournament which is able to find good address assignments in a reasonable amount of time. The recursive partitioning method takes a more heuristic approach: it repeatedly partitions the network to build a tree of subnets. This heuristic approach works quite well; it is able to find solutions that are nearly as good as the graph-theory based RES tournament, and in much less time. Our experiments showed that both methods scale well; they are able to annotate graphs the size of today's largest single-owner networks in under a minute.

Chapter 5 dealt with the problem of selecting physical hardware on which to instantiate an emulated experiment. To find such a mapping, a testbed must solve a constraint satisfaction problem: the host and network constraints specified in the virtual topology must be satisfied by their chosen physical counterparts. It must also do combinatorial optimization: scarce resources must be preserved for other experimenters whenever possible. Because this problem is both NP-hard and on the critical path for experiment creation, it is necessary to use a heuristic solver for it. The solver that we presented in this chapter, `assign`, scales well to the size of today's largest emulation testbeds. To do so, it makes use not only of standard techniques such as simulated annealing and graph partitioning, but also of domain features which we exploit to simplify the mapping problem.

We showed that it is possible to reduce the size of the solution space by exploiting regularity in testbeds' physical topologies. This improves `assign`'s runtime without compromising solution quality. We also showed that graph partitioning, while not by itself capable of satisfying testbed mapping problems, can be used as a coarsening pre-pass to the mapping problem. This technique trades off a modest reduction in solution quality for dramatic improvements in runtimes on large virtual topologies.

These contributions make significant progress towards the goal of large-scale, realistic emulation testbeds.

6.2 Future Research Directions

While this dissertation has made contributions to many of the key scaling and realism problems facing emulation testbeds, it has by no means exhausted them; much remains to be done. We now examine directions for further research.

6.2.1 Realistic End-to-End Conditions

The Flexlab work presented in this dissertation has only begun to explore the large space of possibilities for importing realistic conditions into emulation testbeds. The importance of the Flexlab work extends beyond the few models that we have identified in this dissertation: its primary value is in the *vision* and *mechanism* that we have defined for combining emulation and live-network experimentation. Though the measurement and tomography techniques that we have used in this dissertation are relatively simple, as such techniques improve, Flexlab will be able to incorporate them as new models.

Our Flexlab work has thus far focused on live-network testbeds which are overlays on the Internet; all of the hosts we have experimented with so far are connected by a wired network. There would be significant value to constructing models from other types of live networks, in particular wireless networks. The models used to emulate live wireless networks should fit into the Flexlab framework, but the models themselves will require solving new challenges. There is no limitation in the Flexlab framework that requires it to be used with an emulator that operates at Layer 2 or 3 of the network stack, as Emulab does; this work could be combined with facilities such as the CMU wireless emulator [67], which does Layer 1 emulation of wireless networks.

6.2.2 Finding Structure in Networks

Though we apply the work in Chapter 4 to the problem of annotating graphs with IP addresses, it touches on some much larger issues. It is fundamentally about finding the structure in networks and determining the amount of hierarchy present in them. For example, this work could be applied to the problem of *characterizing* networks: the degree to which a network's addresses can be aggregated is a measure of how "purely" hierarchical its structure is. Using this metric, it may be possible to categorize networks. This metric could also be useful for evaluating topology generators: if the aggregatability of the generated networks differs significantly from that of real networks, this could be a sign that the generated topology is not sufficiently realistic. It may also be possible to directly apply the lessons of RES to the generation of topologies.

6.2.3 Broadening the Testbed Mapping Problem

Future work on `assign` could be taken in a number of different directions.

One branch of work involves further scaling for `assign`. The results presented in this dissertation scale up to physical testbeds of hundreds of nodes, running virtual topologies of thousands. Testbeds with thousands of physical nodes with virtual topologies into the tens of thousands are likely in the future. In order to grow another order of magnitude, further improvements to `assign` will be necessary. Parallelization of simulated annealing has been studied [81, 72, 74], and while applying these techniques to `assign` is likely to results in modest performance improvements, they are not likely to be sufficient by themselves. The most likely avenues for scaling improvement involve partitioning the virtual and physical topologies into sub-problems which can be solved independently; because `assign`'s runtime is super-linear, solving several smaller problems can be faster than solving a single large one. Of course, such partitioning is likely to reduce solution quality, as we saw with the pre-pass, so the key will be finding good-quality partitions. Such work would need to simultaneously partition the virtual and physical topologies, and would thus present some interesting new challenges in graph partitioning.

