Reducing Allocation Errors in Network Testbeds

Jelena Mirkovic
USC/ISI
4676 Admiralty Way, Ste 1001
Marina Del Rey, USA
sunshine@isi.edu

Hao Shi
USC/ISI
4676 Admiralty Way, Ste 1001
Marina Del Rey, USA
shihao@isi.edu

Alefiya Hussain
USC/ISI
4676 Admiralty Way, Ste 1001
Marina Del Rey, USA
alefiya@isi.edu

ABSTRACT

Network testbeds have become widely used in computer science, both for evaluation of research technologies and for hands-on teaching. This can naturally lead to oversubscription and resource allocation failures, as limited testbed resources cannot meet the increasing demand.

This paper examines the causes of resource allocation failures on DeterLab testbed and finds three main culprits that create perceived resource oversubscription, even when available nodes exist: (1) overuse of mapping constraints by users, (2) testbed software errors and (3) suboptimal resource allocation. We propose solutions that could resolve these issues and reduce allocation failures to 57.3% of the baseline. In the remaining cases, real resource oversubscription occurs. We examine testbed usage patterns and show that a small fraction of unfair projects starve others for resources under the current first-come-first-served allocation policy. Due to interactive use of testbeds traditional fair-sharing techniques are not suitable solutions. We then propose two novel approaches – Take-a-Break and Borrow-and-Return – that temporarily pause long-running experiments. These approaches can reduce resource allocation failures to 25% of the baseline case by gently prolonging 1–2.5% of instances. While our investigation is done on DeterLab testbed's data, it should apply to all testbeds that run Emulab software.

Categories and Subject Descriptors

C.2.1 [**Computer Communication Networks**]: Network Architecture and Design; C.2.3 [**Computer Communication Networks**]: Network Operations

Keywords

network testbeds, Emulab, resource allocation

1. INTRODUCTION

The last decade brought a major change in experimentation practices in several areas of computer science, as researchers migrated from using simulation and theory to using network testbeds. Teachers are also shifting from traditional lecture-oriented courses to more dynamic and realistic teaching styles that incorporate testbed use for class demonstrations or student assignments. These diverse groups of users each have important deadlines that they hope to meet with help of testbeds, such as conference and research demonstration deadlines, class projects, class demonstrations, etc.

Current testbed resource allocation practices are not well aligned with these user needs. Most testbeds deploy no automated prioritization of allocation requests, serving them on first-come-first-served basis [24, 3, 2, 7, 10], which makes it impossible to guarantee availability during deadlines. In this paper we focus on testbeds that allow users to obtain exclusive access to some portion of their resources. As user demand grows, these testbeds experience overload that leads to allocation failures. Most testbeds further let users keep allocated resources for as long as needed [24, 3, 2]. While there are idle detection mechanisms that attempt to reclaim unused resources, users can and do opt out of them. Based on our experience from managing DeterLab most experiments are interactive, which means that resource allocations should last at most a day. Thus long-running experiments reflect a user's unwillingness to release resources at the end of their work day. We believe this occurs primarily because: (1) testbeds lack mechanisms to easily save disk state on multiple machines and recreate it the next day, and (2) users have no guarantee that resources will be available the next day. Some users request help of testbed staff to reserve resources for major deadlines. Since many testbeds lack reservation mechanisms [24, 3, 2], these are done manually by staff pulling requested machines out from the available pool. This often occurs early, so staff could guarantee availability, but it wastes resources and increases testbed overload.

Testbeds need better and more predictable allocation strategies as they evolve from a novel to a mainstream experimentation platform. Our main goal in this paper is to understand the reasons for resource allocation failures and to propose changes in testbed operation that would reduce these. We survey related work in Section 2. We then introduce our terminology and data in Sections 3 and 4. We explain the resource allocation problem in network testbeds in Section 5. We then examine reasons for resource allocation failures in the DeterLab testbed's [3, 5] operation during 8 years of its existence in Section 6. DeterLab is a public testbed for security experimentation, hosted by USC/ISI and UC Berkeley. In early 2012 it had around 350 general PCs and several tens of special-purpose nodes. While it is build on Emulab technology, its focus is on cyber security research, test and evaluation. It provides resources, tools, and infrastructure for researchers to conduct rigorous, repeatable experiments with new security technologies, and test their effectiveness in a realistic environment. DeterLab is used extensively both by security researchers and educators. It also experiences intensive internal use for development of new testbed technologies.

We find that 81.5% of failures occur due to a *perceived* resource shortage, i.e., not because the testbed lacks enough nodes, but because it lacks enough of the *right* nodes that the user desires. As user desires and testbed allocation strategies act together to create the shortage of the nodes that are in current demand, we next investigate how much relaxing user constraints (Section 7) or improving resource allocation strategy (Section 8) help reduce allocation failures. Finally, we investigate if changes in testbed resource allocation policy would further improve resource allocation both in cases of perceived and in cases of true resource shortage in Section 9. We conclude in Section 10.

While we only had access to DeterLab's dataset, this testbed's resource allocation algorithms and practices derive from the use of Emulab software [24], which is used extensively by 40+ testbeds around the world [1]. Our findings should apply to these testbeds.

Main contributions of our paper are:

- This is the first analysis of causes for resource allocation failures in testbeds. We find that 81.5% of failures occur due to a perceived resource shortage, when in fact there are sufficient nodes to host a user's request. Around half of these cases occur because of inefficient testbed software, while the rest occur because of over-specification in user's resource requests.

- We closely examine the resource allocation algorithm used in Emulab testbeds – `assign` [20] – and show that it often performs suboptimally. We propose an improved algorithm – `assign+` – that reduces allocation failures to 77% of those generated by `assign`, while running 10 times faster and preserving more of the limited resources, such as interswitch bandwidth.

- We propose improvements to testbed resource allocation strategy by gently relaxing user constraints, to reduce resource allocation failures to 68.1% of those generated by `assign`.

- We propose modifications to testbed resource allocation policy that reshuffle allocated experiments to make space for new ones. This further reduces allocation failures to 57.3% of those generated by `assign`.

- We identify and demonstrate the need for some fair sharing and prioritization of user allocation requests at times of true overload. We propose two ways to modify testbed resource allocation policy to achieve these effects: Take-a-Break and Borrow-and-Return. In both, resources from long-running experiments are reclaimed and offered to incoming ones, but in Take-a-Break they are held as long as needed, while in Borrow-and-Return they are returned back to the original experiment after 4 hours. We show that both these approaches improve fairness of resource allocation, while reducing allocation failures to 25.3 – 25.6% of those generated by `assign`.

- During the course of our study we have also identified tools that testbeds need to develop either to better serve their users or to become more stable. We suggest five such improvements throughout the paper.

All the data used in our study is anonymized and publicly released at `http://nsl.isi.edu/TestbedUsageData`.

2. RELATED WORK

The wide adoption of emulation testbeds in the networking research community has spurred studies on different approaches for designing and managing them. For example, the resource management mechanisms for Globus and PlanetLab are contrasted and compared extensively by Ripeanu [21]. Additionally, Banik et. al [4] conduct empirical evaluations for different protocols that can provide exclusive access to shared resources on PlanetLab. The StarBED project has several unique solutions for emulation that include configuring the testbed and providing mechanisms for experiment management [18, 19]. These works either evaluate pros and cons of specific testbed management mechanisms or propose how to build testbeds, but do not investigate resource allocation algorithms, which is our focus.

Testbed usage and design practices have also attracted research attention. Hermenier and Ricci examine the topological requirements of the experiments on the Emulab testbed [24] over the last decade [9]. They propose a way to build better testbeds by: (1) increasing the heterogeneity of node connectivity, (2) connecting nodes to different switches to accommodate heterogeneous topologies without use of interswitch bandwidth, and (3) purchasing smaller and cheaper switches to save costs. Our work is orthogonal to theirs and focuses on optimizing allocation software and policies, regardless of testbed architecture. Kim et. al characterize the PlanetLab testbed's [7] usage over the last decade [13]. Their results indicate that bartering and central banking schemes for resource allocation can handle only a small percentage of total scheduling requirements. They do not propose better resource allocation algorithms, even though they identify the factors that account for high resource contention or poor utilization.

Yu et al. in [25] propose collecting allocation requests during a time window and then allocating testbed resources to satisfy the constraints of this request group. They employ a greedy algorithm to map nodes and path splitting to map links. Besides, they perform online migration to change the route or splitting ratio of a virtual link, which re-balances the mapping of virtual topologies to maximize the chance of accepting future requests. Their methods consider the general mapping problem at a high-level way, but do not take into account heterogeneity of testbed nodes. Besides, queuing allocation requests in network testbeds would introduce potentially large delays that users would not tolerate. Chowdhury et al. in [6] utilize mixed integer programming to solve the resource allocation problem, but their constraints are limited to CPU capacity and distance between the locations of two testbed nodes. J. Lu et. al [15] develop a method for mapping virtual topologies onto a testbed in a cost-efficient way. They consider traffic-based constraints but do not consider node heterogeneity or node features.

In a broader setting, ISPs tend to address resource allocation problems by over provisioning their resources (bandwidth). This solution does not readily apply to network testbeds. First, testbeds have limits on how many machines they can host that stem from the space, weight, cooling and power capacity of the rooms that host them. Second, testbeds are hosted by academic institutions and funded through grants – this limits both human and financial resources for purchase and maintenance of hardware. Finally, testbed use exhibits heavy tails along many dimensions (see Section 9.2), which prevents prediction of future resource needs.

