VF2x: Fast, efficient virtual network mapping for real testbed workloads

Qin Yin and Timothy Roscoe

Systems Group, ETH Zürich,
Universitätstrasse 6, CH 8092, Zürich,
{qyin,troscoe}@inf.ethz.ch

Abstract. Distributed network testbeds like GENI aim to support a potentially large number of experiments simultaneously on a complex, widely distributed physical network by mapping each requested network onto a share or "slice" of physical hosts, switches and links. A significant challenge is *network mapping*: how to allocate virtual nodes, switches and links from the physical infrastructure so as to accurately emulate the requested network configurations.
In this paper we present the VF2x virtual network mapping system. Based on the VF2 subgraph isomorphism detection algorithm designed for matching large graphs, VF2x incorporates several novel algorithmic improvements. These and careful implementation make VF2x perform more than two orders of magnitude faster than the fastest previously published algorithm.
In evaluating our algorithm, we generated an extensive test workload based on analysis of a 5-year trace of experiments submitted to the popular Emulab testbed, and using the current ProtoGENI topology. We use this test workload to evaluate the performance of VF2x, showing that it can allocate resources to virtual networks on a large testbed in a matter of seconds using commodity hardware.

Key words: Network testbeds, virtual network mapping, subgraph isomorphism

1 Introduction

Network testbeds are used by networking researchers to experiment with new protocols, applications, and systems, and are generally shared between users to reduce the considerable capital cost involved. A testbed consists of a set of physical resources – compute nodes (usually PCs), switches, links, etc. – together with a control plane which allocates resources. Users submit "requests" in the form of specifications of particular network configurations (nodes, topology, etc.) they would like to experiment with, when they require the resources, and for how long. In response, the control plane allocates, if possible, a set of resources to the users at the requested time.

Resources can be physical machines or links, but are often virtual "slices" of real resources. Multiplexing enables a plurality of diverse virtual networks to coexist on a shared physical network infrastructure [1–3]. Widely distributed testbeds like GENI [4] aim to support a potentially large number of experiments simultaneously by mapping each requested network onto a share or "slice" of physical hosts, switches and links.

A significant challenge in the context is *virtual network mapping:* how to map a virtual network (VN) topology with resource constraints to specific nodes and links in

a given physical network (PN) infrastructure so as to accurately emulate the network configurations requested by users. This mapping problem is difficult in theory and in practice due to the four properties summarized in [5]: *diverse topologies, resource constraints, online requests* and *admission control*. Even simplified variants of the mapping problem with relaxed properties turn out to be difficult: assigning nodes in a switched Ethernet-connected testbed without violating bandwidth constraints can be reduced to the NP-hard multiway separator problem [6], and even when the nodes are pre-selected, the link mapping problem for VN requests with link constraints is still NP-hard [5].

In this paper we present a new algorithm, VF2x, based on the VF2 subgraph isomorphism algorithm for matching large graphs [7]. VF2x performs network mapping more than two orders of magnitude faster than the previously-published vnmFlib (also based on VF2) but reduces solving time for near-worst-case problem instances through more careful implementation and several novel algorithmic changes: constructing a candidate queue, applying heuristic sorting, and introducing a new "timeout-and-relax" strategy.

In the rest of this paper, we first provide some background and related work of the problem, and then in Section 3 we describe the VF2x algorithm, discuss the implementation issues and algorithmic improvements, and evaluate its effectiveness through simulation. In Section 4.1 we explain the generation of a more realistic test workload derived from a 5-year trace of experiments submitted to the Emulab testbed. Using the generated test workload, we evaluate the performance of VF2x in Section 4. In Section 5 we make a conclusion.

2 Background and related work

Until recently, most testbeds supporting specification of network properties were centralized, such as Emulab [8] and DETER [9]. These testbeds emulate a variety of network topologies using a small number of high-port-count, high-capacity switches to approximate a physical crossbar between machines. This testbed mapping problem with bandwidth constraints has been proven to be NP-complete [10]. Simulated annealing has been successfully applied to this situation [11], but does not work well for large-scale virtualized network testbeds [12].

Other recent virtual network mapping algorithms make different assumptions in order to apply efficient heuristics to make the problem tractable.

Zhu and Ammar [13] assume unlimited physical network resources and then try to achieve low and balanced load on both physical nodes and links. Their algorithm, VNA-I, subdivides the general topology in the VN request into multiple small star topologies, and then exploits the flexibility of these small topologies.

Lu and Turner [14] consider the offline problem of mapping a virtual network with a backbone-star topology to the physical network with the aim of getting sufficient capacity to accommodate any traffic pattern specified by traffic constraints.

Yu et al. [5] propose a two stage solution to the problem. They first map the virtual nodes, and then map the virtual links using shortest path and multi-commodity flow (MCF) algorithms. The authors focus on improving the link mapping through *substrate support* for path splitting and migration. Razzaq and Rathore [15] also propose a two stage solution, but without the assumption of substrate support: virtual nodes are first

mapped as closely as possible to the physical network, then virtual links are mapped to the shortest paths which satisfy the demands.

Chowdhury et al. [16] propose algorithms which provide better coordination between the two stages. The virtual nodes are mapped to the physical nodes in a way that facilitates the mapping of the virtual links to the physical paths in the subsequent phase. The mapping problem is solved using a Mixed Integer Program (MIP) formulation. The authors also assume substrate support.

