A Solver for the Network Testbed Mapping Problem

University of Utah Flux Group Technical Note 2002-05

December 2002

Revised version to appear in ACM SIGCOMM Computer Communications Review (CCR), issue dated April 2003.

Robert Ricci
School of Computing
University of Utah
ricci@cs.utah.edu

Chris Alfeld *
Mathematics Department
University of Wisconsin-Madison
alfeld@math.wisc.edu

Jay Lepreau
School of Computing
University of Utah
lepreau@cs.utah.edu

Network experimentation of many types require the ability to map virtual resources requested by an experimenter onto available physical resources. These resources include hosts, 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 must be conserved. In the face of these constraints, this mapping becomes an NP-hard problem. Yet, in order to prevent mapping time from becoming a serious hinderance to such experimentation, this process cannot consume an excessive amount of time.

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

1 Introduction

Network experimentation on real hardware requires a mapping from the virtual resources an experimenter requests to available physical resources. This problem arises in a wide range of experimental environments, from network emulation to distributed simulation. This mapping, however, is difficult, as it must take into account scarce physical resources such as bandwidth over network bottlenecks. Poor mapping can degrade performance or introduce artifacts into an experiment. We call this problem the "network testbed mapping problem."

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; and
- Produce a solver that is able to find near-optimal solutions in a modest amount of time.

In pursuit of these goals, this paper makes the following contributions:

- It defines the network testbed mapping problem, and examines the challenges that make it interesting.
- It describes our solver for this problem.
- It presents lessons from our solver's implementation and its use on Emulab [19], a production network testbed.
- It identifies open issues for future work.

In Section 2 we present the domain of the network testbed mapping problem by describing the environments in which it arises, and why good mapping is important for each. In Section 3, we discuss the challenges at the core of the network testbed mapping problem, and present abstractions which can be generalized across our target environments. Next, Section 4 covers the design and implementation of our solver for this problem, called `assign`, and the lessons we learned in this process. Section 5 presents an evaluation of its performance. Finally, Section 6 covers related work, Section 7 discusses directions for future work, and Section 8 concludes.

2 Environment and Motivation

In this section, we describe some of the environments to which the network testbed mapping problem is relevant, and identify the characteristics of these environments that make good mapping necessary, but difficult.

* Work done while at the University of Utah

2.1 Netbed and Emulab

Netbed [19] is a shared public facility for research and education in networking and distributed systems. Versions of it have been in use since April 2000. One of its goals is to transparently integrate a variety of different experimental environments. Currently, Netbed supports three such environments: emulation, simulation, and live-Internet experimentation. Netbed is descended from, and incorporates, Emulab, a time- and space-shared "cluster testbed" whose main goals are to provide artifact-free network emulation for arbitrary experiments, while making that as easy and quick as simulation. Emulab manages a cluster of commodity PC "nodes" with configurable network interconnectivity. The facility is space-shared in the sense that 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 that sense, Emulab is also time-shared.

To run an experiment on Emulab, an experimenter submits a network topology, which is mapped onto available physical resources. This virtual topology can include links and LANs, with associated characteristics such as bandwidth, latency, and packet loss. Limiting and shaping the traffic on a link, if requested, is done by interposing "delay nodes" between the endpoints of the link. Specifications for hardware and software resources can also be included for nodes in the virtual topology.

Once 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. Currently, the Emulab portion of Netbed contains 168 PCs connected, via four interfaces each, to three switches. In general, large scale emulators require multiple switches, because the number of ports an each switch is limited. Our switches are connected via inter-switch links; at the present time, these links are 2Gbps. Since multiple experimenters or very large experiments may be using these links, they become a bottleneck, and overcommitting them could lead to artifacts in experimental results. Emulab aims to create an environment suitable for running real applications without introducing artifacts due to its space-shared nature. Thus, conservative resource allocation is a guiding principle.

The mapping algorithm has a number of simultaneous goals. First, it must economize inter-switch bandwidth by minimizing the total bandwidth of virtual links mapped across physical inter-switch links. 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 requested by the experimenter. Finally, this mapping must be done quickly. Current experiment creation times in Emulab range from three minutes for a single-node topology, to six and a half minutes for an 80-node topology, though we hope to decrease this time dramatically in the future. 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.

2.2 Simulation: Integrated and Distributed

Netbed integrates simulation with the emulation system described above. It uses *nse* [5] to allow the popular *ns* [14] network simulator to generate and interact with live traffic. This also allows packets 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 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 [4]. However, communication between the machines can become a bottleneck, so a "good" mapping of simulated nodes onto physical nodes is important to overall performance. PDNS [13], a parallelized and distributed version of *ns*, is an example of such a distributed simulator. However, except for certain restricted tree topologies, PDNS requires manual partitioning onto physical machines.

2.3 ModelNet

Mapping issues also arise in ModelNet [15], a large-scale network emulator which aims at accurate emulation of the Internet core through simulating 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 ensure that not too many queues are simulated on a single core node, and that the links passing between core nodes are minimized.

Some aspects of ModelNet mapping are different from those outlined above for Emulab. A major difference is that ModelNet does not do conservative resource allocation. 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. Their goal is to load-balance among these resources, rather than use the least number. ModelNet also has some constraints in acceptable solutions, introduced by IP routing semantics, that must be taken account when mapping.

We plan to integrate ModelNet into Netbed as another emulation mechanism; for this to be seamless, mapping will have to take into account both environments' goals and virtual resources.