Another branch of future work involves the model `assign` uses for packet forwarding; currently, the type `switch` has special meaning to `assign`: only `switch`es can be the intermediate nodes in multihop paths. This limits `assign` to mapping topologies in which the infrastructure operates at a single layer of the network. (Though this type is named `switch`, there is nothing inherently specific to Ethernet or Layer 2 in `assign`.) By generalizing support for packet forwarding, `assign` can be made to support multilayer experiments. To do so, physical nodes would be marked with the set of protocols they are able to forward, and a layering of protocols would be established. `assign` would then construct multiple forwarding graphs for calculating multihop paths. This would enable `assign` to support testbeds that include physical-layer switches, Ethernet switches, and IP routers.

6.2.4 Improving Network Experimentation

The work presented in this dissertation is part of a broader context, the more general problem of improving network experimentation. This issue is not limited to improving emulation testbeds or network testbeds in general. It encompasses the environments, tools, and methodologies used for conducting experiments. We close by identifying some of the difficult open problems in network experimentation.

- **Generality:** When designing an experimentation environment, there is a tension between designing a general-purpose facility and designing one that is focused on supporting a particular class of systems or experiments. A focused environment can be more effective or efficient for evaluating its target domain, but may fill too small a niche. Conversely, a general-purpose environment may support many experiments, but not support any of them particularly well, and may rule out some specialized classes altogether. The “sweet spots” in the design space are environments that target a sufficiently large class of important research questions while being focused enough to support that class well. Finding them is a major challenge; while the environments presented in Chapter 2 represent several such points, many parts of the design space remain unexplored.

- **Full Lifecycle Support:** Projects go through many phases, including design, prototyping, development, evaluation, and, if successful, deployment. Each of these stages has different demands. Today’s experimental tools and environments tend to target specific stages in this lifecycle, making transitions between them difficult. The development of a comprehensive suite of tools that seamlessly span the full lifecycle would be a major boost to network research, as it would ease the progression from the conception of an idea through deployment of it in a production environment.

- **End-User Participation:** Some experiments are not well served by being isolated within a testbed; they require interaction with the larger world, providing services to end users or acting as consumers of those services themselves. This raises questions, such as: Do these users need to opt-in or can experiments capture their traffic with explicit user consent? Since services offered by researchers may be unstable or incomplete, how does one build a reasonable failsafe so that end users are not negatively impacted by service failures? When real traffic is used to generate workloads or data for further analysis, how can user privacy be preserved without compromising the value of the data? Participation of end users also raises a host of ethical and legal issues. These questions must be addressed in order to make user participation possible on a large scale.

- **Comparability:** At the heart of most network experimentation is the need to compare systems with each other, such as showing that a new system scales better than an existing one or handles congestion more gracefully. However, comparing networked systems is nuanced; it requires evaluating them under the same conditions, which are particularly challenging to control in a complex network environment. Small changes in the hardware or software of nodes and links, along with the conditions observed on them, can have significant impacts. The ability to package the entire environment in which an experiment is run would allow that experiment to be repeated and improved upon. Some existing evaluation environments provide support for such packaging—for example, it is relatively straightforward to capture all inputs to a simulator. Bringing packaging capabilities to more complicated environments, such as emulation testbeds, would go a long way towards making experiments run on them more comparable.

While significant progress has been made on these fronts in recent years, many opportunities for improvement remain. Due to the critical role that experimentation plays in network research, these problems are deserving of attention.

REFERENCES

[1] Aarts, E. H. L., and Korst, J. *Simulated Annealing and Boltzmann Machines*. John Wiley & Sons, 1989.

[2] Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[3] Albrecht, J., Tuttle, C., Snoeren, A. C., and Vahdat, A. PlanetLab application management using Plush. *ACM SIGOPS Operating Systems Review 40*, 1 (Jan. 2006), 33–40.