Clusters and data centers face similar resource allocation issues as testbeds [8, 11, 23]. In [8], Ghodsi et al. propose dominant resource fairness (DRF) for resource allocation in data centers. This approach achieves fair allocation of heterogeneous resources between users who prioritize them differently. Unfortunately, like other fair-sharing approaches, DRF is not readily applicable to testbeds (see Section 9.2 for more details) due to interactive nature of experimentation and due to different value of long vs short experiments. In [11], Hindman et al. describe a platform called Mesos for sharing clusters, by allowing multiple resource allocation frameworks to run simultaneously. Mesos offers resource shares to the frameworks, based on some institutional policy, e.g., fair share, and they decide which offers to accept and which tasks to allocate on them. Some principles from [11], such as resource offers, may apply to testbeds, but they assume users that are way more sophisticated and informed about resource allocation than testbeds currently have. Condor [23] is a workload management system for compute-intensive jobs that aims to harness unused resources on heterogeneous and distributed hardware and can migrate data and jobs as nodes become available. While some Condor ideas may apply to network testbeds to achieve instance migration (see Section 9) testbed nodes are usually heavily customized by users, which prevents fine-grain migration that Condor excels at.

3. TERMINOLOGY

We now introduce several terms that relate to network testbed use and illustrate them in Figure 1.

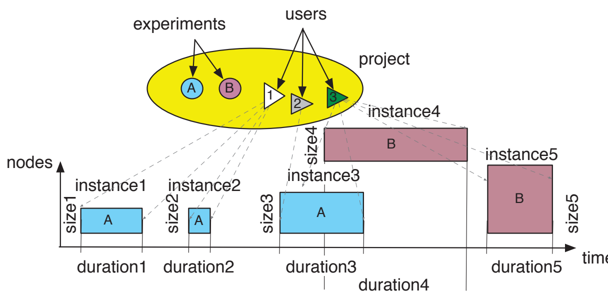


Figure 1: Terminology

An *experiment* is a collection of inputs submitted by a user to the testbed (one or more times) under the same identifier. These inputs describe experimenter's needs such as experiment topology, software to be installed on nodes, etc. We will say that each input represents a *virtual topology*. Experiments can be modified, e.g. by changing number of requested nodes or their connectivity. In Figure 1 there are two experiments A and B.

An *instance* is an instantiation of the experiment at the physical resources of the testbed. We say that an instance has *duration* (how long were resources allocated to it), *size* (how many nodes were allocated) and *virtual topology* (how were nodes connected to each other, what types of nodes were requested, what OS, etc.). The same experiment can result in multiple non-overlapping instances, one for each resource allocation. In Figure 1 there are five instances, three linked to the experiment A, and two linked to B. Release of the resources back to the testbed or instance modification denotes the end of a particular instance.

A testbed *project* is a collection of experiment definitions and authorized users, working on the same common project under a single *head-PI*. In Figure 1 there is one project with two experiments and three users.

An experiment can experience the following events: `preload`, `start`, `swapin`, `swapout`, `swapmod` and `destroy`. Events `preload`, `start` and `destroy` can occur only once during experiment's lifetime, while others can occur multiple times. Each event is processed by one or more testbed scripts and can result in a success or a failure. Figure 2 shows the state diagram of an experiment, where state transitions occur on successful events. A `preload` event leads to experiment's virtual topology being stored on the testbed, but no resources are yet allocated to the experiment – experiment exists in the *defined* state. A `swapin` event leads to resource allocation, changing the experiment's state to *allocated*. A `start` event is equivalent to a `preload` followed by a `swapin`. A `swapout` event releases resources from the experiment, changing its state to *defined*. A `swapmod` event can occur either in *defined* or in *allocated* state. It changes the experiment's definition but does not lead to state change. If a `swapmod` fails while the experiment is in *allocated* state, the testbed software automatically generates a `swapout` event and reverts experiment state to *defined*. A `destroy` event removes an experiment's virtual topology and state from the testbed but history of its events still remains. Table 1 shows the frequency of all experimental events in our dataset, which is described in the following Section.

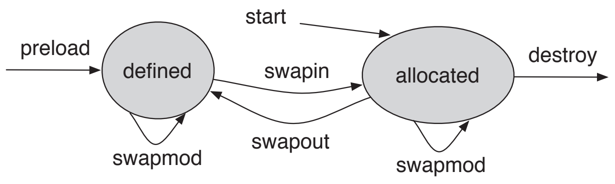


Figure 2: Experiment state diagram

4. DATA

We analyze eight years of data about DeterLab's operation collected from its inception in February 2004 until February 2012. As of February 2012, DeterLab had 154 active research projects (556 research users), 38 active class projects (1,336 users) and 11 active internal projects (95 users). The testbed consisted of 346 general PCs, and some special-purpose hardware. Half of the nodes are located at UCS/ISI, and other half at UC Berkeley. Table 2 shows the features of DeterLab's PCs.

DeterLab runs Emulab's software for experiment control [24], which means that all testbed management events such as node al-

Event	Count	Frequency
preload	10,472	4%
start	16,043	6.1%
swapin	101,275	38.6%
swapmod	36,819	14%
swapout	75,156	28.7%
destroy	22,575	8.6%
total	262,340	100%

Table 1: Frequency of experimental events in the dataset

Type	Disk (GB)	CPU (GHz)	Mem (GB)	Interf.	Count
1	250	2.133	4	4	63
2	250	2.133	4	4	63
3	72	3	2	4	32
4	72	3	2	5	32
5	36	3	2	4	61
6	36	3	2	5	60
7	36	3	2	9	4
8	238	1.8	4	5	31

Table 2: DeterLab's node types as of Jan 2011

Source	Data	Meaning
DB	`events`	Time, experiment, project, size, exit code for each event.
DB	`errors`	Time, experiment, cause and error message for each error.
FS	`/usr/testbed/expinfo`	Virtual topology, testbed resource snapshot, and resource allocation log for all successful and for some unsuccessful resource allocation requests.

Table 3: Types of data analyzed to recreate the experimental events and state of the testbed

location, release, user account creation, etc. are issued from one control node called `boss` and recorded in a database there. Additionally, some events create files in the file system on the `boss` node. We analyze a portion of database and file system state on this node that relates to resource allocations. We have database records about testbed events and any errors that occurred during processing of these events. We further have files describing virtual topologies and testbed state snapshots that were given to the allocation software – `assign` – and the allocation logs showing which physical nodes were assigned to each experiment instance. The virtual topology encodes user desires about the nodes they want, their configuration and connectivity. The testbed state snapshot gives a list of currently available nodes on the testbed, along with their switch connectivity, supported operating system images, features and feature weights (see Section 5 for explanation of these terms). Such snapshots are created on each attempted `start`, `swapin` or `swapmod`. The complete list of our data is shown in Table 3.

In our investigation we found that both database and file system data can be inconsistent or missing. This can occur for several reasons:

1. Different scripts may handle the same event and may generate database entries. It is possible for a script to behave in an unexpected manner or overlook a corner case, leading to inconsistent information. For example, for a small number of experiments we found that database entries show consecutive successful `swapin` events, which is an impossible state transition. We believe that this occurs because one script processes the event and records a success before the event fully completes. In a small number of cases another script detects a problem near the end of resource allocation and reverts the experiment's state to *defined* but does not update the database.

2. State transitions can be invoked manually by testbed operations staff, without generating recorded testbed events. For example, we found a small number of experiments in *allocated* state according to the database, while file system state indicated that they returned the resources to the testbed. This can occur when testbed operations staff manually evicts several or all experiments to troubleshoot a testbed problem.

3. Testbed policies and software evolve, which may lead to different recording of an event over time. For example in 2004–2006, when a user's request for experiment modification had a syntax error this was recoded in the database. This practice was abandoned in later Emulab software releases. Similarly, when a user's request for experiment modification failed due to temporary lack of testbed resources, this request and testbed's state snapshot were recorded in the file system on the `boss` node. This practice was abandoned in early 2007 making it difficult to understand and troubleshoot resource allocation errors.

4. In a small number of cases software generating unique identifiers for file names storing virtual topology and testbed state snapshot had low randomness leading to newer files overwriting older ones within the same experiment. This means that file system state for some instances is missing.

During our analysis, we detect and either correct or discard entries with inconsistencies. We also attempt to infer missing data wherever possible, by combining the database and the file system information.

Suggestion 1: *Testbeds need better software development practices that start from a system model and verify that developed code matches the model, e.g., through model checking and unit testing.* While it is impossible to eliminate all bugs in a large codebase, a systematic tying of code to requirements and models would help eliminate inconsistencies in record-keeping and even facilitate automated detection and forensics of testbed problems.

5. TESTBED MAPPING PROBLEM

We now explain some specifics of testbed operation that relate to resource allocation, using Figure 3 to illustrate them. Many of the concepts in this Section were first introduced in [20].