2.4 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 we integrated other network experimentation mechanisms—geographically distributed nodes, simulated nodes, and soon ModelNet—to form the general Netbed platform. We immediately faced the mapping issue in each of them.

In the wide-area case we chose to develop a separate genetic algorithm-based solver [19], outlined in Section 7. This was partly due to the degree to which the wide-area problem differed from the Emulab problem, and partly due to the exigencies of software development.

However, the simulated and ModelNet environments are more similar in their mapping needs to Emulab. For example, minimizing inter-switch 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 recently 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 paper, 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 accomodated. Indeed, as outlined in Section 7, with modest work our general solver might handle the wide-area case, should that be desirable.

3 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 needs of the virtual nodes are met.
- Map virtual links to physical links, minimizing the use of bottleneck links 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.

3.1 Network Links

At the core of 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. This problem, however, is NP-hard, by reduction to the multiway separator problem [2].

Figure 1 shows a trivial example of this problem. The virtual topology on the left is to be mapped onto

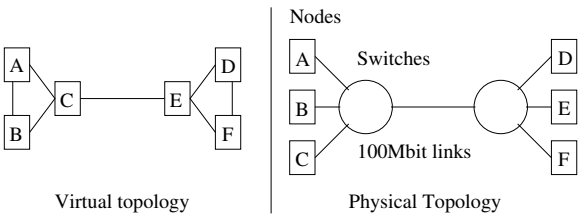

Figure 1: A trivial six-node mapping problem

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

Figure 2: Samples 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 3: Sample nodes in a physical topology

the physical topology shown to its right. The bandwidths of all virtual and physical links in this example are 100Mbps. 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.

At present, `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. Intra-switch links are those that can be satisfied on a single switch. Inter-switch links must cross between switches. Intra-node links connect nodes run on the same hardware; these links do not need to traverse any network hardware at all, and are used to represent flows in distributed simulation or ModelNet that remain on one machine.

When mapping topologies to physical resources, the key limitation is that switch 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 intra-switch links which traverse only that switch's backplane. (This backplane, by design in Emulab, has sufficient bandwidth to handle all nodes connected to it, and can thus be considered to have infinite resources.)

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 "inter-switch 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 inter-switch links, and use intra-switch links instead, wherever possible. As Emulab is a space-shared facility it is important that inter-switch traffic be minimized, rather than simply not oversubscribed. By minimizing such traffic, maximum capacity for future experiments is preserved.

This problem of minimizing inter-switch 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.

3.2 Node Types

A facility like Emulab will generally have distinct sets of nodes with identical hardware. Emulab, for example, has 40 600-MHz PCs, and 128 850-MHz PCs. 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. 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. Of course, experimenters who do not have such requirements should not be burdened with this specification.

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, or ModelNet. One restriction is invariant, however: 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 2 and a physical topology in Figure 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 only be satisfied by pc1 or pc2. In Emulab, 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. 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.

Types themselves are opaque to assign—there are only two types that are treated specially: switch, which is necessary to support inter-switch links, and lan, which will be discussed in Section 4.2. 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.

3.3 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.

$vclasses$ allow, in the virtual topology, the declaration of a virtual equivalence class, along with a list of types that can be used to fulfill the $vclass$. (The latter 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. This is useful in circumstances where, for example, the experimenters wants a set of client machines and a set of servers, each of which can be its own class.

$vclasses$ 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.

3.4 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 also 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 lower 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.

3.5 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. This mapping is made at the start of a run, and is not changed during the algorithm. The biggest benefits from this feature are the ability to use assign for in-

cremental topology enlargement and for replacement of nodes in existing topologies.

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 can be run through the mapping algorithm 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 can be 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.

4 Design, Implementation, and Lessons

Our implementation of a solver for the testbed mapping problem, `assign`, is written in 4,800 lines of C++ code. It uses the Boost Graph Library [3] 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 the design goals of creating a mapper that is able to find near-optimal solutions in a modest amount of time. For `assign`, we have chosen to use simulated annealing.

Simulated annealing [8] 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, this may not be 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 or not, and $pclasses$, which are equivalence classes used to dramatically reduce the search space.

4.1 Initial Configuration

Typically, simulated annealing is started with a randomly-generated configuration [8]. However, `assign` uses a different strategy. Our concept of violations allows us to begin with an empty configuration—one in which no virtual nodes are assigned to physical nodes. 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 type information is taken into account.

4.2 Cost Function

`assign`'s cost function must score a configuration and return a number that indicates how "good," in terms of the goals laid out in Section 2, 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`.

`assign`'s scoring function is split into three parts: $init_score$ initializes the cost for an empty configuration, and computes violations that result from not having any 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 trickier by the fact that other mappings may have been

Physical Resource	Cost
Intra-node Link	0.00
Direct Link	0.01
Intra-switch Link	0.02
Inter-switch Link	0.20
Physical Node	0.20
Switch	0.50
$pclass$	0.50

Table 1: Scores used in `assign`

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 for 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 both have direct links to each other, and inter-switch links. Our experience, however, is that such topologies do not tend to occur in practice. So, while `assign` supports these topologies, it does not include the additional code and time complexity to treat them as first-class entities. Instead, if both intra-switch and inter-switch links are present between a set of nodes, `assign` chooses between them randomly.

