Available online at www.sciencedirect.com

ScienceDirect

Computer Networks 00 (2013) 1–20

Procedia Computer Science

Flow-based Partitioning of Network Testbed Experiments

Wei-Min Yao, Sonia Fahmy

Department of Computer Science, Purdue University, West Lafayette, IN 47907–2107, USA

Abstract

Understanding the behavior of large-scale systems is challenging, but essential when designing new Internet protocols and applications. It is often infeasible or undesirable to conduct experiments directly on the Internet. Thus, simulation, emulation, and testbed experiments are important techniques for researchers to investigate large-scale systems.

In this paper, we propose a platform-independent mechanism to partition a large network experiment into a set of small experiments that are *sequentially* executed. Each of the small experiments can be conducted on a given number of experimental nodes, e.g., the available machines on a testbed. Results from the small experiments approximate the results that would have been obtained from the original large experiment. We model the original experiment using a *flow dependency graph*. We partition this graph, after pruning uncongested links, to obtain a set of small experiments. We execute the small experiments iteratively. Starting with the second iteration, we model dependent partitions using information gathered about both the traffic *and* the network conditions during the previous iteration. Experimental results from several simulation and testbed experiments demonstrate that our techniques approximate performance characteristics, even with closed-loop traffic and congested links. We expose the fundamental tradeoff between the simplicity of the partitioning and experimentation process, and the loss of experimental fidelity.

Keywords: Network simulation, network emulation, network testbeds

1. Introduction

Understanding the behavior of large-scale systems is critical when designing and validating a new Internet protocol or application. Consider the example of studying the impact of a large-scale Distributed Denial of Service (DDoS) attack utilizing a massive botnet. The attack against Estonia is a well-publicized example of this [1]. It is important to explore defenses against this attack under realistic scenarios, but it is undesirable to perform security experiments on the operational Internet.

Since it is often infeasible to perform experiments directly on the Internet or build analytical models for complex systems, researchers often resort to simulation, emulation, and testbed experiments. Simulators scale through abstraction. For example, the popular network simulator ns-2 [2] uses simplified models for physical links, host operating systems, and lower layers of the network protocol stack. Researchers can easily simulate a network topology with hundreds of nodes and links on a single physical machine. Naturally, the simplification of hardware and system properties can adversely impact the fidelity of experimental results [3]. In contrast to simulators, network emulators mostly use the real hardware and software. This allows experimenters to run their unmodified applications. While emulation can provide higher fidelity, scalability is a challenge. Emulation testbeds such as Emulab [4] and the popular cyber-range DETER [5] include a limited set of physical machines that are shared among several users. For fidelity reasons, many testbeds allocate resources conservatively; for example, using a one-to-one mapping between hosts in an experimental topology and machines in the testbed. This implies that if the number of experimental nodes exceeds the number of machines currently available in the emulation testbed, the experiment cannot be executed.

Scalability of network simulation and emulation has been extensively studied in the literature. Ideas from parallel computing [6] and resource multiplexing [7] have been adopted to increase experimental scale. For discrete-event simulators [6, 8], events are distributed among multiple machines to reduce the simulation time and required hardware resources per machine. Additional overhead for inter-machine synchronization and communication depends on how events are partitioned. Emulation testbeds can scale, to a certain extent, via mapping multiple virtual resources onto available physical resources. For example, the Emulab testbed [7] can support experiments which are 20 times larger than the testbed. This network testbed mapping problem is NP-hard [9]. The main challenge, especially with DDoS experiments, is that the mapped experiment can overload physical resources (e.g., CPU or memory of a physical machine) and lead to inaccurate experimental results [3].

In this paper, we present a more versatile solution to the experimental scalability problem. We divide a large network experiment into multiple smaller experiments, each of which is manageable on a testbed. We conduct the smaller experiments *sequentially* on the testbed. The key contributions of our work include (1) a novel approach and tool to automatically partition a large experiment into sequential small experiments based on *network flows and the dependencies among them*, (2) an iterative approach to model the interacting small experiments, and (3) comparisons of different approaches via both simulations and DETER testbed experiments.

Our proposed method, *flow-based scenario partitioning (FSP)*, is *platform-independent* because it does not require any modifications to the simulation, emulation, or physical testbed to be used. Larger experiments can be conducted on a resource-limited experimental platform when partitioned by FSP. FSP can be integrated with most existing scaling solutions. FSP can also be used to analyze dependencies and tune an experiment, even when the experiment is small enough to fit onto a testbed. The partitioning of a large experiment via FSP increases the experimental scalability, at the expense of fidelity. FSP is most appropriate for network experiments that use static routing and focus on coarse-grained performance, such as the average throughput of a data flow.

The remainder of this paper is structured as follows. Section 2 defines our notation and assumptions. Sections 3 to 6 explain our proposed method, FSP. Sections 7 to 9 describe the experiments used to validate FSP and compare it to downscaling approaches. Section 10 discusses the limitations, scalability, and applications of FSP. Section 11 summarizes related work. We conclude in Section 12.

2. Background

In this paper, we focus on performance of data flows. Hence, the term *network experiments* will be used to refer to data plane experiments. A network experiment is represented by a *network scenario*; the smaller experiments generated by our method are referred to as *sub-scenarios* or *partitions*. A network scenario includes the *network topology* and the *flow* information.

We model the network topology as a graph $G = (V, E)$ with vertex set V, representing the routers and end hosts in the network, and edge set E, representing the links in the network. $|V|$ and $|E|$ denote the number of vertices and number of edges in the graph, respectively. The flow information describes all traffic in the experiment. Each flow in F includes information about the network application that generated the traffic flow (e.g., FTP, HTTP), the parameters of the traffic of that application (e.g., request inter-arrival times, file sizes), and the source, destination, route, and directionality of the flow. Traffic flowing between the same source and destination nodes is grouped into the same macro-flow. Unless otherwise specified, the term *flow* in this paper refers to a macro-flow. Depending on the type of network application that generates a flow, the flow can be *open-loop* (e.g., unresponsive CBR UDP flow) or *closed-loop* (e.g., TCP flow). The route of a flow is a sequence of hops from its source to its destination node. The *directionality* of the traffic indicates whether it is unidirectional or bidirectional.

We make the following assumptions to simplify the exposition. First, routes in the network are assumed not to change during the course of the experiment. Second, we assume symmetric routes for bidirectional flows, i.e., the packets in both directions traverse the same route. Third, flows traversing the same router but *not sharing any link* are independent. For example, a flow from port 1 to port 2 of a router does not interfere with a flow from port 4 to port 3. Although this is not always true for low-end routers [3], the assumption holds for typical mid to high-end routers, and in most network simulators, e.g., ns-2 [2]. We can easily relax these assumptions at the expense of higher complexity of the flow partitioning process.

Our approach is based on several simple but important observations. First, a large-scale network experiment involves many nodes and flows but not all flows directly interact with each other, e.g., by sharing a physical link. If we can identify the parts of the network that are not strongly tied, we can initially examine each part independently.

The second observation is that even though a network scenario may contain many flows, researchers are often only interested in fine-grained performance of a few of the flows. The rest of the flows may be used to generate network workload, and are considered as background traffic. For example, when studying performance of a web server, we can set up an experiment with several background FTP flows. Since we are interested in the web server, we need detailed measurements for HTTP connections such as request/response time. We may not need to measure file transfer times for the FTP flows, and the precise arrival processes of these flows are not important as long as they possess certain statistical properties (e.g., average throughput is 1 Mbps or FTP file request frequency is 1 file per second).

3. Overview of FSP

Our proposed method, which we refer to as *flow-based scenario partitioning* (FSP), does not partition the network nodes, like partitioning approaches for parallel and distributed simulation [6] do. This is because our goal is to conduct experiments for each sub-scenario *independently* on a testbed. If we partition the network topology directly as illustrated in Fig. 1(a), some flows may traverse two or more partitions, and we would need to concurrently execute and synchronize more than one sub-scenario experiment. Instead of partitioning the nodes in the topology, we partition the *flows* in the network scenario as illustrated in Fig. 1(b).

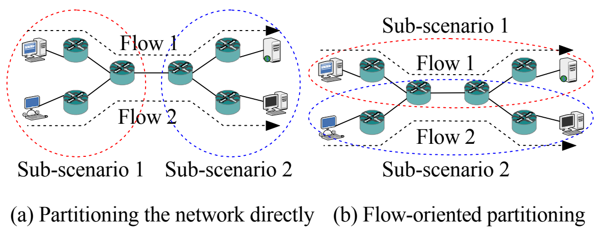


Figure 1. Direct network partitioning versus flow-based network partitioning.

FSP consists of two phases. In the first phase, we automatically split the input scenario into several sub-scenarios. We build a flow dependency graph (FDG) to model the relationship between flows. Each connected component in the FDG constitutes a partition of the graph, which represents a sub-scenario. If any of the connected components is too large for the resources available for an experiment, i.e., it contains too many hosts and routers, we apply a modified recursive bisection algorithm [10] to cut these connected components into partitions that meet the resource constraints. The quantity *maxNode* denotes the upper bound on the number of nodes that can be supported in each sub-scenario. Observe that in emulation testbeds such as Emulab and DETER, we need to take into account additional required testbed nodes, e.g., to emulate link delays, when computing *maxNode*. Section 4 gives the details of this phase.

In the second phase, we conduct experiments for each sub-scenario and collect measurements for the flows of interest. If sub-scenarios do not *interact* with each other, i.e., they are disjoint components in the FDG, we simply conduct experiments for each sub-scenario independently. In most cases, however, there will be interactions among sub-scenarios, i.e., there are edges in the FDG that cross partition boundaries. To account for these interactions, we must conduct experiments iteratively. In the first iteration, we study each sub-scenario independently and collect packet traces that capture information related to dependent flows in interacting sub-scenarios. In each subsequent iteration, we incorporate information *computed from* the traces in the previous iteration (via tools like [11, 12, 13]) into interacting sub-scenarios. In the final iteration, we collect the desired measurements, such as the FTP transfer completion time or HTTP response time. Section 5 gives the details of this phase. The overall FSP approach is summarized in Algorithm 1.

