Partitioning Network Testbed Experiments

Wei-Min Yao, Sonia Fahmy
Department of Computer Science
Purdue University
E-mail: {wmyao, fahmy}@cs.purdue.edu

***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 in two iterations. In the second iteration, we model dependent partitions using information gathered about both the traffic *and* the network conditions during the first 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. [1]

***Keywords*-simulation; emulation; network partitioning;**

I. Introduction

Understanding the behavior of large-scale systems is critical when designing and validating a new Internet protocol or application. However, investigating large systems is difficult. 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. 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 [1]. It is important to explore defenses against this attack using realistic scenarios, but it is undesirable to perform attack experiments on the operational Internet. Testbed experiments are vital for studying how well a defense would work against such a large scale attack. Due to this, several countries have invested in large security testbeds, popularly referred to as *cyber-ranges*.

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 [8], [6], 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 in two iterations.

[1] This work was funded in part by Northrop Grumman Information Systems, and by NSF grant CNS–0831353.

The key contributions of our work include (1) our novel approach and tool to automatically partition a large experiment into sequential small experiments based on *network flows and the dependencies among them*, (2) our iterative approach to modeling interacting small experiments, and (3) our comparisons of different approaches via both simulations and 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. FSP can be integrated with any existing scaling solution. 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 remainder of this paper is structured as follows. Section II defines our notation and assumptions. Sections III, IV, and V explain our proposed method, FSP. Sections VI and VII describe the experiments used to validate FSP.

Section VIII summarizes related work. We conclude in Section IX.

II. 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.

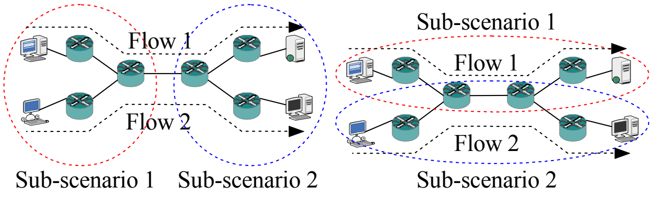
We model the network topology as a graph $G = (V,E)$ with vertex set V, representing the routers and end hosts in the network, and edge set E, representing the links in the network. $|V|$ and $|E|$ denote the number of vertices and number of edges in the graph, respectively. The flow information describes 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 direction of the flow. Traffic flowing between the same source and destination nodes is grouped into the same 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 *direction* of the traffic indicates whether it is unidirectional or bidirectional.

We initially make the following simplifying assumptions. 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 core routers, and in most network simulators, e.g., ns-2 [2].

Our approach relies 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 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).

III. Overview of FSP

Our proposed method, which we refer to as *flow-based scenario partitioning* (FSP), does not partition the network nodes, as with partitioning approaches for parallel and distributed simulation [6]. 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).



(a) Partitioning the network directly (b) Flow-oriented partitioning

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. Section IV gives the details of this phase. 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*.

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 V gives the details of this phase. The overall FSP approach is summarized in Algorithm 1.

IV. 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.

A. 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).

Algorithm 1 Flow-based Scenario Partitioning (FSP)

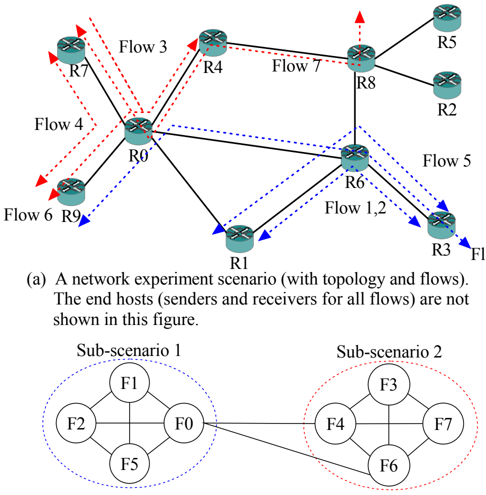
```
FLOW-BASED PARTITIONING($network, flows, maxNode$)
  Input: A network scenario with topology ($network$),
         flow information ($flows$), and $maxNode$
  Output: Estimate of results for original network scenario.

  ▷ Phase 1: Partition the input network scenario
  $fdg \leftarrow$ BUILD-FDG($network, flows$)
  $Parts \leftarrow$ PARTITION($fdg, maxNode$)
  ▷ Phase 2, Iteration 1
  ▷ Collect traces in case of interaction among partitions.
  for ($P \in Parts$)
      do Conduct experiment for sub-scenario $P$
         for each $f \in P$
             do for each $f' \in fdg.neighbors(f)$
                    do if ($f' \notin P$)
                           then 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.
  repeat
         for ($P \in Parts$) ▷ For interacting partitions only
             do for each $f' \notin P$
                    do $sharedPath \leftarrow$ ($f'$'s path) $\cap$ (links in P)
                       if $sharedPath \neq \emptyset$
                           then Import model (e.g., Tmix)
                                of $f'$ on $sharedPath$ (in $P$).
         Conduct experiment for sub-scenario $P$
    until Convergence of results (i.e., twice)
```