To resolve a link, `assign` finds all possible links between the nodes (direct, intra-switch, and inter-switch) and chooses one. Direct links are used first, if they exist, followed by intra-switch and inter-switch links. To find inter-switch 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 1, can be overridden by by passing them to `assign` on the command line. Intra-node 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 intra-switch links. Inter-switch 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 $pclasses$ that the chosen

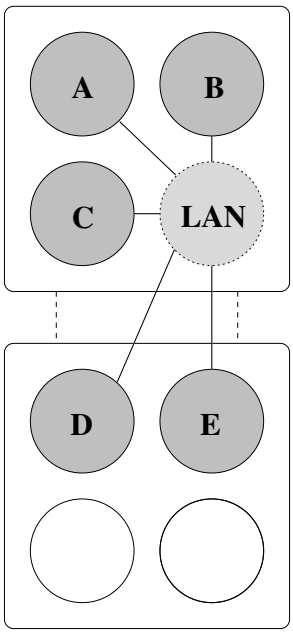

Figure 4: Scoring for LANs is done with a "LAN node," which LAN members have links to. This LAN uses 3 intra-switch links and 2 inter-switch links.

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 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 4, which is used solely for the purpose of assessing scoring penalties. LAN nodes only exist in the virtual topology—since they do not correspond to a real resource, they are not included in the input physical topology. As needed, LAN nodes are dynamically bound to switches in the physical topology, each is attached to the same switch as the majority of its members. Thus, any LAN member that is on another switch will be assessed an inter-switch link penalty. Clearly, then, when LAN members are reassigned, this must be re-calculated, and the LAN node may need to be "migrated" to a new switch, which includes re-scoring all links to it. Doing so is a heavyweight operation, and the time taken can add more than a factor of three to the runtime for LAN-heavy topologies. Instead, we perform migration only occasionally: when the LAN node is selected for re-mapping by the

generation function, and at the end of every temperature step. In practice, we find that this greatly reduces runtime, and has acceptable effects on the solutions found by `assign`.

4.3 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 [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 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 a feature known as "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.

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.

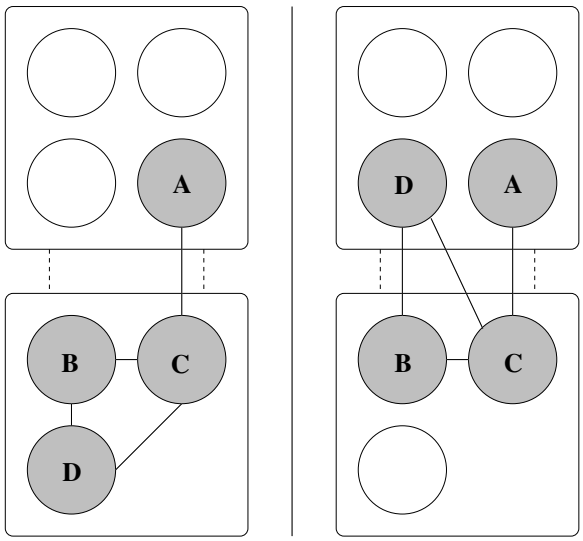

Figure 5: A situation in which allowing solutions with violations helps reach the optimal solution. If the bandwidth between switches is such that only one virtual link can cross between them, the mapping shown on the right is in violation of this constraint. However, it is a necessary intermediate step between the mapping on the left and the optimal mapping, which places all nodes on the upper switch.

One important side effect of violations is that they provide the user of the program with feedback about why a mapping has failed. Six different types of violations are tracked, ranging from overuse of inter-switch 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 re-submit their virtual topology.

4.4 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 non-empty, it picks a member

of the list 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 this is not possible (i.e., there are no free nodes that the virtual node can be mapped to), then it chooses a random physical node, to which the chosen virtual node can be successfully mapped, and unmaps it.

Our generation function avoids certain invalid solutions. When `assign` begins, it pre-computes a list of acceptable assignments for each virtual node. An acceptable assignment is one that is capable of fulfilling the type of the virtual node, and has at least enough physical links to satisfy the virtual node's requirements.

We have discovered that this behavior is important when dealing with scarce resources. For example, in physical topologies containing a few nodes with many network interfaces, any virtual node can be assigned to these nodes, even those that do not have many links, and thus need few interfaces. If the virtual topology contains a node that requires this large number of interfaces, it has to compete for the appropriate physical node against other virtual nodes which do not require so many interfaces. If the generation function is allowed to assign high-degree virtual nodes to low-degree physical nodes (which, of course, introduces a violation), this competition can prevent `assign` from finding a valid solution for large topologies. Thus, we have found that it is critical to avoid violations in node assignment, to cope with scarce resources.

Though some types of violations are useful to explore, as covered in Section 4.3, these are not, because they will not be removed by future assignments; though a future choice may cause an overburdened link to be reduced to an acceptable level of traffic, future choices will not change the type, capacity, or number of interfaces of a physical node. Thus, by not exploring these areas of the solution space, we save on work that we can be sure will not lead to a viable solution.

4.5 Physical Equivalence Classes

4.5.1 Reducing the Solution Space

One of the features of `assign` that has most improved its runtime and quality of solutions in 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 scoring a configuration, 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 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, the number of free nodes in p will tend to be $p - v$ as most or all virtual nodes will be mapped, removing the the physical node from the free pool. 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.

In practice, it is more straightforward, and provides greater benefit, 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 4 $pclasses$, resulting in a branching factor two orders of magnitude smaller. Attempting to reduce v, on the other hand, will generally not lead to such drastic results, since experimenters' topologies are much more heterogenous, and attempting to find symmetries in them would require relatively complicated and computationally expensive graph isomorphism problems.