Several algorithms consider network migration and reconfiguration. Butt et al. [17] note the importance of differentiating physical network resources and argue that topology-aware mapping together with reoptimizing bottleneck mappings can improve the acceptance ratio. Schaffrath et al. [18] formalize the mapping problem as a linear mixed integer program and allow dynamic reconfiguration of existing mappings.

Of particular interest is the vnmFlib network mapping library implemented by Lischka and Karl [19]. Noting the relationship between the network mapping problem and subgraph isomorphism detection, they develop a backtracking algorithm based on the VF2 algorithm used in the pattern recognition community for finding subgraph isomorphisms in large graphs. VnmFlib maps virtual nodes and links in a single stage and achieves better and faster mappings than the two stage approach used by Yu et al. [5], and works especially well for large virtual networks with strong resource constraints.

While many algorithms (including VF2x) can produce optimal results (subject to some utility function), this frequently involves exhaustive search and is therefore expensive. Moreover, it is rarely required; in practice a near-optimal solution in reasonable time is preferable. Emulab's simulated annealing algorithm [11] is a good example of exploiting this property.

However, the Emulab approach suffers from two limitations: firstly, it does not always return the same result due to its use of randomness, which makes the debugging of the algorithm challenging. Furthermore, it sometimes fails to find a solution that satisfies all the constraints even if such a solution exists. This is illustrated by the "ugly" example included in the Emulab source code. In contrast, VF2x is deterministic in all its heuristics, and for this "ugly" mapping problem, can find a solution with no violation in constraints in considerably less time.

3 VF2x algorithm and implementation

Recent proposals for distributed testbeds such as GENI presume a federated, distributed physical infrastructure over which virtual networks ("slices") are instantiated. Low cross-sectional bandwidth is expected in such testbeds. At present, the various proposed GENI frameworks address the mapping problem in different ways: ProtoGENI [20] inherits the existing `assign` mapping algorithm from Emulab; ORCA-BEN [21] uses NDL-OWL ontology language to express substrate and a sequence of request and release operations, and relies on Jena RDF and OWL reasoning engines to perform topology mapping [22]. The cost of the mapping operation is not prohibitive since the BEN network is small.

We found vnmFlib [19] to be especially suitable for the network testbed mapping problem, as explained above. However, in the process of trying to apply vnmFlib di-

rectly to our testbed resource allocator, we find several limitations of both the algorithm and its implementation which we explain in detail below. This motivated us to develop VF2x, our own VF2-derived virtual network mapping algorithm.

3.1 Implementation decisions

As with vnmFlib, VF2x extends VF2 with semantic constraints on virtual node and link attributes (a key requirement for testbed applications) and a pre-defined distance value ϵ which, unlike VF2, allows virtual links to be mapped to multi-hop *paths* in the physical network of length at most ϵ.

Algorithm 1 shows the main skeleton of VF2x mapping algorithm. Variable *depth* maintains account of how many nodes have been successfully mapped so far. At each step, a new pair of nodes (n, m) is generated to try to match against each other. The `feasible`(n, m) function checks the syntactic (structure of the graph) and semantic (node and link attributes) feasibility of the mapping. If it succeeds, the mapping is remembered; otherwise, we try the next pair until we reach a dead end and `backtrack()`.

Algorithm 1: VF2x mapping algorithm

Input: Attributed graph G_{vn} and G_{pn}
Output: Mapping from $M(G_{vn})$ to G_{pn}

```
1  while |M| ≠ |G_vn| and depth > 0 do
2      (n, m) ← genpair();
3      if n < 0 or m < 0 then
4          backtrack();
5          depth ← depth − 1;
6      end
7      else if feasible(n,m) then
8          M(m) ← n;
9          depth ← depth + 1;
10     end
11 end
```

VF2x improves dramatically over vnmFlib through a combination of careful implementation and several algorithmic improvements. The implementation techniques can be summarized as follows: VF2x is implemented entirely in C (about 1,500 lines) based on the existing igraph library [23] implementation of VF2; VF2x supports modelling of network topologies as both directed and undirected graphs, while vnmFlib is restricted to topologies modeled only as directed graphs; unlike vnmFlib, VF2x avoids non-tail recursion; and finally, VF2x computes neighbors in a lazy manner and populates the adjacent neighbor table on demand, while vnmFlib recomputes neighbors every time.

By themselves, these implementation decisions make a dramatic difference in performance. Table 1 compares our VF2x implementation to vnmFlib using two simple mapping problems. To fairly compare the two implementations, we turn off the "splittable" option (to map the flow in a virtual link to multiple paths in the physical network), solve the mapping problems for directed graphs only, and set ϵ to 2 for both implementations. The hardware used is the same as that used in Section 4.

As seen from Table 1, even these better implementation techniques by themselves allow VF2x_base to already solve the problems in roughly *two orders of magnitude* less time than vnmFlib, without employing vnmFlib's sorting heuristics.

We are not the first to observe the performance issues with vnmFlib [24], but our goal here is to demonstrate that VF2-based algorithms in general can be implemented with good performance for reasonable problem sizes.

