High-Performance Simulation of Low-Resolution Network Flows

David M. Nicol
Guanhua Yan
Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
nicol@iti.uiuc.edu

Simulation of large-scale networks demands that we model some flows at coarser time scales than others, simply to keep the execution cost manageable. This article studies a method for periodically computing traffic at a time scale larger than that typically used for detailed packet simulations. Applications of this technique include computation of background flows (against which detailed foreground flows are simulated) and simulation of worm propagation in the Internet. The approach considers aggregated traffic between Internet points of presence (POPs) and computes the throughput of each POP-to-POP flow through each router on its path. This problem formulation leads to a nonlinear system of equations. The authors develop means of reducing this system to a smaller set of equations, which are solved using fixed-point iteration. They study the convergence behavior, as a function of traffic load, on topologies based on Internet backbone networks. They find that the problem reduction method is very effective and that convergence is achieved rapidly. The authors also examine the comparative speedup of the method relative to using pure packet simulation for background flows and observe speedups exceeding 3000 using an ordinary PC. They also simulate foreground flows interacting with background flows and compare the foreground behavior using their solution with that of pure packet flows. They find that these flows behave accurately enough in their approach to justify use of the technique in their motivating application. The authors parallelize the algorithm on a distributed-memory multiprocessor. They exploit the flexibility offered by noncommittal barrier synchronization that permits a processor to handle computation messages even after it invokes a barrier primitive. They also take advantage of application-specific knowledge to minimize synchronization cost, study the performance of their parallel algorithm with both fixed and scaled problem sizes, and observe excellent scalability on a multiprocessor supercomputer.

Keywords: Network simulation, multi-scale modeling, traffic modeling

1. Introduction

Simulation of large-scale wireline networks has many applications. In some of these, only a small fraction of traffic is of specific interest (e.g., the behavior of flows managed by a new transport protocol or control traffic between BGP routers). However, the flows of interest are affected by and may even (to a lesser degree) affect the other “background” flows. There is strong motivation to model background flows with less detail than foreground flows, with the objective of significantly lowering the computational requirements. Control traffic, such as BGP announcements and DNS queries, needs to be modeled at the packet level, yet the network being simulated has hundreds of thousands of devices and (typically) tens of thousands of flows represented at any given time. The only hope we have of meeting real-time constraints is to significantly aggregate our treatment of background traffic. In other applications, the traffic of interest can be modeled at a coarse time scale. For example, simulation of worm spread across the Internet can be usefully modeled at a time scale much larger than is usually used in network simulation, and the volume of traffic is so large that this becomes necessary.

Our problem is this: given a description of flow intensities at ingress points, efficiently determine link loads throughout the network interior and determine flow intensities at egress points. Solution to this problem allows us to represent these flows to a more detailed traffic model in terms of the demand they make on shared bandwidth (and, potentially, buffer space). These demands are periodically recomputed (e.g., every 5 seconds). On large networks, this method will compute bandwidth consumption significantly faster than will packet representation of the same flows.

SIMULATION, Vol. 82, Issue 1, January 2006 21-42
©2006 The Society for Modeling and Simulation International
DOI: 10.1177/0037549706066093

These savings naturally come with a cost. Low-resolution background flows may be less responsive to changes in network state, they may exhibit less burstiness, and the degree to which a simulation using them differs from one with all flows at the same resolution will be unknown. Nevertheless, there are contexts where the trade-offs in favor of computational speed are acceptable; indeed, there are contexts where one really has no other option but to use highly abstracted representation of background traffic. Our research group is developing a network simulator that will be used in cyber-defense training exercises, where the most important accuracy requirements are that the simulated behavior have a realistic "look and feel." The simulator computes the detailed effects on particular traffic flows (e.g., financial transactions) as cyber-attacks occur and as defensive measures are taken. Two defining characteristics are that its application demands real-time performance (e.g., one second of simulation time takes no more than one second of wall-clock time to advance) and that its application involves networks with potentially hundreds of thousands of devices.

The simulation problem becomes nontrivial when we attempt to capture the effects of bandwidth sharing among flows across a common link. In our motivating application, it will run with a periodicity larger than normal end-to-end network latency. This forces us to formulate the problem as though flows pass instantaneously through the network, which leads to a system of nonlinear equations whose solution gives the desired background flows. We propose a fixed-point algorithm for its solution. Our formulation has the property that intermediate-solution approximations are all "plausible" in the sense that important problem constraints are not violated, which means that we can use an intermediate approximation if we run out of time to complete the computation or discover that the fixed-point iterations are not converging.

We empirically analyze this technique on models of real Internet backbone networks, with synthetic traffic generated using Poisson Pareto burst processes (PPBP) [1]. We examine convergence behavior as a function of traffic load, the speedup it offers, and the accuracy of foreground traffic when it is used in place of packet simulation. Our experiments demonstrate the viability of the approach for our motivating application.

We parallelize the algorithm on a distributed-memory multiprocessor. When synchronizing the processors, we encounter the problem that it is difficult for each processor to predict whether new computation messages will arrive in the future. We thus exploit the flexibility offered by non-committal barrier synchronization that allows processors to receive computation messages even after they invoke the barrier primitive. To minimize the synchronization cost, we exploit application-specific knowledge to postpone each processor's entrance into a barrier. We study the performance of our parallel algorithm on a multiprocessor supercomputer. The experiments with both fixed and scaled problem sizes demonstrate excellent scalability of our algorithm.

The remainder of this article is organized as follows. In section 2, we formulate the problem to be investigated. In section 3, a sequential algorithm is presented, followed by a discussion of its convergence behavior, performance, and accuracy. In section 4, we describe how to parallelize the sequential algorithm on a distributed-memory multiprocessor; simulation results on the scalability of the parallel algorithm with both fixed and scaled problem size are also presented. In section 5, some related work is discussed. Section 6 summarizes this article.

2. Problem Formulation

In our model, the simulation time is discretized into units of length Δ, and we use t_k to denote $k \cdot \Delta$ ($k = 0, 1, 2, \ldots$). We assume that the network being studied consists of n points of presence (POP), denoted by $P_1, P_2, \ldots, P_n$. We use $f_{i,j}$ to represent the aggregate flow between ingress-egress pair $< P_i, P_j >$. For any ingress-egress pair $< P_i, P_j >$ ($1 \le i, j \le n$), we use $T_{i,j}(t)$ to denote the ingress rate of the corresponding aggregate flow at time t. Since the simulation time advances by constant time intervals, we need to discretize the ingress rate for every aggregate flow. During the time interval $[t_k, t_{k+1}]$, the traffic volume emitted from ingress node P_i to egress node P_j is

$$\int_{t_k}^{t_{k+1}} T_{i,j}(t)\, dt.$$

We smooth the burstiness at time scales smaller than Δ. To ensure that the same amount of traffic is injected into the network, the ingress rate of aggregate flow $f_{i,j}$ at discretized time $k \cdot \Delta$ is

$$\frac{1}{\Delta} \times \int_{t_k}^{t_{k+1}} T_{i,j}(t)\, dt.$$

The network is modeled as a collection of routers connected with unidirectional links. The sending endpoint of a link is associated with a router's output port. At an output port, there is an output buffer. We assume that every router adopts the output buffering strategy, although other buffering strategies can be easily incorporated into our algorithm described later. Routing protocols are used to direct traffic for each aggregate flow. The routing decisions are assumed to be static within any discretized time interval. In addition, the time step size Δ is relatively large compared to the typical end-to-end latency of an aggregate flow. Hence, all link latencies are assumed to be zero in the network model.

The remainder of the article focuses on the algorithm that is applied at every time step t_k. We will not express the dependence on t_k in the notation—it will be understood

that ingress rates are offered at a time-step epoch, and the goal is to compute the rates at which the flows are delivered at their destinations.

We use set Q to denote the whole set of output ports in the network. For any aggregate flow f, the sequence of output ports that it traverses through at a time-step is by Q_f. Obviously, Q_f is a subset of Q. We use F_q to denote the whole set of aggregate flows that traverse through port q $(q \in Q)$ at a time step. We use $\lambda_{f,q}^{(in)}$ $(f \in F_q)$ to denote the arrival rate of aggregate flow f into port q at a time-step epoch. The sum of all the arrival rates into port q, $\sum_{f \in F_q} \lambda_{f,q}^{(in)}$, is written as $\Lambda_q^{(in)}$. In addition, we use $\widehat{\lambda}_f$ to denote the ingress rate of aggregate flow f from its traffic source at the time-step epoch.

When multiple aggregate flows multiplex at the same output port, how much bandwidth is allocated to each aggregate flow is governed by the port's queuing policy. Consider an output port q implementing the first in, first out (FIFO) queuing policy. Let μ_q denote the link bandwidth associated with q. According to the FIFO service discipline, the departure rate of flow f $(f \in F_q)$ at the time-step epoch, denoted by $\lambda_{f,q}^{(out)}$, can be defined as follows:

$$\begin{aligned} \lambda_{f,q}^{(out)} &= \begin{cases} \lambda_{f,q}^{(in)} & \text{if } \Lambda_q^{(in)} \leq \mu_q \quad \text{(case 1)} \\ \lambda_{f,q}^{(in)} \times \frac{\mu_q}{\Lambda_q^{(in)}} & \text{otherwise} \quad \text{(case 2)} \end{cases} \\ &= \lambda_{f,q}^{(in)} \times \min\{1, \frac{\mu_q}{\Lambda_q^{(in)}}\}. \end{aligned} \tag{1}$$

In the first case where the aggregate arrival rate does not exceed the link bandwidth, the output port can serve all the incoming traffic. In the second case, congestion occurs, and the traffic that can not be served is dropped; the FIFO queuing policy dictates that the loss occurring to each flow be proportional to its arrival rate. In equation (1), we do not model the queuing delay, and therefore, there is no time shift between the departure rate and the arrival rate. This is a reasonable assumption because a coarse time scale is being considered.

The first objective is to compute the aggregate traffic load at each port:

$$\sum_{f \in F_q} \lambda_{f,q}^{(in)} \quad \text{for every } q : q \in Q. \tag{2}$$

The rate at which each aggregate flow departs from the network may also be of interest here. For some applications that use time-stepped coarse-grained traffic simulation, the rate at which a node injects traffic into the network is a function of the rate at which it receives traffic from the network at the previous time step. In these cases, another objective can be defined as computing the aggregate departure rate from each POP. To formulate this objective, we define function $\Pi(f, q)$, where $f \in F_q$, by

$$\Pi(f, q) = \begin{cases} q' & \text{if } q' \ (q' \in Q) \text{ is the next output port} \\ & \text{on flow } f\text{'s path after leaving port } q \\ P & \text{if the next hop on flow } f\text{'s path is } P \\ & (P \in \{P_i\}_{1 \leq i \leq n}) \end{cases} \tag{3}$$

Now if we denote the by E the set of router ports that serve as network egress points, we formulate the second objective as computing

$$\sum_{f \in F_q} \lambda_{f,q}^{(out)}, \quad \text{for every } q \in E. \tag{4}$$

3. Sequential Algorithm

3.1 Time-Step Setup

The starting premise of our approach is that we can compute resource consumption of certain types of flows relatively infrequently and treat that consumption as invariant between updates. These flows may be made sensitive to flows modeled with higher resolution, but only at the fixed update points. Upon reaching an update point, we have the opportunity to reassess the bandwidth and buffer space we will allocate on each link. These decisions may be made as a function of observed past behavior. We also have the opportunity to adjust the offered load rate (e.g., reduce it on flows for which loss has been observed in the last time) to model feedback (such as TCP provides). These issues are important but are not the focal point of this article.

When we use this formulation to compute background traffic intensities, it is important to capture the competition between foreground and background flows for link bandwidth. We use a simple mechanism for including the next epoch's anticipated foreground flow as we compute flow rates for background flows. At a time-step epoch, for each port p_z, we compute an estimated foreground input rate $\lambda_f^{(in)}(p_z)$ based on recent observations (or even known predictions, if such were available). We model foreground flow passing through the port as though it is injected into the network at this point from a traffic source, crosses the link, and immediately disappears into a traffic sink. As we compute new flow rates, for that port, we treat that flow exactly as any other flow. The "fair" contribution of foreground traffic is thus considered as we allocate bandwidth for the background traffic.

Since each flow is recomputed each time step, we also have the opportunity to alter routing. This is particularly useful in applications where routing changes in response to changes in traffic loads or where we simulate attacks on the routing infrastructure by disabling routers, eventually causing routing protocols to respond and create new routes. We assume that the routing for the next time step is known before we compute the flow updates.