A flow dependency graph, FDG = (F_V, F_E, fn_v, fn_e), is a weighted graph with vertex set F_V, edge set F_E, vertex weight function fn_v, and edge weight function fn_e. Algorithm 2 gives the steps for FDG construction. 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 fn_v function we use in our current implementation sets the weight of a vertex (i.e., flow) to the number of nodes (routers or hosts in the network) in the path of the flow. 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 VI.

In Algorithm 2, we insert all flows in scenario S as vertices in F_V. We then insert edges into the FDG based on the routes



(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 $\{F3, F4, F6, F7\}$.

Algorithm 2 Constructing the flow dependency graph

```
BUILD-FDG($network, flows$)
    Input: A network scenario with topology ($network$)
           and flow information ($flows$).
    Output: A flow dependency graph ($fdg$)
    ▷ Initialize the flow dependency graph.

    $fdg \leftarrow$ an empty graph
    for each $f \in flows$
        do $fdg$.addNode($f$)
    ▷ Add edge if the paths of two flows have
      common links in $network$.
    for each $(flowA, flowB) \in fdg$.nodes()
        do $commonlinks \leftarrow flowA.path \cap flowB.path$
           if $commonlinks \neq \emptyset$
              then $newEdge \leftarrow (flowA, flowB)$
                   $newEdge.commonlinks \leftarrow commonlinks$
                   $fdg$.addEdge($newEdge$)
    ▷ Remove uncongested links. Remove the edge if
      all common links are removed.
    $freeLinks \leftarrow$ UNLOADED-LINKS($network, flows$)
    for each $edge \in fdg$.edges()
        do $edge.commonlinks \leftarrow$
              $edge.commonlinks - freeLinks$
           if $edge.commonlinks = \emptyset$
              then $fdg$.removeEdge($edge$)
```

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 no 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.

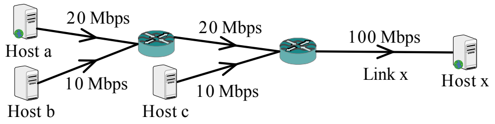


Figure 3. The solid lines are (unidirectional) physical links and the link capacity is shown next to the link. Regardless of the type of flows, the aggregate throughput on link x cannot exceed 30 Mbps.

The connected components in the FDG are automatically assigned as partitions of the graph. The size of a partition is defined as the number of routers and hosts used by the flows in that partition. We *merge* small partitions if their total size is less than available nodes (*maxNode*), in order to limit the number of experiments to be conducted. However, if the size of any partition (connected component in the FDG) exceeds *maxNode*, we must further sub-divide this partition, as discussed next.

B. 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. Our next step is to further divide the FDG such that each partition needs at most *maxNode* nodes (routers and hosts). Ideally, we would like to partition the FDG such that there is as little interaction as possible among the components. 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]. Algorithm 3 gives the complete algorithm.

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 the 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 (line 5 of GGGP()), the algorithm will add new vertices into the region. A vertex that is adjacent to the current region and has the smallest edge-cut (number of edges that connect it to nodes in that region) is selected in each iteration. Since the algorithm is sensitive to the choice of the initial vertex, we randomly select the initial vertex and repeat the process five times. The partition with the smallest edge-cut is selected as the final output.

As discussed above, the size of a partition in the FDG is not sum of the weights of its vertices (i.e., flows). Since there are nodes shared among the flows in the partition, the total size is the number of *distinct network nodes* traversed by the flows. Therefore, the size of a partition that is used in the partitioning algorithm (line 6 of PARTITION() and lines 4 and 5 of GGGP()) represents the number of network nodes required to execute an experiment with that partition.