Table 1. VF2x vs. vnmFlib

	Example1[a]		Example2[b]	
	Steps	Time(ms)	Steps	Time(ms)
vnmFlib	3	2.786	20	18.994
VF2x_base	10	0.130	29	0.370
VF2x_candi	9	0.117	16	0.329
VF2x_candi_sort	4	0.089	14	0.050

[a] VN: 3 nodes and 3 links; PN: 6 nodes and 9 links.
[b] VN: 11 nodes and 10 links; PN: 14 nodes and 15 links.

Besides careful implementation, VF2x_base can be further optimized through algorithmic changes which prune/reorganize the search space further and thus lead to a better solution in less time.

Table 1 includes two of the algorithmic changes we use in VF2x: using a *candidate queue* and applying *heuristic sorting*. With these changes, the number of matching steps is reduced. Details of these and other algorithmic changes, and their performance impact, are given in the rest of this section.

3.2 VF2x with a candidate queue

To map a virtual node n to the physical network, rather than matching n against every node available from the physical network, VF2 matches only against those nodes within ϵ hops of the physical nodes already mapped.

VF2x uses a *candidate queue* to prune the search space still further. We first find all the nodes that n's neighbors have been mapped to. In the example shown in Figure 1, n's neighbors n_1, n_2 are mapped to m_1, m_2. Then, we calculate the intersection of $E(m_1)$ and $E(m_2)$. Here, $E(m)$ is the set of nodes at most ϵ hops away from m in the physical network. In fact, the only possible candidates for n are elements of this intersection and we call it n's *candidate queue*. In Figure 1, the candidates are the light gray ones in the shadowed area. The candidate queue improves solving time by reducing the number of candidates to match: `genpair()` only generates candidate pairs (n, m) where m belongs to the candidate queue. Table 1 shows the improvement obtained (the VF2x_candi row).

3.3 Heuristic sorting of network topologies

Like VF2, VF2x is based on a depth-first search strategy. The sequence of the candidate pairs generated depend on how the nodes are ordered in the virtual and physical net-

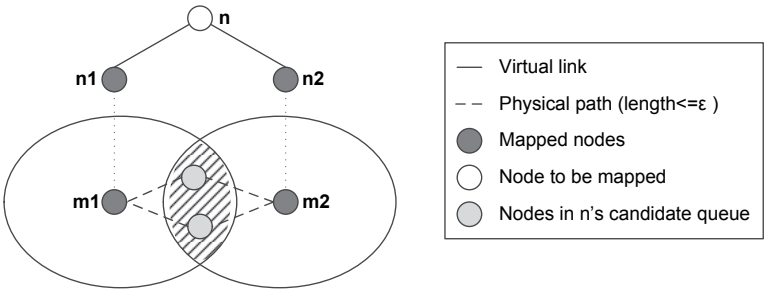


Fig. 1. The generation of n's candidate queue

works. This suggests that sorting the nodes according to heuristics can reorganize the search tree, and find a better solution in less time.

Different heuristics can be applied. vnmFlib sorts the generated candidate pairs based on the resource consumption of the virtual nodes; VF2x sorts graph nodes such that *switches* are ordered before *hosts* (whether a graph node is a switch or a host is indicated by the additional "type" attribute which is introduced for a testbed environment), furthermore high-degree nodes are sorted ahead of low-degree ones. Figure 8(b) in Section 4.2 shows the effectiveness of this heuristic of mapping more-constrained resources first.

3.4 Batching

Another extension VF2x makes for a testbed environment is to support batch solving: mapping several virtual network requests in one solving process even though they share the same physical network resources. The original VF2 algorithm does not support this and can only detect isomorphism for the disconnected subgraphs (VN requests) by mapping the virtual nodes from these requests to different physical nodes. With this extension, the same physical switch can be shared by different requests even though different switches in the same request are mapped to different physical ones.

3.5 Timeout-and-relax for ϵ

The vnmFlib authors propose two algorithms to pick an appropriate ϵ (recall that virtual links are only mapped to paths shorter than ϵ): VnmFlib-simple uses a fixed ϵ, while vnmFlib-advanced starts with $\epsilon = 1$, exhaustively searches for a mapping, and then successively increments ϵ until one is found or until ϵ reaches 10.

For smaller ϵ, fewer candidate nodes need be considered, but the tighter constraints can increase the required exploration of the search space. Informally, for small ϵ, if a solution exists, it is likely that it will be found early in the search.

This observation leads us to adopt an additional heuristic to mitigate occasional worst-case performance of VF2x (the general problem remains NP-hard), which we term *"timeout-and-relax"*. Instead of always running VF2x to completion, we start with small ϵ and impose a small time limit on the solving time, aborting the algorithm if it

exceeds the limit. We then increase ϵ and the value of timeout, and retry. This approach attempts a good compromise between shorter solving time of a smaller ϵ, and higher success rate of a larger ϵ. As shown in Figure 9, this strategy brings down the average solving time considerably compared to fixed ϵ and timeout values.

3.6 Algorithm analysis with synthetic workload

Using the GT-ITM tool, we generated 10 separate physical networks with 100, 150, 200, . . . , 550 nodes. Each node pair is randomly connected with probability 0.1. In this way, the physical network with 100 nodes are connected by around 500 links. We also generated 200 virtual networks. The size of the virtual networks is uniformly distributed between 2 to 20 and the nodes are connected with probability 0.5. The CPU capacity and the link bandwidth follow a uniform distribution ranging from 1 to 100 for the physical networks and ranging from 1 to 50 for the virtual networks. While synthetic, these randomly generated virtual and physical networks are of the same characteristics as those used by other researchers [5, 13, 19].