3.2 Algorithm Description

Description of our algorithm requires notation. We denote by $\vec{\lambda}$ the vector of all flows' arrival rates into all router output ports (and subnetworks, at egress points) through which they flow—it is a complete description of the flow state throughout the system. The variables in $\vec{\lambda}$ are denoted by λ_1, λ_2, ... , and λ_L, where $L = |\vec{\lambda}|$. When we start the algorithm, the variables corresponding to ingress flows are known as boundary conditions. The point of the algorithm is to compute the values of the other variables. Throughout the algorithm, every variable will have an associated state of *unsettled*, *bounded*, or *settled*, depending on whether its value is unknown, has a known upper bound, or is finally known. At any point in the algorithm, notation $S(\lambda_i)$ describes the state of variable λ_i. In the notation we use to describe the algorithm, at any point when $S(\lambda_i) = bounded$, the value of λ_i is the upper bound itself. When $S(\lambda_i) = settled$, the value of λ_i is its final value.

We allow the ingress rate of every flow to change, every time step. For every flow variable in $\vec{\lambda}$ that corresponds to an ingress rate, we set its state to be *settled* and its value to be the new flow ingress rate; for any other variable in $\vec{\lambda}$, we set its state to be *unsettled* and its value to be ∞. As the algorithm develops, the state transition of a flow variable can only be either from *unsettled* to *bounded*, from *unsettled* to *settled*, or from *bounded* to *settled*. Furthermore, in the course of the algorithm, we may change the upper bound on a flow variable, but when we do, the newer bound is always smaller. Hence, a change to the state and value of a flow variable always means that more knowledge of it has been gained.

We also ascribe state to router output ports in the course of the algorithm. We say that a port q is *resolved* once all of its input flows are settled because then equation (1) specifies its output rates. We say that a port is *transparent* at some point in the algorithm if it is not resolved, but the sum of upper bounds on its input rates is less than the port bandwidth; every flow into a transparent port has the same output rate as input rate. A port is *unresolved* if it is neither *resolved* nor *transparent*. We use notation $S(q)$ to name the state of port q; every port is initially given state *unresolved*.

The sequential algorithm consists of four phases: *rule-based flow update computation*, *reduced graph generation*, *fixed-point iterations*, and *residual flow update computation*. As we explain the algorithm, a simple example is used for illustration. The network topology is shown in Figure 1; in the network, there are eight output ports (q_0–q_7) and seven flows (f_0–f_6).

After the time-step setup, every port in the topology is in the *unresolved* state. All ingress flow variables are assigned their ingress flow rates and assigned to the *settled* state, and all other flow variables are in the *unresolved* state.

3.2.1 Phase I: Rule-Based Flow Update Computation

The goal of the first phase is to settle as many flow variables in $\vec{\lambda}$ as possible, based on evaluation of preconditions and execution of actions described in Tables 1 and 2. Rule 1 applies to a port in the *unresolved* state, all of whose input flows are settled. The state of such a port is changed to *resolved*, and for each flow that traverses through it, we compute its departure rate and use that to settle the corresponding input flow variable at the next output port (or subnetwork, if the port is an egress point) on its path.

Rule 2's precondition calls for $\Lambda_q^{(in)} \leq \mu_q$; here we understand $\Lambda_q^{(in)}$ to be the sum of current values assigned to port q's input flows, some or all of which may be upper bounds only. Rule 2 recognizes when first a port becomes transparent; in addition to changing the port state, the associated action marks the output flows of all settled input flows as settled.

Rule 3 may be applied after an input flow to a transparent port becomes settled. Rule 4 may be applied after an input flow to a transparent port becomes bounded or its bounded is lowered. Correspondingly, its output rate is certainly bounded and takes the value of the input flow.

Rules 5 and 6 show how to sharpen upper bounds on flows out of an unresolved port. To understand them, let $R_q^{(in)}$ be the sum of all the arrival rates into port q that have already been settled. Necessarily, we have

$$R_q^{(in)} \leq \Lambda_q^{(in)}. \tag{5}$$

The correctness of rules 5 and 6 is established formally with the following lemmas.

LEMMA 1. Given an input flow variable with a settled rate λ at port q, its departure rate from port q is no greater than $\lambda \times \min\{1, \mu_q/R_q^{(in)}\}$.

Proof. First note that under no circumstances may the output rate of a flow through a port exceed its input rate (this explains the '1' argument of the min operator). If $\mu_q/R_q^{(in)} < 1$, then $\mu_q < R_q^{(in)} \leq R_q^{(in)}(, all)$, where $R_q^{(in)}(, all)$ denotes the sum of the final settled values of all input flows. We know by equation (5) that the final departure rate, $\lambda \times \mu_q/R_q^{(in)}(, final)$, must be no greater than $\lambda \times \mu_q/R_q^{(in)}$. □

Whereas rule 5 shows how a settled input rate can affect its corresponding output rate through an as-yet unresolved port, rule 6 shows how a bounded input rate can do so likewise. A proof of correctness follows below.

LEMMA 2. Given an input flow variable with a bounded rate λ at port q, its departure rate from port q will be no greater than $\lambda \times \min\{1, \mu_q/(\lambda + R_q^{(in)})\}$.

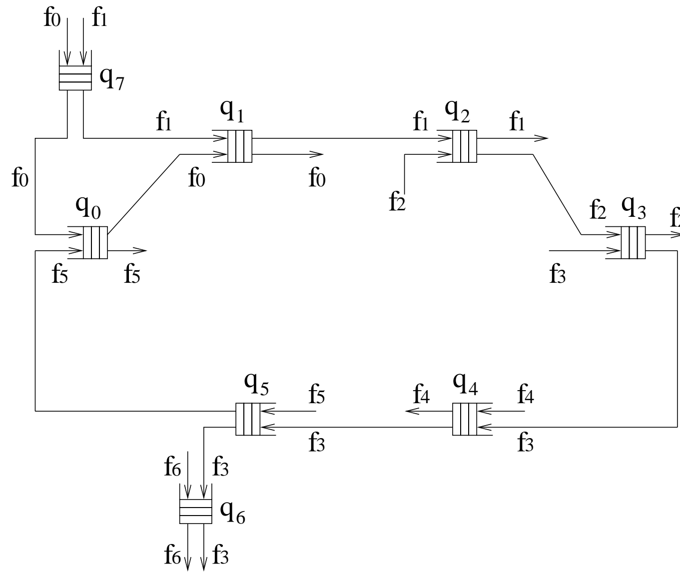

Figure 1. An eight-port topology

Proof. We use λ' to denote the true arrival rate of the flow into port q and λ'' to denote the true departure rate of the flow from port q. Then, $\lambda' \leq \lambda$. We distinguish three cases.

- **Case 1:** if $\lambda + R_q^{(in)} \leq \mu_q$, then the departure rate λ'' satisfies the following:

$$\lambda'' \leq \lambda' \leq \lambda. \tag{6}$$

- **Case 2:** if $\lambda + R_q^{(in)} > \mu_q$ and $\Lambda_q^{(in)} > \mu_q$, then the departure rate λ'' satisfies the following:

$$\begin{aligned}
\lambda'' &= \lambda' \times \frac{\mu_q}{\Lambda_q^{(in)}} \\
&\leq \lambda' \times \frac{\mu_q}{\lambda' + R_q^{(in)}} \\
&\quad (\text{ because } \Lambda_q^{(in)} \geq \lambda' + R_q^{(in)} \text{ }) \\
&= \frac{\mu_q}{1 + \frac{R_q^{(in)}}{\lambda'}} \\
&\leq \frac{\mu_q}{1 + \frac{R_q^{(in)}}{\lambda}} (\text{ because } \lambda' \leq \lambda \text{ }) \\
&= \lambda \times \frac{\mu_q}{\lambda + R_q^{(in)}}.
\end{aligned} \tag{7}$$

- **Case 3:** if $\lambda + R_q^{(in)} > \mu_q$ and $\Lambda_q^{(in)} \leq \mu_q$, then the departure rate λ'' satisfies the following:

$$\begin{aligned}
\lambda'' &= \lambda' \\
&\leq \mu_q - R_q^{(in)} \\
&\leq \mu_q - R_q^{(in)} \times \frac{\mu_q}{R_q^{(in)} + \lambda} \\
&\quad (\text{ because } \lambda + R_q^{(in)} > \mu_q \text{ }) \\
&= \lambda \times \frac{\mu_q}{\lambda + R_q^{(in)}}.
\end{aligned} \tag{8}$$

Combining all three cases, we have proven the lemma. □

Phase I consists of applying these rules and actions until no port satisfies any of the preconditions. After phase I ends, we are ensured that every flow variable is either settled or has a nontrivial upper bound.

Lemma 3. After phase I, every flow variable must be in the settled or bounded state.

Proof. We prove it by contradiction. Suppose that flow variable λ is in the *unsettled* state after phase I. Without loss of generality, suppose that among all flow variables on its flow's path, it appears earliest. Let variable λ' be its predecessor in this sequence (noting that this predecessor must exist, as ingress flow variables are settled), and let q denote the port into which this predecessor flows. If q is in the *resolved* state, then all of its output flows must be settled; if port q is in the *transparent* state, then either rule 3 or rule 4 can be applied to change the state of λ; if port q is in the *unresolved* state, then either rule 5 or rule 6 can be applied on flow variable λ'. Whatever the case, a contradiction follows. □

Table 1. Rules 1, 2, and 3

Rule		
Rule 1	**Precondition:**	$\exists q \in Q\ s.t.$ $S(q) = unresolved \wedge$ $\forall f \in F_q \{S(\lambda^{(in)}_{f,q} = settled)\}$
	Action:	$S(q) \leftarrow resolved$ $\forall f \in F_q$ $\{S(\lambda_{f,\prod(f,q)}) \leftarrow settled$, $\lambda^{(in)}_{f,\prod(f,q)} \leftarrow \lambda^{(in)}_{f,q} \times \min\{1, \mu_q/\Lambda^{(in)}_q\}$
	Explanation:	All the input flow rates to this port are known, so all the output flow rates can be computed.
Rule 2	**Precondition:**	$\exists q \in Q\ s.t.$ $S(q) = unresolved\ \wedge$ $\sim (\exists f \in F_q\ s.t.\ S(\lambda^{(in)}_{f,q}) = unsettled\ ) \wedge$ $\Lambda^{(in)}_q \leq \mu_q\ \wedge$ $\exists f \in F_q\ s.t.\ S(\lambda^{(in)}_{f,q}) = bounded\ \}$
	Action:	$S(q) \leftarrow transparent$, $\forall f \in F_q\ s.t.\ S(\lambda^{(in)}_{f,q}) = settled$ $\{S(\lambda^{(in)}_{f,\prod(f,q)}) \leftarrow settled$, $\lambda^{(in)}_{f,\prod(f,q)} \leftarrow \lambda^{(in)}_{f,q}\}$
	Explanation:	When we first discover that the sum of flow rate upper bounds is no larger than the port bandwidth, we can mark the port as transparent and mark as settled the output flow of any input flow known to be settled.
Rule 3	**Precondition:**	$\exists q \in Q\ s.t.$ $S(q) = transparent \wedge$ $\exists f\ \in\ F_q\ s.t.$ $S(\lambda^{(in)}_{f,q}) = settled\ \wedge$ $\sim (S(\lambda^{(in)}_{f,\prod(f,q)}) =\ settled\ )$
	Action:	$S(\lambda^{(in)}_{f,\prod(f,q)}) \leftarrow\ settled$, $\lambda^{(in)}_{f,\prod(f,q)} \leftarrow \lambda^{(in)}_{f,q}$
	Explanation:	When an input flow to a transparent port becomes settled, we can mark the corresponding output flow as settled also and give it the settled rate.

We use the example topology in Figure 1 to illustrate how the six rules work in phase I. We consider two sets of conditions among flow variables and link bandwidth:

- **Flow Condition Set 1:** Table 3 presents how the rule-based flow update computation process works on this set of flow conditions. Step 1 applies rule 1 to resolve port q_7. Step 2 applies rule 5 to compute the upper bound on $\lambda^{(in)}_{f_0,q_1}$. Because $\lambda^{(in)}_{f_0,q_1} + \lambda^{(in)}_{f_1,q_1} \leq \mu_{q_7}$, port q_1 can be identified as a transparent port; hence, the settled arrival rate of flow f_1 into port q_1 can be propagated through it and then be used to settle $\lambda^{(in)}_{f_1,q_2}$. Thereafter, rule 1 can be used to resolve ports q_2, q_3, q_4, q_5, q_6, and q_0 successively.
- **Flow Condition Set 2:** Table 4 presents how the rule-based flow update computation process works on this set of flow conditions. Step 1 applies rule 1 to resolve port q_7. Step 2 applies rule 5 to compute the upper bound on $\lambda^{(in)}_{f_0,q_1}$. Because $\lambda^{(in)}_{f_0,q_1} + \lambda^{(in)}_{f_1,q_1} > \mu_{q_7}$, port q_1 cannot be identified as transparent (as it was in flow condition set 1). Step 3 applies rule 5 to compute the upper bound on $\lambda^{(in)}_{f_1,q_2}$; step 4 applies rule 5 to compute the upper bound on $\lambda^{(in)}_{f_2,q_3}$; step 5 applies rule 5 to compute the upper bound on $\lambda^{(in)}_{f_3,q_4}$. Port q_4 is identified as transparent in step 6 because $\lambda^{(in)}_{f_4,q_4} + \lambda^{(in)}_{f_3,q_4} \leq \mu_{q_4}$. Step 7 uses rule 4 to compute the upper bound on $\lambda^{(in)}_{f_3,q_5}$. The final step applies rule 5 to compute the upper bound on $\lambda^{(in)}_{f_3,q_6}$ and $\lambda^{(in)}_{f_5,q_0}$.