4.5.2 *pclasses*

In order to effect this reduction in the physical topology, `assign` defines an equivalence relation. Any equivalence relation on a set will partition that set into disjoint subsets in which all members of a subset are equivalent (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 in invoked, rather than choosing a physical node from among a large set of potentially equivalent nodes, it instead selects a $pclass$, and a node is chosen from that $pclass$ and used. This technique reduces the size of the search space dramatically, without degrading the the results of the scoring function; it remains unchanged.

$pclasses$ have an interesting effect on the way that the solution space is explored. To illustrate this issue, consider an extreme case: n identical nodes connected to one switch, A, and n identical nodes each connected to their own switches, $B_{1..n}$. Now, consider a virtual topology with v nodes, where $v < n$. Clearly, the ideal solution is to map all virtual nodes onto physical nodes

connected to A. Without pclasses, when the generation function selects from among all nodes, it has approximately a $\frac{1}{2}$ chance of selecting one of these ideal nodes. With $pclasses$ however, this drops to $\frac{1}{n+1}$. If n is large, this makes it more difficult for `assign` to find a good mapping, as the probability of finding a "good" node is lower. On the converse side, this increases the chance of selecting a scarce type of node, which may improve the mapping, from $\frac{1}{2n+1}$ to $\frac{1}{n+1}$. This circumstance is not likely to occur in practice, so it does not greatly decrease the utility of $pclasses$. However, it is still an issue that warrants further study.

There are some circumstances, however, when $pclasses$ are not appropriate. When mapping large numbers of virtual nodes onto a small set of physical nodes, 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 intra-node links or not. As a result, when mapping simulated or ModelNet topologies, we disable $pclasses$. Fortunately, these mappings tend to involve smaller numbers of physical nodes than full Emulab-style mappings, and are thus still able to complete in reasonable time.

4.6 Cooling Schedule

By default, `assign` uses the polynomial-time cooling schedule described in [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 4.5) 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 should scale linearly in two dimensions: the number of virtual nodes, and the number of $pclasses$. The temperature decrement function and termination condition, however, will depend 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 Evaluation

In this section, we evaluate the performance of `assign`. First, we consider the performance of `assign` on a real workload—a set of 3,113 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. Then, we examine `assign`'s ability to map simulated and ModelNet topologies. Finally, we compare `assign` to a re-implementation of a mapper using 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 still 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.

5.1 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, and the physi-

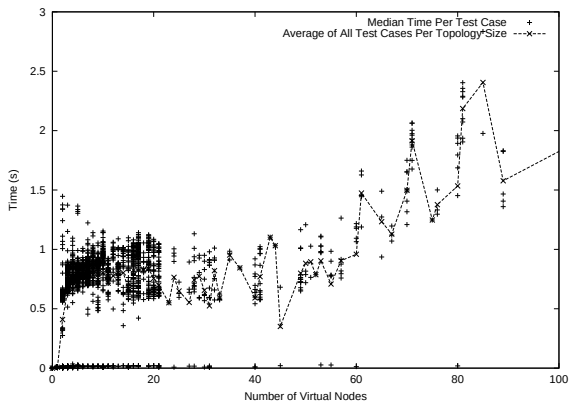

Figure 6: Runtimes for Emulab topologies. Each test case was run 10 times. The scatter-plot shows the median runtime for each test case. The line shows the average across all topologies of the same size.

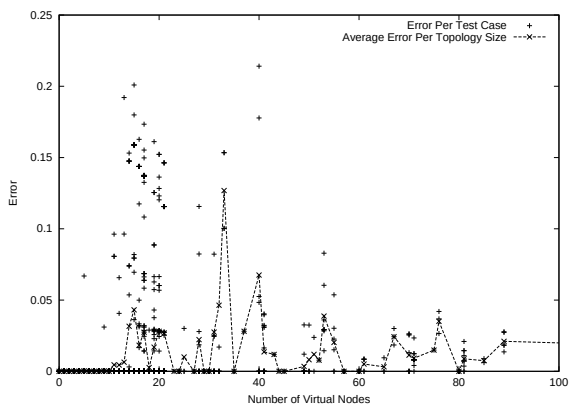

Figure 7: Error for Emulab topologies.

cal topology available at the time the experiment was submitted. Since virtual topologies vary widely, along with available physical resources, 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 6 shows runtimes for the test cases. This graph shows three important things. First, the majority of experiments run on Emulab, and thus, the typical workload for `assign`, consists 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` will run until it determines that it is done. Finally, we can see that `assign` always completes quickly for its historical workload, in less than 2.5 seconds.

Figure 7 shows the amount of error for the same test

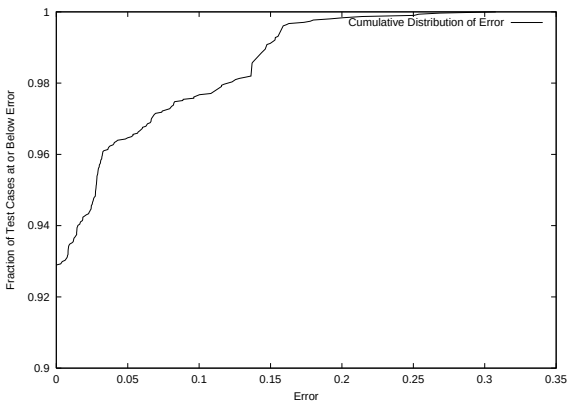

Figure 8: CDF of error on on Emulab topologies. The line represents how many topologies had an error of a given value or smaller. Note that the y-axis for this graph begins at .90.

cases, which were each 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 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 aim to show that `assign` will scale to larger Emulab-like facilities, in addition to being general enough for other environments.