4. Phase I: Scenario Partitioning

In the first phase of our approach, the input network scenario is partitioned into sub-scenarios. By carefully selecting which flows to include in each sub-scenario, flows can have as little interaction as possible with flows in other sub-scenarios. Given a network scenario (S) which includes the network topology ($G = (V,E)$) and flow information (F), we divide S into sub-scenarios ($S_1, S_2, \cdots, S_j$) such that the number of hosts and routers in each of the sub-scenario (S_i) is $\leq$ *maxNode*. An example of this FDG construction and partitioning (tiling) process is illustrated in Fig. 2.

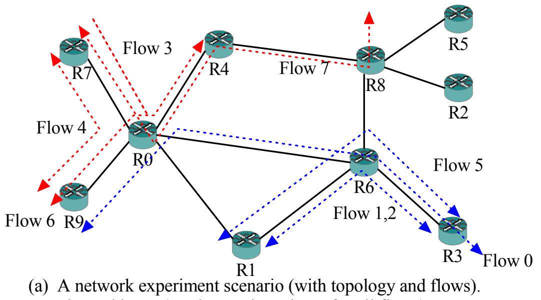
Algorithm 1 Flow-based Scenario Partitioning (FSP)

```
FlowBased Partition($network, flows, maxNode$)
Input: A network scenario with topology ($network$), flow information ($flows$), and $maxNode$
Output: Estimate of results for original scenario
    // Phase 1: Partition the input network scenario
 1: Construct a flow-dependency graph ($fdg$) as described in Sec. 4.1
 2: Partition the $fdg$ into multiple partitions ($Parts$) where each partition contains no more than $maxNode$ network nodes (Sec. 4.2)
    // Phase 2, Iteration 1:
    // Collect traces in case of interaction among partitions.
 3: for $P \in Parts$ do
 4:     Conduct experiment for sub-scenario $P$
 5:     for each $f \in P$ do
 6:         for each $f' \in fdg.neighbors(f)$ do
 7:             if $f' \notin P$ then
 8:                 Collect $f$'s packet traces on $f.path \cap f'.path$
    // Phase 2, Iterate for interacting partitions:
    // Incorporate traces collected from first iteration and acquire experimental results.
 9: repeat
10:     for ($P \in Parts$) do
            // For interacting partitions only
11:         for each $f' \notin P$ do
12:             $sharedPath \leftarrow$ ($f'$'s path) $\cap$ (links in P)
13:             if $sharedPath \neq \emptyset$ then
14:                 Import model (e.g., Tmix) of $f'$ on $sharedPath$ (in $P$).
15:                 Conduct experiment for sub-scenario $P$
16: until Interacting flows propagate their influence (Sec. 6.2)
```

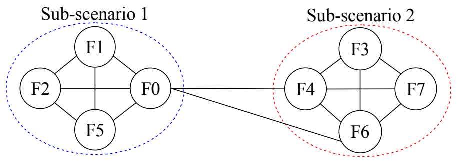
4.1. Flow Dependency Graph (FDG) Construction

Our first step is to identify the relationship among flows in the network scenario. We consider two flows to be *directly dependent* if they both compete for the same resources such as network buffers or link bandwidth. In our current implementation, two flows directly depend on each other if they share at least one common link in the network in the same direction during a time window. We model this relationship using a flow dependency graph (FDG).

A flow dependency graph, FDG = (F_V, F_E, fn_e), is a weighted graph with vertex set F_V, edge set F_E, and edge weight function fn_e. A vertex in F_V represents a flow in the given scenario S and an edge (f_1, f_2) in F_E denotes that flows f_1 and f_2 are *directly* dependent on each other. All FDG edges are bidirectional. Note that two flows u and v may impact each other if there is a path from u to v in the FDG. This follows from the transitivity property of dependence. Unless two flows belong to different connected components in the FDG, they may affect each other in the experiment. The weight of an edge fn_e is set to the number of nodes shared by the two directly dependent flows. We evaluate this choice experimentally in Section 7.



(a) A network experiment scenario (with topology and flows). The end hosts (senders and receivers for all flows) are not shown in this figure.



(b) The flow dependency graph of (a).

Figure 2. Example of transforming a network scenario into a flow dependency graph. According to the partitioning in (b), the network scenario in (a) can be divided into two sub-scenarios with five routers each. Sub-scenario 1 contains routers $\{R0, R1, R3, R6, R9\}$ and flows $\{F0, F1, F2, F5\}$. Sub-scenario 2 contains routers $\{R0, R4, R7, R8, R9\}$ and flows $\{F3, F4, F6, F7\}$.

When constructing an FDG, we insert all flows in scenario S as vertices in F_V. We then insert edges into the FDG based on the routes and the directions of the flows. Recall that an edge between two vertices in the FDG indicates that the two flows will compete for resources. We need to predict the existence of such competition *without actually conducting the original large experiment*. Unfortunately, such a priori prediction is challenging, especially for closed-loop flows. Therefore, we resort to using flow path and direction. For example, if the set of flows that will traverse link l *at any time during the experiment* is $\{a, b, c\}$, we insert the three edges, (a, b), (b, c), and (a, c), into the FDG. Of course, even though flows a, b, and c all traverse link l, it is possible that only a single flow traverses link l at any given time.

Edge pruning. Extra FDG edges unnecessarily limit our ability to partition the experiment. Therefore, we prune edges in cases of underload. Previous work [14] shows that when flows are competing for the same link bandwidth, if the capacity of the link is large enough, i.e., there are no packet drops and only a few packets in the buffers, each flow will utilize this link as if there are no other flows on the same link. Therefore, we identify "uncongested links" in the network and remove these links from the FDG. (An edge in the FDG represents a set of links in the network that are shared by two directly dependent flows, and an edge is removed from the FDG when it contains only uncongested links.)

Using flow path information and physical link capacity, we can estimate an approximate upper bound on the workload that can appear on any single link. If the upper bound is less than the physical link capacity, we mark this link as "uncongested" and remove it from the FDG. For example, in Fig. 3, assuming that there are three flows (from hosts a, b, and c to host x) in the network, the aggregate throughput of the three flows on link x cannot exceed 30 Mbps. Since the physical capacity of link x exceeds 30 Mbps, we predict that link x will not be significantly congested during the experiment. We have implemented an automated tool to identify such links and delete them from the FDG.

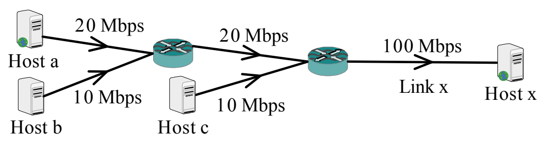


Figure 3. The solid lines are (unidirectional) physical links and the link capacity is shown next to the link. Regardless of the type of flows, the aggregate throughput on link x cannot exceed 30 Mbps.

4.2. Partitioning the FDG

The FDG created in the previous step may have several connected components, and one or more of these components may be too large to fit onto an experimentation platform. The size of a component is defined as the number of routers and hosts used by the flows in that component. Our next step is to divide large components such that the size of each component is smaller than the given *maxNode* value. After this partitioning process, each flow in the original network scenario is included in only a single FDG partition (sub-scenario). For example, the connected component in Fig. 2(b) is partitioned into two sub-scenarios. Ideally, we would like to partition the FDG such that there is as little interaction as possible among the sub-scenarios. Since the optimal solution to this graph partitioning problem is computationally intractable, we employ an approximation that repeatedly computes two-way partitions (i.e., bisections) of the graph [15].

We leverage the greedy graph growing partitioning approach (GGGP) [16]. GGGP is a simple approach to bisect a graph. It starts from a vertex and grows its region in a greedy and breadth-first fashion. While the number of nodes in the region is smaller than half of the nodes in the graph, the algorithm will add new vertices into the region. In each iteration, a vertex (from the vertices that are adjacent to the current region) is selected if moving it into the current region results in the smallest edge-cut between the two regions. Since the algorithm is sensitive to the choice of the initial vertex, we randomly select the initial vertex and repeat the process multiple times. The partition with the smallest edge-cut is selected as the final output. The number of initial vertices to be selected is configurable in our implementation and the default value is set to five– the default in [16]. We found that five initial vertices are sufficient in all our evaluation experiments, but users may select a higher value to produce better partitions when the input FDG is large.

5. Phase II: Sub-scenario Experiments

After determining the partitions (sub-scenarios) (S_1, $S_2, \cdots, S_j$), our goal is to obtain the desired performance measurements, such as the goodput of flows, from these sub-scenarios. Without loss of generality, let S_1 be a sub-scenario containing the flows of interest, and let there be m sub-scenarios that *interact* with S_1, i.e., they belong to the same connected component in the FDG. We define a *shared link* as a link in the original network topology that is shared by flows in more than one sub-scenario. Assume that there are n shared links in S_1. In order to obtain measurements for the flows of interest (in S_1), we need to generate workloads on these n shared links for flows in $\{S_2, \cdots, S_m\}$ (since these flows are not in S_1). To achieve this, we propose to conduct the experiments in multiple iterations.

5.1. First Iteration

In the first iteration, we conduct experiments independently for each sub-scenario. For interacting sub-scenarios, there will be flows *missing* on the shared links, compared to the original large scenario. For example, Fig. 2 contains two sub-scenarios (S_1 and S_2). When we conduct the experiment for S_1 in the first iteration, flow 4 and flow 6 will not generate any workload on link R0-R9 since they are not included in S_1, and flow 0 will also be missing from S_2. As a result, the measurements in this iteration may be dramatically different, compared to those in the original scenario, e.g., the throughput of flow 0 may increase since flow 0 does not need to compete for bandwidth with flow 4 and flow 6. Therefore, we collect packet traces for these interacting flows, in order to later use information computed from these traces to generate workload that models the missing flows.

5.2. Subsequent Iterations

In subsequent iterations, we sequentially conduct experiments for each sub-scenario that interacts with others, but we now incorporate information computed from the previous iteration to model its interacting sub-scenarios. Multiple iterations may be required to propagate the interactions among flows. Measurements collected in the final iteration approximate the results of the original experiment. The number of required iterations varies depending on the partitioning of the FDG. As we will discuss in Section 6, in most cases, FSP requires no more than five iterations – typically two iterations suffice.

5.2.1. Workload and network modeling

Since many flows are *closed-loop*, we **cannot simply replay** the collected packet traces on the shared links. We must model the workload of flows at the application level, and model the **conditions experienced by these flows in the non-shared links** in interacting sub-scenarios. This is crucial so that the missing flows are *no more aggressive* than they would have been in the original unpartitioned experiment. In other words, the conditions in the network, such as congestion level and delays experienced by the missing flows during the second iteration, must mimic the original unpartitioned experiment, so that the transport and application layers at the end hosts can react similar to their reaction in the original experiment. This is critical when the flows are bottlenecked in another partition or their propagation delays in another partition are high.