Both Tables 3 and 4 provide only a sample execution path for each flow condition set because as the processing progresses, there are multiple choices on which ports

Table 2. Rules 4, 5, and 6

Rule		
Rule 4	**Precondition:**	$\exists q \in Q\ s.t.$ $S(q) = transparent \wedge$ $\exists f \in F_q\ s.t.$ $S(\lambda^{(in)f,q}) = bounded \wedge$ $(S(\lambda^{(in)}_{f,\Pi(f,q)}) = unsettled \vee \lambda^{(in)}_{f,\Pi(f,q)} > \lambda^{(in)}_{f,q})$
	Action:	$S(\lambda^{(in)}_{f,\Pi(f,q)}) \leftarrow bounded$, $\lambda^{(in)}_{f,\Pi(f,q)} \leftarrow \lambda^{(in)}_{f,q}$
	Explanation:	When the upper bound on a flow into a transparent port is lowered, we can give the corresponding output flow the same bound.
Rule 5	**Precondition:**	$\exists q \in Q\ s.t.$ $S(q) = unresolved \wedge$ $\exists f \in F_q\ s.t.$ $S(\lambda^{(in)}_{f,q}) = settled \wedge$ $(S(\lambda^{(in)}_{f,\Pi(f,q)}) = unsettled \vee$ $\lambda^{(in)}_{f,\Pi(f,q)} > \lambda^{(in)}_{f,q} \times \min\{1, \mu_q/R^{(in)}_q\})$
	Action:	$S(\lambda^{(in)}_{f,\Pi(f,q)}) \leftarrow bounded$, $\lambda^{(in)}_{f,\Pi(f,q)} \leftarrow \lambda^{(in)}_{f,q} \times \min\{1, \mu_q/R^{(in)}_q\}$
	Explanation:	When an input flow to an unresolved port becomes settled, we can compute an upper bound on the corresponding output flow by assuming that all the port capacity is allocated to the input flows that are currently settled.
Rule 6	**Precondition:**	$\exists q \in Q\ s.t.$ $S(q) = unresolved \wedge$ $\exists f \in F_q\ s.t.$ $S(\lambda^{(in)}_{f,q}) = bounded \wedge$ $(S(\lambda^{(in)}_{f,\Pi(f,q)}) = unsettled \vee$ $\lambda^{(in)}_{f,\Pi(f,q)} > \lambda^{(in)}_{f,q} \times \min\{1, \mu_q/(R^{(in)}_q + \lambda^{(in)}_{f,q})\})$
	Action:	$S(\lambda^{(in)}_{f,\Pi(f,q)}) \leftarrow bounded$, $\lambda^{(in)}_{f,\Pi(f,q)} \leftarrow \lambda^{(in)}_{f,q} \times \min\{1, \mu_q/(R^{(in)}_q + \lambda^{(in)}_{f,q})\}$
	Explanation:	When the upper bound on an input flow to a unresolved port drops, we can recompute an upper bound on the corresponding output flow by assuming that all the port capacity is allocated to the input flows that are currently settled and the flow whose bound just dropped.

should be processed. With flow condition set 1, all ports in the topology are resolved after phase I; it is evident that with the aid of transparency of port q_1, the circular dependence among flow variables in the topology can be entirely removed. When this occurs, the algorithm stops after phase I. With flow condition set 2, however, all ports but port q_7 are still unresolved. If there still exist some unresolved ports after phase I, the algorithm proceeds into phase II.

3.2.2 Phase II: Dependence Graph Generation

After phase I, some output ports may remain unresolved. In the example topology under the second flow condition set, there exists a cycle of unresolved ports: $q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4 \rightarrow q_5 \rightarrow q_0 \rightarrow q_1$. Specifically, this means that q_2 cannot be resolved before a flow from q_1 is settled; likewise q_3 cannot be resolved before a flow from q_2 is settled (which cannot happen before q_2 is resolved) and so on, with a circular dependency emerging. The second phase of our algorithm identifies all the critical flow variables that are involved in such circular dependences. Some unsettled flow variables need not be involved in phase II and III analysis because their values will be completely determined after the critical flow values on cycles are solved. For example, a flow variable out of a transparent port is not critical; it may depend on a critical variable upstream

Table 3. Flow update computation for the topology in Figure 1 (flow condition set 1)

Flow Conditions		
$\lambda^{(in)}_{f_0,q_7} + \lambda^{(in)}_{f_1,q_7} \le \mu_{q_7}$ and $\lambda^{(in)}_{f_0,q_7} + \lambda^{(in)}_{f_1,q_7} \le \mu_{q_1}$		
Step	**Rule**	**Processing**
1	1	$S(q_7) \leftarrow resolved,\ S(\lambda^{(in)}_{f_0,q_0}) \leftarrow settled,\ S(\lambda^{(in)}_{f_1,q_1}) \leftarrow settled$
2	5	$S(\lambda^{(in)}_{f_0,q_1}) \leftarrow bounded,\ \lambda^{(in)}_{f_0,q_1} \leftarrow \lambda^{(in)}_{f_0,q_0}$
3	2	$S(q_1) \leftarrow transparent,\ S(\lambda^{(in)}_{f_1,q_2}) \leftarrow settled,\ \lambda^{(in)}_{f_1,q_2} \leftarrow \lambda^{(in)}_{f_1,q_1}$
4	1	$S(q_2) \leftarrow resolved,\ S(\lambda^{(in)}_{f_2,q_3}) \leftarrow settled,\ \lambda^{(in)}_{f_2,q_3} \leftarrow \lambda^{(in)}_{f_2,q_2} \times \min\{1, \mu_{q_2}/\Lambda^{(in)}_{q_2}\}$
5	1	$S(q_3) \leftarrow resolved,\ S(\lambda^{(in)}_{f_3,q_4}) \leftarrow settled,\ \lambda^{(in)}_{f_3,q_4} \leftarrow \lambda^{(in)}_{f_3,q_3} \times \min\{1, \mu_{q_3}/\Lambda^{(in)}_{q_3}\}$
6	1	$S(q_4) \leftarrow resolved,\ S(\lambda^{(in)}_{f_3,q_5}) \leftarrow settled,\ \lambda^{(in)}_{f_3,q_5} \leftarrow \lambda^{(in)}_{f_3,q_4} \times \min\{1, \mu_{q_4}/\Lambda^{(in)}_{q_4}\}$
7	1	$S(q_5) \leftarrow resolved,\ S(\lambda^{(in)}_{f_3,q_6}) \leftarrow settled,\ \lambda^{(in)}_{f_3,q_6} \leftarrow \lambda^{(in)}_{f_3,q_5} \times \min\{1, \mu_{q_5}/\Lambda^{(in)}_{q_5}\}$
		$S(\lambda^{(in)}_{f_5,q_0}) \leftarrow settled,\ \lambda^{(in)}_{f_5,q_0} \leftarrow \lambda^{(in)}_{f_5,q_5} \times \min\{1, \mu_{q_5}/\Lambda^{(in)}_{q_5}\}$
8	1	$S(q_0) \leftarrow resolved,\ S(\lambda^{(in)}_{f_0,q_1}) \leftarrow settled,\ \lambda^{(in)}_{f_0,q_1} \leftarrow \lambda^{(in)}_{f_0,q_0} \times \min\{1, \mu_{q_0}/\Lambda^{(in)}_{q_0}\}$

Table 4. Flow update computation for the topology in Figure 1 (flow condition set 2)

Flow Conditions		
$\lambda^{(in)}_{f_0,q_7} + \lambda^{(in)}_{f_1,q_7} \le \mu_{q_7},\ \lambda^{(in)}_{f_0,q_5} + \lambda^{(in)}_{f_5,q_5} > \mu_{q_0},\ \lambda^{(in)}_{f_0,q_7} + \lambda^{(in)}_{f_1,q_7} > \mu_{q_1},\ \lambda^{(in)}_{f_2,q_2} + \lambda^{(in)}_{f_1,q_7} > \mu_{q_2},$ $\lambda^{(in)}_{f_2,q_2} + \lambda^{(in)}_{f_3,q_3} > \mu_{q_3},\ \lambda^{(in)}_{f_3,q_3} + \lambda^{(in)}_{f_4,q_4} \le \mu_{q_4},\ \lambda^{(in)}_{f_5,q_5} + \lambda^{(in)}_{f_3,q_3} > \mu_{q_5}$, and $\lambda^{(in)}_{f_3,q_3} + \lambda^{(in)}_{f_6,q_6} > \mu_{q_6}$		
Step	**Rule**	**Processing**
1	1	$S(q_7) \leftarrow resolved,\ S(\lambda^{(in)}_{f_0,q_0}) \leftarrow settled,\ S(\lambda^{(in)}_{f_1,q_1}) \leftarrow settled$
2	5	$S(\lambda^{(in)}_{f_0,q_1}) \leftarrow bounded,\ \lambda^{(in)}_{f_0,q_1} \leftarrow \lambda^{(in)}_{f_0,q_0}$
3	5	$S(\lambda^{(in)}_{f_1,q_2}) \leftarrow bounded,\ \lambda^{(in)}_{f_1,q_2} \leftarrow \lambda^{(in)}_{f_1,q_1}$
4	5	$S(\lambda^{(in)}_{f_2,q_3}) \leftarrow bounded,\ \lambda^{(in)}_{f_2,q_3} \leftarrow \lambda^{(in)}_{f_2,q_2}$
5	5	$S(\lambda^{(in)}_{f_3,q_4}) \leftarrow bounded,\ \lambda^{(in)}_{f_3,q_4} \leftarrow \lambda^{(in)}_{f_3,q_3}$
6	2	$S(q_4) \leftarrow transparent$
7	4	$S(\lambda^{(in)}_{f_3,q_5}) \leftarrow bounded,\ \lambda^{(in)}_{f_3,q_5} \leftarrow \lambda^{(in)}_{f_3,q_4}$
8	5	$S(\lambda^{(in)}_{f_3,q_6}) \leftarrow bounded,\ \lambda^{(in)}_{f_3,q_6} \leftarrow \lambda^{(in)}_{f_3,q_5}$
		$S(\lambda^{(in)}_{f_5,q_0}) \leftarrow bounded,\ \lambda^{(in)}_{f_5,q_0} \leftarrow \lambda^{(in)}_{f_5,q_5}$

but will be identical to that variable. The dependency analysis we do "sees through" transparent ports and limits the work done to only flow variables that, once settled, define downstream variables of the same flow, through a sequence of transparent ports. Likewise, we need not be concerned with flow variables whose values will be completely determinable once the critical flow variables are solved for.

To identify all the critical flow variables, we construct a dependence graph G whose vertex set V is composed of all output ports that are in the *unresolved* state after phase I:

$$V = \{q \mid q \in Q \wedge S(q) = unresolved\}. \qquad (9)$$

Note that the output ports in the *transparent* state may also have some unsettled input flow variables, but they are excluded from set V. An edge will be defined from vertex q to vertex q' if there is a flow that enters q, passes through a sequence (possibly empty) of settled or transparent ports, and enters q'. For every such edge $e(q, q')$, we define a set $F_G(e(q, q'))$ of flows into q' that first pass through q.

To discover the edge set E, we initialize it as empty and discover edges that belong, as follows. For every port q in V, we process each of its input flow variables as follows: trace the flow's path downstream through successive ports until the first one in V is encountered; call it q'. If q' exists,

a directed edge from port q to port q', denoted by $e(q, q')$, is added to E (if it is not in E yet), and the corresponding input flow variable into q' is added into $F_G(e(q, q'))$. If q' does not exist, then no additional edge is included in the graph. After all the ports in V have been so processed, no vertex in graph G has indegree 0. If there exists such a port, then its resolution does not depend on any other port's state, and it must have already been resolved in phase I, which contradicts it being in set V.