[4] Alderson, D., Doyle, J., Govindan, R., and Willinger, W. Toward an optimization-driven framework for designing and generating realistic Internet topologies. *ACM SIGCOMM Computer Communications Review* (Jan. 2003).

[5] Andersen, D. G. Theoretical approaches to node assignment, December 2002. Unpublished Manuscript. `http://nms.lcs.mit.edu/papers/andersen-assign.ps`.

[6] Andersen, D. G., Balakrishnan, H., Kaashoek, F., and Morris, R. Resilient overlay networks. In *Proc. of the 18th ACM Symposium on Operating Systems Principles* (Mar. 2001), pp. 131–145.

[7] Andersen, D. G., and Feamster, N. Challenges and opportunities in Internet data mining. Tech. Rep. CMU–PDL–06–102, Carnegie Mellon University Parallel Data Laboratory, Jan. 2006.

[8] Anderson, T., Peterson, L., Shenker, S., and Turner, J. Overcoming the Internet impasse through virtualization. *IEEE Computer* (Apr. 2005).

[9] Awerbuch, B., Bar-Noy, A., Linial, N., and Peleg, D. Improved routing strategies with succinct tables. *J. of Algorithms 11*, 3 (1990), 307–341.

[10] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles* (Oct. 2003), pp. 164–177.

[11] Bavier, A., Feamster, N., Huang, M., Peterson, L., and Rexford, J. In VINI veritas: Realistic and controlled network experimentation. In *Proc. of SIGCOMM 2006* (Sept. 2006), pp. 3–14.

[12] Berge, C. *Graphs and Hypergraphs*, second ed., vol. 6. North-Holland, Amsterdam, 1976, pp. 389–390.

[13] Bharambe, A. R., Herley, C., and Padmanabhan, V. N. Analyzing and improving a BitTorrent networks performance mechanisms. In *Proc. of IEEE INFOCOM* (Apr. 2006).

[14] Boost C++ libraries. `http://www.boost.org/`.

[15] Boukerche, A., and Tropper, C. A static partitioning and mapping algorithm for conservative parallel simulations. In *Proc. of the Eighth Workshop on Parallel and Distributed Simulation* (1994).

[16] Brakmo, L., O'Malley, S., and Peterson, L. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. SIGCOMM* (Aug.–Sept. 1994), pp. 24–35.

[17] Breslau, L., Estrin, D., Fall, K., Floyd, S., Heidemann, J., Helmy, A., Huang, P., McCanne, S., Varadhan, K., Xu, Y., and Yu, H. Advances in network simulation. *IEEE Computer 33*, 5 (May 2000), 59–67.

[18] Bu, T., and Towsley, D. On distinguishing between Internet power law topology generators. In *Proc. of IEEE INFOCOM* (July 2002), pp. 1587–1596.

[19] Cappos, J., Baker, S., Plichta, J., Nyugen, D., Hardies, J., Borgard, M., Johnston, J., and Hartman, J. H. Stork: Package management for distributed VM environments. In *In Proc. of the 21st Large Installation System Administration Conference (LISA)* (Nov. 2007).

[20] Chambers, B. A. The grid Roofnet: a rooftop ad hoc wireless network. Master's thesis, Massachusetts Institute of Technology, June 2002.

[21] Chen, J., Gupta, D., Vishwanath, K. V., Snoeren, A. C., and Vahdat, A. Routing in an Internet-scale network emulator. In *Proc. of MASCOTS* (2004).

[22] Cheng, Y.-C., Hölzle, U., Cardwell, N., Savage, S., and Voelker, G. M. Monkey see, monkey do: A tool for TCP tracing and replaying. In *Proc. of the 2004 USENIX Annual Technical Conf.* (Boston, MA, June–July 2004), pp. 87–98.

[23] Coates, M., Hero, A. O., Nowak, R., and Yu, B. Internet tomography. *IEEE Signal Processing Mag. 19*, 3 (May 2002), 47–65.

[24] Considine, J., Byers, J. W., and Mayer-Patel, K. A constraint satisfaction approach to testbed embedding services. In *Proc. of ACM HotNets-II* (Nov. 2003).