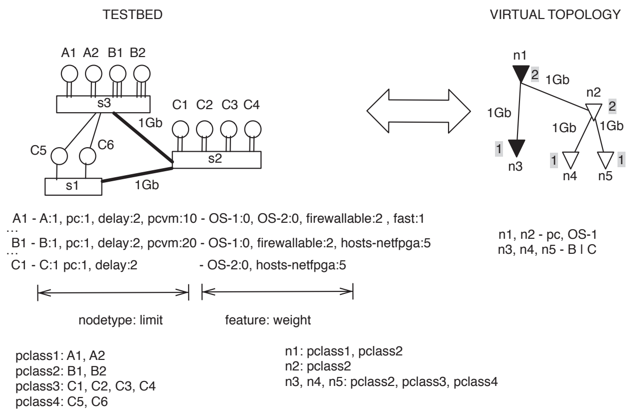


Figure 3: Illustration of the network testbed mapping problem

Over time network testbeds acquire nodes of different hardware types leading to heterogeneity. Types can differ in number of network interfaces, processor speed, memory size, disk space, etc. In Figure 3 the drawing on the left shows a sample testbed architecture. There are three hardware types: *A*, *B* and *C*, with 2, 2 and 6 nodes respectively. Each physical node is connected to a switch. Because a single switch has a limited number of ports a testbed may have multiple switches connected by limited-bandwidth links, each hosting a subset of nodes. In Figure 3 there are three switches – *s1*, *s2* and *s3* – with interswitch links shown as thick lines between them. Often nodes of the same type are connected to the same switch. Sometimes it is beneficial to connect different node types to the same switch (e.g., nodes of type *A* and *B* are connected to *s1*) or to connect some nodes to two different switches (e.g., nodes *C5* and *C6* connect to *s1* and *s3*). DeterLab has instances of all three node-to-switch connection types in its current architecture.

Users submit their experiment configuration requests to the testbed as a *virtual topology*. One such topology is shown in the right drawing in Figure 3. A resource allocation algorithm attempts to solve the *testbed mapping problem* [20]. It starts from the virtual topology and a snapshot of the testbed state and attempts to find the best selection of hardware that satisfies experimenter-imposed and testbed-imposed constraints.

Testbed-imposed constraints consist of limitations on available nodes of any given type, limitations on number of node interfaces, and limited interswitch link bandwidth. Experimenter-imposed constraints are encoded in the virtual topology as *desires* and consist of: (1) **Node type constraints** – a virtual node must be mapped to specific hardware type, (2) **OS constraints** – a virtual node must run specific OS, (3) **Connectivity constraints** – a virtual node should have specific number of network interfaces and must be connected to another node by a link of specific bandwidth. Node type and OS constraints are encoded explicitly by annotating nodes in the virtual topology, and the connectivity constraints are implied in the topology's architecture. For example, in Figure 3, explicit constraints request nodes *n1* and *n2* to be of type *pc* and run OS 1, while nodes *n3*, *n4* and *n5* should be of type *B* or *C*. Implicit connectivity constraints require that *n2* be mapped to a node with at least 3 network interfaces, *n1* to a node with at least 2 interfaces, and the rest to nodes with at least 1 interface. Each link is required to have 1 Gbit bandwidth. This limits the number of virtual links that can be allocated to an interswitch link and in turn invalidates some mapping of virtual to physical nodes that would oversubscribe interswitch bandwidth. Emulab software further lets users specify **fixed mappings**: virtual nodes that map to specific physical nodes (e.g. a user may request *n1* to be mapped to *A1*). This sometimes helps `assign` algorithm to find a solution, where it would otherwise miss it. We elaborate on reasons for fixed mappings in the next Section.

The notion of the *node type* [20] extends beyond simple hardware types in two ways. First, a physical node can "satisfy" multiple node types and may host multiple instances of the same type. For example, node *A1* (see annotations at the bottom left in Figure 3) can host one virtual node of type *A*, one virtual node of type *pc*, two virtual nodes of type *delay*, or ten virtual nodes of type *pcvm* (virtual machine installed on a physical node). Second, a user can specify a *vclass* – a set of node types instead of the single type for any virtual node, e.g. in Figure 3 a user has asked for nodes *n3*, *n4* and *n5* to be either of type *B* or of type *C*. *vclasses* can be hard – requiring that all nodes be assigned to the same node type from *vclass* (e.g., all are *B* or all are *C*) – and soft – allowing mixed type allocations, from the same *vclass* (e.g., each could be *B* or *C*). In DeterLab's operation we have only encountered use of soft *vclasses*. Corresponding to experimenter's desires, physical nodes have *features*. For example, in Figure 3 there are the following features: (1) OS 1 runs on types *A* and *B*, (2) OS 2 runs on types *A* and *C*, (3) firewallable feature is supported by types *A* and *B*, (4) hosts-netfpga feature is supported by types *B* and *C*. Each feature is accompanied by a *weight* that is used during resource allocation process to score and compare different solutions.

Testbeds create *base* OS images for all their users, for popular OS types like Linux, Windows and Free BSD. Over time testbed staff creates newer versions of base images but the old ones still remain on the testbed and are used, we believe due to inertia. Testbeds further allow users to create custom disk images as a way of saving experimental state between allocations. These images are rarely upgraded to new OS versions. As testbeds grow, old custom and base images cannot be supported by new hardware. Thus virtual topologies with such images can be allocated only to a portion of the testbed and OS desires turn into mapping constraints.

Suggestion 2: *Testbeds need mechanisms that either provide state saving without disk imaging, or help users to upgrade their custom images automatically to new OS versions. Experiment specifications (virtual topologies) should also be upgraded automatically to use newer base OS images. This would eliminate OS-based constraints and improve allocation success.*

An *acceptable* solution to the testbed mapping problem meets all experimenter-imposed and testbed-imposed constraints. We note that honoring an interswitch bandwidth constraint is a choice and not a must. Testbed software can allocate any number of virtual links onto the interswitch substrate, but if it oversubscribes this substrate and if experimenters generate full-bandwidth load on the virtual links they may experience lower than expected performance. In our example in Figure 3 it is possible to allocate links *n2-n4* and *n2-n5* on the same 1 Gbit interswitch link, but if the experimenter sends 1 Gbit of traffic on each of them at the same time half of the traffic will be dropped. There are two choices when evaluating if interswitch bandwidth constraint is met: (1) evaluation can be done only within the same experiment assuming no other experiment uses the same interswitch link, and (2) evaluation can be done taking into account all experiments that use the same interswitch link. In practice, solution (1) is chosen because it improves the resource allocation success rate. Risk of violating experimenter's desires is minimal because the incidence of multiple experiments using the same interswitch link and generating high traffic at the same time is low.

The *best* solution to the testbed mapping problem is such that minimizes interswitch bandwidth consumption and minimizes unwanted features on selected physical nodes – these are the features that are present on the nodes but were not desired by the experimenter. Doing so improves the chance of success for future allocations. In face of these allocation goals the testbed mapping problem becomes NP-hard, because the number of possible solutions is too large to be exhaustively searched for the best one.

6. WHY ALLOCATIONS FAIL

A resource allocation may fail for a number of reasons such as a syntax error in the user's request, a testbed software failure, a policy violation by a user's request, etc. In this paper we only investigate resource allocation failures that occur due to temporary shortage of testbed resources. This means that the same virtual topology would successfully allocate on an empty testbed. We will call these TEMP failures and classify them into the following categories:

1. **FIXED:** Virtual topology specified a fixed mapping of some virtual nodes to specific physical nodes but testbed could not obtain access to these nodes.

2. **TYPE:** Virtual topology had node type constraint that could not be met by the testbed.

3. **OS:** Virtual topology had OS constraint that could not be met by the testbed.

4. **CONNECT:** The testbed could not find a node with sufficient interfaces.

5. **INTERSWITCH:** The allocation algorithm found a solution but the projected interswitch bandwidth usage exceeded link capacity.

6. **TESTBED:** There is a problem in the testbed's software that only becomes evident during resource allocation. One such problem occurs when `assign` [20] – the current allocation algorithm – fails to find a solution even though one exists.

Categories FIXED, TYPE, OS, CONNECT and INTERSWITCH stem directly from the way testbeds address the testbed mapping problem (see the previous Section) – a failure to satisfy user or testbed constraints will fall into one of these five categories. In our analysis of TEMP failures on DeterLab we further find that bugs in testbed configuration and software occasionally lead to TEMP failures, *when in reality there are available resources to satisfy user and testbed constraints*. This leads us to create the TESTBED category. One could view TEMP failures that fall into TESTBED category as false TEMP failures, since they do not occur due to a temporary resource shortage.

We first investigate why TEMP failures occurred historically on the DeterLab's testbed. We start with the records from the DeterLab's database that contain experiment identifier, time and alleged cause of each failure, as well as the error message generated by the testbed software. The database only has records for failures that occurred after April 13, 2006. There are 24,206 records, out of which 11,176 have their cause classified as TEMP in the database. We use the error messages to classify these TEMP failures into the categories above. We find that 47.5% are TYPE failures, 18.5% are FIXED failures, 15.7% are OS failures, 3.8% are CONNECT failures and only 0.5% are INTERSWITCH failures. In 13.5% of cases the error message indicates that mapping failed, but does not give the specific reason. Finally there are 0.2% of failures that occur due to a policy violation or a semantic problem in the experimenter's request but are misclassified as TEMP failures.

While the above analysis offers a glimpse into why specific allocations failed, we would like to know how many failures occur due to *true overload* – there are not enough nodes on the testbed – and how many occur due to *perceived overload* – there are enough nodes on the testbed but experimenter or testbed constraints are violated. Cases of *perceived overload* could be eliminated either by relaxing experimenter's constraints or by improving testbed software. To answer these questions we need to match each TEMP failure to the virtual topology and the testbed state snapshot that were given to `assign` so we could mine the desired and the available number of resources. We perform this matching in the following way:

1. We link each TEMP failure to a resource allocation log file showing details of the allocation process, by matching the time of the TEMP failure with the timestamp of the file.

2. From the log file we mine the file names of the virtual topology and the testbed state snapshots that were used by the resource allocation software, i.e. the `assign` algorithm.