However, it is possible that some vertices in V have outdegree 0. This happens when some unresolved ports after phase I are not in any cycle. In the above example, port q_6 has outdegree 0 because flow variable $\lambda^{(in)}_{f_3,q_6}$ is not in the cycle formed by others. From dependence graph G, we remove every vertex with outdegree 0 and all the edges that point to them. After removing a vertex with outdegree 0, we decrease the outdegrees of those vertices that have an edge pointing to it by 1; if the outdegree of a vertex becomes 0 thereby, it should also be removed from graph G. This process repeats until no vertex with outdegree 0 exists in G. Phase II finishes when the process terminates. Note that when we prune the vertices with outdegree 0 from graph G, no vertex with indegree 0 should be introduced. Hence, the following lemma must hold:

LEMMA 4. In the final dependence graph after phase II, no port has indegree 0 or outdegree 0.

In the example, the final dependence graph G is shown in Figure 2.

3.2.3 Phase III: Fixed-Point Iterations and Residual Flow Update Computation

The dependence graph produced in phase II contains all the ports whose unsettled input flow variables are critical in order to resolve the circular dependencies in the original network topology. We define vector Λ to be the vector of all critical flow variables; each member of Λ is found as a flow across some edge $e(q, q')$ in G, having been placed in set $F_G(e(q, q'))$. Indeed, Λ contains all such variables. Now for each variable, we can instantiate equation (1), where the variable itself ($\lambda^{(out)}_{f,q}$ in the equation) is expressed as a function of flow values into q, including that flow's own rate into q ($\lambda^{(in)}_{f,q}$ in the equation). In each such equation, any flow variable that is already settled is treated as a constant.

To illustrate, for the graph expressed in Figure 2, we have Λ constructed as

$$\Lambda =< \lambda^{(in)}_{f_0,q_1}, \lambda^{(in)}_{f_1,q_2}, \lambda^{(in)}_{f_2,q_3}, \lambda^{(in)}_{f_3,q_5}, \lambda^{(in)}_{f_5,q_0} > . \qquad (10)$$

The associated set of equations is given below. Flow variables that are settled and those that appear in Λ are notationally distinguished from each other by putting the settled values in bold face.

$$\begin{aligned}
\lambda^{(in)}_{f_0,q_1} &= \lambda^{\mathbf{(in)}}_{\mathbf{f_0,q_0}} \times \min\{1, \mu_{q_0}/(\lambda^{\mathbf{(in)}}_{\mathbf{f_0,q_0}} + \lambda^{(in)}_{f_5,q_0})\}\\
\lambda^{(in)}_{f_1,q_2} &= \lambda^{\mathbf{(in)}}_{\mathbf{f_1,q_1}} \times \min\{1, \mu_{q_1}/(\lambda^{\mathbf{(in)}}_{\mathbf{f_1,q_1}}) + \lambda^{(in)}_{f_0,q_1})\}\\
\lambda^{(in)}_{f_2,q_3} &= \lambda^{\mathbf{(in)}}_{\mathbf{f_2,q_2}} \times \min\{1, \mu_{q_2}/(\lambda^{\mathbf{(in)}}_{\mathbf{f_2,q_2}} + \lambda^{(in)}_{f_1,q_2})\}\\
\lambda^{(in)}_{f_3,q_5} &= \lambda^{\mathbf{(in)}}_{\mathbf{f_3,q_3}} \times \min\{1, \mu_{q_3}/(\lambda^{\mathbf{(in)}}_{\mathbf{f_3,q_3}} + \lambda^{(in)}_{f_2,q_3})\}\\
\lambda^{(in)}_{f_5,q_0} &= \lambda^{\mathbf{(in)}}_{\mathbf{f_5,q_5}} \times \min\{1, \mu_{q_5}/(\lambda^{\mathbf{(in)}}_{\mathbf{f_5,q_5}} + \lambda^{(in)}_{f_3,q_5})\}. \qquad (11)
\end{aligned}$$

To simplify the notation, we may write $\Lambda =< \lambda_1, \lambda_2, \cdots, \lambda_m >$, and then, for each λ_i, denote the right-hand-side of the equation that defines it by $f_i(\Lambda)$. The dependency of the equation on the specific variable is denoted by the subscript; the fact that the right-hand-side may contain any subset of the other variables in Λ is denoted by including that dependence in the notation. Our problem then is to find assignment of values to components of Λ that simultaneously satisfies all equations $f_i(\Lambda), i = 1, \ldots, m$. The general problem of solving nonlinear systems is known to be very hard. A nonlinear system may have between 0 and many solutions, depending on the problem specifics. Solution approaches to nonlinear systems are typically iterative, meaning that given an estimated solution Λ_k, one creates another estimate Λ_{k+1} that (hopefully) is closer to a true solution. Typically, one iterates until some norm of the difference between successive iterations $||\Lambda_{k+1} - \Lambda_k||$ is less than a tolerance ϵ. Any given iterative algorithm may or may not converge to a solution, depending on the starting point Λ_0.

Equation (1) itself suggests an iterative method. We define the first approximation Λ_0 by assigning to each unknown flow variable the smallest upper bound determined for it in phase I processing; the approximation is good if there is relatively little loss. Given approximate solution Λ_n, we compute

$$\Lambda_{n+1} = f(\Lambda_n) =< f_1(\Lambda_n), f_2(\Lambda_n), \ldots, f_m(\Lambda_n) > .$$

This formulation is also known as a *fixed-point* computation. The intuition is that if we hold the flow rates into the network constant, over time the flows across each link should stabilize (although as yet we have no formal proof of this). If this were the case, then the suggested iterative method is equivalent to pushing the network state along in time until it settles. The converged state is the "fixed point," in the sense that reapplying the traffic-shaping rules expressed by equation (1) does not alter the flow rates.

Naturally, a very significant question to ask is whether the fixed-point computation will converge to some solution $\Lambda^* = f(\Lambda^*)$. Formally proving convergence, or circumstances under which convergence is achieved, remains an important goal of our research. It is a challenging goal, though, as it seems that to achieve it, we will have to use specific properties of our problem domain. Our experimental section will directly study how convergence behaves on certain motivating networks.

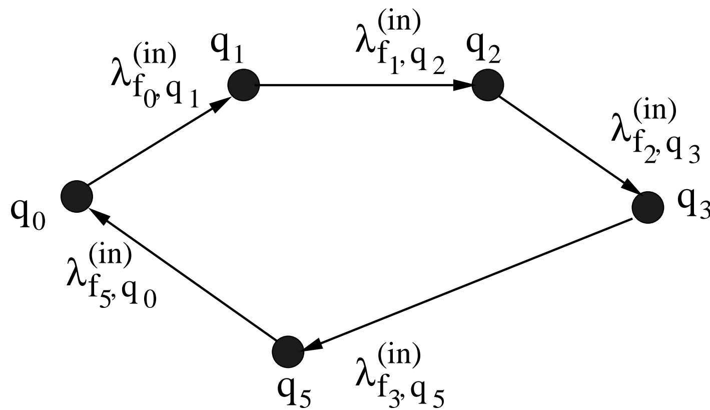

Figure 2. Dependence graph for the example topology with flow condition set 2

Although convergence has been achieved on every experiment we have run, lacking proof of convergence, we must be prepared to deal with divergence. Should a solution fail to emerge within the allocated computing time budget, we have still to construct *some* representation of background traffic. In our training application context, we can justify using a traffic matrix that does not satisfy equation (1) but is in some weaker sense "plausible," in the sense that it obeys certain constraints implied by that equation. We formalize this respect by the definition of a plausible approximation.

DEFINITION 1. Approximation Λ_n is *plausible* if

1. every flow rate expressed by Λ_n is nonnegative;
2. across every link, the sum of flow rates assigned by Λ_n to that link does not exceed the link bandwidth;
3. for every pair of POPs P_i and P_j, the volume of P_i-to-P_j traffic leaving the subnetwork during a time step is no larger than the sum of the volume of P_i-to-P_j traffic entering the subnetwork.

The fixed-point method suggested by equation (1) ensures plausibility.

THEOREM 1. Let Λ_n be any approximation with nonnegative flow values. Then $\Lambda_{n+1} =< f_1(\Lambda_n), f_2(\Lambda_n), \ldots, f_m(\Lambda_n) >$ is a plausible approximation.

Proof. Given nonnegative values for flow variables, it is not possible for equation (1) to create negative values; thus, $f_i(\Lambda_n) \geq 0$ for all i, and the first condition for plausibility is satisfied. By construction, it is not possible for equation (1) to assign flows across a link that exceed the link's allocated capacity for background traffic, so the second condition for plausibility is also satisfied. To establish the third condition, choose any flow P_i-to-P_j that passes through the subnetwork and consider the sequence of ports $q_1, q_2, \ldots, q_z$ that it follows through the subnetwork. Now in equation (1), the value of the expression used in case 1 is always larger than the value used in case 2. Therefore, an upper bound on the flow value associated with the flow leaving the subnetwork is obtained by assuming that the case 1 expression is used at each port along the path. That upper bound is simply the sum of the rate at which the P_i-to-P_j flow enters the subnetwork. This establishes the third condition for plausibility. □

After the fixed-point iteration establishes values for critical flow variables, we mark each as settled. Now the set of all settled flow variables implicitly contains all the information needed to settle the remaining variables by direct computation—we need only resume the rule-based flow update computation of phase I. In the example, $\lambda^{(in)}_{f_3,q_6}$ and $\lambda^{(in)}_{f_3,q_4}$ are not settled after phase III. Because both input flow variables at port q_3 are settled, rule 1 can be used to compute $\lambda^{(in)}_{f_3,q_4}$; $\lambda^{(in)}_{f_3,q_6}$ can also be calculated by applying rule 1 on port q_5.

After phase III, all output ports in the network have been resolved. The algorithm ends here.

It is worth noting that efficient implementations of this algorithm are possible. It is possible to organize the rule selection process in Phase I such that the next rule to evaluate is chosen in O(1) time and that the rule so chosen needs to be evaluated because a state change requires that we reevaluate the precondition. Furthermore, it is possible to organize the computation such that each precondition can be evaluated in O(1) time, using space that is linear in the size of the problem. Finally, in the event that offered loads are such that no port is congested, the algorithm will settle all flow variables in Phase I, in time proportional to the size of the problem. Formal proofs of these results may be found in Guanhua [2].

Table 5. Four topologies used in the simulations

Topology	# Nodes	# Directional Links	# Flows	Link Bandwidth
Top-1	27	88	702	100 Mbps
Top-2	244	1080	12,200	2488 Mbps
Top-3	610	3012	61,000	2488 Mbps
Top-4	1126	6238	168,900	2488 Mbps

3.3 Experimental Results

We now turn to an experimental evaluation of our algorithm. Table 5 summarizes four real topologies that are used in the simulation experiments. Top-1 is a POP-level ATT USA backbone network; it has 27 POPs. A flow is created between any pair of POPs, which thus leads to 702 flows in total. The link bandwidth (100 Mbps) is artificially low to enable us to make a direct comparison of accuracy and performance with equivalent packet-based flows. The other three topologies are obtained from the Rocketfuel project [3]. Top-2 is the router-level Exodus backbone; Top-3 consists of two router-level ISP backbones, the Exodus backbone and the Above.Net backbone, which are connected through some peering links; Top-4 further augments Top-3 by adding another router-level ISP backbone, Sprint backbone, to it. In Top-2, Top-3, and Top-4, every router picks 50 routers from its own backbone and directs a flow to each of them; it also picks 50 routers from each other backbone in the topology and directs a flow to every one of them. Hence, we get the total numbers of flows shown in the table. We use the PPBP traffic model to generate the ingress rates for each flow. We adjust its input parameters to achieve two desired statistics, Hurst parameter and coefficient of variance (COV) [4]. In all the experiments throughout this article, the Hurst parameter is always 0.8, but we vary the COV parameter to obtain different traffic bursty levels.

3.3.1 Convergence Behavior

Our convergence experiments simulate Top-2, Top-3, and Top-4 for 200 time steps, each spanning 5 seconds. In all of these experiments, we consider a solution to be converged when the maximum relative difference between successive approximations to any flow variable is 0.001, for example, when $|\lambda_{n+1} - \lambda_n|/\lambda_n \leq 0.001$ for all flow variables λ. In addition, the desired COV of each flow's ingress traffic is 5, and the desired average link utilization is 50% in all the experiments.

Figure 3 depicts the histogram of the number of ports on the dependence graph in phase III, as well as the number of iterations used to reach fixed-point solutions, on single runs of 200 time steps. From the experimental results, our algorithm is able to significantly reduce the number of ports in the fixed-point iterations. The largest proportion of ports that appear on the dependence graph in phase III is 4.4%, 2.8%, and 3.4% for Top-2, Top-3, and Top-4 respectively. Furthermore, the fixed-point solution in our algorithm can converge within a small number of iterations. In all the experiments, we observe that at most 12 iterations are necessary to reach a fixed-point solution.