[25] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, Cambridge, Mass., 1990.

[26] Cowen, L. Compact routing with minimum stretch. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)* (1999), pp. 255–260.

[27] Dabek, F., Cox, R., Kaashoek, F., and Morris, R. Vivaldi: A decentralized network coordinate system. In *Proc. of SIGCOMM 2004* (Aug.–Sept. 2004), pp. 15–26.

[28] Diaz, J., Petit, J., and Serna, M. A survey of graph layout problems. *ACM Computing Surveys 34*, 3 (2002), 313–356.

[29] Dischinger, M., Haeberlen, A., Beschastnikh, I., Gummadi, K. P., and Saroiu, S. SatelliteLab: Adding heterogeneity to planetary-scale network testbeds. In *Proc. of SIGCOMM 2008* (Aug. 2008).

[30] Draves, R., King, C., Venkatachary, S., and Zill, B. Constructing optimal IP routing tables. In *Proc. of IEEE INFOCOM* (1999), pp. 88–97.

[31] Duerig, J., Ricci, R., Zhang, J., Gebhardt, D., Kasera, S., and Lepreau, J. Flexlab: A realistic, controlled, and friendly environment for evaluating networked systems. In *Record of the 5th Workshop on Hot Topics in Networks: HotNets V* (Nov. 2006), pp. 103–108.

[32] Eide, E., Stoller, L., and Lepreau, J. An experimentation workbench for replayable networking research. In *Proc. of the Fourth USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Apr. 2007), pp. 215–228.

[33] List of Emulab-based testbeds. `http://users.emulab.net/trac/emulab/wiki/OtherEmulabs`.

[34] Emulab bibliography. `http://www.emulab.net/expubs.php`.

[35] Emulab pc3000 hardware specification. `http://users.emulab.net/trac/emulab/wiki/pc3000`.

[36] Ettus research (website). `http://www.ettus.com/`.

[37] Fall, K. Network emulation in the Vint/NS simulator. In *Proc. of the IEEE Symposium on Computers and Communications* (July 1999).

[38] Fiedler, M. Algebraic connectivity of graphs. *Czechoslovak Mathematical J. 23*, 98 (1973), 298–305.

[39] Flammini, M., van Leeuwen, J., and Marchetti-Spaccamela, A. The complexity of interval routing on random graphs. *The Computer Journal 41*, 1 (1998), 16–25.

[40] Floyd, S., and Kohler, E. Internet research needs better models. *ACM SIGCOMM CCR (Proc. HotNets-I) 33*, 1 (Jan. 2003), 29–34.

[41] Floyd, S., and Paxson, V. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking 9*, 4 (Aug. 2001), 392–403.

[42] Flux Research Group, University of Utah. The Emulab Web site. `http://www.emulab.net/`.

[43] Francis, P., Jamin, S., Jin, Y., Raz, D., Shavitt, Y., and Zhang, L. IDMaps: A global Internet host distance estimation service. *IEEE/ACM Transactions on Networking 9*, 5 (Oct. 2001), 525–540.

[44] Frederickson, G. N., and Janardan, R. Designing networks with compact routing tables. *Algorithmica 3* (1988), 171–190.

[45] Freedman, M. J., Freudenthal, E., and Maziéres, D. Democratizing content publication with Coral. In *Proc. of the first USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)* (Mar. 2004).

[46] Fuller, V., Li, T., Yu, J., and Varadhan, K. RFC 1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, Sept. 1993.

[47] Gavoille, C., and Peleg, D. The compactness of interval routing. *SIAM J. on Discrete Mathematics 12*, 4 (1999), 459–473.

[48] GENI: Exploring networks of the future (web site). `http://www.geni.net/`.

[49] Gerich, E. RFC 1466: Guidelines for management of IP address space, May 1993.

[50] Glover, F., and Laguna, M. *Tabu Search.* Kluwer Academic Publishers, 2007.

[51] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[52] Hendrickson, B., and Leland, R. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. on Scientific Computing 16*, 2 (1995), 452–469.

[53] Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., and Lepreau, J. Large-scale virtualization in the Emulab network testbed. In *Proc. of the 2008 USENIX Annual Technical Conf.* (June 2008), pp. 113–128.