5.1.1 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, with virtual topologies consisting of Emulab's historical data of over 3000 experiments. In each, 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. I.e., the random version still observes physical link limits, experimenters' constraints on node types, etc.

For the first test, we measured simulated 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 inter-switch 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.[1] Mapping using `assign` processed the queue in 194 virtual days, while random mapping took 604 days, a factor of 3.1 longer. 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 inter-switch bandwidth as our metric. First, we altered the physical topology to show infinite bandwidth between switches. As above, we first generated a randomly-ordered work queue, then removed and mapped experiments until one failed to map by exceeding the number of available nodes. We recorded bandwidth consumption on the inter-switch links. To prepare for the next iteration, we emptied the testbed and re-shuffled the queue. The result, after 30 iterations, was that `assign`-based mapping used an average of 0.28Gbps across both links, while random mapping used 7.4Gpbs, a factor of 26 higher.[2]

To gain further insight into `assign`'s value, we are writing a "simple mapper" against which to compare. It will use a simple heuristic such as flood-filling the switches.

5.2 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 [11], 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 lower 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.

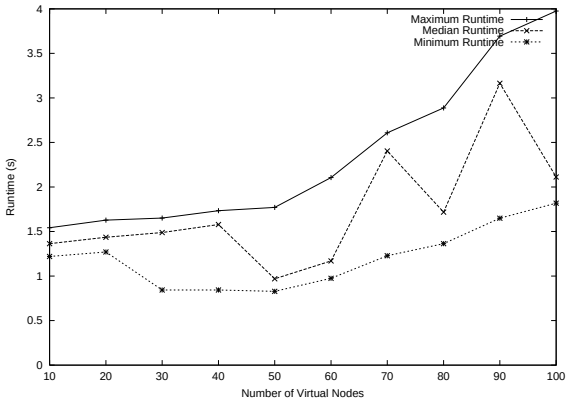

Figure 9: Runtimes for the brite100 test set

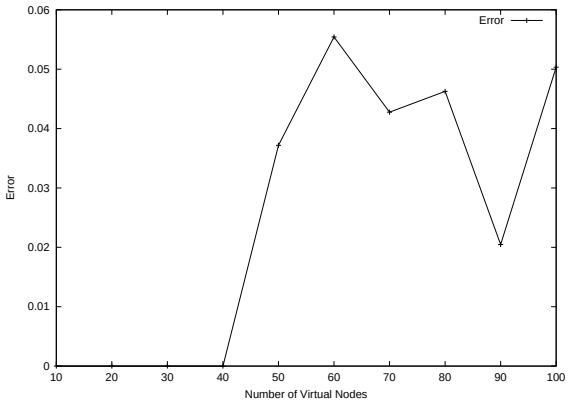

Figure 10: Solution quality for the brite100 test set

5.2.1 Scaling

Figure 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.

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

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

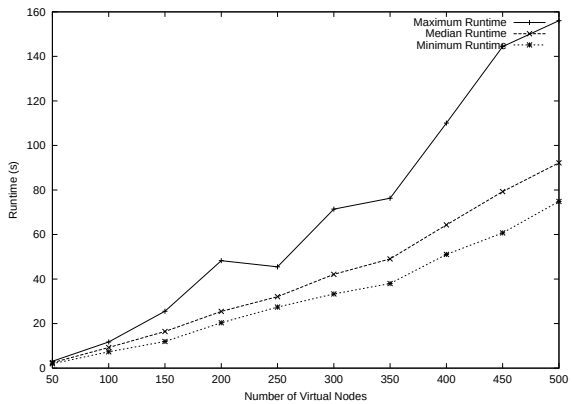

Figure 11: Runtimes for the brite500 test set

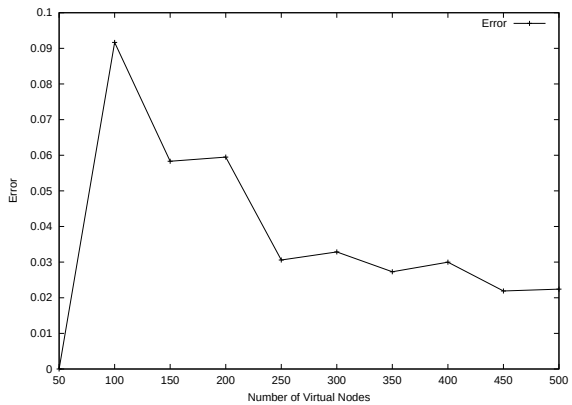

Figure 12: Solution quality for the brite500 test set

Figure 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 11 and 12 show, respectively, the runtimes and waste 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.

5.2.2 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 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.