3.3.2 Performance

We examine performance using these same topologies, with traffic loads designed to yield two different average link utilizations: 20% and 50%. Again, the time step size is 5 seconds. We use the PPBP traffic model to generate the ingress rates for each flow. The Hurst parameter is 0.8, and the COV is 0.5. All the experiments are done on a 1.5-GHz PC with 2 GB of memory. The average execution times per time step for these four topologies under different traffic loads are presented in Table 6; the confidence intervals derived from 10 runs are too small to display. Memory limitations prevented us from completing experiments for Top-4 when the COV is 0.5.

The simulated time step is 5 seconds, and the average execution per time step in each case is significantly smaller—these models run faster than real time. Table 6 suggests that when traffic is more bursty, less simulation execution time is required. This is counterintuitive because high variations usually cause heavier congestion, thus putting more ports on the dependence graph in phase III. However, when the COV is large, traffic generated from the PPBP model is very bursty, implying times of high intensity *and* longer "silent" periods. Our implementation takes advantage of time steps when a flow's offered load rate is zero—it does not contribute and so is not explicitly handled during that time step. Obviously, if more flows have ingress rate 0 at a time step, there are fewer flow variables that need to be settled, and there is less work to accomplish.

To evaluate relative speedup (and accuracy), we use a baseline pure packet-level network model. This model executes in the same simulation framework as the flow-based model; the only difference is that traffic is expressed using packets. Top-1 is the only topology where the packet-level simulation completes in reasonable time. The relative execution speedups, as a function of average link utilization, are presented in Figure 4. It is clear that time-stepped coarse-grained traffic simulation has achieved execution speedups at several orders of magnitude under both traffic

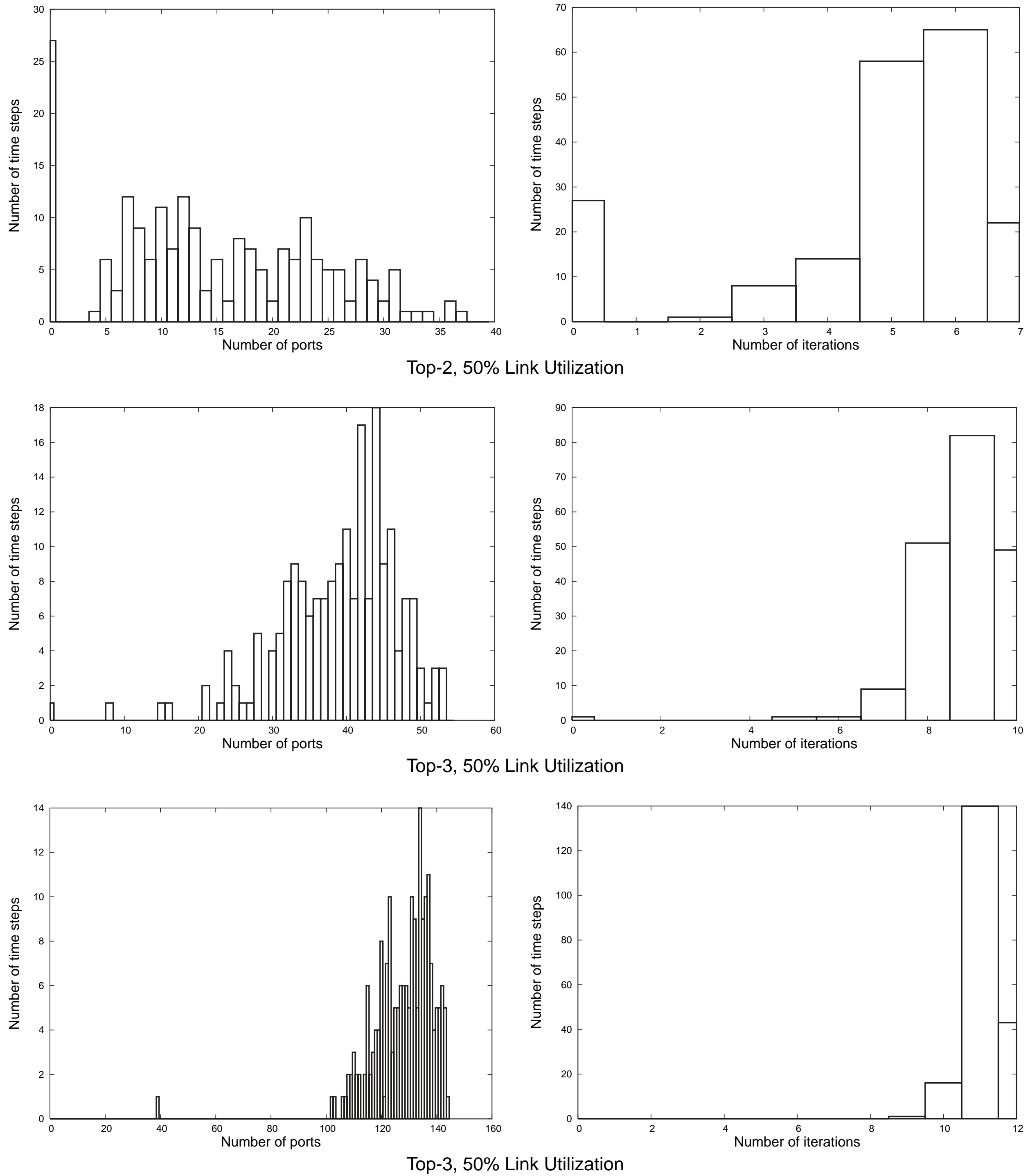

Figure 3. Histogram of ports on circular dependencies, and iterations per time step

Table 6. Average execution time per time step

	COV = 5		COV = 0.5	
Topology	**Secs/Round (20% Link Utilization)**	**Secs/Round (50% Link Utilization)**	**Secs/Round (20% Link Utilization)**	**Secs/Round (50% link Utilization)**
Top-1	0.0026	0.0026	0.0051	0.0052
Top-2	0.051	0.051	0.1988	0.2380
Top-3	0.283	0.285	1.4895	1.8740
Top-4	0.852	0.907	—	—

loads. We also observe that with increasing average link utilization, the speedup over the pure packet-level simulator also improves. This is because the performance of a packet-level traffic simulator is directly affected by the amount of traffic traversing through the network, but performance of the time-stepped coarse-grained traffic simulator is relatively insensitive to the absolute traffic intensities in the network.

From Figure 4, we have noticed that the relative speedup curve grows *sublinearly* as the average link utilization exceeds 50%. There are two reasons for this. First, when the traffic load increases up to a high level, the pure packet-level traffic simulator drops a significant portion of the traffic because of heavy congestion. Second, heavy traffic load causes more output ports to appear on resolution cycles, and the fixed-point algorithm thus has to operate on more unsettled flow variables in phase III.

Comparing the two curves with different COVs, we find that a larger COV leads to a smaller relative speedup. The reason is that bursty traffic leads the packet-level simulation to drop more packets and thereby reduce the volume of execution work needed to push them along their paths. Referencing Table 6, evidently the effect of burstiness on the packet simulator is more pronounced than on the fluid simulator. Nevertheless, the speedups produced by the fluid method are very significant.

3.3.3 Accuracy

We also consider the accuracy of our approach when used as a background traffic generator, as compared to the pure packet-oriented approach. The idea is to study the properties of a small number of detailed packet-oriented TCP and UDP flows, while the background traffic is either packet based or flow based. Toward this end, we modify topology Top-1 by attaching an end host to each POP node. The background traffic is simulated with two different techniques: our time-stepped coarse-grained simulation and conventional packet-level simulation. We parametrically control the traffic parameters to cause the average background traffic load on a link to vary, between experiments, from 10% to 80%. The PPBP traffic model is used to generate ingress traffic for each flow. The Hurst parameter is 0.8. The COV is 1.0 throughout all the experiments.

In this model, each end host has a TCP server, a TCP client, a UDP server, and a UDP client. For every background traffic load, we simulate 10 experiments. In each experiment, a TCP client randomly picks a TCP server on any other end host. A TCP client's behavior can be modeled as an ON/OFF process: it connects with the TCP server it has chosen, requests a data transfer of 5 Mbytes, and then waits for the requested bytes from the server; after the client receives all the data it has requested, it remains idle for an exponentially distributed period with a mean of 5 seconds; when it wakes up, it starts another request, and the above process repeats until the simulation is over. Upon receiving a request, a TCP server uses TCP protocol to transfer requested bytes to the client from which the request comes. In each experiment, a UDP server also randomly chooses a UDP client on any other host. A UDP server's behavior can also be modeled as an ON/OFF process: it uses UDP protocol to send a file of 5 Mbytes at the constant rate of 1 Mbps, and after it finishes the transfer, it remains idle for an exponentially distributed period with a mean of 5 seconds; after the off period finishes, it sends another 5 Mbytes at the same rate, and the above process repeats until the simulation is over. Every experiment is simulated for 1000 seconds (in simulation time).

We compare the behavior of the TCP and UDP flows using different methods for background traffic generation. The experiments are coupled, in the sense that the same pattern of requests is simulated for the TCP and UDP flows in the two experiments that use different background flow generation, and furthermore, the background flow generation in the two experiments is driven by the same offered load. In this way, we ensure that on an experiment-by-experiment basis, we are comparing precisely the same context for measuring TCP and UDP flows against different background generation techniques.

Figure 5 presents three graphs describing the results of these experiments. The top two graphs plot the sample means and 95% confidence intervals of the goodput measured in 10 TCP experiments, as well as the delivered fraction of UDP traffic, also measured in 10 experiments.

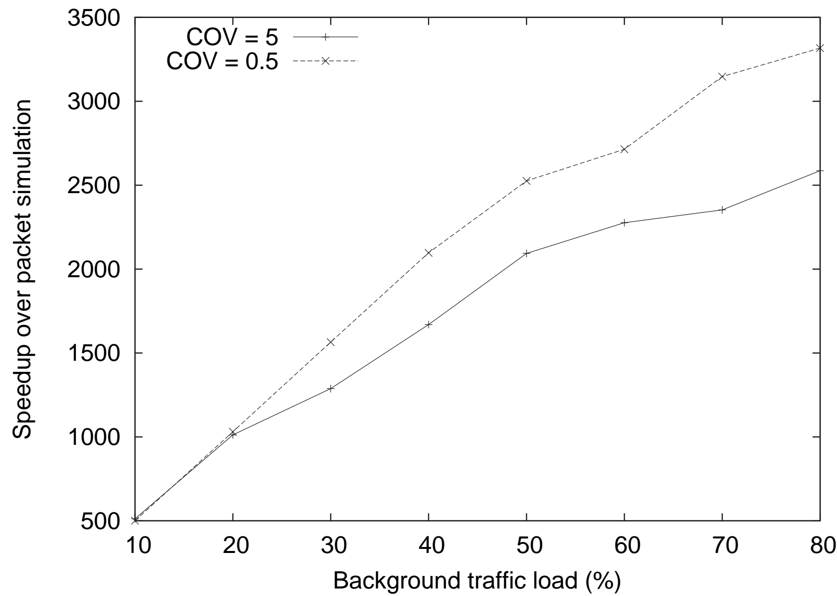

Figure 4. Relative speedup of time-stepped coarse-grained traffic simulator (Top-1)

We see that goodput decreases with link utilization, owing to congestion. Likewise, the fraction of UDP traffic that is delivered decreases with the link utilization. The third graph plots the average relative error in goodput and round-trip time (RTT) for these same TCP experiments and the delivered fraction of traffic for the UDP experiments. The most important feature of this latter graph is that the error is small. It is interesting to note that accuracy of TCP goodput increases with network load, while the accuracy of TCP RTT tends to increase (although it is quite small). The goodput error settles down under high load because the congestion is so large that TCP send windows are essentially one packet in size, and the bandwidth use optimizations of TCP send windows are not operating. On the other hand, under high network load, the relative error of UDP increases. The explanation here is that we have to approximate packet loss at a congested router and that approximation is the cause of the error. The more heavily congested the network, the more heavily the approximation is exercised. This same effect will affect TCP goodup accuracy but under higher load than shown in these experiments.

4. Parallel Algorithm

The sequential algorithm described in section 3 can simulate a reasonably large network on a uniprocessor. However, simulation of very large networks requires that we use multiple processors to gain access to larger pools of memory and more computational power. It is natural to consider parallelizing the sequential algorithm on a distributed-memory multiprocessor. We can then combine both advantages of high abstraction-level models and high-performance computing techniques to further improve the performance of large-scale network simulation.