To evaluate each algorithmic change introduced above, we run a set of experiments to map 200 virtual networks to each of the 10 physical networks using variants of VF2x. In these experiments, ϵ is fixed and set to 2 and *timeout* is set to 5*s*. The VF2x algorithm variants compared are: VF2x_base, VF2x_candi and VF2x_candi_sort. VF2x_base is an extended VF2 which supports semantics constraints and maps virtual links to multi-hop paths. VF2x_candi extends VF2x_base with a candidate queue which further prunes the search space. VF2x_candi_sort applies heuristic sorting to VF2x_candi and sorts the nodes by the degree. Heuristic sorting reorganizes the search space in a way to find a better mapping in less time.

We ran all the experiments on a machine with an Intel Core2 Q6700 (quad-core 2.66 GHz) CPU and 4GB memory. The machine was running Ubuntu Linux 10.04 LTS Lucid Lynx, and we used version 0.5.4 of the *igraph* library.

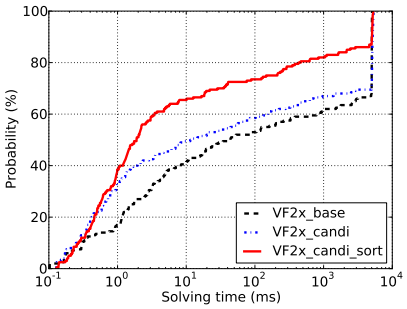


Fig. 2. Solving time CDF for VF2x variants

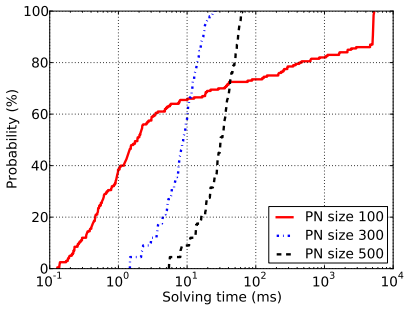


Fig. 3. Solving time CDF for various PN sizes

Figure 2 compares the three algorithm variants by mapping 200 virtual networks to the same 100-node physical network. As we can see, algorithmic improvements dramatically reduce the solving time and improve the mapping success rate: among 200

requests, within 5s, VF2x_base fails to map 65 requests, VF2x_candi fails to map 61 requests, while VF2x_candi_sort manages to bring the number down to 26. If we set the timeout to 500s, one request fails after exhausting the search space and 11 requests fails due to timeout.

Figure 3 plots the solving time of VF2x_candi_sort to map 200 randomly generated virtual networks to three different physical networks: 100 nodes with 483 links, 300 nodes with 4513 links, and 500 nodes with 12458 links. As shown in the figure, for more than 65% of the requests, VF2x uses less time to map them to the 100-node physical network. However, it takes substantially more time for about 20% of the requests, and even times out for 26 requests out of 200. Physical network size does play an important role in the algorithm and we investigate this subject further in Section 4.3. Moreover, within the same time limit, the *timeout-and-relax* strategy can improve the success rate significantly, as we show in Section 4.2.

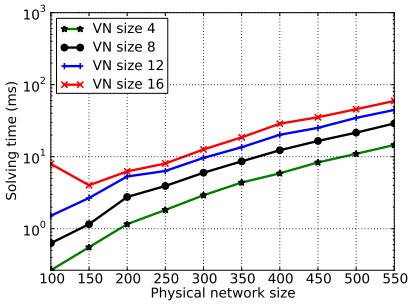


Fig. 4. Solving time CDF for various VN sizes

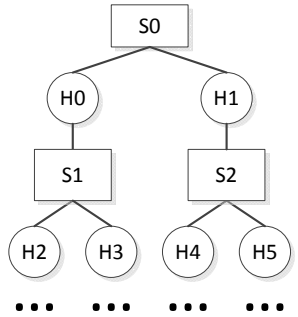


Fig. 5. Generated tree topology

Figure 4 plots the solving time of VF2x_candi_sort to map 4 virtual networks of 4 nodes / 4 links, 8 nodes / 12 links, 12 nodes / 34 links, and 16 nodes / 54 links separately to the 10 different physical networks. As the figure shows, the solving time increases roughly exponentially with the physical network size as well as the virtual network size. However, this is still acceptable for these problem sizes since VF2x is able to map the virtual network of 16 nodes / 54 links to the physical network of 500 nodes / 12458 links within 100ms.

4 Evaluation using real testbed workloads

Since networking testbeds are still relatively new, particular those which are distributed in nature and permit complex topological requests, it is not clear what kind of workload is representative for a testbed resource allocator. Much related work uses GT-ITM to simulate requests and testbed network topologies, as we have done to analyze different VF2x algorithm variants in Section 3.6.

However, these randomly generated undirected graphs cannot faithfully represent real testbed requests and infrastructures. Therefore, in this section, we describe the generation of a more plausible test workload and use it to evaluate VF2x. While we make no authoritative claims that the workload generated is representative, we do argue that it is based on plausible data and assumptions. The hardware used is the same as that used in Section 3.6.

4.1 Workload generation

Physical topology We extracted physical topology information from ProtoGENI[1] and build our physical topology with 627 nodes (including switches and hosts) and 1163 links based on the resource information from the Utah ProtoGENI site. We simplify the topology by summing up the bandwidth of all links from the same host to the same switch instead of including all "duplicated" links. This physical topology is used for all the experiments shown in this section.