3. In 2007, DeterLab testbed stopped saving the virtual topology and the testbed state files for failed allocations so we must infer them from other data. To infer the virtual topology we identify the testbed event (`swapmod` or `swapin`) that led to that specific TEMP failure. For failures that occurred on a `swapin` event, we attempt to find a previous successful `swapmod` or `swapin` of the same experiment and link it to a virtual topology using the same process from steps 1 and 2. We associate this topology with the TEMP failure. To infer the testbed state at the time of TEMP failure, we process the testbed state snapshots chronologically up to the time of the failure and infer from those the physical node features and testbed architecture (connections and bandwidth between nodes and switches). We also take the last testbed snapshot created before the TEMP failure and extract the list of available nodes at the time. We then process any `swapout` events between the time of the last snapshot and the TEMP failure, and add the released nodes to the available pool. This gives us the testbed state at the time of TEMP failure. Then we combine all this information and generate the testbed snapshot in the format required by the `assign` algorithm. This inference process may result in an incorrect testbed state only if some of the available nodes become unavailable in the time between the last testbed snapshot and the TEMP failure. This can happen due to a hardware error, a manual reservation by the testbed staff, or because some of the nodes released by the `swapout` events in that short time interval failed to reload the default OS and required manual intervention by testbed staff. While hardware errors and manual reservations are rare on DeterLab, reload failures occur daily but usually affect a handful of nodes. We thus believe that most of our inferred testbed snapshots are correct.

We were able to match 9,066 out of 11,176 TEMP failures in this manner – they form the *matched-failure* set that we analyze further. We focus only on demand and availability of general PC nodes, since only a small fraction of instances request special hardware. Only 1,679 of TEMP errors or 18.5% occur because of a true overload, meaning that there are less PCs than desired. This means that 81.5% of TEMP errors could potentially be reduced or eliminated by improving testbed software or by educating users how to minimize their use of constraints. To identify TESTBED errors we run both the `assign` and our `assign+` allocation algorithm (described in the Section 8.2) on the remaining 7,387 pairs in the *matched-failure* set.

We next modify virtual topologies in the *matched-failure* set to remove fixed mappings, because they seem to often harm allocations, as evidenced by a high number of FIXED failures. In many cases fixed mappings are inserted not by a user but by the testbed software when a running instance is being modified, e.g. by adding or removing nodes. This enables the testbed to keep the currently allocated nodes associated with the instance and just drop some (in case of removing nodes) or add a few more. However, if some of the allocated nodes become unresponsive the entire resource allocation fails. We believe that this is an incorrect model, and the testbed should fall back to the strategy of releasing all nodes and allocating from the entire available node pool. Another reason for fixed mappings occurs in a case when nodes of a given type may differ based on their location, and a user prefers some locations over others. We argue that these cases would be better handled through node features or through creation of location-specific node types, since fixed mappings allow users to select only one out of several possible node choices.

We find that `assign+` can successfully allocate resources in 1,392 cases, or 15.3% of our *matched-failures* set. We further find that both `assign` and `assign+` succeed in 2,251 or 24.8% of cases. It is possible that the original failure, recorded in the database, was a "bad run of the luck" event for `assign`, due to its randomized search strategy (see Section 8.1 for more details). It is further possible that the original failure occurred due to a failure of some other testbed software and was recorded as a TEMP failure. Either way, we classify these failures as TESTBED failures. Finally, we find that in 456 cases or 5% allocation failed due to a spelling error in the database in some switch names. These entries are used when testbed snapshots are created, and a spelling error leads to a disconnected testbed. We thus conclude that 3,288 or 36.3% of TEMP errors occur due to experimenter's constraints, 4,099 or 45.2% occur due to testbed software and 1,679 or 19.5% occur due to true overload. There are thus three ways of addressing the allocation problem: (1) helping users understand and reduce the constraints on their topologies, (2) designing better resource allocation algorithms and (3) enforcing some fair sharing of resources. We explore each of these strategies in the following sections.

Suggestion 3: *Testbeds should develop automated self-checking software that detects events such as spelling errors in the database records, real switch and node disconnections, etc., well before they lead to resource allocation failures.*

7. RELAXING USER CONSTRAINTS

We now explore how much user constraints influence allocability of instances on DeterLab. We first match all the successfully allocated instances in our dataset with their virtual topology and the state of the empty testbed that existed at the time of their allocation. For each topology, we simulate the checks in the testbed mapping software for node type, OS and connectivity constraints on this empty testbed. We limit our checks only to those nodes in the virtual topology that can be allocated on general PCs, and the testbed state only includes these PCs. For each node we record the *nodescore*, showing the percentage of testbed that can satisfy this node's constraints. For example, if a user asked for a node of type A or B with OS 1 and if there are 30 nodes of type A, 30 of type B and 20 of type C in the testbed, with OS 1 supported on A and C, the *nodescore* for this node would be $30/80 = 0.375$ because it can only be allocated to nodes of type A. We then calculate the *topscore*, averaging all the *nodescore*'s in the virtual topology.

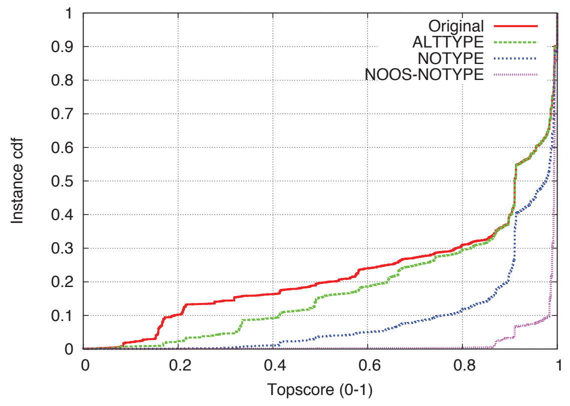


Figure 4: *Topscore* values when we vary type restrictions

The red line in the Figure 4 shows the cumulative distribution function (cdf) of all the *topscores* in the original topologies. There are 30% of topologies that can allocate on less than 80% of the testbed, 20% can allocate on less than half of the testbed and 10% can allocate on less than 20% of the testbed. To identify the effect of the node type, OS and connectivity constraints on the allocability we modify virtual topologies in the following ways: (1) **ALTTYPE:** We allow use of alternative node types that have similar or better hardware features than the user-specified node type; these are described in more detail in Section 8.4, (2) **NOTYPE:** We completely remove the node type constraint, (3) **NOOS-NOTYPE:** We remove both the node type and the OS constraints; the OS constraint can be removed by users upgrading their experiments to use newer OS versions that are supported by all testbed hardware. Effect of these strategies on the allocability is also shown in Figure 4. Use of alternative types improves the allocability, especially of

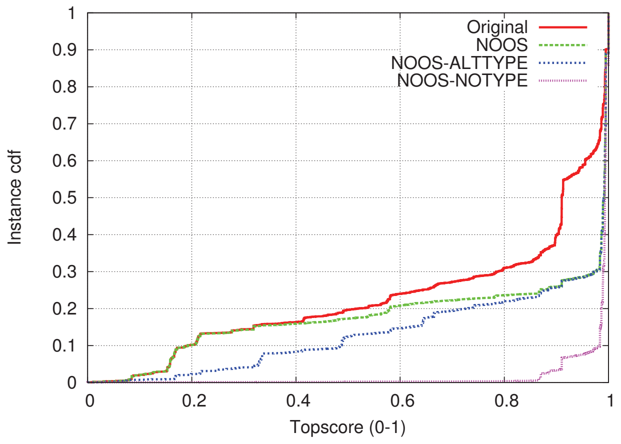


Figure 5: *Topscore* values when we vary OS restrictions

those topologies that were previously severely restricted. There are now 15% of topologies now allocate on less than half of the testbed and only 2% allocate on less than 20% of the testbed. Removal of node type constraints has a profound effect. Only 11% of topologies now allocate on less than 80% of the testbed, only 3% on less than half of the testbed and only 0.1% on less than 20% of the testbed. Finally, removing all node type and OS constraints leads to only 0.3% of topologies to allocate on less than 80% of the testbed.

We next explore the effect of (1) **NOOS:** Removing OS restrictions, (2) **NOOS-ALTTYPE:** Removing OS restrictions and using alternative node types. The effect of these strategies is shown in Figure 5. Removal of OS constraints leads to 23% of topologies that can allocate on less than 80% of the testbed, 17% can allocate on less than half of the testbed and 10% can allocate on less than 20% of the testbed. If we add to this use of alternative types, 21% of topologies that can allocate on less than 80% of the testbed, 12% can allocate on less than half of the testbed and only 2% can allocate on less than 20% of the testbed.

We do not explore how changing connectivity would influence allocability, because connectivity constraints only affect a small number of topologies, and lower the allocability by a small value. This is reflected in **NOOS-NOTYPE** line in Figure 5 where it departs from 1 to values 0.9–1 for about 15% of topologies.

This section has laid out strategies that users can deploy themselves to improve allocability of their topologies. But, how likely is this to happen, i.e. are users flexible about their constraints? To answer this, we try to characterize evolution of virtual topologies in our dataset by first pairing failed allocations with the first following successful allocation in the same experiment and then comparing their topologies. We manage to pair 2,124 out of our 9,066 virtual topologies from the *matched-failures* set. In 956 of those pairs the topologies differ: in 639 cases a user has modified node type or OS constraint, and in 322 cases a user has reduced the topology's size. We conclude that users naturally relax their constraints when faced with an allocation failure about half of the time. When we examine how long it takes a user to converge to a "good" set of constraints we find that half of the users discover the constraint set that leads to successful allocation within one hour, 66% within 4 hours, and 78% within a day. But this distribution is heavy-tailed with the tail going into year-long values, possibly due to the user abandoning the experiment and returning to it much later.