The example network shown in Figure 1 is used to illustrate how we parallelize each phase in the sequential algorithm. Suppose that the original topology is divided into three partitions, each simulated by a single processor. This is shown in Figure 6. Flow f_1's path is on both processors p_1 and p_2; flow f_3's path is on both processors p_2 and p_3; flow f_5's path is on both processors p_3 and p_1.

In the following subsections, we describe parallelization of our algorithm and investigate its performance and scalability.

4.1 Algorithm Description

We now suppose that the network description is distributed across the memory spaces of multiple processors. Each processor executes the phase I/II/III work that we identified for the sequential algorithm but on the pieces of the network description it is (statically) assigned. Within each phase, the processors exchange messages, to be described. Global synchronization is used to determine when a phase has completed. Detecting when a phase completes is tantamount to detecting when every processor has completed all known work for that phase *and* that there are no as-yet-unreceived messages in flight. For this, we use a specialized algorithm called a noncommittal barrier. Our al-

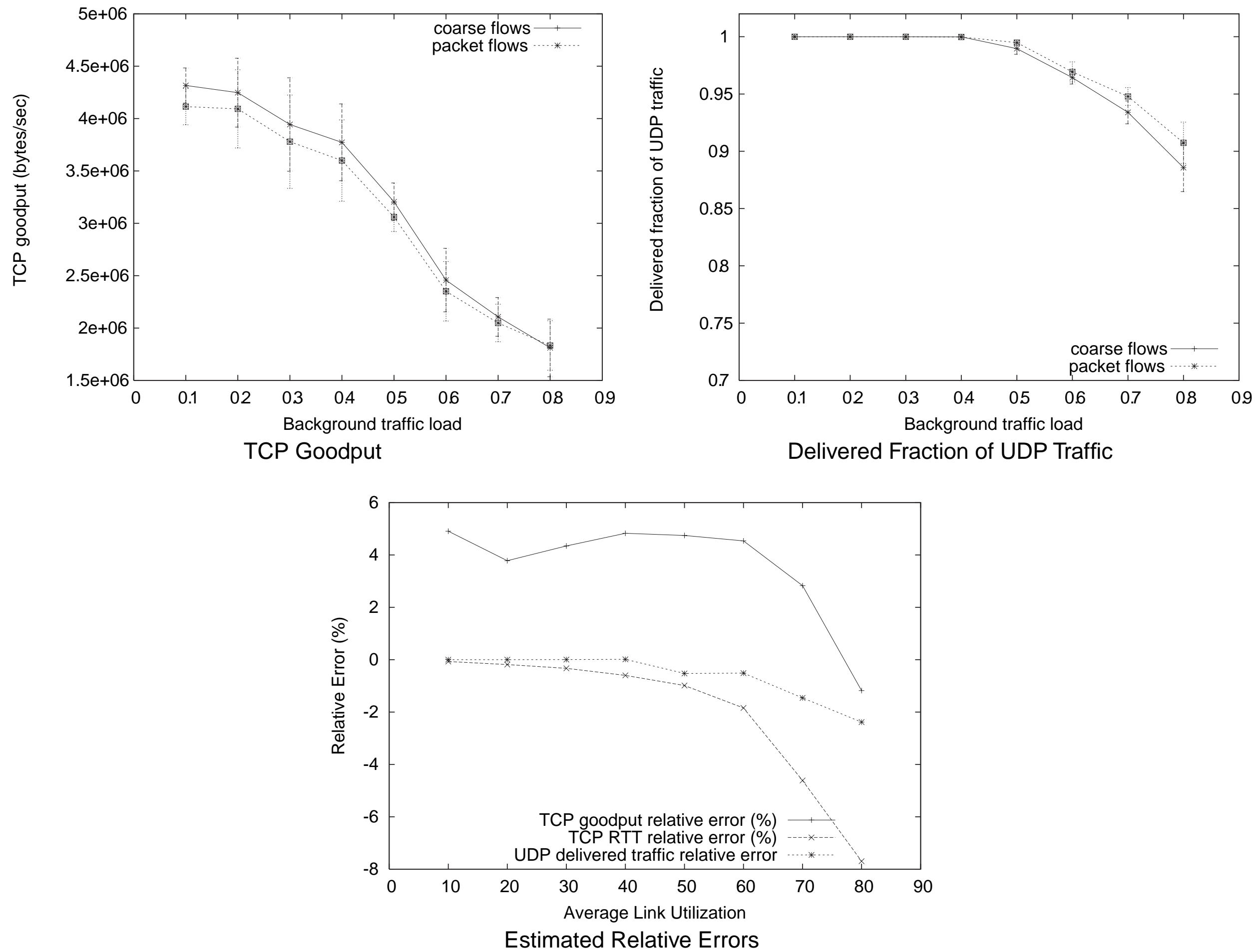

Figure 5. Accuracy results with time-stepped coarse-grained traffic simulation

gorithm takes advantage of domain knowledge to reduce the overhead of executing that construct.

Next we discuss the particulars of each phase.

4.1.1 Phase I: Rule-Based Flow Update Computation

On entering phase I, each processor executes the same logic as the sequential algorithm, in choosing ports whose preconditions require reevaluation. As in the sequential algorithm, after a processor applies a rule on a flow traversing through a local output port, it updates its state at the next output port. In the event the next port is located on another processor, a *flow update* message is sent to convey the state change. Handling of this message is identical to processing of a state-change contained on-processor—the flow's state is updated, and the target port is added to the list of ports whose preconditions must be reevaluated.

Eventually, a processor will have no ports needing reevaluation. However, it may be the case that the processor will later be sent a *flow update* message that must be processed and that may trigger further local work (as well as additional outbound messages). For a processor to safely advance to phase II, it must engage in a computation that establishes when (1) no processor has ports requiring reevaluation and (2) every *flow update* message that has been sent has also been received. This problem has been addressed before in the parallel processing context by the so-called noncommittal barrier [5]. In contrast to a standard barrier synchronization, the noncommittal barrier can be "reset" in the event that the processor receives another message. After notifying the noncommittal barrier that it has no known work, a processor alternates between monitoring for incoming message traffic (receipt of which triggers a reset and rollback of the noncommittal barrier state) and

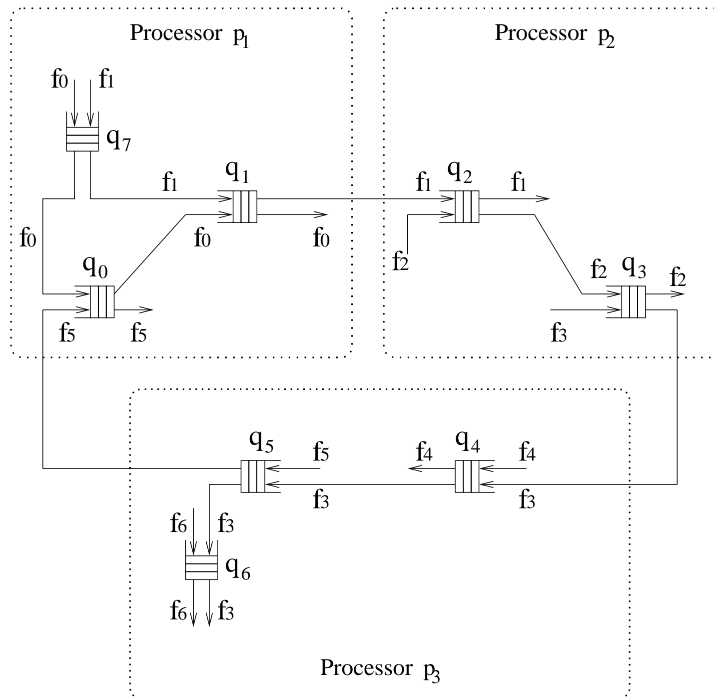

Figure 6. Partitions of the topology in Figure 1

polling the noncommittal barrier for indication of phase completion.

In the noncommittal barrier algorithm, every processor maintains both counts on the computation messages it has sent and received. With aid of a tree structure, each processor keeps a set of neighbor processors in different dimensions of a logical hypercube. It exchanges the counts on computation messages sent and received with its neighbor processors from the lowest dimension to the highest dimension. A processor is able to progress past the barrier only after it has agreed on the counts on the computation messages sent and received with its neighbor processors in all the dimensions. A processor may receive a computation message while it is checking the counts on the computations messages sent and received with the neighbor processor in a dimension; when this occurs, the algorithm rolls back to the lowest dimension. The noncommittal barrier algorithm requires $O(log^2 P)$ space on each of P processors and, in the absence of rollbacks, requires $O(log^2 P)$ parallel execution time.

A significant communication overhead can be associated with rollbacks. A simple observation helps to reduce the number of rollbacks. Recalling that every non-ingress flow variable begins in the *unsettled* state, as well as knowledge (by lemma 3) that at the end of phase I, no flow variable can be in that state, we impose the rule that a process not engage in the noncommittal barrier before all of its flow variables have transitioned out of the *unsettled* state. Even if a processor has no port preconditions to reevaluate, if it has any *unsettled* flows still, it knows there is another message coming.

Once the noncommittal barrier releases all processors, they advance to phase II processing.

4.1.2 Phase II: Dependence Graph Generation

Recall that in the sequential algorithm, phase II starts by creating a graph whose vertices represent *unresolved* ports and whose edges reflect input dependencies between them. A second step recursively prunes vertices with outdegree 0 and their edges. Implementation on a distributed platform requires interprocessor communication and synchronization.

The ports in the *unresolved* state can be located on multiple partitions, each of which then has only partial knowledge of the whole dependence graph. For example, suppose that processor p_a processes port q_a in phase II. For every input flow variable $\lambda^{(in)}_{f,q_a}$ $(f \in F_q)$ in the *unsettled* state, the local processor needs to find the next port q_x on flow f's path that is also in the *unresolved* state (if such q_x exists). Port q_x, however, may not be found on the local partition. When this occurs, the local processor sends a *query* message to the processor to which flow f goes after departing

from processor p_a, say, processor p_b. The query message holds the identified port q_a and processor p_a that originate the edge whose other endpoint is sought. Upon receiving the message, p_b continues searching along the flow's path, looking for the first unresolved port. If found, p_b sends a *positive reply* message to processor p_a and adds a *backward* interprocessor edge from $< p_a, q_a >$ to $< p_b, q_x >$ to the part of the dependence graph that is maintained locally; if the flow ends in a sink without encountering an *unsettled* port, p_b sends a *negative reply* message back to processor p_a. A third possible situation is when the flow does not encounter any *unsettled* ports on p_b and passes to a port on a different processor p_c. Processor p_b forwards the query to p_c and has no further part to play in the query processing (unless the flow later snakes back to a port that p_b holds). When processor p_c receives the *query* message, it processes it as though it received it directly from p_a, using the same logic as did p_b. Eventually, processor p_a receives a reply to its query, establishing either that the flow does not encounter another *unsettled* port before reaching its destination or that it does (and the identity of the port and processor where it does).

In the sequential algorithm, the two steps in this phase can be executed sequentially. The parallel algorithm avoids the implicit synchronization between steps, as follows. A processor sends out all its query messages and, for each port, records the number of such messages sent. A count of responses is maintained as well. When the counts match, we know the node's outdegree and can act accordingly as per the second step. After a port is identified as having outdegree 0, it is specially marked as *detached* and is removed from the vertex set; all the intraprocess edges pointing to it are removed from the partial dependence graph maintained locally. Some *forward* interprocessor edges on other processors may still point to this port. For each of such edges, an *edge-removing* message is sent to the processor maintaining the port at the other end. On receiving this message, the processor removes the corresponding *forward* interprocessor edge and checks whether the port's own outdegree decreases to 0; if so, it repeats the above process. In addition, owing to asynchrony, a *query* message may arrive at a port that is marked as *detached*. When this occurs, a *negative reply* message is sent to the processor whose port initiates the *query* message. We illustrate messages communicating between processors in Figure 7.

All the processors are synchronized before they proceed into phase III, using a noncommittal barrier (for the same reasons as we used one between phases I and II). As with phase I, we are able to identify conditions to be satisfied before entering the noncommittal barrier (and after the processor has done all the local phase II work it can).

1. Every *query* message has been acknowledged by either a *positive* or *negative reply* message.
2. if a flow that is not yet in the *settled* state originates on a different processor, then a *query* message associated with that flow has been received and fully processed.

If either of these conditions is not met, the processor is certain to receive another message, and so it is pointless to engage in the noncommittal barrier.