In our experiments in this paper, we investigate the use of the three tools (1) Tmix [11], (2) Harpoon [13], and (3) Swing [12] to (i) process packet traces collected during an iteration, and (ii) model non-shared network conditions and generate application workloads in the next iteration. These tools capture application traffic characteristics (e.g., *connection vectors* representing requests, responses, and think times), as well as network conditions (e.g., round-trip-time (RTT) and packet loss) on the parts of the network that are **not** shared among partitions.

Observe that additional experimental nodes may be required to generate the workload to represent the interacting partitions. Depending on the testbed resources, these additional nodes can be hosted on additional testbed machines or virtualized on the nodes in the sub-scenario. We can consider this when configuring the *maxNode* parameter.

5.2.2. Modeling example

Consider the simple network scenario with three flows shown in Fig. 4. After the first phase of FSP, the scenario is partitioned into two sub-scenarios where Flow 1 and Flow 2 belong to the first sub-scenario, and Flow 3 belongs to the second sub-scenario.

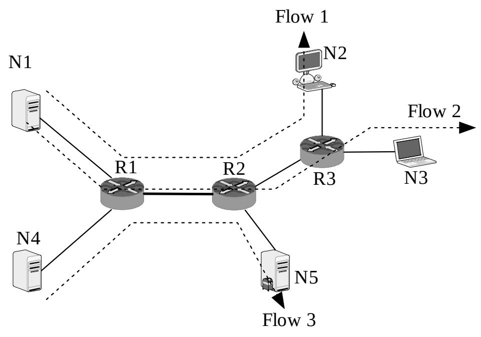


Figure 4. An example scenario with three flows. The scenario is partitioned into two sub-scenarios where the first sub-scenario contains Flow 1 and Flow 2 and the second sub-scenario contains Flow 3.

When conducting the first iteration of sub-scenario one in the testbed (Fig. 5), we collect the packet trace on the shared link (R1-R2) using tcpdump. This packet trace is processed by Tmix to extract the application workload model of Flow 1 and Flow 2. This application workload model includes a vector for each connection representing request sizes, response sizes, and application think times. Other tools such as Swing [12] additionally capture session-level characteristics, e.g., correlations among different TCP connections. We will compare these tools in Sections 9 and 10.

For each flow that traverses a shared link, we also need to model the network conditions experienced by the flow *in the non-shared links* (i.e., the paths before and after the shared link). To balance the tradeoff between experimental complexity and fidelity, FSP currently models the network conditions via the bottleneck link capacity, the average packet drop ratio, and the average packet delay on the non-shared links.

The bottleneck link capacity can be observed directly from the network topology. For example, the bottleneck capacity for Flow 2 on path N1-R1 is 10 Mbps and on path R2-R3-N3 is 5 Mbps from Fig. 5. The average packet loss and delay can be calculated from the per-flow packet loss and delay on the associated non-shared links. For instance, the average packet delay on path R2-R3-N3 is the sum of delays measured on links R2-R3 and R3-N3.

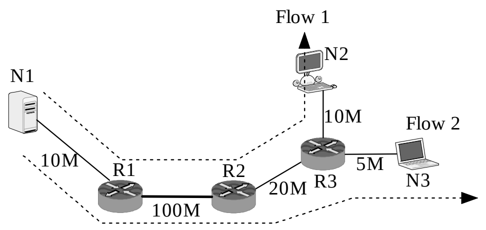


Figure 5. The first iteration of sub-scenario one.

In subsequent iterations, additional nodes are attached to the shared links to generate workload that represents the interacting partitions. For example, in the second iteration of sub-scenario two (Fig. 6), two pairs of Tmix processes, (T1, T3) and (T2, T4), are generating workload to represent Flow 1 and Flow 2 using the application-level model extracted from the previous iteration of sub-scenario one (Fig. 5).

Each Tmix process is connected to a "delay box" [11] before connecting to the shared link. The delay boxes are configured to reflect the network conditions that the Tmix traffic should encounter. For instance, delay box D4 is configured to have 5 Mbps link capacity and introduces additional loss and delay on transmitted packets according to the measurements on path R2-R3-N3 in Fig. 5.

Note that it may be possible to host multiple Tmix processes and delay boxes onto the same testbed machine. For example, as illustrated in Fig. 6, T1 and T2 are two Tmix processes running on machine Tmix1. The two delay boxes, D1 and D2, can be implemented as Linux traffic shaping rules on Tmix1 and R1.

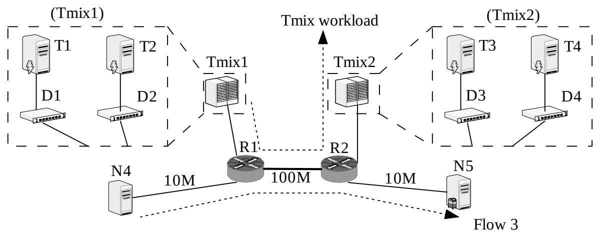


Figure 6. The second iteration of sub-scenario two. Two additional machines, Tmix 1 and Tmix 2 are used to generate the workload for Flow 1 and Flow 2.

5.2.3. *Importance of modeling interacting partitions*

The key to obtaining correct results from sub-scenarios is to utilize a modeling procedure (such as that discussed in the previous subsection) to model the traffic and network conditions in missing partitions, as opposed to simply replaying collected packet traces. Consider a dumbbell topology with N flows and $2N+2$ nodes. Let the link between the two routers in the dumbbell be the bottleneck link (C Mbps) and each sub-scenario contains only a single flow.

Fig. 7 illustrates the average throughput of the N flows (on the x-axis) in the dumbbell topology. As shown in the figure, since there is a single flow in each sub-scenario, all flows are able to utilize the full link capacity C in experiments in the first iteration. In subsequent iterations, the workload of the flows from $N-1$ sub-scenarios is generated at the application level on top of the TCP protocol and experiences conditions of the missing partitions. Therefore, the throughputs of flows on the bottleneck link in the second iteration are close to C/N, for any value of N.

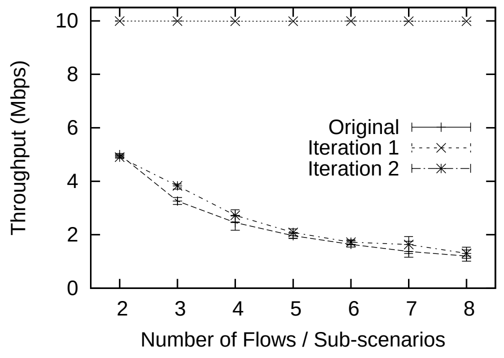


Figure 7. Average/Max/Min goodput of the flows in a dumbbell topology. Each sub-scenario contains a single flow and the bottleneck link is 10 Mbps with 10 ms delay.

5.3. *Illustrative Examples*

To further understand the second phase of FSP, we use a set of simple illustrative examples. We study network scenarios with FTP and HTTP flows using the popular network simulator ns-2 (Version 2.31) [2]. We use the topology given in Fig. 2, and set all "last-mile" links to 100 Mbps to create more interaction among flows. The FDG for the closed-loop scenarios is given in Fig. 2(b). For FTP, a client at the source host will send requests to download files from an FTP server at the destination host. Each time the client downloads a 5 MB file, and the interval between requests is exponentially distributed with the rate parameter (λ) set to 0.1, 1, or 2. We generate HTTP flows using the PackMime-HTTP [17] traffic generator. We control the rate parameter in the traffic generator to study network scenarios under different loads.

5.3.1. Uncongested scenarios

We first study the performance of our method in lightly loaded network scenarios. We use 8 FTP flows and the rate parameter (λ) is 0.1. We measure the goodput and packet drop rate for all 8 flows. As expected, the measurements from the original scenario and sub-scenarios (iteration 1) are almost identical (results omitted for brevity), and we observe similar results for lightly loaded network scenarios with 8 HTTP flows (rate = 1). This confirms the intuition that under lightly loaded scenarios, results from a single iteration suffice to accurately approximate results of the original scenario, which is the rationale for pruning uncongested links in Section 4.

5.3.2. Congested scenarios

We now increase the load in the network to increase interaction among flows. Table 1 lists the average results for a heavily loaded network with 8 FTP flows ($\lambda = 2$), repeating each experiment 10 times.

Table 1. The goodput (Mbps) of 8 FTP flows collected from the original scenario and the second iteration of FSP.

Flow	Original	FSP	Difference
0	30.65	32.22	1.57 (5.11%)
1	48.60	48.57	-0.03 (-0.06%)
2	48.59	48.34	-0.25 (-0.51%)
3	61.12	60.55	-0.57 (-0.93%)
4	34.65	35.70	1.05 (3.03%)
5	57.09	58.37	1.27 (2.23%)
6	34.55	36.73	2.18 (6.30%)
7	60.86	59.05	-1.81 (-2.97%)

As depicted in Fig. 2, we have 2 sub-scenarios (P_1, P_2) and link R0-R9 is shared among them. In the first iteration, we conduct an experiment for P_1, and collect a packet trace on link R0-R9, which contains packets for flow 0. We collect another packet trace on link R0-R9 which includes flow 4 and flow 6 when running the experiment for P_2.

In the second iteration, the packet trace on link R0-R9 is input to the Tmix tool [11] to generate workloads that represent the missing flows on the link, i.e., flow 4 and flow 6 for P_1 and flow 0 for P_2. When connecting the Tmix workload generator to link R0-R9, we insert a delay box [11] between the link and each Tmix traffic generator. As discussed in Section 5.2.2, the delay box introduces delays representing the one way portion of the RTT of each flow, minus the delay of the shared path. The capacity of the delay box is configured to be the bottleneck link capacity of a flow. We assign loss rates to the delay box to model the network conditions encountered in interacting partitions. For each path, we compute the packet loss rates and delays of the non-shared links from the trace collected in the previous iteration. For example, let n_1, n_2, n_3, and n_4 be the nodes on the path of flow 0, where n_1 and n_4 are the source and destination of the flow and n_2 and n_3 are the two end points of the shared link. The four loss rates on the non-shared parts of the flow (in both directions), i.e., $(n_1, n_2), (n_3, n_4), (n_4, n_3)$, and (n_2, n_1), are used in the delay box configuration. Tmix also infers detailed application behavior from the trace and represents it as connection vectors [11].