[54] Hibler, M., Stoller, L., Lepreau, J., Ricci, R., and Barb, C. Fast, scalable disk imaging with Frisbee. In *Proc. of the 2003 USENIX Annual Technical Conf.* (June 2003), pp. 283–296.

[55] Hubbard, K., Kosters, M., Conrad, D., Karrenberg, D., and Postel, J. RFC 2050: Internet registry IP allocation guidelines, Nov. 1996.

[56] Hussain, A., Kapoor, A., and Heidemann, J. The effect of detail on Ethernet simulation. In *Proc. of the ACM Workshop on Parallel and Distributed Simulation* (May 2004), ACM.

[57] Internet2 (website). `http://internet2.edu/`.

[58] Jacobson, V., Braden, R., and Borman, D. Tcp extensions for high performance. Internet RFC 1323, IETF, May 1992.

[59] Jain, M., and Dovrolis, C. Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In *Proc. of the 4th ACM SIGCOMM Conference on Internet Measurement (IMC 2004)* (Oct. 2004), pp. 272–277.

[60] Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., and Towsley, D. Inferring TCP connection characteristics through passive measurements. In *Proc. INFOCOM* (Mar. 2004), pp. 1582–1592.

[61] Jiang, X., and Xu, D. vBET: A VM-based emulation testbed. In *Proc. of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools)* (Aug. 2003).

[62] Johnson, D., Gebhardt, D., and Lepreau, J. Towards a high quality path-oriented network measurement and storage system. In *Proc. of the Ninth Passive and Active Measurement Conference (PAM)* (Apr. 2008).

[63] Johnson, D., Stack, T., Fish, R., Flickinger, D. M., Stoller, L., Ricci, R., and Lepreau, J. Mobile Emulab: A robotic wireless and sensor network testbed. In *Proc. IEEE INFOCOM 2006* (Apr. 2006).

[64] Johnson, D. S., Demers, A., Ullman, J. D., Garey, M. R., and Graham, R. L. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. on Computing 3*, 4 (1974), 299–325.

[65] Johnson, E., and Kunze, A. *IXP1200 Programming.* Intel Press, 2002.

[66] Judd, G., and Steenkiste, P. Repeatable and realistic wireless experimentation through physical emulation. In *Record of the 2nd Workshop on Hot Topics in Networks: HotNets-II* (Nov. 2003).

[67] Judd, G., and Steenkiste, P. Using emulation to understand and improve wireless networks and applications. In *Proc. of the Second Symposium on Networked Systems Design and Implementation (NSDI)* (May 2005), pp. 203–216.

[68] Juvan, M., and Mohar, B. Optimal linear labelings and eigenvalues of graphs. *Discrete Applied Mathematics 36*, 2 (1992), 153–168.

[69] Kamp, P.-H., and Watson, R. N. M. Jails: Confining the omnipotent root. In *Proc. 2nd Intl. SANE Conference* (May 2000).

[70] Kannan, R., Vempala, S., and Vetta, A. On clusterings: Good, bad and spectral. *J. of the ACM 51*, 3 (May 2004), 497–515.

[71] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. on Scientific Computing 20*, 1 (1998), 359–392.

[72] King-Wai Chu, Yuefan Deng, J. R. Parallel simulated annealing by mixing of states. *J. of Computational Physics 148* (1999), 646–662.

[73] Kirkpatrick, S., Gelatt, Jr., C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science 220*, 4598 (1983), 671–680.

[74] Kliewer, G. A general software library for parallel simulated annealing, 2000.

[75] Kotz, D., Newport, C., Gray, R. S., Liu, J., Yuan, Y., and Elliott, C. Experimental evaluation of wireless simulation assumptions. In *Proc. of the ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems MSWiM* (Oct. 2004).

[76] Krioukov, D., Fall, K., and Yang, X. Compact routing on Internet-like graphs. In *Proc. IEEE INFOCOM* (2004), pp. 209–219.

[77] Krishnamurthy, B., Madhyastha, H. V., and Spatscheck, O. ATMEN: A triggered network measurement infrastructure. In *Proc. of the 14th International Conf. on World Wide Web* (May 2005), pp. 499–509.