When computing the weight of the edge-cut between two partitions (line 8 of GGGP()), we do not simply add the weights of edges; rather, we compute the distinct network nodes included in the edge-cut. We evaluate this choice in our simulation experiments.

V. 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 certain 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 workload on these n shared links for flows in $\{S_2, \cdots, S_m\}$ (since these

Algorithm 3 Graph partitioning algorithm

```
PARTITION($fdg, maxNode$)
    Input: A flow dependency graph ($fdg$) and
           a positive integer $maxNode$.
    Output: A set of $fdg$ partitions (each partition
           is a set of vertices in $fdg$).
1   $maxFlowNode \leftarrow$ the maximum number of nodes
        a flow will visit in the original scenario.
2   if $maxNode < maxFlowNode$
3       then return $\emptyset$ ▷ Error, $maxNode$ too small.
    ▷ Partition $fdg$ such that each partition needs no
        more than $maxNode$ nodes.
4   $initPart \leftarrow fdg$.nodes()
5   $PartSet \leftarrow \{initPart\}$
6   while ($P \in PartSet$) and ($\text{REQNODES}(P) > maxNode$)
7       do $PartSet \leftarrow PartSet - \{P\} + \text{GGGP}(P)$
8   return $PartSet$

GGGP($P$)
     Input: A partition $P$
     Output: Two partitions $(P_1, P_2)$
     ▷ Generate two initial partitions of $P$.
 1   $v \leftarrow$ a random vertex in $P$
 2   $P_1 \leftarrow \{v\}$
 3   $P_2 \leftarrow P - \{v\}$
     ▷ Move vertices from $P_2$ to $P_1$ until their weights
        are balanced.
 4   $halfNodes \leftarrow \text{REQNODES}(P)/2$
 5   while ($\text{REQNODES}(P_1) < halfNodes$)
     ▷ Move the best candidate vertex ($candV$) from $P_2$ to $P_1$.
 6       do for each $candV \in P_2$
 7               do $cutEdges \leftarrow$ all $fdg$ edges between
                        $(\{P_1 + candV\}, \{P_2 - candV\})$.
 8                  $cutWeight[candV] \leftarrow$
                        $\text{WEIGHT}(cutEdges)$
 9           Let $cutWeight[vertex]$ be the smallest
             value in $cutWeight[\cdot]$
10           $P_1 \leftarrow P_1 + \{vertex\}$
11           $P_2 \leftarrow P_2 - \{vertex\}$
12   return $\{P_1, P_2\}$

REQNODES($s$)
    Input: A set of vertices in an $fdg$
1   for each $flow \in s$.nodes()
2       do $nodeSet \leftarrow nodeSet \cup$ {nodes on $flow$'s path}
3   return $|nodeSet|$

WEIGHT($s$)
    Input: A set of edges in an $fdg$
1   for each $(f_1, f_2) \in s$
2       do $sharedLinks \leftarrow$ $f_1$'s path $\cup$ $f_2$'s path
3          $nodeSet \leftarrow nodeSet \cup$ {nodes on $sharedLinks$}
4   return $|nodeSet|$
```

flows are not in S_1). To achieve this, we propose to conduct the experiments in two iterations.

A. *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, there can be a network link l that exists in both sub-scenarios S_1 and S_2, and there are two flows $f_1 \in S_1$, $f_2 \in S_2$ on link l in the original scenario. When we conduct the experiment for S_1 in the first iteration, flow f_2 will not generate any workload on link l since it is not included in S_1, and f_1 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 f_1 may increase since f_1 does not need to compete for bandwidth with f_2. Therefore, we collect packet traces for these interacting flows, and then use information computed from these traces to generate workload that models the missing flows.

B. *Second Iteration*

In the second iteration, we sequentially conduct experiments for each sub-scenario that interacts with others, but we now incorporate information computed from the first iteration to model its interacting sub-scenarios. Measurements collected in this final iteration approximate the results of the original experiment.

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 three tools (1) Tmix [11], (2) Harpoon [13], and (3) Swing [12] to (i) process packet traces collected during the first iteration, and (ii) model non-shared network conditions and generate application workloads in the second 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.