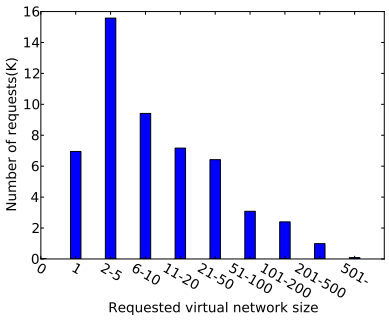


Fig. 6. Distribution of the requested VN size

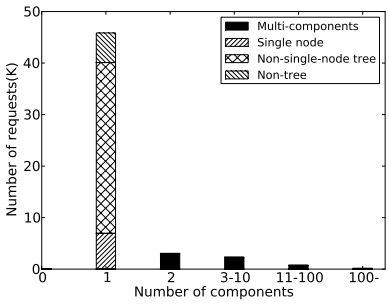


Fig. 7. Topology of the requested VNs

Request stream We then derive a sequence of virtual network requests from a complete sequence of Emulab topology requests for 5 years up to June 2007, essentially covering every experiment submitted to Emulab before that date. This trace consists of 23818 projects, 127586 topology requests among which 52089 are unique (the Emulab `.top` files are not identical). Figure 6 shows the distribution of the requested network sizes.

Temporal characteristics We assume that virtual network requests arrive dynamically. The users submit "requests" stating desired network resources, when they are needed, and for how long. In response, the testbed resource allocator will decide whether to accept the requests and if so, which specific physical resources will be allocated.

In order to construct a realistic request workload, two important temporal variables must be modeled: the request *arrival rate* and *duration* which defines for how long the user actively uses the allocated resources.

[1] http://www.protogeni.net/trac/protogeni

The arrivals of the requests are modeled as a Poisson process with a mean of λ requests per time window. As a reference, we roughly calculate the request arrival rate for the Emulab request trace. Based on the fact that Emulab received 127586 top requests over 4 years, we conclude that Emulab received 4 requests per hour on average. Using 4 as a base value, in the following experiment, we vary λ to increase or decrease offered load. This distribution model naturally does not take into account load spikes on the system (such as before conference deadlines).

Request duration is modeled as a Gamma distribution. which have long been used for modeling demand distribution in queuing systems [25, 26]. We choose the parameters $shape = 0.3$ and $scale = 20$, and generate 1000 duration values with the distribution as follows: 266 values are less than 10 minutes, 117 between 10 and 30 minutes, 90 between 0.5 and 1 hour, 459 between 1 and 24 hours, 68 between 1 and 7 days.

Based on the distribution models, we annotate the Emulab request stream with two time parameters – when to request resources and when to release them – and generate our workload from this. The workload is used in Section 4.4 to evaluate our testbed resource allocation with VF2x mapping algorithm.

4.2 Additional heuristics for the test workload

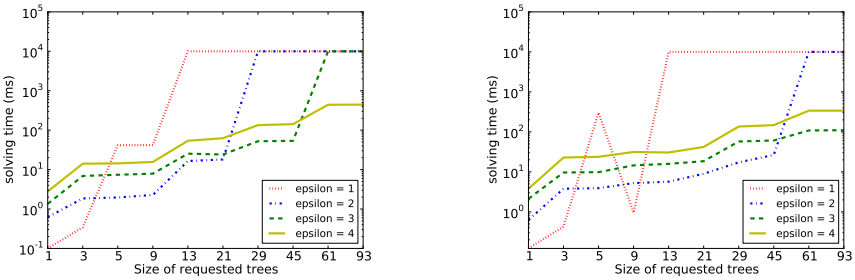
Analysis of the test workload Unlike the random virtual and physical networks generated by GM-ITM, nodes both in the physical topology described above and in our new request stream have an additional "type" attribute indicating whether they are switches or nodes. By analyzing the number of components of the requested virtual network graph, we found that the majority of the requests are connected graphs. Among them, most have tree topology and 20% of the trees are single nodes, as shown in Figure 7. Among these tree-topology requests, the structure in Figure 5 is common, where the hosts between switches are running DummyNet and acting as delays.

To explore the best heuristics we can apply to the test workload, we use the physical network described in Section 4.1, and generate a series of trees of various sizes following the structure pattern shown in Figure 5. We ran the VF2x mapping algorithm with different ϵ values (the maximum length of the physical path that a virtual link can be mapped to) and different sorting heuristics.

ϵ values are 1, 2, 3 and 4. Our baseline for comparison is "doing nothing": ordering virtual and physical nodes as they are declared in the specification; the second heuristic is the one we describe in Section 3: sorting virtual and physical switches and hosts in descending order of their degrees: first switches from high to low degree, then hosts from high to low degree (so that switches are mapped first).

By comparing Figures 8(b) and 8(a), we can see that the heuristic to order/map switches and hosts based on their degrees can reduce the solving time and increase the success rate within a solving time limit of $10s$.