4.1.3 Phase III: Fixed-Point Iterations and Residual Flow Update Computation

Fixed-point iteration is applied in phase III. The necessary communication pattern is plainly evident from the reduced graph structure—wherever a graph edge is directed from a port in one processor to a port in another, every iteration's new values for all flows associated with that edge will be communicated. A processor may apply its updates when all necessary input values (from the previous iteration) have been received, after which it communicates new values for the next iteration and engages in convergence testing. A processor computes the maximum relative difference between successive values computed for any flow variable and offers that value to a *max* reduction operator [6]. The result of this operator is the maximum value offered among all processors, and this value is returned to all processors. Each processor then independently observes whether the global termination criteria are satisfied.

After the fixed-point iterations terminate, processors can resume the rule-based flow update computation to settle the remaining unsettled flow variables. Completion of this final step is terminated using a noncommittal barrier.

4.2 Scalability Analysis

In this section, we examine the algorithm's performance on a distributed-memory parallel machine. The iSSFNet simulator used to study sequential performance maps also supports parallel execution—in this case, on the Tungsten supercomputer at the National Center for Supercomputing Application. Each of Tungsten's nodes is composed of a 32-bit 3.2-GHz Intel Xeon dual-processor motherboard and 3 GB memory; the nodes are interconnected with Myrinet 2000. Each nodes runs Linux 2.4.20 (Red Hat 9.0). Our experiments use up to 16 nodes.

In the following subsections, we study how parallel performance behaves as a function of the number of processors and problem size.

4.2.1 Speedup on Fixed Problem Size

We first consider how execution time of a given problem changes as we use increasingly more processors to solve it. We construct a model problem whose mapping to a parallel machine is straightforward. The model problem is built up from 32 replications of the ATT backbone archi-

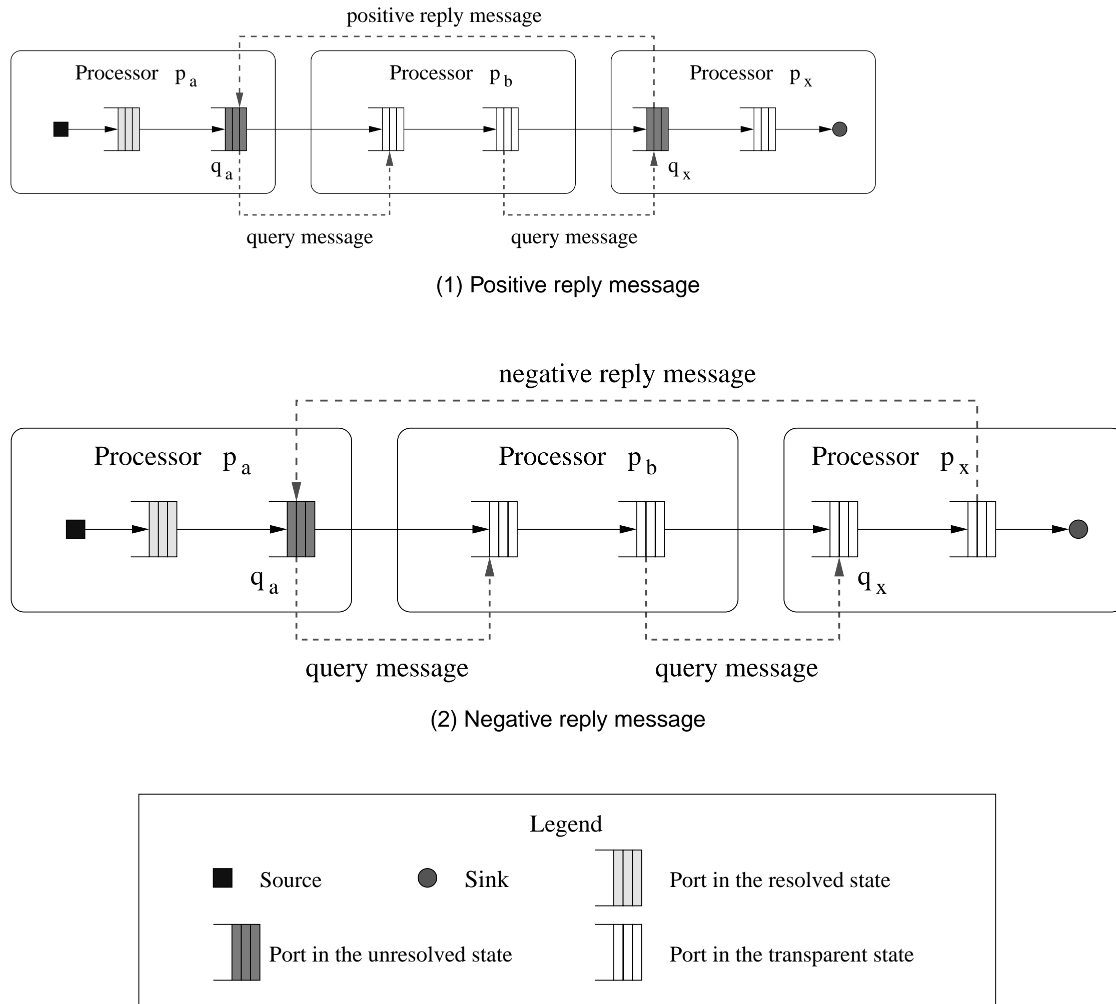

Figure 7. Dependence graph generation across multiple processors

tecture considered earlier (i.e., Top-1 in Table 5, which has 27 nodes). The three most highly connected nodes in that backbone structure each serve as gateways to the other backbone replications. Each gateway connects to every one of its topological equivalents in other backbones, so that between the backbones, there are three cliques. The links internal to a backbone are modeled as having 100 Mbps, while the links between backbones are given 1000 Mbps. The traffic pattern is all-to-all; every node directs a flow to every other node. There are then $(32 \times 27)^2 = 745,632$ distinct flows in the model. The ingress traffic of each flow is generated by a PPBP traffic model, whose COV (i.e., ratio of standard deviation to mean) is 1 and Hurst parameter is 0.8. We vary the mean of each flow's ingress rates to obtain two desired average link utilizations, 20% and 50%. We eliminate the execution time required to generate this traffic from our experiments by precomputing the load rate offered by each flow, each time step. Doing so better isolates the performance of the algorithm of interest from that of one of the many ways one might create traffic rates for the algorithm.

We partition the problem onto 1, 2, 4, 8, 16, and 32 processors in the obvious way, assigning the same number of backbones to each processor. The parallel program is executed for 200 time steps. Figures 8 and 9 present the execution time per time step and speedup (relative to optimized single-processor execution using the same iSSFNet framework, using one processor of the multiprocessor) for two workload cases, 50% and 80% link utilization. Each graph separately plots the contribution of each of the three phases, as well as the combined metric. Notable features of these results include

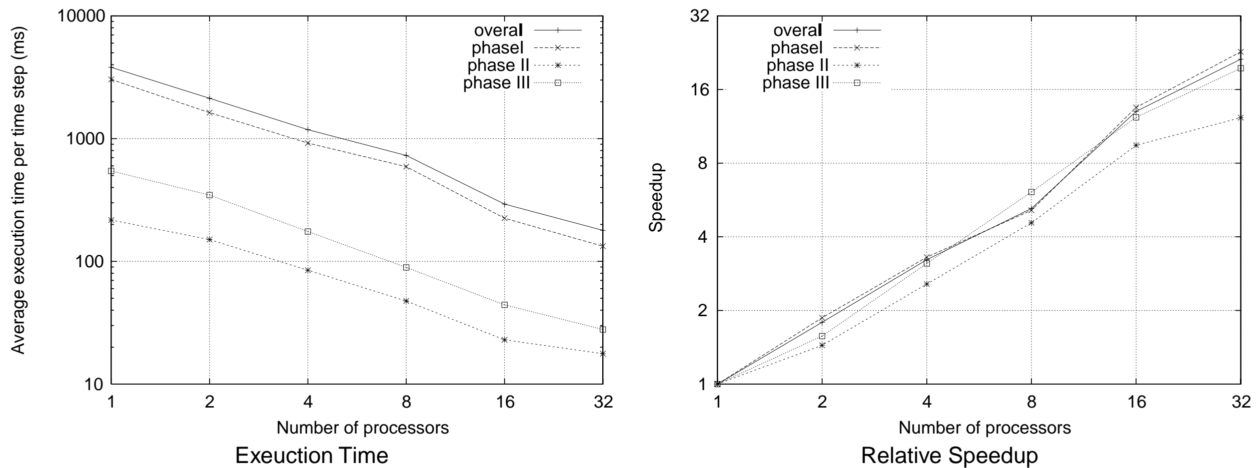

Figure 8. Scalability results with fixed problem size (link utilization 50%)

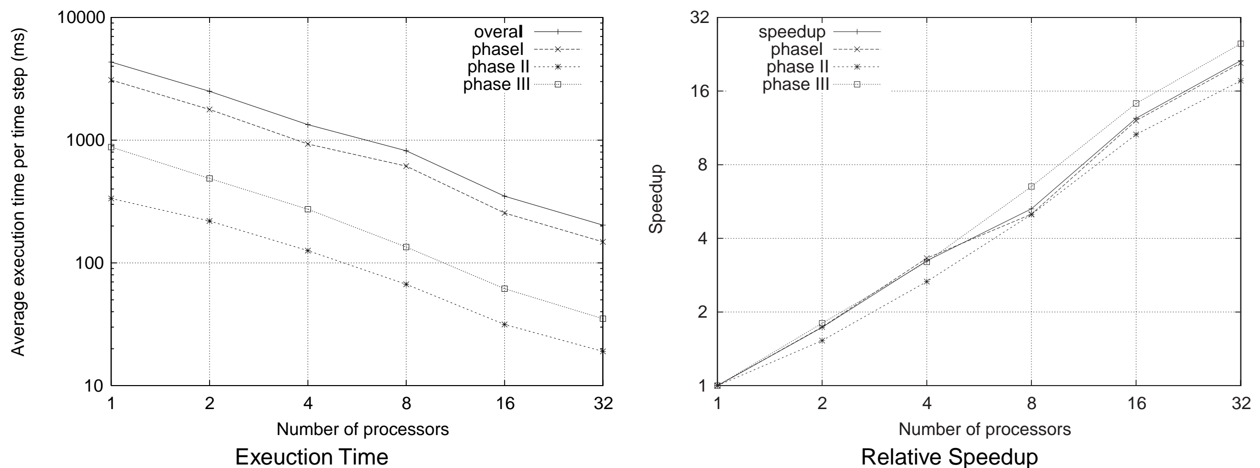

Figure 9. Scalability results with fixed problem size (link utilization 80%)

- performance increases monotonically with additional processes;
- phase I computation dominates throughout, and the relative contribution by each phase varies little with increasing numbers of processors;
- faster than real-time performance is achieved using sufficiently many processors.

Table 7 shows the proportion of the execution time used within each phase. These show that for a certain traffic intensity, the fraction of time spent in each phase does not change significantly as the number of processors is increased. We also see that phase I dominates the computation. The aggressive removal of ports from the subsequent fixed-point iterations proves to be effective. Under the 50% link utilization load, only 5% of the ports are involved with the fixed-point step; under the 80% utilization load, only 8% are.

4.2.2 Scalability

Scalability analysis asks how performance behaves as the model size and number of processors simultaneously increase. Demonstration of such scalability shows that there are no inherently limiting barriers to parallelization, at least over the range of problem and architecture size considered.

Table 7. Proportion of execution time consumed by each phase with respect to the average execution time per time step

Link Utilization	Phase	Timelines 1	2	4	8	16	32
50	I	79.9	76.6	78.0	81.2	77.0	74.5
	II	5.7	7.1	7.2	6.5	7.9	9.9
	III	14.3	16.3	14.8	12.3	15.2	15.6
80	I	71.8	71.6	70.0	75.3	73.2	73.3
	II	7.8	8.8	9.5	8.2	9.1	9.4
	III	20.4	19.6	20.6	16.5	17.7	17.3

We examine scalability using a model architecture designed to keep the same amount of computation and communication load on each processor as the model size and number of processors are increased. We start with a baseline subnetwork, the Exodus backbone (listed as Top-2 in Table 5, with 244 routers); in every experiment, each processor simulates one of these subnetworks. The connectivity between subnetworks varies depending on the number of processors used. Given k processors, we identify the k most highly connected nodes. Each of these nodes is connected to all other topologically equivalent nodes in the other subnetworks. Thus, we form k cliques, each formed using the "same" node in each of the subnetworks.