C. *Illustrative Examples*

To understand how the two iterations of the second phase of FSP work, 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.

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 IV.

Congested scenarios. We now increase the load in the network to increase interaction among flows. Table I lists the average results for a heavily loaded network with 8 FTP flows ($\lambda = 2$), repeating each experiment 10 times. The left side of the table gives the results of the original network scenario, and the rightmost columns list the percentage difference between the original scenario and the partitioned sub-scenarios. For example, if flow 0 sent 100 packets in the original network and 120 packets in the sub-scenarios, we indicate the difference as 20%.

Table I
RESULTS FOR 8 FTP FLOWS ($\lambda = 2$).

	Original		Difference	
Flow	Sent (Packets)	Goodput (Mbps)	Sent (%)	Goodput (%)
0	89189.2	38.52	-1.57	-1.42
1	46636.2	21.58	1.59	1.78
2	89164.5	51.73	-0.69	-0.71
3	103391.8	40.74	0.66	0.69
4	77328.8	36.03	-1.66	-1.70
5	77194	29.83	0.22	0.33
6	81920.4	43.97	-1.39	-1.47
7	62159.5	30.65	1.43	1.47

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” between the link and each Tmix traffic generator. This delay box introduces delays representing the one way portion of the RTT of each flow, minus the propagation 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 of the non-shared links* from the trace collected in the first 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 application behavior from the trace and represents it as *connection vectors* [11]. The results in Table I demonstrate that long-term metrics, such as the average goodput of a flow, can be reasonably predicted using our method. Transient behavior of the flows is not preserved due to this simple Tmix configuration that does not capture dynamics of the interacting partitions, and simply uses average values over the entire experiment.

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 II. 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 16.22, 11.16, and 13.75 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 II
GOODPUT OF 8 FTP FLOWS (MBPS) IN DIFFERENT ITERATIONS.

Flow	Original	Iteration 1	Iteration 2
0	29.55	45.77	33.68
1	40.41	38.99	40.37
2	49.88	38.63	49.53
3	46.16	39.07	47.45
4	29.03	40.19	29.17
5	40.20	38.67	39.42
6	32.18	45.93	33.23
7	39.72	33.98	41.24

After the second iteration, we are better able to predict the goodput of flows 4 and 6. However, the goodput of flow 0 is still 5.32 Mbps higher 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 should 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.

D. Examples with Multiple Interacting Partitions

We now examine other aspects of the tradeoff between experimental fidelity and complexity. Consider open-loop flows and assume we will simply replay traces during the second iteration and *not* use a tool like Tmix to model network conditions. 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.

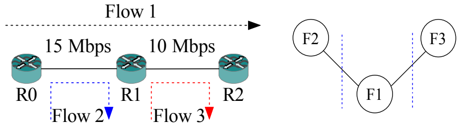


Figure 4. A scenario with a sub-scenario that interacts with two others.

For example, consider the network scenario shown in Fig. 4. In this simple scenario, there are three UDP flows, each belonging to a different partition ($F1 \in P_1, F2 \in P_2, F3 \in P_3$). Table III gives the goodput for all three flows after each iteration. The goodputs of flow 1 and flow 2 can be collected after iteration 2, as they do not change in subsequent iterations. The goodput of flow 3 is incorrect until the third iteration. This is because, in this network, the traffic on the bottleneck link R1-R2 depends on all three flows, and the effect of flow 2 on flow 1 will not have propagated to the packet trace on link R1-R2 after the first iteration.

Now consider closed-loop flows and the use of Tmix. We again use the network scenario shown in Fig. 4, with three FTP flows, and 1 ms propagation delay on all links. The results are given in Table IV. We observe that the results stabilize after the second iteration.

Table III
GOODPUT OF UDP FLOWS IN KBPS.

Flow	Original	Iteration 1	Iteration 2	Iteration 3
1	4902	6064	4850	4850
2	5057	5644	4469	4469
3	1513	2646	1225	1546

Table IV
GOODPUT OF FTP FLOWS IN KBPS.