[78] Kumar, A., Rastogi, R., Silberschatz, A., and Yener, B. Algorithms for provisioning virtual private networks in the hose model. In *Proc. of SIGCOMM 2001* (August 2001).

[79] Lakhina, A., Crovella, M., and Diot, C. Mining anomalies using traffic feature distributions. In *Proc. of SIGCOMM 2005* (Aug. 2005), pp. 217–228.

[80] Lakshman, T. V., and Madhow, U. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking 5*, 3 (1997), 336–350.

[81] Lee, F. H. A. *Parallel Simulated Annealing on a Large Message-Passing Multicomputer.* PhD thesis, Utah State University, 1995.

[82] Lee, S.-J., Sharma, P., Banerjee, S., Basu, S., and Fonseca, R. Measuring bandwidth between PlanetLab nodes. In *Passive and Active Network Measurement: 6th International Workshop, (PAM)* (Mar.–Apr. 2005), pp. 292–305.

[83] Lischka, J., and Karl, H. A virtual network mapping algorithm based on subgraph isomorphism detection. In *Proc. of the first ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures (VISA)* (Aug. 2009).

[84] Liu, X., and Chien, A. Realistic large-scale online network simulation. In *Proc. Supercomputing* (Nov. 2004).

[85] MacDonald, J. E. Use of tabu search in a solver to map complex networks onto Emulab testbeds. Master's thesis, Air Force Institute of Technology, Mar. 2007. AFIT/GCE/ENG/07-07.

[86] Madhyastha, H. V., et al. iPlane: An information plane for distributed services. In *Proc. OSDI* (Nov. 2006), pp. 367–380.

[87] Mahadevan, P., Hubble, C., Huffaker, B., Krioukov, D., and Vahdat, A. Orbis: Rescaling degree correlations to generate annotated internet topologies. In *Proc. of SIGCOMM* (2007).

[88] Mahadevan, P., Krioukov, D., Fomenkov, M., Huffaker, B., Dimitripoulos, X., k. claffy, and Vahdat, A. The Internet AS-level topology: Three data sources and one definitive metric. In *SIGCOMM CCR* (Jan. 2006).

[89] Martin, O. C., and Otto, S. W. Combining simulated annealing with local search heuristics. *Annals of Operations Research 63* (1996), 57–75.

[90] M-lab. `http://www.measurementlab.net/`.

[91] Medina, A., Lakhina, A., Matta, I., and Byers, J. BRITE: An approach to universal topology generation. In *Proc. of MASCOTS 2001* (August 2001).

[92] Miyachi, T., ichi Chinen, K., and Shinoda, Y. Automatic configuration and execution of Internet experiments on an actual node-based testbed. In *Proc. of the International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)* (Feb. 2005).

[93] Mohar, B. The Laplacian spectrum of graphs. In *Graph theory, combinatorics, and applications* (New-York, 1991), Y. Alavi, G. Chartrand, O. Ollermann, and A. Schwenk, Eds., vol. 2, John Wiley and Sons, Inc., pp. 871–898.

[94] Monien, B., and Sudborough, H. *Embedding One Interconnection Network in Another.* Springer-Verlag/Wien, 1990, pp. 257–282. Computing Supplementum 7: Computational Graph Theory.

[95] Nakao, A., Peterson, L., and Bavier, A. A routing underlay for overlay networks. In *Proc. of SIGCOMM 2003* (Aug. 2003), pp. 11–18.

[96] NetFPGA (web site). http://netfpga.org/.

[97] Ng, T. S. E., and Zhang, H. Predicting Internet network distance with coordinates-based approaches. In *Proc. INFOCOM* (June 2002), pp. 170–179.

[98] NIST Internetworking Technology Group. NIST Net home page. http://www.antd.nist.gov/itg/nistnet/.

[99] Noble, B., Satyanarayanan, M., Nguyen, G. T., and Katz, R. H. Trace-based mobile network emulation. In *Proc. SIGCOMM* (Sept. 1997), pp. 51–61.