These two experiments both show some interesting properties of ϵ. First, the smaller the ϵ value is, the fewer candidate nodes are to be considered, which results in less solving time. Second, the smaller the ϵ value is, the more constrained it is to find a satisfiable solution. This can result in longer solving time for the cases where a solution with smaller ϵ exists but requires more exploration of the search space, or when failing to find a mapping solution due to a timeout. Third, the smaller the ϵ value is, the more



(a) "Doing nothing" heuristic sorting (b) Degree-based heuristic sorting

Fig. 8. Solving time for trees of various sizes

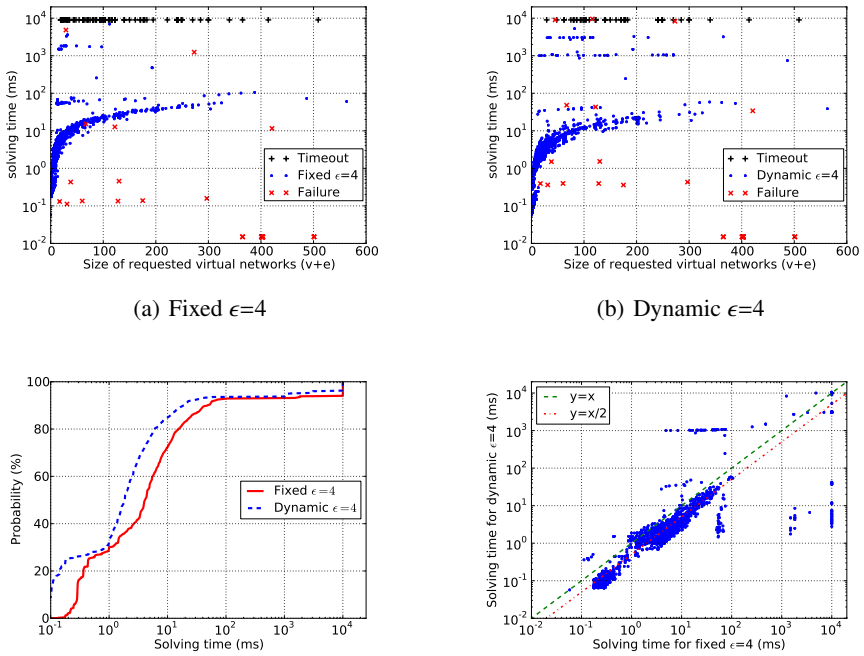
sensitive the solving time is to the order of the nodes. In Figure 8(b), a tree of size 5 takes more time to solve than tree of size 9. The 5-tree orders the nodes as $S0, S1, S2, H0, H1$ while the 9-tree orders them as $S1, S2, S0, H0, H1, H2, ...H5$ according to the sorting heuristic. These observations led us to adopt the "timeout-and-relax" technique: we start with small ϵ and small timeout, and increase ϵ and the value of timeout after each timeout. This approach aims at a good compromise between faster average solving time of smaller ϵ, and higher success rate of larger ϵ.

Applying heuristics to the test workload Having shown the effectiveness of the sorting heuristic, in the next experiment we use our request stream to investigate the effectiveness of the "timeout-and-relax" strategy. We randomly choose 2000 requests out of 52089 Emulab requests and individually map each of them to the *same* physical topology introduced in Section 4.1. We compare two different strategies: the simple strategy in which ϵ is fixed and set to 4 with $timeout = 10s$; the timeout-and-relax strategy in which ϵ is dynamic with different timeouts: $(\epsilon = 2, 1s), (\epsilon = 3, 2s), (\epsilon = 4, 7s)$. In both experiments, the sorting heuristic described above is applied.

Figure 9(a) and Figure 9(b) plot the solving time against the virtual network size. Here, the virtual network size is defined as the total number of nodes and links. Figure 9(c) depicts the solving time CDF for fixed $\epsilon = 4$ and dynamic $\epsilon = [2, 3, 4]$. From these figures, we can see that: with timeout-and-relax, VF2x solves most (more than 85%) mapping problems in less time, while it also achieves a higher success rate and the number of timeout cases decreases from 119 to 74.

Figure 9(d) shows this comparison in more detail by plotting the solving time with *fixed* ϵ against the solving time with *dynamic* ϵ. As we can see, most of the mapping problems are solved in less time with *dynamic* ϵ and a majority of them are twice as fast. The dots plotted in the center right of the figure are the 45 requests which fail with fixed ϵ but succeed with dynamic ϵ. The dots plotted in the top right corner are the 74 requests (out of 2000) which fail to be mapped within 10s.

The algorithm used in all the next set of experiments is VF2x_candi with the above sorting heuristic and timeout-and-relax strategy applied.



(a) Fixed ϵ=4 (b) Dynamic ϵ=4

(c) Solving time CDF (d) Solving time: fixed vs. dynamic ϵ

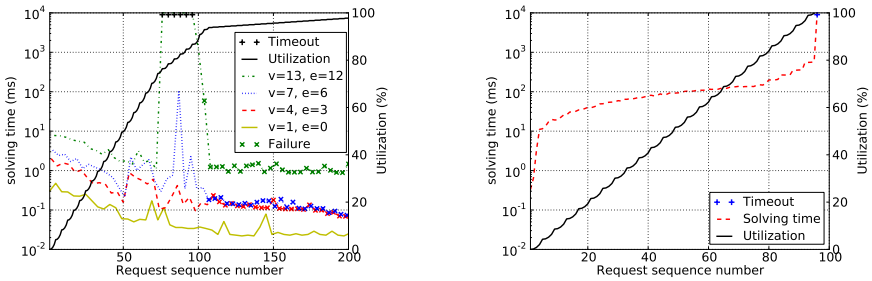
Fig. 9. Solving 2000 Emulab requests

4.3 Repeated requests to exhaust physical resources

In this section, we use a simple test workload to investigate the *sequential* and *global* allocation strategies. The test workload is a round-robin sequence of four requests from Emulab request history which contain one node, 4 nodes, 7 nodes and 13 nodes separately. For the two allocation strategies, we investigate the solving time to map physical resources to the successive requests, as well as the total proportion of physical resources that are allocated. In the experiment, we deliberately repeat the request in order to investigate, for one specific request, the influence of the shrinking physical resource pool.