Flow	Original	Iteration 1	Iteration 2	Iteration 3
1	4014	9333	5416	5416
2	10684	12705	8542	8542
3	5756	10039	6600	6600

Unlike the open-loop case, where we naively replayed the packet trace, we employ Tmix to *generate the application workload* and to *model the network conditions* in interacting partitions. Two iterations are now sufficient because, in the second iteration, we are already modeling network conditions that a flow encounters on parts of the network not represented in this sub-scenario. The application behavior extracted by Tmix for all three 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) inferred by Tmix from the traces are stable from the second iteration onwards. We repeat this experiment with HTTP flows, with different levels of network congestion (with high and low packet loss) and the results are always consistent: results after the second iteration are close to the original scenario. Table V gives a sample result from a highly congested scenario.

Table V
GOODPUT OF HTTP FLOWS IN KBPS. ITERATIONS 6 ONWARDS YIELD THE SAME RESULTS AS ITERATION 5.

Flow	Original	Iter. 1	Iter. 2	Iter. 3	Iter. 4	Iter. 5
1	3607	9748	3610	3617	3563	3529
2	10961	14490	10487	10801	10550	10744
3	5902	9734	5585	5684	5640	5649

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 RTT 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.

VI. 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 [18] topologies representing the backbone network using the Rocketfuel-to-ns tool [19]. 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.

A. 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 network topology) among all the shared network links represented in the cut (Section IV). 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 sample alternative 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. 5 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 averaged results from 30 experiment runs are plotted. From Fig. 5, 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.

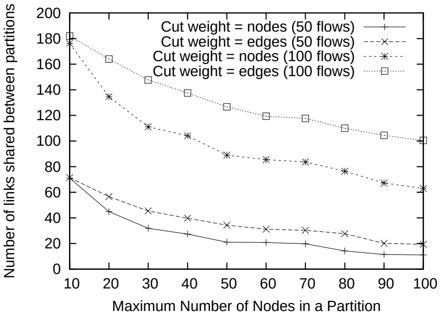


Figure 5. Number of shared links among sub-scenarios with two methods to calculate the weight of an edge cut.

B. 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 According to Algorithm 1, the time complexity of the first phase of FSP, i.e., partitioning a large network scenarios into sub-scenarios, is decided by the graph partitioning algorithm (Algorithm 3). 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. This includes our functions to determine the size of a partition and weight of an edge cut (REQNODES() and WEIGHT() in Algorithm 3).

We computed the run-time for partitioning scenarios with 100 to 500 flows, randomly generating 30 scenarios for each value of the number of flows. As expected, 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 infrequently, and hence FSP is still feasible. Moreover, our current implementation 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 data structures and program optimization techniques.

Due to space limitations, more partition characteristics such as the number of partitions can be found in [20].

VII. BOTNET EXPERIMENTS

Denial of Service (DoS) attacks have been launched against Internet sites for decades, and distributed DoS attacks are one of the hardest assaults 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 in this network topology. 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.

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 continues 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 and 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. 6 and Fig. 7 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.

A closer look at the results in Fig. 6 and Fig. 7 reveals that a few flows (e.g., flow 9) have a lower success ratio in the partitioned experiments. Comparing Fig. 6 and Fig. 7, the success ratios for HTTP/1.1 sessions are lower than the HTTP/1.0 ones, and the results from the second iteration have a greater error for HTTP/1.1. This is because HTTP/1.1 uses 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

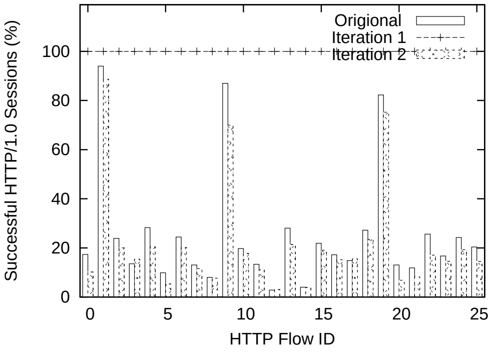


Figure 6. Percentage of successful HTTP/1.0 sessions. Only the first 25 flows are shown due to space constraints.

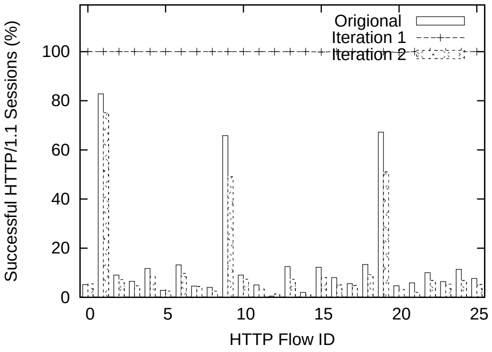


Figure 7. Percentage of successful HTTP/1.1 sessions. Only the first 25 flows are shown due to space constraints.