The results in Table 1 demonstrate that long-term metrics, such as the average goodput of a flow, can be reasonably predicted using our method.

5.3.3. Shared links are bottlenecks

In the previous experiment, although the network is congested, the shared link R0-R9 is not a bottleneck in the original network because flows 4 and 6 and flow 0 are downloading files in opposite directions. We now reverse the direction of flow 0 to make link R0-R9 the bottleneck link. The goodput of the flows is given in Table 2. We are especially interested in flows 0, 4, and 6, since they are the flows on the shared link R0-R9. In the first iteration, the goodput of flows 0, 4, and 6 is 21.31, 13,81, and 13.17 Mbps higher than the goodput they obtain in the original scenario. This is due to the missing flows in each sub-scenario, e.g., flow 0 does not exist in P_2.

Table 2. The goodput of 8 FTP flows (Mbps) in different iterations.

Flow	Original	Iteration 1	Iteration 2
0	30.50	51.81	27.64
1	48.51	48.03	51.21
2	48.80	49.02	45.95
3	61.19	49.81	59.60
4	34.25	48.07	36.85
5	57.44	44.00	61.51
6	35.09	48.26	38.22
7	60.19	49.83	58.07

After the second iteration, we are better able to predict the goodput of all flows. Note that in both examples, there are small variances between the results collected from the original scenario and from the second iteration of FSP. For example, the goodput of flow

0 in Table 2 is still 2.85 Mbps lower than the correct value. Recall that when we generate the workload for flow 4 and flow 6 into P_1, the network conditions of P_2 are modeled as the delay, link capacity, and loss on the Tmix delay boxes. Ideally, the loss rates and delays accurately capture the impact of other flows in P_2 (flow 3 and flow 7). However, to reduce complexity, we do not capture the dynamics encountered by each flow in the interacting partition. As a result, the workload generated by Tmix in the second iteration of P_1 fails to accurately constrain the goodput of flow 0. We are currently investigating alternative Tmix configurations that more accurately represent the workload of missing flows, at the expense of space and time complexity.

5.3.4. *Fine-grained and coarse-grained metrics*

Fig. 8 plots the packet delay distribution of flow 1 in the previous experiment. Table 3 lists the percentage difference of the results between the original scenario and the second iteration of FSP in that experiment. We observed that the overall statistics (coarse-grained metrics) such as the average goodput, packet delay, and delay jitter as well as their cumulative distribution functions (CDFs) can be accurately predicted by FSP.

Since the delay and loss in the interacting partitions are abstracted into simplified models, such as the average packet loss on non-shared links, fine-grained metrics are not preserved by FSP. For example, Fig. 9 illustrates the one-way packet delay of the first 100 packets in flow 1. Even though flow 1 does not directly interact with flows in other partitions, the packet delays of individual packets are different from the original scenario. As a result, FSP does not target experiments that require fine-grained metrics to be preserved.

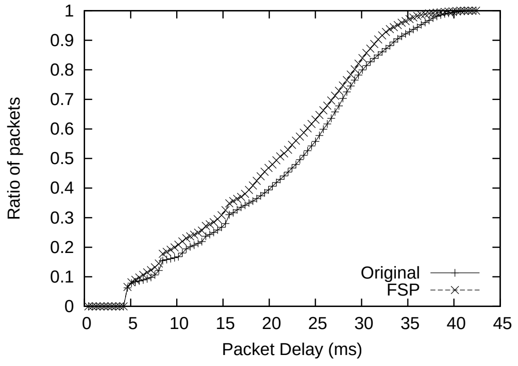


Figure 8. The packet delay distribution of the packets in flow 1.

Table 3. Percentage difference of the results between the original scenario and the second iteration of FSP. The is the same experiment as Table 2.

Flow	Goodput	Sent Packets	Dropped Packets	Packet Delay	Delay Jitter
0	-9.37%	-9.13%	-6.75%	-12.38%	10.30%
1	5.57%	5.35%	1.09%	-1.71%	-5.28%
2	-5.85%	-5.85%	-5.82%	-3.11%	6.20%
3	-2.60%	-1.89%	23.92%	3.53%	2.68%
4	7.59%	6.42%	-7.24%	-5.99%	-7.06%
5	7.09%	7.19%	10.79%	8.00%	-6.62%
6	8.92%	7.93%	-4.12%	-4.25%	-8.18%
7	-3.52%	-2.66%	27.62%	6.45%	3.66%

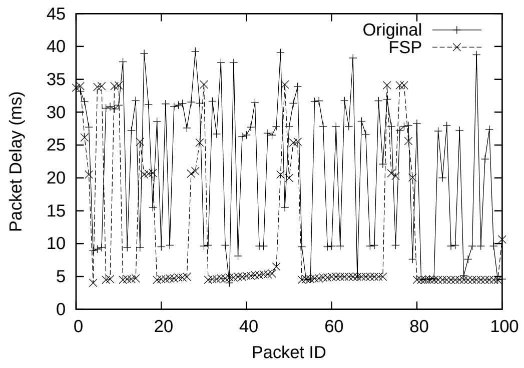


Figure 9. The packet delay of the first 100 packets in flow 1.

6. Multiple Interacting Partitions

We now examine an important aspect of the tradeoff between experimental fidelity and complexity. As discussed earlier, FSP represents the relationship between flows by an FDG, where each flow is a vertex in the graph and two flows can influence each other as long as there is a path between them. If there is no direct edge between two flows, the interaction among the two flows can only be propagated transitively via other flows. If the flows belong to different sub-scenarios, this propagation process can only take place in a subsequent iteration. Naturally, this leads to the following question: how many iterations are required when there are multiple interacting sub-scenarios?

6.1. *Four Interacting Sub-scenarios*

Consider the network scenario shown in Fig. 10. In this simple scenario, all links are 10 Mbps with 5 ms delay and there are four macro-flows, each belonging to a different partition ($\forall 1 \leq x \leq 4 : Fx \in P_x$). Each macro-flow contains a number of concurrent TCP connections sending traffic continuously from the source to the destination node for 600 seconds. In all experiments, we

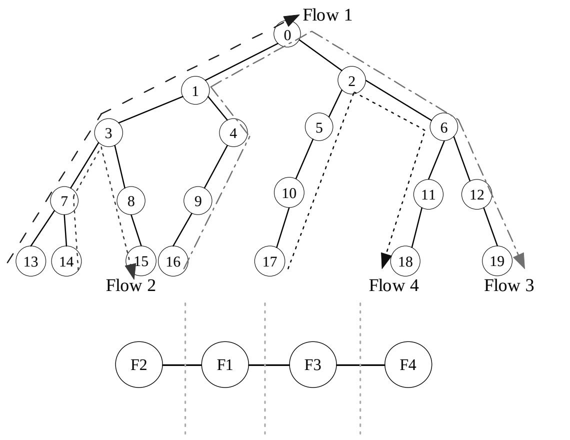


Figure 10. A scenario with up to four (indirectly) interacting sub-scenarios and its flow-dependency graph.

employ Tmix to *generate the application workload* and *model the network conditions* via losses and delays in interacting partitions.

Tables 4 and 5 give the goodput of the four macro-flows, each containing one connection or three TCP connections, respectively. When the network is lightly loaded (Table 4), two iterations are sufficient as the results stabilize in later iterations. The application behavior extracted by Tmix for all four macro-flows, each a single long-term TCP flow, remains unchanged in all iterations, since we are using the same application traffic model. Tmix correctly infers that model from the traffic traces. The network conditions (delay and loss) extracted from the traces vary within a small range from the second iteration onwards as most links have low packet losses and queuing delays in this lightly loaded network.

Note that the interactions among partitions that are not directly connected in the FDG (e.g., P_4 and P_1) are propagated through the network conditions of intermediate partition(s) (e.g., P_3). The similarity between the network conditions measured in different iterations also implies that the partitions which do not have common shared links have little influence on each other in this scenario. For example, flow 4 interacts with all other flows in subsequent iterations (3rd and 4th iterations for flow 1 and flow 2). However, as seen in Table 4, the goodputs of flow 1 and 2 remained almost unchanged after iteration 2. Although the goodput of flow 3 becomes more accurate in iteration 3, the improvement is insignificant (within 5% of the link capacity).

As we increase the workload in the network, interactions between any two sub-scenarios are no longer dominated by the application workload generated by Tmix and two iterations are insufficient to obtain accurate results. As illustrated in Table 5, it takes four iterations to obtain results with reasonable accuracy. The number of iterations to propagate the interaction from one sub-scenario to any other sub-scenario is upper-bound by the number of sub-scenarios in the largest connected FDG component, e.g., if the largest FDG component needed to be partitioned into four sub-scenarios, four iterations are sufficient to propagate the interaction between any two interacting sub-scenarios. This maximum iteration value is labeled *maxChain*.

Note that despite executing additional iterations, the results from the second experiment (Table 5) are less accurate then those from the first experiment (Table 4). Since in both experiments the application traffic models are correctly inferred by Tmix, the loss of fidelity stems from the simplified network condition model which is amplified when propagated through multiple sub-scenarios.

The examples in this section highlight a fundamental tradeoff: fidelity of the results versus the time and space complexity of the experimentation process. When simple aggregate measurements, e.g., average delay or loss over the entire experiment, are input to a tool like Tmix to model network conditions encountered by a flow, loss of fidelity will occur, compared to having more detailed representations of network conditions, e.g., a time series of packet loss over the entire experiment duration. We are currently exploring this tradeoff in greater depth. The choice of which tool to use (Tmix, Harpoon, Swing) is a critical aspect of the tradeoff, and we examine this via testbed experiments in Section 9.

6.2. Required Iterations

Let us return to the question posed at the beginning of this section. In the two previous examples, FSP requires two to *maxChain* iterations. The value of *maxChain* depends on the input scenario and the value of *maxNode*. For example, Fig. 16 in Section 7.3 illustrates the *maxChain* value calculated for a set of randomly generated network scenarios. We have observed that only a few iterations are needed for all scenarios we generated: we need four to five iterations in most cases, based on the size of the largest set of interacting sub-scenarios.

Naturally, the examples illustrated in this section cannot represent all possible network scenarios and fewer or more iterations may be required. To speed-up the termination of FSP, several heuristics can be used as discussed in the following subsections.

Table 4. The goodput of TCP flows in kbps. Each macro-flow contains one TCP connection.