All links are modeled as having bandwidth 2488 Mbps. The intradomain traffic pattern is *all-to-all*; that is, from each router in any backbone, there is a flow directed to every other router in the same backbone. Hence, there are $59,292 \cdot k$ (i.e., $k \times 243 \times 244$) intradomain flows. The interdomain traffic pattern is localized, as follows. Each router in the ith backbone ($0 \leq i \leq k$) directs a flow to every router in the $((i+1) \bmod k)$th backbone. Hence, there are $59,536 \cdot k$ (i.e., $k \times 244 \times 244$) interdomain flows. The total number of flows in the simulation using k processors is $118,828 \cdot k$. As before, the ingress traffic of every flow is modeled with a PPBP traffic model whose COV parameter is 1 and Hurst parameter is 0.8. We vary the mean of the ingress rates of each flow to obtain two traffic intensity levels, one with average link utilization 50% and the other with average link utilization 80%.

From this description, we see that the number of routers, number of connections, and number of flows all increase linearly with the number of processors used. The interdomain traffic pattern keeps the number of domains a flow may cross limited to 1, but it does cause the amount of communication on a processor to grow linearly with the number of processors. Even though most of the edges in an interdomain clique do not carry flows, the existence of the cliques contributes to a linear growth in the memory used (to store the connections) as the number of processors grows.

Our experiments simulate 200 time steps, varying the number of processors among the powers of 2 from 2 to 32. Figure 10 plots the average execution time per time step, for each phase (and collectively), as a function of the number of processors. When the average link utilization is 50%, the average execution time per time step increases from 780.6 milliseconds with 2 timelines to 794.7 milliseconds with 32 timelines, only by 1.8%; when the average link utilization is 80%, the average execution time per time step increases from 807.9 milliseconds with 2 timelines to 844.6 milliseconds with 32 timelines, only by 4.5%. The flatness of the curves shows that the increases in communication and extent of interprocessor synchronization do not hamper performance with linearly increasing model size and numbers of processors.

Neither the speedup nor scalability experiments challenged the problem with load imbalance. Our goal in this is to isolate the intrinsic performance and scalability characteristics of the algorithm from its dependencies on load distribution. As we use the simulation system in real contexts, we will be faced with real load-balancing issues. While iSSFNet does have automated partitioning code, there is much more work to be done. Existing work that considers the problem includes Liu and Chien [7, 8], but a general solution that works well under varied simulation configurations still remains as an open problem. A seemingly as-yet unconsidered issue is created by our approach of mixing traffic at different levels of resolution, for with different resolutions come different load demands. Our own future work will address these issues.

5. Related Work

Our problem is related to the area of network tomography, which likewise seeks to determine some network characteristics (e.g., the traffic matrix—volume of flows between any pair of ingress and egress points) from other traffic measurements (e.g., measured link loads on the network edge). There is a large and fascinating literature on the topic (e.g., see [9-16]). A key difference between our prob-

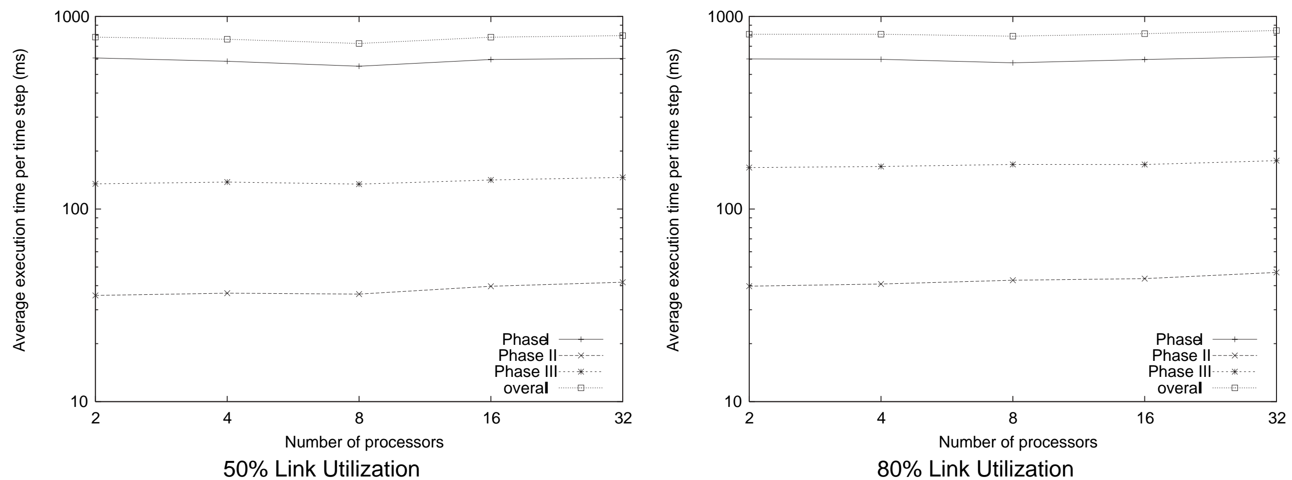

Figure 10. Average execution time with scaled problem size

lem and network tomography is that the latter is driven by difficulties in obtaining measurements of certain important quantities and so works to use models to infer them from measurements that are available. In the virtual world of simulation, we can measure anything we like; our problem is computing those measurements given the offered load.

Our approach includes ideas found elsewhere. The notion of simulating traffic using simple rate functions was introduced almost 10 years ago [17], a formulation that observes that first come, first serve (FCFS) service at a congested router port can be modeled by scaling the flows in portion**[*PROPORTION?*]** to their input rates, which is a central facet of our model. Further development is found in Nicol, Goldsby, and Johnson [18]; application to TCP is developed in Nicol and Yan [19], which treats issues in the simultaneous simulation of packet and fluid flows, as do other authors [20-22]. Specific application to the global Internet and practical problems therein are discussed in Barakat et al. [23].

More detailed fluid models found in the literature include treatments of TCP [24], models containing stochastic elements [25], a time-stepped model [26], and a discussion of trade-offs [27, 28]. Use of fixed-point iteration to solve for network measures of interest also appears in Bu and Towsley [29], a model that focuses much more on TCP and less on link loads. A very abstract rate-based model is discussed in Vahdat et al. [30], work that does not try to capture interflow dependencies in detail, as we do.

Against this backdrop, our contribution is the development, optimized solution, and empirical study of a coarse-grained traffic model that focuses on the impact that bandwidth sharing on congested links has on flow rates throughout the entire network. Our fixed-point approach is unique in that each approximated solution has internal consistency that can, in some contexts, allow it to be used even if the solution has not yet converged. The approach is a valuable and necessary component of our real-time cyber-defense training simulator.

6. Summary

This article proposes an approach for simulating network traffic flows at a coarse scale. The approach is motivated by an application where we must deliver real-time performance and “look-and-feel” accuracy. We develop an optimized means of performing the simulation and study its behavior on a set of large topologies based on known Internet backbones. We find that it is fast—very, very fast relative to packet-based flows—and accurate enough for our purposes. We parallel the algorithm on a multiprocessor supercomputer. We observe good speedup and excellent scalability.

7. Acknowledgments

This research was supported in part by DARPA contract N66001-96-C-8530 and NSF grant CCR-0209144. Accordingly, the U.S. government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. government purposes. In addition, this project was supported under award no. 2000-DT-CX-K001 from the Office for Domestic Preparedness, U.S. Department of Homeland Security. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security.

A preliminary version of this article appeared in the *Proceedings of the 19th Workshop on Parallel and Distributed Simulation* [31].

8. References

[1] Zukerman, M., T. D. Neame, and R. G. Addie. 2003. Internet traffic modeling and future technology implications. In *Proceedings of IEEE Infocom'03*.
[2] Guanhua, Y. 2005. Improving large-scale network traffic simulation with multi-resolution models. Report TR2005-558, Dartmouth College, Hanover, NH.
[3] Project RocketFuel. http://www.cs.washington.edu/research/networking/rocketfuel/
[4] Leland, W. E., and D. V. Wilson. 1991. High time-resolution measurement and analysis of LAN traffic: Implication for LAN interconnection. In *Proceedings of 10th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'91)*.
[5] Nicol, D. M. 1995. Noncommittal barrier synchronization. *Parallel Computing* 1995;21:529-49.
[6] Pacheco, P. S. 1997. *Parallel programming with MPI*. New York: Morgan Kaufmann.
[7] Liu, X., and A. A. Chien. 2003. Traffic-based load balance for scalable network emulation. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, Phoenix, AZ.
[8] Liu, X., and A. A. Chien. 2004. realistic large-scale online network simulation. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, Pittsburgh, PA.
[9] Zhang, Y., M. Roughan, C. Lund, and D. Donoho. 2003. An information-theoretic approach to traffic matrix estimation. In *Proceedings of the 2003 ACM SIGCOMM Conference*, Karlsruhe, Germany.
[10] Zhang, Y., M. Roughan, N. Duffield, and A. Greenberg. 2003. Fast accurate computation of large-scale IP traffic matrices from link loads. In *International Conference on Measurement and Modeling of Computer Systems*, San Diego, CA, pp. 206-17.
[11] Feldmann, A., A. G. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. 2000. Deriving traffic demands for operational IP networks: Methodology and experience. In *SIGCOMM*, pp. 257-70.
[12] Vardi, Y. 1996. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American Statistical Association* 91:365-77.
[13] Tebaldi, C., and M. West. 1998. Bayesian inference on network traffic using link count data. *Journal of the American Statistical Association* 93:557-76.
[14] Cao, J., D. Davis, S. V. Wiel, and B. Yu. 2000. Time-varying network tomography. *Journal of the American Statistical Association* 95:1063-75.
[15] Adams, A., T. Bu, R. Cáceres, N. Duffield, T. Friedman, J. Horowitz, et al. 2000. The use of end-to-end multicast measurement for characterizing internal network behavior. *IEEE Communications Magazine*, May.
[16] Coates, M., A. Hero, R. Nowak, and B. Yu. 2002. Internet tomography. *IEEE Signal Processing Magazine*, May.
[17] Kesidis, G., A. Singh, D. Cheung, and W. W. Kwok. 1996. Feasibility of fluid-driven simulation for ATM network. In *Proceedings of IEEE Globecom'96*, London.
[18] Nicol, D., M. Goldsby, and M. Johnson. 1999. Fluid-based simulation of communication networks using SSF. In *Proceedings of the 1999 European Simulation Symposium*, Erlangen, Germany.
[19] Nicol, D., and G. Yan. 2004. Discrete event fluid modeling of background TCP traffic. *ACM Transactions on Modeling and Computer Simulation* 14:1-39.
[20] Kiddle, C., R. Simmonds, C. Williamson, and B. Unger. 2003. Hybrid packet/fluid flow network simulation. In *Proceedings of the Seventeenth Workshop on Parallel and Distributed Simulation (PADS'03)*, San Diego, CA.
[21] Zhou, J., Z. Ji, M. Takai, and R. Bagrodia. 2004. MAYA: Integrating hybrid network modeling to the physical world. *ACM Trans on Modeling and Computer Simulation* 14 (2): 149-69.
[22] Nicol, D., M. Liljenstam, and J. Liu. 2003. Large-scale network simulation using SSF. In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA.
[23] Barakat, C., P. Thiran, G. Iannaccone, C. Diot, and P. Owezarski. 2002. A flow-based model for Internet backbone traffic. In *Internet Measurement Workshop'02*, San Francisco.
[24] Padhye, J., V. Firoiu, D. Towsley, and J. Kurose. 1998. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of ACM SIGCOMM'98*, Vancouver, Canada.
[25] Liu, Y., F. L. Presti, V. Misra, D. Towsley, and Y. Gu. 2004. Scalable fluid models and simulations for large-scale IP networks. *ACM Transactions on Modeling and Computer Simulation* 14 (3): 305-24.
[26] Yan, A., and W. B. Gong. 1998. Time-driven fluid simulation for high-speed networks with flow-based routing. In *Proceedings of the Applied Telecommunications Symposium'98*, Boston.
[27] Liu, B., Y. Guo, J. Kurose, D. Towsley, and W. Gong. 1999. Fluid simulation of large scale networks: Issues and tradeoffs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV.
[28] Liu, B., D. R. Figueiredo, Y. Guo, J. Kurose, and D. Towsley. 2001. A study of networks simulation efficiency: Fluid simulation vs. packet-level simulation. In *Proceedings of IEEE Infocom'01*, Anchorage, AK.
[29] Bu, T., and D. Towsley. 2001. Fixed point approximations for TCP behavior in an AQM network. In *Proceedings of ACM SIGMETRICS 2001*, Cambridge, MA.
[30] Vahdat, A., K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, et al. 2002. Scalability and accuracy in a large-scale network emulator. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*.
[31] Nicol, D. M., and G. Yan. 2005. Simulation of network traffic at coarse timescales. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation (PADS'05)*, Monterey, CA.

David M. Nicol *is Professor of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign.*

Guanhua Yan *is a post-doc in the Discrete Simulation Science division at Los Alamos National Laboratory.*