With the sequential solving strategy, we allocate resources to each request, mark the resources allocated as unavailable and never reallocate them, and continue until we exhaust all physical resources. During this process, physical resource utilization is increasing and the free physical resources are decreasing. Figure 10(a) plots the relationship between the solving time for one specific request and the shrinking physical network. It shows how, initially, the solving time decreses since the available resources are abundant the search space is shrinking. This trend stops in the middle when the platform has much fewer resources available and the solver has to nearly exhaust the (smaller) search space to find a solution.

With the global solving strategy, whenever we process a new request, we reassign resources to all the requests seen so far in one mapping process, without respecting any



(a) Solving 4 requests repeatedly

(b) Solving accumulated requests

Fig. 10. Exhausting physical resources

previous resource assignment. As we have observed in [27], with global solving, the execution time increases exponentially as more constraint programs are solved simultaneously in the ECLiPSe solver we used [28]. To our surprise, with the batch solving described in Section 3.3, the solving time in VF2x does not increase exponentially. As shown in Figure 10(b), after a sharp increase at the beginning, the execution time increases linearly. This is due to the fact the requests are repeating themselves. At the beginning, VF2x takes some time (still within 10ms) to find mappings for the first 4 requests. Then, since successive requests are identical, they reuse the switch mappings generated beforehand and this significantly prunes the search space. This shows that batch solving is a surprising promising technique for solving a set of structurally recurring requests efficiently.

4.4 Long trace behavior

In this section, we use the full test workload generated in Section 4.1 to evaluate our testbed resource allocator with the VF2x mapping algorithm.

We run the simulations for 200 time windows with different request arrival rates: $\lambda = 4, 8, 16$ which correspond to $800, 1600, 3200$ requests in each simulation instance. The generation of the request workload and the physical network topology used in the simulations is described in Section 4.1. We use a client simulator to send resource requests and release requests to the allocator. Upon receiving a resource request, the allocator runs VF2x to decide whether or not to accept the request, if yes, it decides which physical resources to allocate and removes them from the physical network; upon receiving a release request, the allocator will revoke the resources and return them to the physical network.

In Figure 11, the dotted line depicts the utilization of the physical network (dividing the number of allocated hosts by the total number of physical hosts) under a request arrival rate of $\lambda = 4$. The solid line shows the utilization in an "ideal" scenario where all the requests are accepted and satisfied. Of 800 requests, the allocator refuses 69 due to resource insufficiency and fails to map 7 requests to the physical network within $10s$.

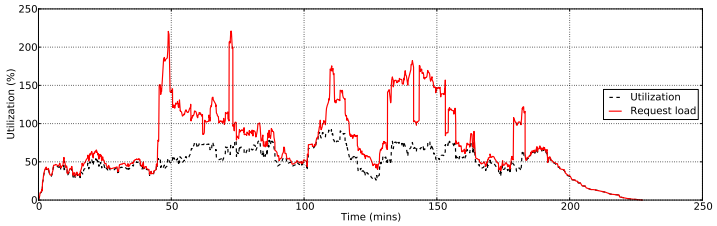


Fig. 11. Executing the generated trace of arrival rate $\lambda = 4$

For these timeout cases, we can relax their resource constraints and retry the mapping with VF2x.

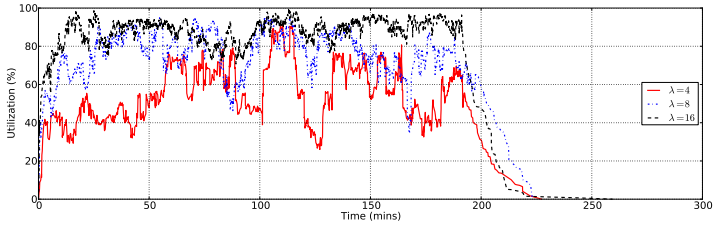


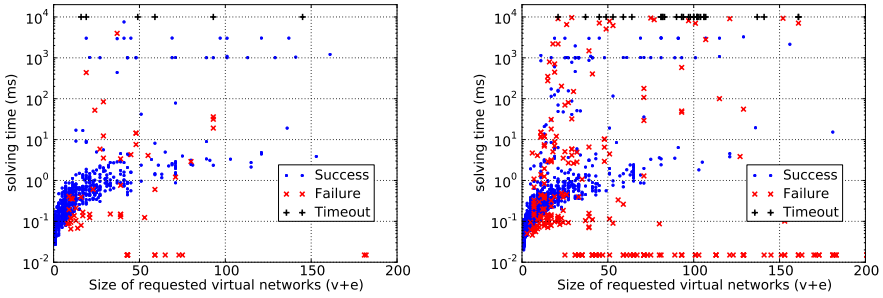
Fig. 12. Executing the generated trace of arrival rate $\lambda = 4, 8, 16$

Figure 12 shows the physical network utilization under different request loads. After some warm-up time, the system enters a more steady state as leases are requested and released "evenly". In the end, when no more requests are received, the allocator gradually releases all the resources. By comparing the lines for different λ, we can see that the higher the load, the more likely it is that the allocator can fit small requests into the network and achieve higher utilization.