Flow	Original	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
1	4998	8544	4998	4998	4998	4998
2	4998	8544	4998	5075	5075	5075
3	4228	4270	3716	4214	4217	4216
4	5618	5695	5657	5675	5642	5610

Table 5. The goodput of TCP flows in kbps. Each macro-flow contains three TCP connections.

Flow	Original	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
1	4895	9997	3161	4254	4455	4331
2	5098	9996	3089	3492	4247	4017
3	723	9992	2919	917	1045	945
4	9270	9992	9373	9286	9324	9309

6.2.1. *FDG edge-pruning*

As mentioned in Section 4.1, uncongested links in the network can be identified and the associated FDG edges are removed when constructing the FDG. This reduces the size of the connected components in the FDG and reduces *maxChain* value.

6.2.2. *Minimize interactions among partitions*

In addition to identifying uncongested links conservatively (e.g., Fig. 3), one can also predict the uncongested links based on known traffic workload [18] or simulation results. For example, before conducting a testbed experiment using FSP, a full-scale simulation with simplified application workloads can be conducted to predict potential uncongested links. Although we may not wish to prune all their associated FDG edges during the FSP partitioning phase as these links are not guaranteed to be uncongested, one can easily modify the GGGP algorithm to select them as edge cuts to minimize the interaction among partitions.

6.2.3. *Early termination*

As seen in Table 4, it is often possible to obtain accurate results before *maxChain* iterations. This occurs when the extracted application and network models from all partitions converge to the same values in two consecutive iterations. Note that the inputs to the workload generation tool (e.g., Tmix) for the $(n+1)$st iteration are extracted from the *n*th iteration and FSP can terminate immediately without conducting the $(n+1)$st iteration.

6.2.4. *Partial results and feedback mechanism*

If FSP fails to converge within a specified number of iterations, FSP can terminate and report only the results from partitions that converged. If not all flows of interest are covered by these partitions, the user can increase the *maxNode* value and repeat the FSP procedure. By merging interacting partitions that do not converge into a single partition, we are able to estimate a recommended *maxNode* value for the next FSP procedure.

7. Partitioning Experiments

In this section, we investigate the first phase of FSP. Given the size of a backbone network (the number of routers) and the number of flows we wish to have in a network scenario, we generate a set of Rocketfuel [19] topologies representing the backbone network using our Rocketfuel-to-ns tool [20]. For each flow, we insert two end hosts as the source and the destination of this flow, and randomly attach these end hosts to the backbone network. The end hosts are only attached to routers with degree no larger than three and, to avoid trivial cases, the source and destination nodes of a flow are not attached to the same router.

7.1. *Weights in Partitioning*

We first evaluate our choice of weight function in the first phase of FSP. Recall that we compute the weight of an edge cut in the FDG as the number of distinct nodes (hosts and routers in the original network topology) that are shared by flows represented in the cut (Section 4). Since the graph partitioning algorithm in our method aims to select partitions with low edge cut weight, the function we choose to calculate the weight of an edge cut should help reduce the interactions among partitions. In this section, we show how our weight function compares to a simpler function that uses the number of FDG edges on an edge cut as the weight. Since an edge in the FDG implies dependence among two flows, fewer FDG edges between partitions also implies less dependence among partitions.

Fig. 11 demonstrates the average number of shared links between partitioned network scenarios when using these two methods of computing the weight of an edge cut. In this experiment, we generate different network scenarios by randomly assigning 50 or 100 flows with their end hosts onto a fixed Rocketfuel backbone with 100 routers. We compute the average number of shared links among partitions and the average results from 30 experiment runs are plotted. From Fig. 11, we find that using the number of distinct nodes as the weight of an edge cut can lead to fewer shared links among sub-scenarios than simply using the number of dependent flows. This not only indicates that we have fewer packet traces to collect in the second phase of our method, but also implies that there may be less complex interactions among sub-scenarios.

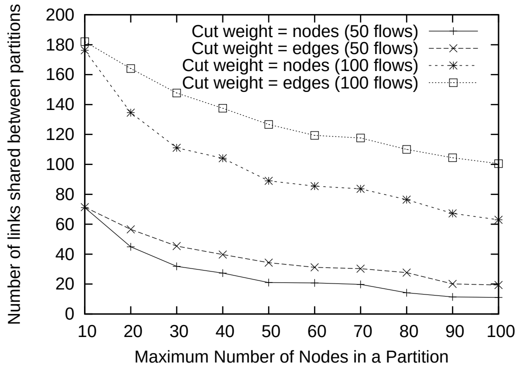


Figure 11. Number of shared links among sub-scenarios with two methods to calculate the weight of an edge cut.

7.2. Time Complexity

The time required for the second phase of FSP depends on the number of sub-scenarios and the tool used (e.g., Tmix vs. Harpoon) which can significantly vary (see Section 9). As seen in Algorithm 1, the time complexity of the first phase of FSP, i.e., partitioning a large network scenarios into sub-scenarios, is dominated by the graph partitioning algorithm. Our current implementation uses the recursive bisection algorithm with complexity $O(|F_E|\log k)$, where F_E denotes the edges in the FDG and k denotes the number of partitions generated by the algorithm [16]. The worst case time complexity of our complete algorithm is $O(|F|^4)$ where $|F|$ is the number of input flows.

We computed the runtime for partitioning scenarios with 100 to 500 flows, randomly generating 30 scenarios for each value of the number of flows. As illustrated in Fig. 12, the runtime is proportional to the number of flows in the network scenario. The runtime can vary from seconds to hours for the same number of flows depending on the complexity of the scenario or, in other words, the number of edges in the FDG. Despite the fact that FSP took up to a few hours for some scenarios with 500 flows, partitioning will typically be invoked offline, and hence FSP is still feasible. Moreover, the current FSP prototype does not take advantage of possible performance optimizations; we will develop a faster implementation by selecting a faster graph partitioning algorithm, e.g., the k-way multilevel partitioning algorithm with $O(|E|)$ [16], and using more sophisticated program optimization techniques.

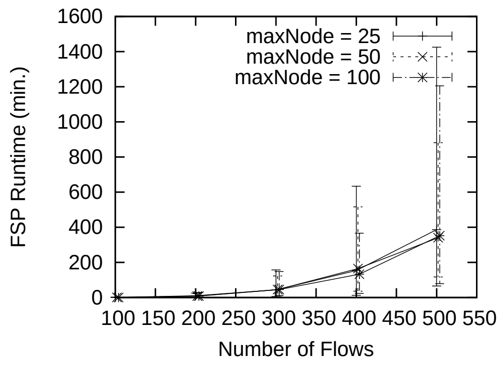


Figure 12. The average runtime of FSP phase 1 versus the number of flows in randomly generated network scenarios.

7.3. Partition Characteristics

We now take a closer look at the partitioning results. We generate sets of random network scenarios, each with 50 routers. We generate 50 to 250 unidirectional open-loop flows in the network, i.e., there are $50+2\times|F|$ total nodes (hosts and routers) in the network. For each set of experiments, we generate 50 random scenarios and execute the FSP partitioning phase with different *maxNode* values. We investigated (i) the number of partitions, (ii) the value of *maxChain* (defined in Section 6), and (iii) the number of links and routers shared among partitions.

Fig. 13 and Fig. 14 show the number of directly dependent flows, and the number of links shared among partitions. Since the number of dependent flows is large, and there are many links shared among partitions, we conclude that the random network scenarios we are studying are complex enough to represent interesting scenarios.

Fig. 15 shows the number of partitions generated by our algorithm, and Fig. 16 shows the *maxChain* value computed after partitioning, which typically influences the required number of iterations. As expected, we find that the larger the value of *maxNode*, the fewer the number of partitions and the smaller the *maxChain* value. A flow in any of our randomly generated scenarios involves two end hosts and several routers on its path. Although a flow can only appear in one partition, the routers on its path can be *duplicated* in multiple partitions. We observed that a router may appear in 10 to 20 partitions. The number of duplicated routers is significantly reduced when *maxNode* increases.

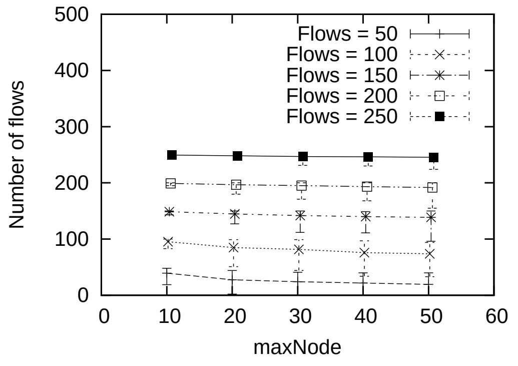


Figure 13. Average/min/max number of flows involved among partitions.

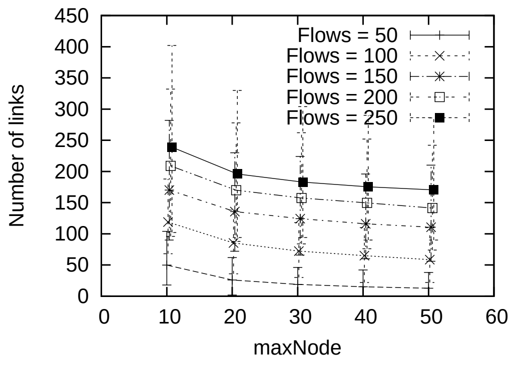


Figure 14. Average/min/max number of links shared among partitions.

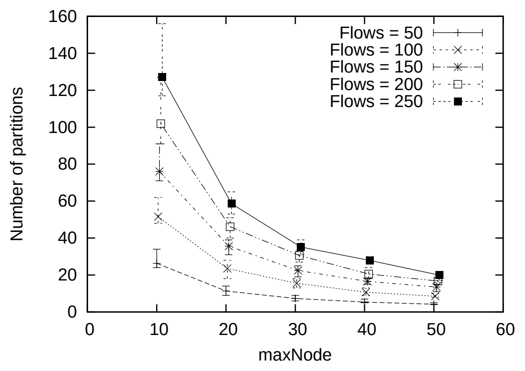


Figure 15. Average/min/max results of FSP partitioning under different network scenarios and *maxNode* values.