Having automated tools that identify and propose alternative constraints to users would improve this convergence time, and would improve user experience. We believe that such an interactive dialogue with the user would work better then letting users specify how important certain constraints are to them, since this more actively engages the user and informs them about possible trade-offs.

Suggestion 4: *Testbeds need tools that help users evaluate trade-offs between different constraint sets and automatically suggest modifications that improve allocability. This could be done prior to the actual attempt to allocate resources.*

8. IMPROVING RESOURCE ALLOCATION

We now explain how `assign` algorithm works and how we improve it in `assign+`.

8.1 `assign`

In [20] Ricci et al. propose and evaluate the `assign` algorithm as a solver for the testbed mapping problem. Because this problem is NP-hard, Ricci et al. propose to solve it using simulated annealing [14] – a heuristic that performs a cost-function-guided exploration of a solution space. Simulated annealing starts from a random solution and scores it using a custom *cost function* that evaluates its quality. It then perturbs the solution using a *generation function* to create the next one. If this solution is better than the previous one it is always accepted; otherwise it is accepted with some small probability, controlled by *temperature*. This helps the simulated annealing to get out of the locally optimal solutions and find the global optimum. At the beginning of the search, the temperature is set to a high value, leading to most solutions being accepted. Over time the temperature is lowered, following a custom *cooling schedule*, making the algorithm converge to a single “best” solution. There is no guarantee that the algorithm will find the best solution but it should find one that is much better than a random assignment, and fairly close to the best one. Obviously, as algorithm runs longer its chance of finding the global optimum increases but so does the runtime. To guarantee time-bounded operation, `assign`’s runtime is limited, which may sometimes make it miss a possible solution.

To condense the search space, Ricci et al. introduce the concept of *pclasses* – sets of nodes that have the same node types, features, network interfaces and switch connections. In Figure 3 we identify four *pclasses*. Virtual nodes are then mapped to *pclasses*. The `assign` algorithm starts from the set of all *pclasses* and precomputes for each virtual node a list of *pclasses* that are acceptable candidates. It then moves all the virtual nodes into *unassigned* list and, at each step, tries to map one node from this list to a *pclass*. When all the nodes have been assigned, the algorithm tries in each step to remap one randomly selected virtual node to another *pclass*. Each solution is scored by calculating a penalty for used interswitch bandwidth and for unwanted features. The actual scoring function is quite complex but it approximately adds up unwanted feature weights and fixed link penalties. A lower score denotes a better solution. In the end `assign` selects the solution with the lowest score as the best one. Algorithm 1 gives a high-level overview of `assign`’s operation.

Algorithm 1 `assign` pseudocode.

```
 1: generate $pclasses$; put all virtual nodes into $unassigned$ set
 2: map each virtual node to candidate $pclasses$
 3: repeat
 4:     assign one node from $unassigned$ to a $pclass$
 5: until $unassigned = \emptyset$
 6: repeat
 7:     $solution$ = remap one virtual node to a different $pclass$
 8:     score $solution$; $solutions += solution$
 9: until sufficient iterations or average score low
10: select the lowest scored $solution$ as best
```

8.2 `assign+`

In designing `assign+` our main insight was to use expert knowledge of network testbed architecture to identify allocation strategies that lead to minimizing interswitch bandwidth. These strategies are deployed deterministically to generate candidate solutions, instead of exploring the entire space of possible allocations via simulated annealing, which significantly shortens the run time. We also recognized that allocating strongly connected node clusters in experiment topologies together leads to preservation of interswitch bandwidth and shortens the run time. In the long run these strategies also lead to better distribution of instances over heterogeneous testbed resources.

Like `assign`, `assign+` generates *pclasses* and precomputes for each virtual node a list of *pclasses* that are acceptable candidates. It then generates *candidate lists*, aggregating virtual nodes that can be satisfied by the same candidate *pclasses*. For example, in Figure 3 *n1* can be satisfied by *pclass1* or *pclass2*, *n2* can be satisfied only by *pclass2* because it requires three network interfaces, and *n3*, *n4* and *n5* can be satisfied by *pclass2*, *pclass3* or *pclass4*. Each *pclass* has a size which equals the number of currently available testbed nodes that belong to it. Next, the program calls its `allocate` function five times, each time exploring a separate allocation strategy.

The main idea of the `allocate` function is to divide the virtual topology into several connected subgraphs and attempt to map each subgraph or its portion in one step, if possible. Only if this fails, the function attempts to map individual virtual nodes. This reduces the number of allocation steps, while minimizing the interswitch bandwidth, because connected nodes are mapped in one step whenever possible.

The `allocate` function first breaks the virtual topology into several connected partitions attempting to minimize the number of cut edges. Our partitioning goal is to create a large number of possible partitions, where smaller partitions can be subsets of larger ones. This allows us flexibility to map these partitions to different-sized *pclasses*. We achieve this goal by traversing the topology from edges to the center and forming parent-child relationships, so that nodes closer to the center become parents of the farther nodes.

Graph partitioning problem has many well-known solutions (e.g. [12]), but these either require the number of partitions to be known in advance – whereas we want to keep this number flexible – or they are too complex for our needs. We employ the following heuristic to generate the partitions we need. We start from virtual nodes with the smallest degree and score them with number 1, also initializing round counter to 1. In each consecutive round, links that are directly connected to the scored nodes are marked, if the peer on the other side of the link is either not scored yet or is scored with the higher number. The peer becomes a “parent” of the scored node if it does not already have one. The process stops when all nodes in the virtual topology have been scored. We illustrate the scores for nodes in the virtual topology in Figure 3. Black nodes belong to one partition and white ones to the second one. Node *n2* is the parent of nodes *n4* and *n5* and node *n1* is the parent of the node *n3*.

Next, the `allocate` function traverses the candidate list from the most to the least restricted, attempting to map each virtual node and, if possible, its children. Let us call the virtual node that is currently being allocated the *allocating* node. The most restricted candidate list has the smallest number of *pclasses*. In our example this is the list for node *n2*. The function calculates the number of virtual nodes that must be allocated to this list and the number of physical nodes available in the list. If the first is larger than the second the entire mapping fails. Otherwise, we calculate for each parent node in the candidate list two types of children pools: *minimum pool* and *maximum pool*. Both calculations only include those children that have not yet been allocated. The minimum pool relates to the candidate list and contains all the children of the node that must be allocated to this list. The maximum pool relates to each *pclass* in the candidate list and contains all the children of the given parent node that can be allocated to this *pclass*. In our example, when we allocate *n1* its minimum pool would be empty because neither *n4* nor *n5* must be allocated to *pclass2*, while the maximum pool would contain *n4* and *n5* for *pclass2*. The `allocate` function traverses each *pclass* in the current candidate list in an order particular to each allocation strategy we explore. This order is always from the most to the least desirable candidate. It first tries to allocate the allocating node and its maximum pool. If there are no resources in any of the *pclasses* of the candidate list it tries to allocate the allocating node and its minimum pool. If this also fails, it tries only to allocate the allocating node. If this fails the entire mapping fails.

Algorithm 2 `assign+` pseudocode.

```
1: generate pclasses
2: map each virtual node to candidate pclasses
3: generate candidate lists
4: for strategy = (PART, SCORE, ISW, PREF, FRAG) do
5:     solution = allocate(strategy)
6:     score solution; $solutions+ = solution$
7: end for
8: select the solution with the lowest interswitch bw as best
9: break ties by selecting solution with the lowest score
```

There are five allocation strategies we pursue in the calls to the `allocate` function: PART, SCORE, ISW, PREF and FRAG. Each strategy uses expert knowledge of possible network testbed architectures to generate candidate solutions that are supposed to minimize interswitch bandwidth use. The success of each strategy depends on the available resources and the size and user-specified constraints in a given virtual topology. The first strategy – PART – minimizes partitions in the virtual topology by allocating *pclasses* from largest to smallest size. This improves packing of future instances and also reduces number of interswitch links. The second – SCORE – minimizes the score of the allocation by allocating *pclasses* from those with the smallest to those with the largest score. We explore different ways to score a *pclass*, e.g., based on how many features it supports, based on how often it is requested, or a combination of both. When we score by features we do not use feature weights, but instead just add up counts of supported features. The next three allocation strategies prefer those *pclasses* that already host parents or children of the allocating node, thus minimizing interswitch bandwidth demand. In addition to parent/child host preference, the ISW strategy also prefers those *pclasses* that have high-bandwidth interswitch links to *pclasses*, which host neighbors of the allocating node. This only makes a difference when interswitch links have different capacities, in which case ISW will minimize the risk of mapping failure due to interswitch bandwidth oversubscription. In addition to parent/child host preference, the PREF strategy also prefers those *pclasses* that share a switch with *pclasses*, which host neighbors of the allocating node. This minimizes use of interswitch bandwidth because communication is contained within one switch, even though it occurs among different *pclasses*. The PREF strategy tries to both minimize the interswitch bandwidth and to minimize partitions in the virtual topology by allocating from *pclasses* with the largest to those with the smallest product of their preference and size. The FRAG strategy only deploys parent/child preference and tries to use the smallest number of *pclasses* by allocating from *pclasses* with the largest to those with the smallest product of their preference and size.

At the end, the `allocate` function records the candidate solution and then tries to further reduce interswitch bandwidth cost by running Kernighan-Lin graph partitioning algorithm [12] to exchange some nodes between *pclasses* if possible. Each exchange generates a new candidate solution. The algorithm stops when no further reduction is possible in the interswitch bandwidth. Each solution's score is the sum of scores of all the physical nodes in it.