Error in the solution found went down significantly due to the longer runtimes, as shown in Figure 14. The

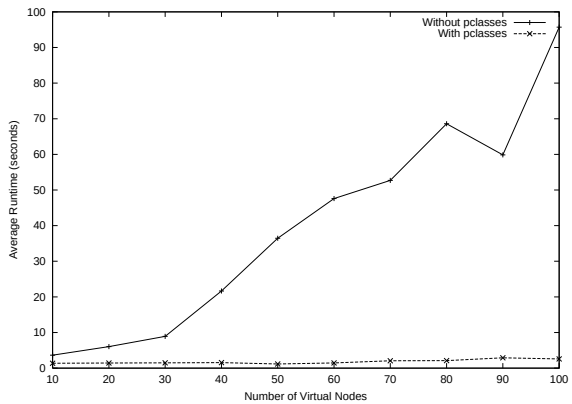

Figure 13: Runtimes for the brite100 test with and without pclasses

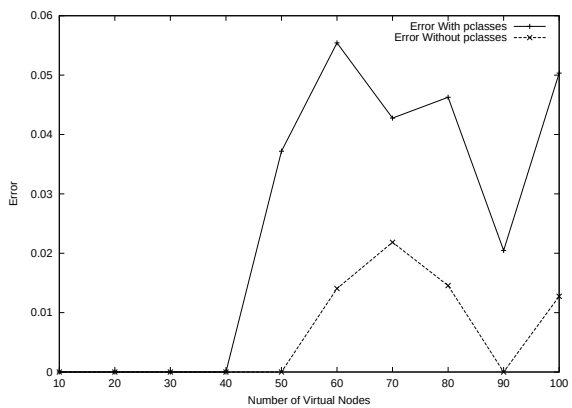

Figure 14: Solution quality for the brite100 test with and without pclasses

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.

5.2.3 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 inter-switch link. This feature was given to all nodes on one of the three switches, so it does not introduce additional $pclasses$, which would have lengthened the runtime.

We found that, in all runs, `assign` properly avoided using `undesirable` nodes. Up to 80, the number

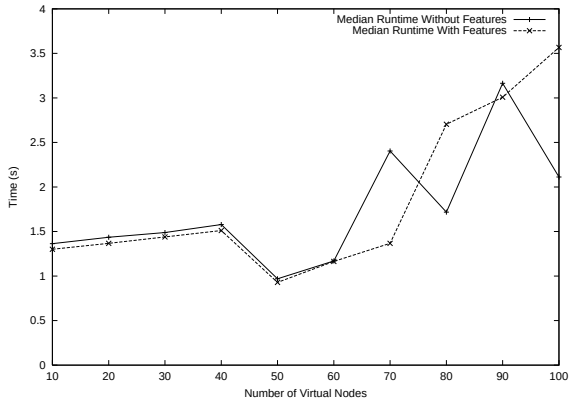

Figure 15: Runtimes for the brite100 test set when avoiding undesirable features

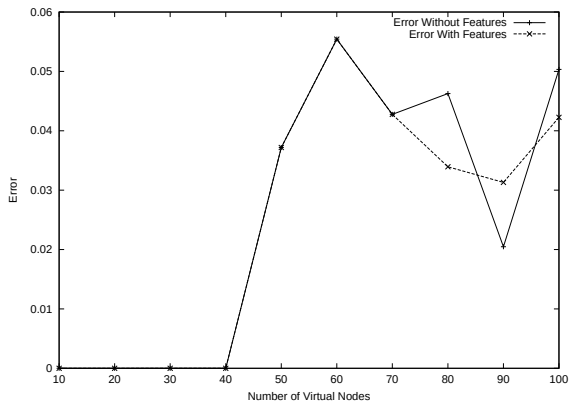

Figure 16: Solution quality for the brite100 test set when avoiding undesirable features

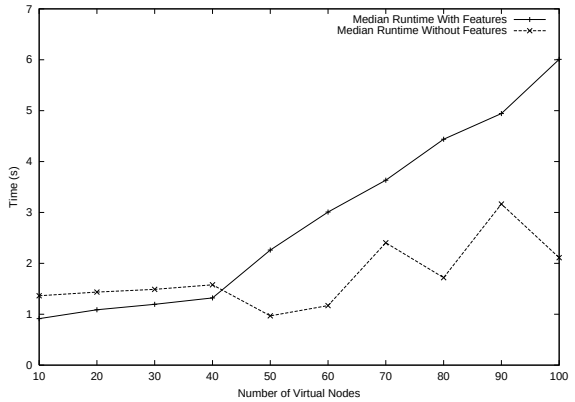

Figure 17: Runtimes for the brite100 test set, when attempting to satisfy desires

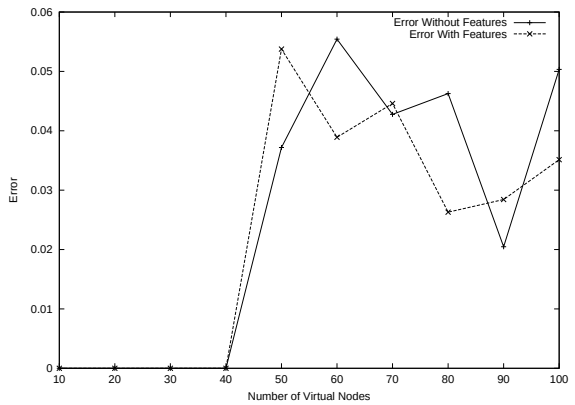

Figure 18: Solution quality for the brite100 test set, when attempting to satisfy desires

of non-`undesirable` nodes in the physical topology, `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 15 shows runtimes for this test. As we can see, features used in this manner do not adversely affect runtime. Figure 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 the nodes with feature `B`.