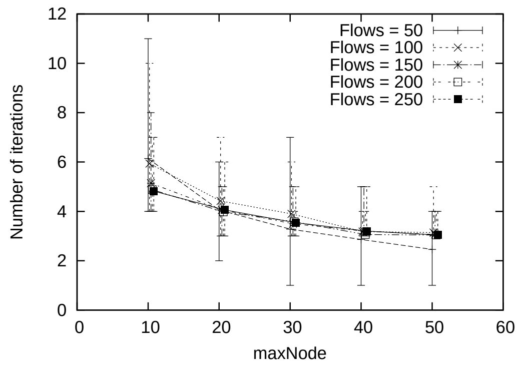


Figure 16. The *maxChain* value for randomly generated network scenarios.

8. Simulation Experiments

8.1. Botnet Experiments

Denial of Service (DoS) attacks have been launched against Internet sites for decades, and distributed DoS attacks are extremely difficult to defend against. With the prevalence of botnets in today's Internet, individuals can easily launch a massive DDoS attack from a rented botnet for just a few hundred dollars per day. In this section, we use both phases of FSP on a scenario that studies the impact of a large-scale DDoS attack targeting a busy web server at Purdue University.

To understand the availability of our web server to visitors during the attack, we selected 200 domains as sources of the legitimate users and 50 subnets as the attackers. The 200 (out of 14407) domains cover more than 70% of the service providers of all visitors to our web server between May 2009 and May 2010, and the 50 subnets are selected from the black list generated by DShield.org [21] in June 2010. We use traceroute from the web server to all 200+50=250 /24 subnets to generate the network topology, and find 1232 routers. Several heuristics are then applied to reduce the number of routers. For example, if the last 8 hops of a traceroute record are not used by any other flow, we aggregate the delays between them and remove the 7 intermediate hops. After reductions, there are 438 nodes and 462 links in this network topology as shown in Fig. 17. Note that the end hosts for legitimate users and attackers are aggregated. For instance, the 50 attack flows represent thousands of attackers from the 50 /24 subnets. All links are set to 100 Mbps. During the attack, 67% of the links in the network are highly congested, including the link directly linking to the web server.

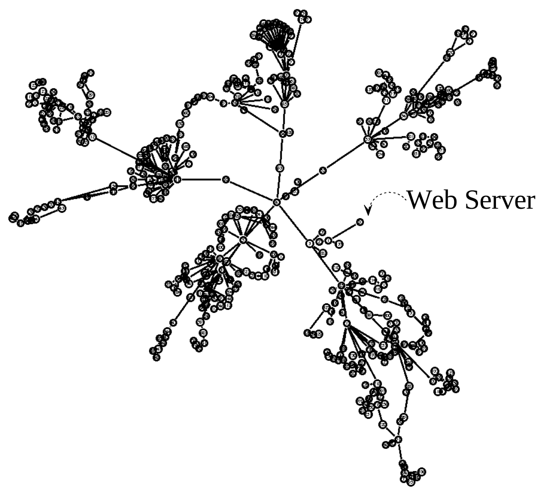


Figure 17. The network topology of the botnet experiment.

Since the size of this topology is larger than the DETER testbed, we use ns-2 to compare between the original and the partitioned experiments. The 200 legitimate flows are generated by the PackMime-HTTP module in ns-2 with 2 requests per second using both HTTP/1.0 and HTTP/1.1. For each HTTP/1.0 session, the client requests a 36 kB page, which is the size of the most popular page in our web site, and terminates the TCP connection once it is received. For HTTP/1.1 sessions, the client first requests the same page as in HTTP/1.0, but requests up to two other pages using the same persistent connection. This 3-page per session is based on the fact that most of our site visitors (85.89%) view at most three pages during their visit. Each page contains several objects and the size and number of the objects are generated by PackMime-HTTP. For the attack flows, we send UDP packet bursts to the web server at 5 Mbps and exponential on/off time with mean set to 2 seconds.

We execute FSP on this scenario with *maxNode* set to 100 to generate sub-scenarios which can easily fit onto a testbed like DETER. The large scenario with 438 nodes is partitioned by FSP into 8 sub-scenarios where the largest one contains 83 nodes – a reasonable size for DETER.

We use a user-perceived metric, the ratio of successful HTTP sessions, in both the original and the partitioned experiments. An HTTP session is successful if its duration is less than 60 seconds or the delay between receiving objects from the server is less than 4 seconds [22]. Fig. 18 and Fig. 19 give the percentage of successful HTTP/1.0 and HTTP/1.1 sessions in the 300-second period when the server is under DoS. We also examined the download time distributions for pages and objects. Clearly, results from the first iteration are erroneous (100% success) since attack flows and legitimate flows are mostly in separate partitions, while the results from the second iteration reasonably match the original scenario.

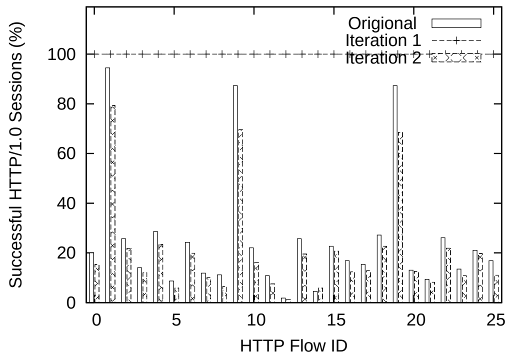


Figure 18. Percentage of successful HTTP/1.0 sessions. Only the first 25 flows are shown due to space constraints.

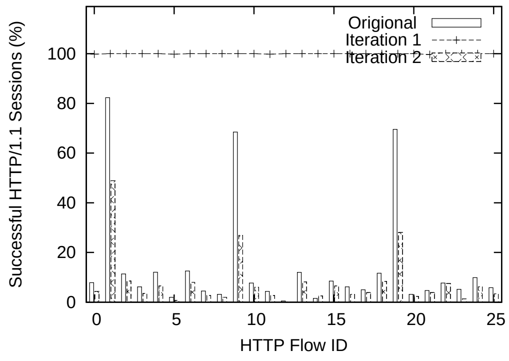


Figure 19. Percentage of successful HTTP/1.1 sessions. Only the first 25 flows are shown due to space constraints.

A closer look at the results in Fig. 18 and Fig. 19

reveals that, while most flows have similar results in both experiments, a few flows (e.g., flow 9) have a significantly lower success ratio in the partitioned experiments. Comparing Fig. 18 and Fig. 19, the success ratios for HTTP/1.1 sessions are lower than the HTTP/1.0 sessions, and the results from the second iteration have a greater error for HTTP/1.1. This is due to the HTTP/1.1 persistent connections. The HTTP/1.1 session has more objects and pages in the first iteration than in the original scenario and the Tmix-injected TCP flows are thus more aggressive. This is because when a requested page is dropped in the original scenario, a client will not request the objects in that page. Since there are no dropped requests in the first iteration (the horizontal line in the figure for iteration 1), a client will request more objects and pages in a connection. *Such changes in user behavior are hard to capture by workload generators unless they have application-layer knowledge, which is avoided by Tmix because it hinders scalability and extensibility to new applications*. As we will discuss in Section 9, Tmix [11], Harpoon [13], and Swing [12] make different choices in terms of the user, session, connection, and network characteristics that they extract and model, and hence the fidelity of the results obtained varies according to which of these tools we use, and how we configure the selected tool. *As we extract and model more information, space and time complexity increase, but fidelity also increases*. This tradeoff must be balanced according to the goals of the experiment to be partitioned, and the time and space constraints.

8.2. Comparisons to Downscaling

In this section, we compare FSP with a downscaling technique. We select the TranSim time-sampling approach [23] since it is designed for arbitrary traffic (whereas other downscaling techniques in the literature [24, 14] make assumptions about traffic models). TranSim aims to accelerate a simulation experiment by reducing the number of packet events in the simulation. While TranSim can also be applied to testbed experiments to reduce the hardware requirements, perhaps by running experiments on slower testbed machines, TranSim was designed to speed up simulations [23] and it is not entirely clear how it can reduce the number of machines required on a testbed. Thus, we compare TranSim and FSP via simulation experiments.

In our first experiment, we consider a scenario where 50 backbone routers are generated using Rocketfuel-to-ns [20] and there are 50 pairs of end hosts randomly attached to the backbone. The topology of our second experiment is a simple dumbbell as shown in Fig. 20. For each pair of hosts, a client continuously requests a file from a server at a fixed rate. The first experiment has 50 long-lived FTP flows where the requested file size is 5 MB and the request rate is one per second. In the second experiment, there are 25 long-lived (class 0) HTTP flows and 25 short-lived (class 1) HTTP flows. The requested file sizes are 100 kB and 1 kB, and the request rates are 2 and 50 per second, respectively. In both experiments, the downscaling parameter α in TranSim is set to 0.5 and 0.25. To correspond to this, the $maxNode$ parameter in FSP is configured so as to partition the 50 flows into 2 and 4 sub-scenarios. We use ns-2 with Tmix as the modeling tool in FSP.

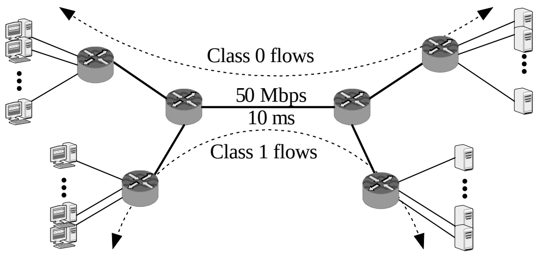


Figure 20. The network topology of 25 long-lived TCP flows and 25 short-lived TCP flows. Unless otherwise specified, all links are 100 Mbps and 10 ms delay.

We use throughput as the metric in the first experiment since there are many active TCP connections when the simulation ends. In the second experiment, we compute the success ratio of HTTP requests [22]. In this experiment, we consider an HTTP request successful if it is completed within 10 seconds. We find that throughput CDFs for FSP and TranSim are similar for the first experiment in Fig. 21. In the second experiment (Fig. 22), the fidelity of FSP is significantly higher than TranSim, especially compared to the 0.25 downscaling parameter. These TranSim results agree with our previous study [25]. TranSim loses fidelity with short-lived flows, since control packets, such as the TCP handshake packets and HTTP request packets, cannot be sampled. The short-lived flows are thus relatively more aggressive.

As with FSP, TranSim is not designed to preserve fine-grained metrics. When conducting a downscaled experiment using TranSim, only α percentage of the packets in a flow are transmitted. The sampling of packets can significantly impact the fidelity of packet-level metrics such as the packet delay jitter. Table 6 gives the differences in the delay jitter between the original and the downscaled experiments measured in the second network topology (Fig. 20). We observe that the fidelity of packet-level metrics decreases as TranSim selects fewer packets to represent a flow in a downscaled experiment.