[100] The network simulator: ns-2 (web site). http://www.isi.edu/nsnam/ns/.

[101] The ns-3 network simulator (web site). http://www.nsnam.org/.

[102] The OpenVZ web site. http://openvz.org/.

[103] Oppenheimer, D., Albrecht, J., Patterson, D. A., and Vahdat, A. Distributed resource discovery on PlanetLab with SWORD. In *Proc. of the First Workshop on Real, Large Distributed Systems (WORLDS '04)* (Dec. 2004).

[104] Oppenheimer, D., Chun, B., Patterson, D., Snoeren, A. C., and Vahdat, A. Service placement in a shared wide-area platform. In *Proc. of the 2006 USENIX Annual Technical Conf.* (May–June 2006), pp. 273–288.

[105] Park, K., and Pai, V. CoMon: A mostly-scalable monitoring system for PlanetLab. *ACM SIGOPS Operating Systems Review 40*, 1 (Jan. 2006), 65–74.

[106] Peterson, L., Anderson, T., Culler, D., and Roscoe, T. A blueprint for introducing disruptive technology into the Internet. In *Proc. of HotNets-I* (Oct. 2002).

[107] Peterson, L., and Wroclawski, eds., J. Overview of the GENI architecture. GENI Design Document GDD–06–11, GENI Planning Group, Jan. 2007. Draft. http://geni.net/GDD/GDD-06-11.pdf.

[108] ProtoGENI (web site). http://www.protogeni.net/.

[109] Rao, S., and Richa, A. W. New approximation techniques for some ordering problems. In *Proc. of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (1998), pp. 211–218.

[110] Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., and Yu, H. OpenDHT: A public DHT service and its uses. In *Proc. of ACM SIGCOMM* (Aug. 2005).

[111] Ricci, R., Alfeld, C., and Lepreau, J. A solver for the network testbed mapping problem. *ACM SIGCOMM Computer Communications Review 33*, 2 (Apr. 2003), 65–81.

[112] Ricci, R., Duerig, J., Sanaga, P., Gebhardt, D., Hibler, M., Atkinson, K., Zhang, J., Kasera, S., and Lepreau, J. The Flexlab approach to realistic evaluation of networked systems. In *Proc. of the Fourth USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Apr. 2007), pp. 201–214.

[113] Riley, G. F., Ammar, M. H., and Fujimoto, R. Stateless routing in network simulations. In *MASCOTS 2000* (2000), pp. 524–531.

[114] Riley, G. F., Fujimoto, R., and Ammar, M. H. A generic framework for parallelization of network simulations. In *Proc. of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (1999).

[115] Riley, G. F., and Reddy, D. Simulating realistic packet routing without routing protocols. In *19th Workshop on Principles of Advanced and Distributed Simulation (PADS'05)* (2005), IEEE, pp. 151–158.

[116] Rizzo, L. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Computer Communication Review 27*, 1 (Jan. 1997), 31–41.

[117] The RON/IRIS testbed (website). `http://www.datapository.net/tb/`.

[118] Sanaga, P., Duerig, J., Ricci, R., and Lepreau, J. Modeling and emulation of Internet paths. In *Proc. of NSDI* (Apr. 2009).

[119] Soltesz, S., Pötzl, H., Fiuczynski, M., Bavier, A., and Peterson, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. of EuroSys* (Mar. 2007).

[120] Sommers, J., Barford, P., Duffield, N., and Ron, A. Improving accuracy in end-to-end packet loss measurement. In *Proc. of SIGCOMM 2005* (Aug. 2005), pp. 157–168.

[121] Spring, N., Mahajan, R., and Wetherall, D. Measuring ISP topologies with Rocketfuel. In *Proc. of SIGCOMM 2002* (Aug. 2002), pp. 133–145.

[122] Spring, N., Peterson, L., Pai, V., and Bavier, A. Using PlanetLab for network research: Myths, realities, and best practices. *ACM SIGOPS Operating Systems Review 40*, 1 (Jan. 2006), 17–24.

[123] Spring, N., Wetherall, D., and Anderson, T. Scriptroute: A public Internet measurement facility. In *Proc. of USENIX USITS* (2003).