Figure 13 shows the solving time (including failures and timeouts) of VF2x in mapping the requests of different sizes to the changing physical network in the continuous resource allocation process. As we can see, with a bigger λ value, the provider receives more requests in a given time slot, and this results in a higher failure rate because of its limited capacity.

5 Conclusion

We have presented VF2x, a new virtual network mapping algorithm based on the VF2 subgraph isomorphism detection algorithm. VF2x supports semantic constraints and is able to map virtual links to multi-hop paths. Several algorithmic improvements are introduced: using a candidate queue, applying heuristic sorting and adopting timeout-and-relax strategy.



(a) $\lambda = 4$ (b) $\lambda = 8$

Fig. 13. Executing the generated trace of arrival rate $\lambda = 4, 8$

The design of suitable network mapping algorithms for network testbeds (and, indeed, similar scenarios such a datacenter networks and cloud facilities) is still in its infancy, and the idealized problem is, in theory, computationally hard.

However, we have shown that a combination of good choice of algorithm, pragmatism with regard to timeouts, and careful attention to implementation details can provide a fast way of embedding virtual networks in a physical substrate – when combining algorithmic optimizations, implementation, and heuristics like sorting and timeout-and-relax, VF2x is over two orders of magnitude faster than previous systems like VnmFlib.

References

1. Thomas Anderson, Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the internet impasse through virtualization. *IEEE Computer Magazine*, 38:34–41, April 2005.
2. Jon Turner and David Taylor. Diversifying the internet. In *GLOBECOM*, pages 755–760. IEEE, 2005.
3. N.M. Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. *Comput. Netw.*, 54:862–876, April 2010.
4. GENI. http://www.geni.net/.
5. Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *SIGCOMM Comput. Commun. Rev.*, 38:17–29, March 2008.
6. David G. Andersen. Theoretical approaches to node assignment. Unpublished Manuscript, December 2002.
7. Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26:1367–1372, October 2004.
8. Jay Lepreau. Emulab - Network Emulation Testbed. http://www.emulab.net/.
9. The DETER Testbed: Overview. www.isi.edu/deter/docs/testbed.overview.pdf.
10. Rick McGeer, David G. Andersen, and Stephen Schwab. The network testbed mapping problem. In *Proc. 6th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*, May 2010.

11. Robert Ricci, Chris Alfeld, and Jay Lepreau. A solver for the network testbed mapping problem. *SIGCOMM Comput. Commun. Rev.*, 33:65–81, April 2003.
12. Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 113–128, Berkeley, CA, USA, 2008. USENIX Association.
13. Yong Zhu and Mostafa H. Ammar. Algorithms for assigning substrate network resources to virtual network components. In *INFOCOM*. IEEE, 2006.
14. J. Lu and J. Turner. Efficient mapping of virtual networks onto a shared substrate. Technical Report WUCSE-2006-35, Washington University, September 2006.
15. Adil Razzaq and Muhammad Siraj Rathore. An approach towards resource efficient virtual network embedding. In *Proceedings of the 2010 2nd International Conference on Evolving Internet*, INTERNET '10, pages 68–73, Washington, DC, USA, 2010.
16. N. M. Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Virtual network embedding with coordinated node and link mapping. In *INFOCOM*, pages 783–791. IEEE, 2009.
17. Nabeel Farooq Butt, N. M. Mosharaf Kabir Chowdhury, and Raouf Boutaba. Topology-awareness and reoptimization mechanism for virtual network embedding. In *Networking*, volume 6091 of *LNCS*, pages 27–39. Springer, 2010.
18. Gregor Schaffrath, Stefan Schmid, and Anja Feldmann. Generalized and resource-efficient vnet embeddings with migrations. *CoRR*, abs/1012.4066, 2010.
19. Jens Lischka and Holger Karl. A virtual network mapping algorithm based on subgraph isomorphism detection. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, VISA '09, pages 81–88, New York, NY, USA, 2009. ACM.
20. ProtoGENI. http://www.protogeni.net/trac/protogeni.
21. Jeff Chase. ORCA control framework architecture and internals. Technical report, Duke University, September 2009.
22. Ilia Baldine, Yufeng Xin, Daniel Evans, Chris Heerman, Jeff Chase, Varun Marupadi, and Aydan Yumerefendi. The missing link: Putting the network in networked cloud computing. In *International Conference on the Virtual Computing Initiative (ICVCI 2009)*, October 2009.
23. The igraph library. http://igraph.sourceforge.net/.
24. Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 15:1–12, New York, NY, USA, 2010. ACM.
25. J.R. Smith, P.A. Golden, and B. Appleton. *Airline: a strategic management simulation*. Prentice Hall, 1991.
26. Raj K. Jain. The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling. page 720, April 1991.
27. Qin Yin and Timothy Roscoe. A better way to negotiate for testbed resources. In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2011)*, Shanghai, China, July 2011.
28. Krzysztof R. Apt and Marg G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.