Figures 17 and 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 size 30 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 a role in the optimal solution. This weight places the feature as being more valuable than two inter-switch links, but less valuable than three. Thus, depending on the virtual topology, it may be desirable for `assign` to conserve inter-switch links rather than these nodes. Table 2 shows the number of nodes with feature `B` in the minimally-scored solution,

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

Table 2: assign's performance in avoiding feature B

and the median number chosen. If we placed more value on feature B, we could give it a higher weight, so that its cost is higher than a larger number of inter-switch links.

5.3 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 PDNS. 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 to use only 56 links between nodes. 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 make it unable to accept arbitrary network partitions, such as those generated by assign. We are working to eliminate these restrictions, so that we can perform 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 to be multiplexed 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 [16], this is likely best attacked by improving the generation function.

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

Table 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.

5.4 ModelNet

To apply assign to mapping ModelNet, we developed tools to convert ModelNet's topology representation into assign's. We then mapped the topology used in [15] 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 155Mbps, transit-stub links have a bandwidth of 45Mbps, and stub-stub links are 100Mbps. The results of mapping this topology to differing numbers of core nodes is shown in Table 3. Though the error is significantly higher than for the Emulab topologies that assign has been tuned for, the average bandwidth to each core node stays near 1000Mbps, which is the speed of the core nodes' links.

The ModelNet 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 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

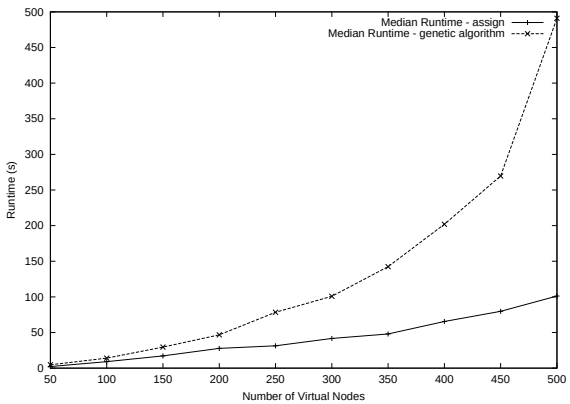

Figure 19: Runtimes for the brite500 test set for `assign` and our genetic algorithm

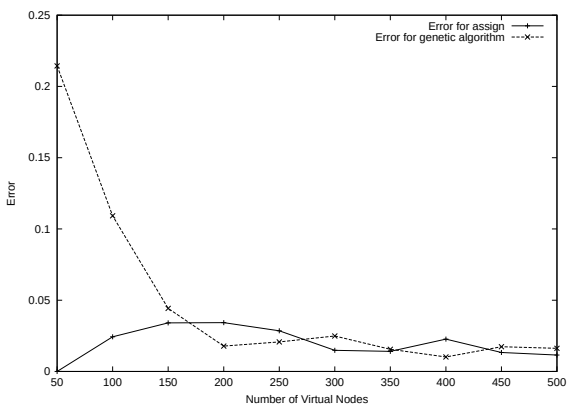

Figure 20: Solution quality for the brite500 test set for `assign` and our genetic algorithm

comparison between the solutions from `assign` and ModelNet's mapper is problematic. The best way, which we are pursuing, would be to run both mappers and the resulting emulations, and compare the details of their performance and behavior.

5.5 Comparison to Genetic Algorithm

Finally, we compared our simulated annealing approach to the testbed mapping problem to another general-purpose and randomized heuristic approach, a genetic algorithm (GA) [6]. 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 numbers of nodes, where it was worse, the quality of solutions found by the genetic algorithm, shown in Figure 20, is close to simulated annealing's. Performance is a different story. For the brite100 topologies (not shown), the GA was faster when mapping 40 or fewer virtual nodes. However, as shown in Figure 19, the GA exhibited much worse scalability than simulated annealing; for all of the brite500 test cases, the GA was slower, on average. In the most complex topology tested, simulated annealing far outperformed the genetic algorithm.

The key reason for this disparity in performance appears to be incremental scoring, which cannot be done in GA's 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 reevaluated. 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 superlinear scaling as both the cost of evaluations and the number of evaluations required increase.

6 Related Work

Simulated annealing was first proposed for use in VLSI design [8]. Much literature is available on aspects of it [1, 18, 17]. 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 inter-chip dependencies, which require communication over expensive pins and busses. In this way, this problem is similar to ours, but does not have the unique challenges described in Section 3. Simulated annealing is also used in combinatorial optimization in various Operations Research fields.

Similar partitioning problems arise on parallel multiprocessor computers [7]. Some network mapping algorithms can also be found in the literature. For example, [4] discusses partitioning of distributed simulation using simulated annealing. [9] discusses algorithms for network resources when providing bandwidth guarantees for VPNs. None of these, however, meet our goal of being more generally applicable across a range of experimentation environments.

We know of ongoing work on ModelNet's mapper, with the goal of allowing re-mapping in real-time based on observed network load. It seems likely that this work will be complementary to ours, and that some of the lessons learned in each mapper will be applicable to the other.

7 Future Work

7.1 Wide-Area Assignment

As network testbeds expand into the wide-area, such as Netbed's wide-area nodes [19] and PlanetLab [12], 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 describe in this paper.

Netbed 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 each set of resources in each phase does not depend on choices made in the other phase. However, we plan to investigate whether it appropriate to use the same program, or at least, the same approach, for both phases.

Another potential approach to wide-area mapping is the simplification of the problem into mapping "last-mile" characteristics of network links. For some types of network experimentation, the primary concern is whether, for example, a node is connected to a DSL line, a cable modem, or Internet2. Though it fails to capture all of the global behavior characteristics of the node, this approach makes mapping considerably easier, and eases the specification burden on the experimenter. Netbed currently supports a crude form of this type of mapping using its type system, but more work is required to determine the proper specification and use of this style of mapping.