After all calls to the `allocate` function return, `assign+` chooses the best solution. This solution has the smallest interswitch bandwidth. If multiple such solutions exist, the one with the smallest score is selected. Algorithm 2 gives a high-level overview of `assign+`'s operation.

8.3 Evaluation

Algorithm	Failed allocations (out of 19,258)	% Baseline
`assign`	1,176 (mean)	100%
`assign+.1`	905	77%
`assign+.1m`	983	83.6%
`assign+.2m`	917	78%
`assign+.at`	801	68.1%
`assign+.mig`	701	59.6%
`assign+.atmig`	674	57.3%
`assign+.tb`	298	25.3%
`assign+.borrow`	301	25.6%

Table 4: Evaluation summary for allocation failure rates

To compare the quality of found solutions and the runtime of `assign` and `assign+`, we needed a testbed state (hardware types and count, node types, features, weights and operating systems supported by each hardware type, node and switch connectivity), and a set of resource allocation requests. We reconstruct the state of the DeterLab testbed on January 1, 2011 using virtual topology and testbed state snapshot data from the filesystem. To make the allocation challenging, we permanently remove 91 PC nodes from the available pool, leaving 255. While this may seem extreme, our analysis of testbed state over time indicates that often this many or more PCs are unavailable due to either reserved but not yet used nodes or to internal testbed development. We seed the set of resource allocation requests with all successful and failed allocations on DeterLab in 2011. Each request contains the start and end time of the instance and its virtual topology file. For failed allocations, we generate their desired duration according to the duration distribution of successful allocations. Finally, we check that there are no overlapping instances belonging to the same experiment. If found, we keep the first instance and remove the following overlapping instances from the workload. We test this workload both with `assign` and `assign+` on empty testbed and remove 1.3% of instances that fail with both algorithms, because the reduced-size testbed does not meet experimenter's constraints. We will label this final simulation setup "2011 synthetic setup". We then attempt to allocate all workload's instances, and release them in order dictated by their creation and end times, evolving the testbed state for each allocation and each release. Evaluation results are summarized in Table 4.

Figure 6 shows the allocation failure rate over time on this setup for both algorithms. Since `assign` deploys randomized search,

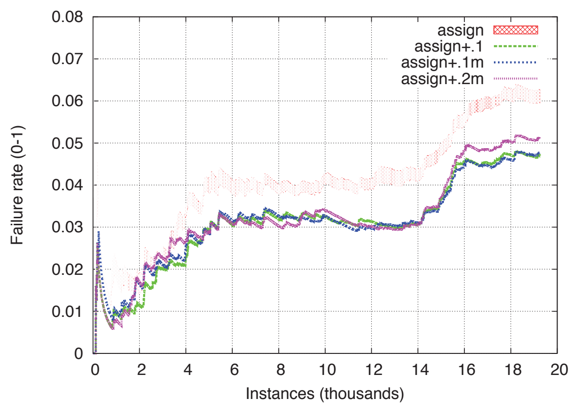


Figure 6: Allocation failure rates

we show the mean and the standard deviation of its 10 runs. We test several approaches to score calculation: `assign+.1` deploys `assign`-like approach, where node types with more unwanted features are penalized higher, (2) `assign+.1m`, same as `assign+.1` but with memory, so node types that are requested more often receive a higher penalty when allocated, and (3) `assign+.2m` scores nodes only by how often they are requested. In case of `assign`, the failure rate starts small and increases to almost 6% by the end of the simulation. Curves for different flavors of `assign+` have the similar shape but the failure rate is always below that for `assign`, reaching 4.7% at the end. Overall, `assign+` creates only 77% of allocation failures generated by `assign`. `assign`-like score calculation outperforms the one that depends only on user request frequency. In the rest of the paper we use `assign+.1` under `assign+` label.

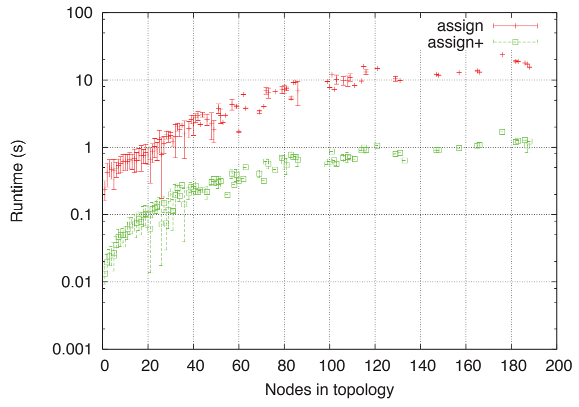


Figure 7: Runtime versus topology size

Figure 7 shows the runtime of `assign` and `assign+` versus topology size, with y-axis being in the log scale. For both algorithms the runtime depends on the size and complexity of the virtual topology, and the number and diversity of available nodes on the testbed. We show the average of runtimes for each topology size and the error bars show one standard deviation around the mean. `assign+` is consistently around 10 times faster than `assign`, thanks to its deterministic exploration of the search space.

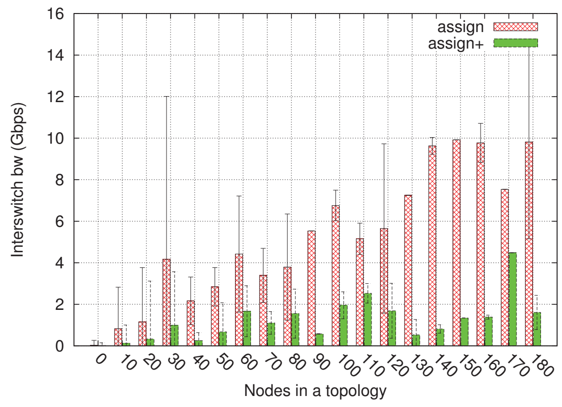


Figure 8: Interswitch bandwidth versus topology size

Figure 8 shows the interswitch bandwidth allocated by `assign` and `assign+` versus topology size. We group allocations into bins based on the virtual topology size, with step 10, and show the mean, with error bars showing one standard deviation. Here too, `assign+` significantly outperforms `assign` on each topology size. On the average, `assign+` allocates only 23% of the interswitch bandwidth allocated by `assign` .

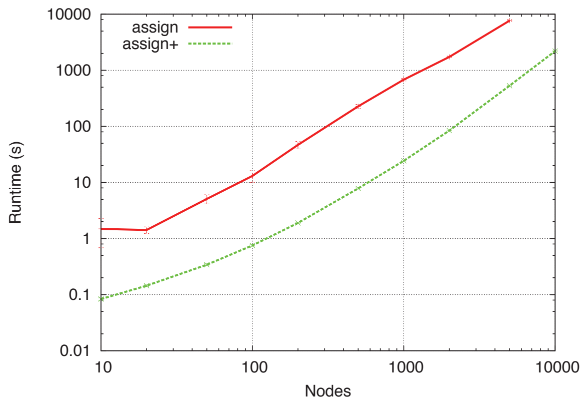


Figure 9: Scalability

We further test the scalability of both algorithms by using Brite [16] to generate realistic, Internet-like topologies of larger sizes. These topologies are more complex and more connected than those found in most testbed experiments [9] and thus challenge allocation algorithms. We generate 10 topologies each of the size 10, 20, 50, 100, 200, 500, 1,000, 2,000, 5,000 and 10,000 nodes. For each node we request `pcvm` type (virtual machines), making it possible to allocate large topologies on our limited testbed architecture. Each allocation request runs on an empty testbed. Figure 9 shows means and standard deviations for runtime of `assign` and `assign+` versus the topology size, both on log scale. `assign+` again outperforms `assign` having about 10 times shorter runtime. `assign` further fails to find a solution for 10,000-node topologies, while assign+ finds it. assign+ only allocates interswitch bandwidth in 5,000 and 10,000 node cases, while assign allocates it for much smaller topologies. We believe that mechanisms that limit assign's runtime for large topologies interfere with its ability to find a good solution that minimizes the interswitch bandwidth.

8.4 Alternative Types

We now assume that users request specific node types because they need some well-provisioned resource such as a large disk or a fast CPU. We explore if we can further improve resource allocation success by expanding experimenter's node type constraint to equivalent or better hardware. In real deployment this would have to be done only with user's consent. We only consider disk, CPU and memory specifications. We start from DeterLab's node types in January 2011, as shown in Table 2, and identify for each type *alternative types* that have same or better features in these three dimensions.

Figure 10 shows the effect of using alternative types on allocation success of assign+, under the label assign+.at. It creates 68.1% of failed allocations generated by assign.

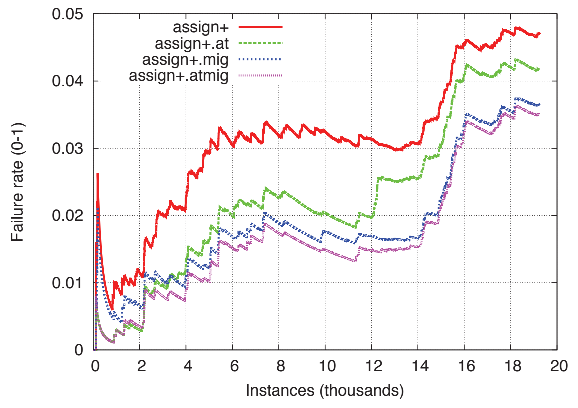


Figure 10: Allocation failure rates for assign+, when using alternative types and migration

9. CHANGING ALLOCATION POLICY

The approaches in the previous section change the resource allocation strategy, but do not change the allocation policy on testbeds that allows users to hold resources for arbitrarily long times without interruption. In this section we investigate if changes in resource allocation policy would further improve allocation success. We examine two such changes:

1. *Experiment migration* – where running instances can be migrated to other resources in the testbed (that still satisfy user-specified constraints) to make space for the allocating instance.