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. 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. Readers can refer to [20] for a detailed comparison between the three tools and the comparison between FSP and a downscaling technique.

VIII. 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.* [23] 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. Instead of sampling traffic flows, Kim *et al.* propose TranSim [24] 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.* [25]. DSCALE includes two methods, DSCALEd and DSCALEs, that prune uncongested network links, based on earlier work on queuing networks [26]. Petit *et al.* [27] 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 [28].

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], [29]. For example, Walker *et al.* [30] 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 such as in [7], [31], [32]. 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, such as DoS experiments [3].

Our approach is orthogonal to techniques in both categories, and can be easily combined 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], [28].

IX. Conclusions and Future Work

In this paper, we have proposed a platform-independent mechanism, FSP, to partition flows in a large network experiment into a set of smaller experiments that can be *sequentially* executed in two iterations. 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 several 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.

References

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

[2] "Ns-2," http://www.isi.edu/nsnam/ns/.

[3] R. Chertov and S. Fahmy, "Forwarding devices: From measurements to simulations," *ACM Transactions on Modeling and Computer Simulation*, 2010, accepted for publication.

[4] "Emulab," http://www.emulab.net/.

[5] "Deter," http://www.isi.edu/deter/.

[6] "Pdns - 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, and 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, and J. Lepreau, "A solver for the network testbed mapping problem," in *SIGCOMM Comput. Commun. Rev.*, 2003.

[10] K. Schloegel, G. Karypis, and 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, and 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 and A. Vahdat, "Realistic and responsive network traffic generation," in *Proc. of SIGCOMM*, 2006.

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

[14] F. Papadopoulos and K. Psounis, "Efficient identification of uncongested Internet links for topology downscaling," *SIGCOMM Comput. Commun. Rev.*, 2007.

[15] M. Berger and S. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Transactions on Computers*, vol. 100, no. 36, pp. 570–580, 1987.

[16] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, 1999.

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

[18] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring ISP topologies with rocketfuel," *Networking, IEEE/ACM Transactions on*, vol. 12, no. 1, pp. 2–16, 2004.

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

[20] W. Yao and S. Fahmy, "Partitioning network experiments," Department of Computer Science, Purdue University, Tech. Rep., 2011, http://www.cs.purdue.edu/homes/wmyao/papers/fsp2011.pdf.

[21] "Dshield," http://www.dshield.org/.

[22] J. Mirkovic, A. Hussain, S. Fahmy, P. Reiher, and R. Thomas, "Accurately measuring denial of service in simulation and testbed experiments," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 81–95, Apr/June 2009.

[23] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik, "SHRiNK: a method for enabling scalable performance prediction and efficient network simulation," *IEEE/ACM Transactions on Networking*, 2005.

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

[25] F. Papadopoulos, K. Psounis, and R. Govindan, "Performance preserving topological downscaling of Internet-like networks," *IEEE Journal on Selected Areas in Communications*, 2006.

[26] D. Eun and N. Shroff, "Simplification of network analysis in large-bandwidth systems," in *Proc. of INFOCOM*, 2003.

[27] B. Petit, M. Ammar, and R. Fujimoto, "Scenario-specific topolgy reduction in network simulations," in *Proc. of SPECTS*, 2005.

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

[29] D. M. Nicol, J. Liu, M. Liljenstam, and G. Yan, "Simulation of large-scale networks using SSF," in *Proc. of the 35th conference on Winter simulation*, 2003.

[30] B. Walker, J. Seastrom, G. Lee, and K. Lin, "Addressing scalability in a laboratory-based multihop wireless testbed," *Mobile Networks and Applications*, August 2009.

[31] D. Gupta, K. Vishwanath, and A. Vahdat, "DieCast: Testing distributed systems with an accurate scale model," in *Proc. of NSDI*, April 2008.

[32] K. Yocum, E. Eade, J. Degesys, D., Becker, J. Chase, and A. Vahdat, "Toward scaling network emulation using topology partitioning," in *Proc. of MASCOTS*, 2003.