7.2 Resource Descriptions

One potential avenue for further work on `assign` is the introduction of arbitrary resource descriptions for virtual and physical nodes. For example, a given virtual node may specify that it will require approximately X amount of memory and Y amount of processor cycles per second. When multiplexing onto a physical node the resource requirements of the assigned virtual nodes would be subtracted from the resources available on the physical node.

Our current method for representing such things involves our type system. For example, we may determine empirically how many simulated nodes can be handled on a physical node, to get a "packing factor" X. Then, we declare the simulated virtual nodes to be of type `sim`, and allow physical nodes to satisfy X `sim` nodes. This works reasonably well, but can make sub-optimal choices, since all simulated nodes must be assumed to consume the same resources. Alternately, simulated nodes can be classified by their resource consumption, say into "heavyweight" and "lightweight" nodes, but these cannot be mixed on a single physical node, since a physical node is only permitted to act as one type at a time.

The main problem with this modification will be in our generation function. Currently we are able to avoid certain types of violations, such as multiplexing too many virtual nodes on to a physical node, with minimal processing cost. This is simple, because the type system can know that no virtual node consumes more than a single "slot" on a physical node. With arbitrary resource costs on virtual nodes, however, maintaining a structure that allows us to efficiently find a physical node into which a given virtual node "fits" becomes much more complicated. This could make our generation function slower, or reduce the quality of solutions, and more time is spent exploring invalid solutions. In essence this adds a bin-packing aspect, another NP-complete problem, to an already complicated solution space. It remains to be seen whether the better packing allowed by these resource descriptions can be done with a minimum of increase in runtimes.

7.3 Dynamic Delay Nodes

Emulab's delay nodes present an interesting mapping challenge. In the current Emulab environment, where all nodes are connected using 100Mbps Ethernet, it is possible to determine the necessity of traffic-shaping nodes before mapping is done; all links that are not 100Mbps require them. However, in an environment with mixed link speeds, which Emulab hopes to have with the addition of gigabit Ethernet, this can not effectively be done outside the mapper. For example, if only gigabit links are available, but an experimenter desires a 100Mbps link, a delay node may need to be inserted, where it would not if a 100Mbps link were available. Since these decisions about which links to use are known only to `assign`, it becomes necessary for `assign` to be able to introduce delay nodes when appropriate. This dynamic addition of nodes to the virtual topology, however, presents interesting challenges for the generation and cost functions. We have an initial implementation of dynamic delay nodes, but more work is needed.

7.4 Local Search

A possible way to improve `assign`'s performance would be to combine it with local search, another strategy for combinatorial optimization. [10] proposes a combination of simulated annealing with local search, in such a way that simulated annealing is performed

on local minima, rather than on all states. The basic algorithm for this 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 re-assigning 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 is applied, to decide whether or not to move to the new minima.

8 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. We have presented our solver for this problem, discussing its design, implementation, and lessons learned in the process. Through evaluation on real and synthetic workloads, we have shown its effectiveness on a range of problems. Finally, we have identified interesting problems that remain as future work.

Acknowledgments

We are very grateful to several who helped with this paper. Chad Barb implemented alternative versions of `assign`, including a GA, which we used for comparison and validation. Mac Newbold worked on an initial framework to transfer topologies from ModelNet's input format into `assign`'s, and ran the utilization tests. Shashi Guruprasad is working on the modifications to PDNS necessary to test it. We also thank Dave Andersen for his early work on the conception of `assign` and its early versions.

This research was sponsored by many, especially NSF under grants ANI-0082493 and ANI-0205702, Cisco Systems, and DARPA/Air Force under grant F30602-99-1-0503.

References

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

[2] D. Anderson. Theoretical Approaches To Node Assignment, December 2002. Unpublished Manuscript. http://nms.lcs.mit.edu/papers/andersen-assign.ps.

[3] Boost C++ Libraries. http://www.boost.org/.

[4] A. Boukerche and C. Tropper. A Static Partitioning and Mapping Algorithm for Conservative Parallel Simulations. In *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation*. ACM Press, 1994.

[5] K. Fall. Network Emulation in the Vint/NS Simulator. In *Proc. 4th IEEE Symp. on Computers and Communications*, 1999.

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

[7] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[8] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[9] A. Kumar, R. Rastogi, A. Silberschatz, and B. Yener. Algorithms for Provisioning Virtual Private Networks in the Hose Model. In *Proceedings of SIGCOMM '01*. ACM Press, August 2001.

[10] O. C. Martin and S. W. Otto. Combining Simulated Annealing with Local Search Heuristics. *Annals of Operations Research*, 63:57–75, 1996.

[11] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An Approach to Universal Topology Generation. In *MASCOTS '01*, Cincinnati, Ohio, August 2001.

[12] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets-I*, Princeton, NJ, Oct. 2002.

[13] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. A Generic Framework for Parallelization of Network Simulations. In *Proceedings of MASCOTS '99*, October 1999.

[14] The VINT Project. *The* ns *Manual*, Apr. 2002. http://-www.isi.edu/nsnam/ns/ns-documentation.html.

[15] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 271–284, Boston, MA, Dec. 2002. USENIX Association.

[16] P. van Hentenryck. Personal communication, Nov. 2001.

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

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

[19] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.