[124] Taylor, W. A. Change-point analysis: A powerful new tool for detecting changes. `http://www.variation.com/cpa/tech/changepoint.html`, Feb. 2000.

[125] Thorup, M., and Zwick, U. Compact routing schemes. In *ACM Symposium on Parallel Algorithms and Architectures* (2001), pp. 1–10.

[126] Touch, J. Dynamic Internet overlay deployment and management using the X-Bone. *Computer Networks* (July 2001), 117–135.

[127] Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., and Becker, D. Scalability and accuracy in a large-scale network emulator. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Dec. 2002), pp. 271–284.

[128] van Laarhoven, P. J. M. *Theoretical and Computational Aspects of Simulated Annealing.* Centrum voor Wiskunde en Informatica, 1988.

[129] van Laarhoven, P. J. M., and Aarts, E. H. L. *Simulated Annealing: Theory and Applications.* D. Reidel, 1987.

[130] van Leeuwen, J., and Tan, R. Interval routing. *The Computer Journal 30* (1987), 298–307.

[131] Vishwanath, K., and Vahdat, A. Realistic and responsive network traffic generation. In *Proc. of SIGCOMM 2006* (Sept. 2006).

[132] Vishwanath, K., and Vahdat, A. Evaluating distributed systems: Does background traffic matter? In *Proc. of the USENIX Annual Technical Conference* (June 2008).

[133] VMware, Inc. VMware: A virtual computing environment (web site). `http://www.vmware.com/`.

[134] Wang, L., Park, K., Pang, R., Pai, V., and Peterson, L. Reliability and security in the CoDeeN content distribution network. In *Proc. of the 2004 USENIX Annual Technical Conf.* (June–July 2004), pp. 171–184.

[135] Webb, K., Hibler, M., Ricci, R., Clements, A., and Lepreau, J. Implementing the Emulab-PlanetLab portal: Experience and lessons learned. In *Proc. First Workshop on Real, Large Distributed Systems* (Dec. 2004).

[136] Wei, Y., and Cheng, C. Ratio cut partitioning for hierarchical designs. *IEEE Trans. on Computer-Aided Design 10*, 7 (July 1997), 911–921.

[137] Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., and Welsh, M. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2006).

[138] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Dec. 2002), pp. 255–270.

[139] Winick, J., and Jamin, S. Inet-3.0: Internet topology generator. Tech Report CSE–TR–456–02, University of Michigan, 2002.

[140] Xu, K., Zhang, Z.-L., and Bhattacharyya, S. Profiling Internet backbone traffic: Behavior models and applications. In *Proc. of SIGCOMM 2005* (Aug. 2005), pp. 169–180.

[141] Yalagandula, P., Sharma, P., Banerjee, S., Lee, S.-J., and Basu, S. S3: A scalable sensing service for monitoring large networked systems. In *Proc. SIGCOMM Workshop on Internet Network Management (INM)* (Sept. 2006), pp. 71–76.

[142] Yu, M., Yi, Y., Rexford, J., and Chiang, M. Rethinking virtual network embedding: Substrate support for path splitting and migration. *ACM SIGCOMM Computer Communications Review 38*, 2 (Apr. 2008), 19–29.

[143] Zegura, E., Calvert, K., and Bhattacharjee, S. How to model an internetwork. In *Proc. of IEEE INFOCOM* (Mar. 1996), pp. 594–602.

[144] Zeng, X., Bagrodia, R., and Gerla, M. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proc. of the 12th Workshop on Parallel and Distributed Simulations (PADS)* (May 1998).

[145] Zhang, M., Zhang, C., Pai, V., Peterson, L., and Wang, R. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *Proc. of the Sixth Symposium on Operating Systems Design and Implementation* (Dec. 2004), pp. 167–182.

[146] Zhang, Y., Du, N., Paxson, V., and Shenker, S. On the constancy of internet path properties. In *Proc. SIGCOMM Internet Meas. Workshop (IMW)* (Nov. 2001), pp. 197–211.

[147] Zhang, Y., Paxson, V., and Shenker, S. The stationarity of Internet path properties: Routing, loss, and throughput. Tech. rep., ACIRI, May 2000.