2. *Fair sharing* – where running instances can be paused if they have been running for a long time and are currently idle, so their resources can be borrowed by an allocating instance.

We acknowledge that either of these changes would represent a major shift in today's network testbeds' philosophy and use policy. Yet such shift may be necessary as testbeds become more popular and the demand on their resources exceeds capacity. Our work helps evaluate potential benefits of such policy changes.

Further, the above changes are potentially disruptive for some instances that rely on the constancy of hardware allocated to them for the duration of their lifetime. We assume that users would have mechanisms to opt out of these features, i.e. they could mark their experiment as "do not migrate" or "do not pause". We further assume that, if the benefits of these policy changes seem significant, testbeds would develop mechanisms to seamlessly migrate or pause and restart instances that would be minimally disruptive to users. Finally, we emphasize that instances could be migrated and/or paused only during idle times, when allocated machines exhibit no significant terminal, CPU, disk or network activity. Emulab software looks for such idle machines each 10 minutes, and records each machine's state (idle/non-idle) in the database, overwriting the previous state. We have collected these reports for one year to investigate the extent of idle time in instances. Figure 11 shows the distribution of the total idle time in four classes of instances: those that last < 12, 12-24 h, 1-7 days and > 7 days. All instances have significant idle time and long-lived instances are often idle for more than a week! This leads us to believe that our proposed policy modifications would apply to many instances and would not disrupt their existing dynamics.

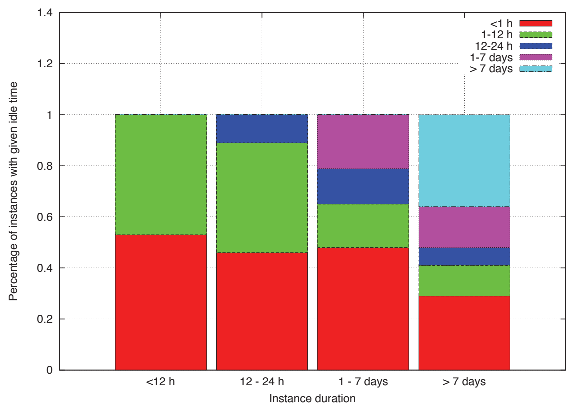


Figure 11: Idle times for instances of different duration

9.1 Migration

We now explore how much we can improve resource allocation by performing experiment migration. This would include stopping some allocated instances and moving their configuration and data to other testbed machines to "make space" for the allocating instance. While there exist techniques in distributed systems that can migrate running processes to another physical node [17], we propose a much lighter-weight migration that would only move instances that are idle at the time. The simplest implementation of such migration would be to image all disks of the experimental machines, and load those images to new machines, but that would require a lot of time and disk space.

We test the migration on our "2011 synthetic setup". If a regular resource allocation fails, we identify any instance that holds node types requested by the allocating instance and is idle as the *migration candidate*. We then order candidates from the smallest to the largest and attempt to migrate them one at a time. To do so we reclaim resources from the migration candidate, try to allocate the allocating instance, and then try to reallocate the migration candidate. Allocation succeeds only if both of these actions succeed. Otherwise we restore the old state and try the next candidate. We only record a failure for the allocating instance if all migrations fail. In real deployment this can be easily simulated, without disturbing any instances, until the successful combination is found.

Figure 10 shows the allocation failure rate when using migration during `assign+` under `assign+.mig` label, and when combining migration and alternative types, under `assign+.atmig` label. We assume that a migration candidate is always idle in our simulation. Migration lowers the allocation failure rate to 59.6% of that generated by `assign`. Adding the alternative types to migration has a minor effect, lowering the allocation failure rate to 57.3% of that generated by `assign`.

Suggestion 5: *Testbeds need better state-saving mechanisms that go beyond manual disk imaging, to support migration.*

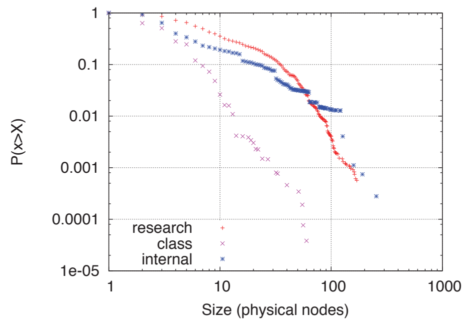
9.2 Fair-Sharing

When demand for a shared resource exceeds supply, a usual approach is to enforce fair sharing and penalize big users. Traditional fair sharing where every user (or in testbed case every project) receives a fair share of resources works well when: (1) users have roughly similar needs for the resource, or (2) the demand does not heavily depend on the resource allocation success and jobs are scheduled in fixed time slots. In the first case, one can implement quotas on use, giving each user the same amount of credit and replenishing it on periodic basis. In the second case, one can implement fair queuing (e.g., [22]), allocating jobs from big users whenever there is leftover resource from the small ones. Unfortunately, neither of these approaches works well for testbeds.

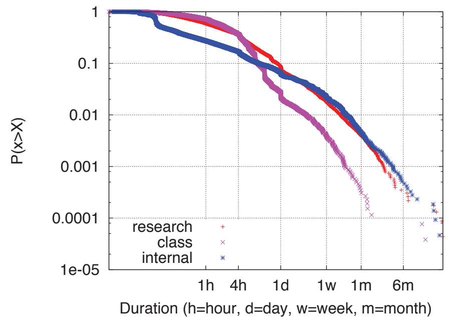
Many measures of testbed usage exhibit heavy-tail properties that violate the first assumption about users having similar needs. For example, the distributions of instance size and duration is heavy-tailed (Figure 12(a) and 12(b)): most instances are short and small, but there are a few very large or very long instances that dominate the distribution. If we assume that fairness should be measured at the project level, heavy tail property manifests again. The distribution of node-hours per project (Figure 12(c)), obtained by summing the products of size and duration in hours for all its instances, is also heavy tailed. The second assumption is violated because most fair-sharing algorithms are designed for fixed-size jobs while testbed instances have a wide range of durations that are not known in advance.

We now quantify the extent of unfairness on DeterLab testbed. We define a project as "unfair" if it uses more than its fair share of PCs in a week. While we focus on PC use, similar definitions can be devised for specific node types. We choose a week-long interval to unify the occurrence of heavy use due to any combination of large instances, long instances or many parallel instances in a project. A fair share of resources is defined as total number of possible node-hours in a week, taking into account available and allocated PCs, divided by the total number of active projects in that week. A project can be classified as unfair one week and fair the other week.

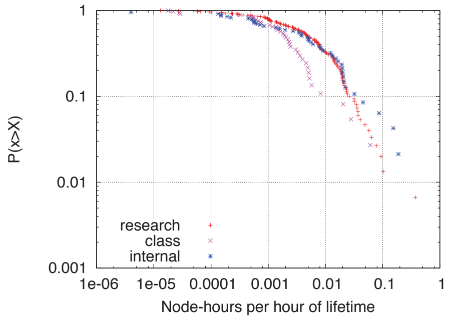
Figure 13 shows the percentage of total possible node-hours used by unfair and by fair projects each week during 2011. There are 114 projects active during this time, each of which has been fair at some point during the year; 27 projects have also been unfair. There were total of 3,126 TEMP failures in 2011, averaging 58.7 failures per project when it is unfair, and 26.1 failures per project, when it is fair. While an unfair project has more than double the errors of a fair project, this is expected, because unfair projects request more



(a) Instance size



(b) Instance duration



(c) Node-hours/hour project lifetime

Figure 12: Heavy tails in testbed use measures

allocations per second, their allocations are larger and have longer duration.

Penalizing big testbed users is difficult because large use in research projects seems to be correlated with publications. We manually classify research projects on DeterLab as "outcome projects" if they have published one or more peer-reviewed publication, or MS and PhD thesis, in which they acknowledge use of DeterLab. We find 48 outcome projects and 104 no-outcome projects. We then define an instance as big if it uses 20 nodes or more, and as long if it lasts one day or longer. Only 9% of instances are big and 5% are long. Outcome projects have on the average 99 big and 33 long instances, while no-outcome projects have 10 big and 8 long instances. Thus big instances and long instances that lead to unfair testbed use seem to be correlated with research publications, and good publicity for the testbed, and are thus very valuable to testbed owners. It would be unwise to alienate these users or discourage heavy testbed use. Instead, we want to gently tip the scale in favor of small users when possible.

We identify three design goals for fairness policy on testbeds:

1. **Predictability.** Any fairness approach must allow users to accurately anticipate when they may be penalized.
2. **User control.** Actions taken to penalize a user must depend solely on their actions, and testbeds should offer opt-out mechanisms.
3. **On-demand.** Resources should be reclaimed only when there is an instance whose allocation fails, and whose needs can be satisfied by these resources.

One approach to tipping the scale would be to reclaim some resources from unfair projects until their use is reduced to a fair share. This would violate all three design goals, because unfair status changes depending on how many other projects are active, and freed resources could sit unused on the testbed. Another approach would be to reclaim resources on demand from an instance that has used the most node-hours. Again this leads to unpredictable behavior from the user's point of view, and it may interrupt short-running but large instances that are difficult to allocate again. We opt for the strategy that reclaims resources on demand from the longest-running instance, as long as it has been running for more than one day and is currently idle. This lets users identify which of their instances may be reclaimed in advance. We propose two possible approaches to fair allocations: Take-a-Break and Borrow-and-Return.