Compared to TranSim, FSP requires additional traffic modeling processes and longer run-time to conduct sub-scenario simulations in multiple iterations. The key advantage of FSP is that it can be directly applied to experimental and emulation testbeds to reduce the number of required machines, whereas prior work only considered how to reduce simulation time [24, 14, 23].

Table 6. Percentage difference of average packet delay jitter between the original and the downscaled experiments.

	Short-lived TCP	Long-lived TCP
FSP (maxNode=60)	0.84%	3.08%
FSP (maxNode=30)	1.34%	2.90%
TranSim (α=0.5)	51.06%	6.43%
TranSim (α=0.25)	74.70%	39.09%

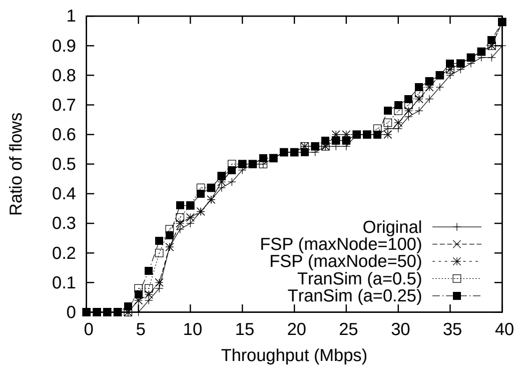


Figure 21. The throughput distribution of the 50 long-lived flows. Note that the results from TranSim are normalized ($\times 1/\alpha$).

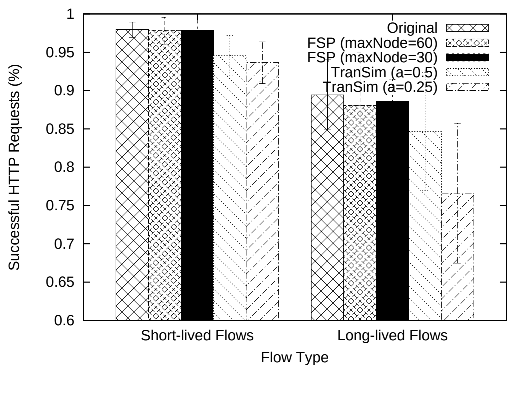


Figure 22. The success ratio of long-lived and short-lived HTTP flows.

9. DETER Testbed Experiments

In this section, we discuss two sample experiments on the DETER testbed [5] that compare Harpoon [13], Swing [12], and Tmix [11]. We directly use the author-provided Harpoon implementation, but port Tmix and Swing to DETER. The network topology is the same as in Fig. 2(a) where all links are 10 Mbps with 10 ms delay. Only 3 macro-flows, flow 0, flow 4, and flow 6, are included to simplify comparisons among the tools.

Flow 4 is an HTTP flow generated by httperf [26]. The requested file size is 100 kB and the request rate is 5 per second for a total of 600 requests. The request timeout is set to 3 seconds to make the flow more sensitive to the dynamics of flow 0. Flow 6 (only used in the second experiment) emulates a pulsing DoS attack. The idle period is 5 seconds and 40 TCP connections arrive during the attack period. This arrival rate and the flow size of 50 kB saturate the 10 Mbps link during the attack periods. In the first experiment, flow 0 (labeled Flow I) is also an httperf flow with the same parameters as flow 4, but with an exponentially distributed request inter-arrival time (mean=0.15 seconds). In the second experiment, flow 0 (labeled Flow II) consists of 10 clients downloading a series of files from a server. All files are 500 kB and the “think time” between consecutive downloads is 2 seconds. In iteration 1 of the second phase of FSP, we collect traces for all flows on link R9-R0, and use them to model missing flows in iteration 2. We repeat each experiment 5 times and average the results.

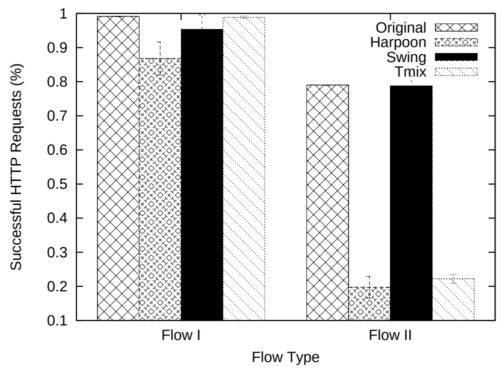


Figure 23. The success ratio of flow 4.

The results of the two experiments are depicted in Fig. 23. In the first experiment (Flow I), results from Tmix and Swing are close to the original scenario. Although all three tools correctly model the traffic and network conditions, Harpoon has the lowest fidelity because of its use of time granularity. Unlike the other two methods that inject exactly 600 TCP connections, Harpoon divides time into intervals of length *Interval-Duration*, configured to 60 seconds in this experiment, and does not aim to accurately model dynamics within an interval. Due to this, Harpoon generates more aggressive flow 0 traffic, which reduces the success ratio for flow 4. In the second experiment (Flow II), Swing yields the highest fidelity, because connections within a session are not modeled independently. Unlike Harpoon and Tmix, a connection will not start before the previous one is finished, even during the attack period when connections take longer. The think time between downloads is preserved, and the modeled traffic in iteration 2 is no more aggressive than the original scenario.

Comparing the three methods, Harpoon has the lowest fidelity but exhibits the lowest complexity. Unlike Swing and Tmix that require packet-level traces, a much smaller Netflow trace is used by Harpoon. A trace that includes 0.8 M packets or 12 k connections required 74 MB from tcpdump but was only 1.2 MB in Netflow format. Collecting flow-level traces takes less effort compared to tcpdump, since Netflow is typically supported by routers. Swing and Tmix have similar storage requirements. However, while Tmix models connections independently, Swing's structural model has the highest fidelity since correlations among connections in a session are captured. Tmix gives good results with moderate complexity in all our experiments when connections are *not* highly correlated. In our implementations of Tmix and Swing, a 74 MB tcpdump trace can be processed in only 35 seconds by Tmix, whereas an additional 40 seconds are required by Swing.

Table 7. Properties captured by different traffic modeling tools.

Property	Swing	Tmix	Harpoon
Average traffic volume	Yes	Yes	Yes
Connection start time/size	Yes	Yes	No
Session-level think time	Yes	No	No
Packet timing/order	No	No	No

Table 7 summarizes example properties that can be extracted using different tools. None of the three tools can precisely capture the original packet-level behavior such as the timing and order of packets or the packet delay jitter. While Swing and Tmix are able to extract more details from the traces, all three tools can be used to model the average traffic volume of flows, and can be integrated into FSP to provide results with different degrees of fidelity according to the experimenter requirements.

10. Discussion

10.1. Choice of Modeling Tool

The choice of the traffic modeling tool does not alter the FSP algorithms. The procedures to extract the network conditions and the locations where traces must be collected are independent of the tool. Using Table 7 as a guideline, we can identify the appropriate tool that can be used to capture the required flow and network characteristics. For example, consider a one-day trace with macro-flows that are composed of a large number of independent TCP connections. One should use Harpoon in this scenario since it is not required to model the exact order of TCP connections to generate workload that matches the traffic volume. Harpoon has the lowest complexity among the tools.

Although it is possible to integrate more sophisticated modeling tools in FSP for higher fidelity, current tools focus on transport-level and session-level characteristics, and require the flows to have similar application-level behavior in both the original and the partitioned scenarios. For instance, consider a video streaming application that automatically adjusts its bit rate according to the encountered network conditions. Compared to the original scenario, the application transmits more traffic in a less congested sub-scenario experiment. Since the highly variable application behavior cannot be easily extracted from the different iterations of the sub-scenario experiments using current modeling tools, results from FSP may have lower fidelity.

10.2. Scalability of FSP

Given a network scenario and the available machines in a testbed (*maxNode*), FSP can partition the scenario, regardless of the size of the network, as long as the longest flow in the network traverses no more than *maxNode* nodes. In the extreme case, each sub-scenario will contain only a single flow and all traffic generation nodes (e.g., Tmix nodes and delay boxes) are embedded on the routers.

We define the scaling factor of FSP as the size of the original network scenario divided by *maxNode*. As one increases the scaling factor in an FSP experiment (i.e., selecting a smaller *maxNode* value), the number of interacting sub-scenarios increase and can adversely impact the experimental fidelity. The optimal scaling factor is often challenging to predict without conducting a series of FSP experiments. Our experience from the evaluation experiments indicates that FSP can provide acceptable accuracy when applied to a network scenario that is one order of magnitude larger than the testbed size.

10.3. Applications of FSP

FSP can be applied to both network simulation and testbed experiments. Since the partitioned sub-scenarios require fewer computational resources for a simulator or fewer testbed machines, larger experiments can be conducted on a resource-limited experimental platform when partitioned by FSP.

FSP can aid parallel simulation. Unlike most parallel simulators that require manual partitioning of the network topology, FSP can automatically partition a large experiment into sub-scenarios that can be executed in parallel.

The first phase of FSP can also be used to analyze dependencies and tune an experiment. For example, consider assigning nodes randomly in a network to generate background traffic. The flow dependency graph (FDG) can be used to analyze if the nodes are assigned properly, i.e., ensure that the background traffic interacts with target flows.

11. Related Work

The experimental scalability problem has been studied in the context of simulation, emulation, and testbed experiments. The proposed approaches can be broadly classified into two categories: (1) approaches that reduce the size or events in a given experimental scenario, and (2) approaches that perform intelligent resource allocation to map a given scenario onto available resources.

The goal of the approaches in the first category is to generate a *downscaled* version of the original network scenario that preserves important properties of the original scenario. For example, Pan *et al.* [24] propose Small-scale Hi-fidelity Reproduction of Network Kinetics (SHRiNK). Using SHRiNK, one can construct a downscaled network replica by sampling flows, reducing link speeds, and downscaling buffer sizes and Active Queue Management (AQM) parameters. The intuition behind their approach is to reduce the traffic and the resources consumed by that traffic by the same factor.

Instead of sampling traffic flows, Kim *et al.* [23] propose TranSim to slow down the simulation and sample *time intervals* (also referred to as *time expansion*). By maintaining the bandwidth-delay product invariant, network dynamics (such as queue sizes) and TCP dynamics (such as congestion windows) remain unchanged in the process of network transformation.

Another noteworthy approach in the first category is DSCALE, proposed by Papadopoulos *et al.* [14]. DSCALE includes two methods, DSCALEd and DSCALEs, that prune uncongested network links, based on earlier work on queuing networks [27]. Their follow-up work considers how to identify uncongested links for pruning [18]. Petit *et al.* [28] investigate methods similar to DSCALE, and point out that downscaling methods are highly sensitive to network traffic, topology size, and performance measures. This is consistent with our findings in [25].

Instead of pruning uncongested links in a path, the path emulator proposed by Sanaga *et al.* [29] simplifies a network path into a single hop. By collecting parameters from end-to-end observations of the Internet, the emulator abstracts characteristics of the path and can emulate the path without the detailed router-level topology. Other work has also considered the downscaling problem in specific contexts. For example, Weaver *et al.* [30] focus on downscaling worm attacks. Carl *et al.* [31] study how to preserve routing paths among Autonomous Systems (ASes) while reducing the number of ASes through Gaussian elimination. Krishnamurthy *et al.* [32] consider preserving topological characteristics when reducing a network graph.

Approaches in the second category map an experimental scenario onto available resources. These approaches include the application of a range of parallel and distributed simulation techniques such as in [6, 33]. For example, Walker *et al.* [34] employ software virtualization to migrate running node images from one switch to another, in order to maintain the proximity of the nodes attached to each RF switch.

An important technique in this category is virtualization. The capacity of a testbed scales when multiple experimental nodes are hosted as virtual machines on a single physical machine [7, 35]. Container based network emulators [36, 37, 38] utilize lightweight OS-level virtualization techniques. These tools run high fidelity, but small scale, emulation testbeds on a single machine. Real-time network simulators such as [39, 40, 41, 42] are able to interact with real network traffic and further improve the capacity of existing testbeds. In contrast to FSP, approaches in this second category typically map multiple nodes in the original scenario to a single node in the experiment, potentially introducing artifacts in experiments that overload resources, especially in DoS experiments [3, 37].

Our approach is orthogonal to techniques in both categories, and can be integrated with them. For example, downscaling techniques can be applied to an experimental scenario before or after FSP to simplify the original scenario or to speed up the execution of a sub-scenario. Virtualization techniques can also be used on a sub-scenario when appropriate, allowing it to be executed on a smaller number of testbed machines. FSP is a simple platform-independent approach for different types of experiments, including DoS experiments that pose significant challenges with other approaches [3, 25, 37].

12. Conclusions and Future Work

In this paper, we have proposed a platform-independent mechanism, Flow-based Scenario Partitioning (FSP), to partition flows in a large network experiment into a set of smaller experiments that can be *sequentially* executed. The results of the smaller experiments approximate the results of the original experiment. FSP is platform-independent since it does not require any modifications to the experimentation testbed. Since the original large experiment and the partitioned smaller experiments can be viewed as independent experiments, our approach can be integrated with existing scaling solutions. For example, we can use our tools to understand or simplify a large network experiment before applying virtualization or hybrid simulation techniques.

Our results from simulation and DETER testbed experiments indicate that FSP approximates application performance under different levels of congestion and open/closed-loop traffic. Our future work plans include optimization of the partitioning process, and an in-depth analysis of the granularity of modeling interacting partitions.

We are also experimenting with FSP on a variety of topologies and flow mixes, including large CDNs and P2P systems. The focus of these experiments is to further explore the fundamental tradeoff between experimental fidelity and space/time complexity of the experimentation process. Finally, we are examining the integration of hybrid simulation, emulation, and experimentation techniques into FSP, in order to relax the assumptions we made, especially regarding dynamic route changes and route symmetry.

Acknowledgments

This work has been sponsored in part by Northrop Grumman Information Systems, and by NSF grant CNS–0831353. Part of this work appeared in the Proceedings of the IEEE International Conference on Distributed Computing Systems in June 2011.

References

[1] Digital fears emerge after data siege in Estonia, `http://www.nytimes.com/2007/05/29/technology/29estonia.html` (2007).

[2] The network simulator, `http://www.isi.edu/nsnam/ns/`.

[3] R. Chertov, S. Fahmy, Forwarding devices: From measurements to simulations, ACM Transactions on Modeling and Computer Simulation 21 (2) (2011) 12:1–12:23.

[4] Network emulation testbed, `http://www.emulab.net/`.

[5] Cyber-defense technology experimental research laboratory testbed, `http://www.isi.edu/deter/`.

[6] Parallel/distributed ns, `http://www.cc.gatech.edu/computing/compass/pdns/index.html`.

[7] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, J. Lepreau, Large-scale virtualization in the Emulab network testbed, in: Proc. of USENIX ATC, 2008.

[8] S. Robinson, Simulation: The Practice of Model Development and Use, Wiley, 2004.

[9] R. Ricci, C. Alfeld, J. Lepreau, A solver for the network testbed mapping problem, in: SIGCOMM Comput. Commun. Rev., 2003.

[10] K. Schloegel, G. Karypis, V. Kumar, The sourcebook of parallel computing, Morgan Kaufmann Publishers Inc., 2003, Ch. Graph partitioning for high-performance scientific simulations.

[11] M. C. Weigle, P. Adurthi, F. Hernández-Campos, K. Jeffay, F. D. Smith, Tmix: a tool for generating realistic TCP application workloads in ns-2, in: SIGCOMM Comput. Commun. Rev., 2006.

[12] K. V. Vishwanath, A. Vahdat, Realistic and responsive network traffic generation, in: Proc. of SIGCOMM, 2006.

[13] J. Sommers, P. Barford, Self-configuring network traffic generation, in: Proc. of IMC, 2004, pp. 68–81.

[14] F. Papadopoulos, K. Psounis, R. Govindan, Performance preserving topological downscaling of Internet-like networks, IEEE Journal on Selected Areas in Communications 24 (12) (2006) 2313–2326.

[15] M. Berger, S. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, IEEE Transactions on Computers 100 (36) (1987) 570–580.

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

[17] J. Cao, W. Cleveland, Y. Gao, K. Jeffay, F. Smith, M. Weigle, Stochastic models for generating synthetic HTTP source traffic, in: Proc. of INFOCOM, 2004.

[18] F. Papadopoulos, K. Psounis, Efficient identification of uncongested Internet links for topology downscaling, in: SIGCOMM Comput. Commun. Rev., 2007.

[19] N. Spring, R. Mahajan, D. Wetherall, T. Anderson, Measuring ISP topologies with rocketfuel, Networking, IEEE/ACM Transactions on 12 (1) (2004) 2–16.

[20] Rocketfuel to ns tool suite, `http://www.cs.purdue.edu/homes/fahmy/routing.html`.

[21] Distributed intrusion detection system, `http://www.dshield.org/`.

[22] J. Mirkovic, A. Hussain, S. Fahmy, P. Reiher, R. Thomas, Accurately measuring denial of service in simulation and testbed experiments, IEEE Transactions on Dependable and Secure Computing 6 (2) (2009) 81–95.

[23] H. Kim, H. Lim, J. C. Hou, Accelerating simulation of large-scale IP networks: A network invariant preserving approach, in: Proc. of INFOCOM, 2006.

[24] R. Pan, B. Prabhakar, K. Psounis, D. Wischik, SHRiNK: a method for enabling scalable performance prediction and efficient network simulation, IEEE/ACM Transactions on Networking 13 (5) (2005) 975–988.

[25] W. Yao, S. Fahmy, Downscaling network scenarios with denial of service (DoS) attacks, in: Proc. of Sarnoff Symposium, 2008.

[26] D. Mosberger, T. Jin, httperf-a tool for measuring web server

performance, in: SIGMETRICS Perform. Eval. Rev., 1998.
[27] D. Eun, N. Shroff, Simplification of network analysis in large-bandwidth systems, in: Proc. of INFOCOM, 2003.
[28] B. Petit, M. Ammar, R. Fujimoto, Scenario-specific topolgy reduction in network simulations, in: Proc. of SPECTS, 2005.
[29] P. Sanaga, J. Duerig, R. Ricci, J. Lepreau, Modeling and emulation of Internet paths, in: Proc. of NSDI, 2009.
[30] N. Weaver, I. Hamadeh, G. Kesidis, V. Paxson, Preliminary results using scale-down to explore worm dynamics, in: Proc. of ACM workshop on Rapid malcode, 2004.
[31] G. Carl, S. Phoha, G. Kesidis, B. B. Madan, Path preserving scale down for validation of internet inter-domain routing protocols, in: Proc. of Winter Simulation Conference (WSC), 2006.
[32] V. Krishnamurthy, M. Faloutsos, M. Chrobak, L. Lao, J.-H. Cui, A. Percus, Reducing large Internet topologies for faster simulations, in: Proc. of IFIP Networking, 2005.
[33] D. M. Nicol, J. Liu, M. Liljenstam, G. Yan, Simulation of large-scale networks using SSF, in: Proc. of Winter Simulation Conference (WSC), 2003.
[34] B. Walker, J. Seastrom, G. Lee, K. Lin, Addressing scalability in a laboratory-based multihop wireless testbed, in: Mobile Networks and Applications, 2009.
[35] K. Yocum, E. Eade, J. Degesys, D., Becker, J. Chase, A. Vahdat, Toward scaling network emulation using topology partitioning, in: Proc. of MASCOTS, 2003.
[36] B. Lantz, B. Heller, N. McKeown, A network in a laptop: Rapid prototyping for software-defined networks, in: Proc. of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, 2010.
[37] B. Heller, N. Handigol, V. Jeyakumar, B. Lantz, N. McKeown, Reproducible network experiments using container based emulation, in: Proc. of CoNEXT, 2012.
[38] J. Ahrenholz, C. Danilov, T. Henderson, J. Kim, Core: A real-time network emulator, in: Military Communications Conference, 2008. MILCOM 2008. IEEE, 2008.
[39] J. Liu, Y. Li, Y. He, A large-scale real-time network simulation study using prime, in: Proc. of Winter Simulation Conference (WSC), 2009.
[40] ns-3, `http://www.nsnam.org/`.
[41] S. Wang, Y. Huang, NCTUns distributed network emulator, Internet Journal 4 (2) (2012) 61–94.
[42] D. Gupta, K. Vishwanath, A. Vahdat, DieCast: Testing distributed systems with an accurate scale model, in: Proc. of NSDI, 2008.