9.3 Take-a-Break

In Take-a-Break approach, when a resource allocation fails, we identify any instance that holds node types that are requested by the allocating instance and is currently idle, as the *break candidate*. We then select the candidate that has been running the longest and, if its age is greater than one day, we release its resources to the allocating instance. The break candidate is queued for allocation immediately, and an attempt is made to allocate it after any resource release.

Figure 14 shows the rate of allocation failures for Take-a-Break approach under the `assign.tbreak` label. We also deploy migration and alternative node types. We assume that a break candidate is always idle in our simulation. Failure rate is strikingly

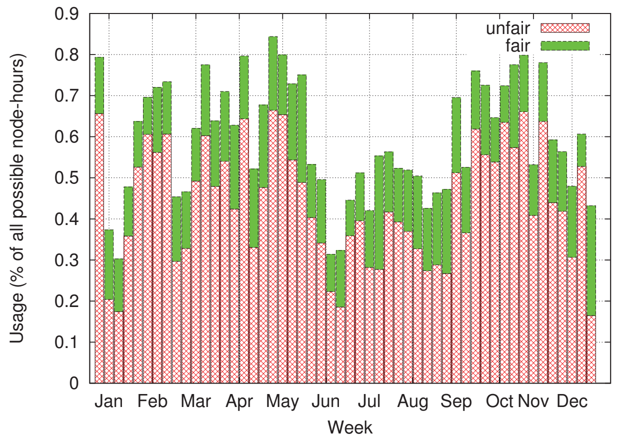


Figure 13: Usage of fair and unfair projects in 2011.

low, reaching 1.5% by the end of our simulation even though the density of allocated instances is increased. Overall, `assign+` with Take-a-Break creates only 25.3% of failed allocations generated by `assign`. This comes at the price – duration of 177 instances is prolonged. Half of these instances experience delays of up to 1 hour, 79% up to 4 hours, and 97% up to 1 day. Only six instances are delayed more than one day, the longest delay being 1.67 days. Looking at relative delay, 72% of instances are only delayed up to 1% compared to their original duration, 94% of instances are delayed by at most 10% and the worst delay doubles the instance's duration.

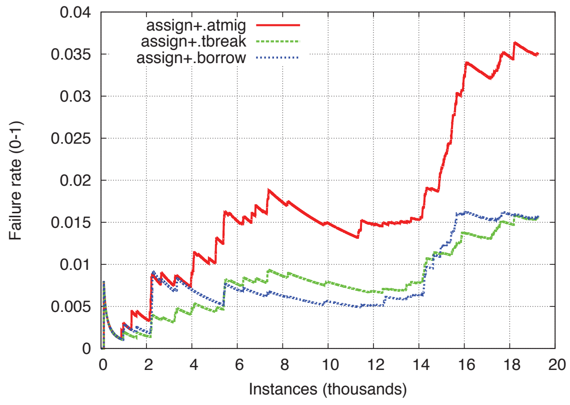


Figure 14: Error rates for `assign+` when using Take-a-Break and Borrow-and-Return approaches.

We now verify if we have tipped the scale in favor of fair projects. We first apply the fairness calculation to the allocations resulting from running `assign+` on our "2011 synthetic setup" and obtain similar usage patterns, to those seen in the real dataset, except that the allocation failures are reduced because we were removing overlapping instances during workload creation and we started with an empty testbed. There were 14.5 failures per project when it is unfair, and 3.98 failures per project, when it is fair. We then apply the same calculation to the allocations resulting from running `assign+` with Take-a-Break on our "2011 synthetic setup" and we count both allocation failures and forcing an instance to take a break as "failures". We find that there are 18.6 failures per project when it is unfair, and 1.62 failures per project, when it is fair. Thus unfair projects are slightly penalized and the failure rate of fair projects is more than halved.

9.4 Borrow-and-Return

While Take-a-Break approach helps fair projects obtain more resources, it forces instances whose resources have been reclaimed to wait for unpredictable duration. Borrow-and-Return approach amends this. Its design is the same as Take-a-Break approach, but resources are only "borrowed" from long-running instances for 4 hours, after which they are returned to their original owner. Users receiving these borrowed nodes would be alerted to the fact that the nodes will be reclaimed at a certain time. Instances interrupted this way are queued and allocated as soon as possible.

Figure 14 shows the rate of allocation failures for Borrow-and-Return approach, under the `assign.borrow` label. We also deploy migration and alternative node types. Allocation failure rate is similar to that of Take-a-Break approach – 25.6% of that of the `assign`. Duration of 583 instances is prolonged. 80% of these instances experience delays of up to 1 hour, 96% up to 4 hours, and 99% up to 1 day. Only five instances are delayed more than one day, the longest delay being 1.9 days. Looking at relative delay, 25% of instances are delayed up to 1% compared to their original duration, 76.1% of instances are delayed by at most 10%, 97% are delayed by at most 100% and the worst delay extends the instance's duration 4.5 times. This approach seems to find a good middle ground between heavily penalizing long instances, like Take-a-Break does, and doing nothing. Fairness of Borrow-and-Return approach is slightly worse than that of Take-a-Break, leading to the average of 18.9 failures for unfair projects, and 2.3 for fair projects.

10. CONCLUSIONS

Network testbeds are extensively used today, both for research and for teaching, but their resource allocation algorithms and policies have not evolved much since their creation. This paper examines the causes for resource allocation failures in Emulab testbeds and finds that 31.9% of failures could be avoided through: (1) providing better information to users about the cost of and the alternatives to their topology constraints, and (2) better resource allocation strategies. The remaining failures can be reduced to 25.3% of the original by applying a gentle fair-sharing strategy such as Take-a-Break or Borrow-and-Return. The main challenge in designing fair testbed allocation policies lies in achieving fairness, while being sensitive to human user needs for predictability and while nurturing heavy users that bring most value to the testbed. Our investigation is just the first of many that need to be undertaken to reconcile these conflicting, but important goals.

11. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1049758. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

12. REFERENCES

[1] Other Emulab Testbeds. `https://users.emulab.net/trac/emulab/wiki/OtherEmulabs`.

[2] Schooner WAIL (Wisconsin Advanced Internet Laboratory). `http://www.schooner.wail.wisc.edu`.

[3] R. Bajcsy, T. Benzel, M. Bishop, et al. Cyber Defense Technology Networking and Evaluation. *Communications of the ACM*, 2004.

[4] S. M. Banik, L. P. Daigle, and T. H. Chang. Implementation and Empirical Evaluations of Floor Control Protocols on Planetlab Network. In *Proceedings of the 47th Annual Southeast Regional Conference, ACM-SE 47*, 2009.

[5] T. Benzel. The Science of Cyber Security Experimentation: The DETER Project. In *Proc. of 2011 Annual Computer Security Applications Conference (ACSAC)*, December 2012.

[6] N. M. M. K. Chowdhury, M.R. Rahman, and R. Boutaba. Virtual network embedding with coordinated node and link mapping. In *Proc. of INFOCOM 2009*, pages 783–791, 2009.

[7] B. Chun, D. Culler, et al. Planetlab: an Overlay Testbed for Broad-coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[8] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[9] F. Hermenier and R. Ricci. How to Build a Better Testbed: Lessons From a Decade of Network Experiments on Emulab. In *TridentCom*, 2012.

[10] J.P. Herron, L. Fowler, and C. Small. The GENI Meta-Operations Center. In *Proc. of IEEE Conf. on eScience*, pages 384–385. IEEE Computer Society, 2008.

[11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, 2011.

[12] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[13] W. Kim, A. Roopakalu, K.Y. Li, and V.S. Pai. Understanding and Characterizing PlanetLab Resource Usage for Federated Network Testbeds. In *Proc. of the ACM SIGCOMM*, pages 515–532. ACM, 2011.

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

[15] J. Lu and J. Turner. Efficient mapping of virtual networks onto a shared substrate. *Washington University in St. Louis, Tech. Rep*, 2006.

[16] A. Medina, A. Lakhina, I. Matta, et al. BRITE: An Approach to Universal Topology Generation. In *Proc. of MASCOTS*, pages 346–. IEEE Computer Society, 2001.

[17] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, September 2000.

[18] T. Miyachi, K. Chinen, and Y. Shinoda. Automatic Configuration and Execution of Internet Experiments on an Actual Node-based Testbed. In *Proc. of Tridentcom*, 2005.

[19] J. Nakata, S. Uda, T. Miyaclii, et al. Starbed2: Large-scale, Realistic and Real-time Testbed for Ubiquitous Networks. In *Proc. of TridentCom*, 2007.

[20] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. *SIGCOMM Comput. Commun. Rev.*, 33(2):65–81, 2003.

[21] M. Ripeanu, M. Bowman, S. Chase, et al. Globus and Planetlab Resource Management Solutions Compared. In *Proc. of International Symposium on High Performance Distributed Computing*, pages 246 – 255, 2004.

[22] I. Stoica, S. Shenker, and H. Zhang. Core-stateless Fair Queueing: a Scalable Architecture to Approximate Fair Bandwidth Allocations in High-speed Networks. *IEEE/ACM Trans. Netw.*, 11(1):33–46, 2003.

[23] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: the Condor Experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[24] B. White, J. Lepreau, L. Stoller, et al. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of OSDI*, pages 255–270, December 2002.

[25] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration. *ACM SIGCOMM Comp. Comm. Rev.*, 38(2):17–29, 2008.