SIMULATION OF LARGE-SCALE NETWORKS USING SSF

David M. Nicol
Jason Liu
Michael Liljenstam

Department of Electrical and Computer Engineering
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St.
Urbana, IL 61801, U.S.A.

Guanhua Yan

Department of Computer Science
Dartmouth College
6211 Sudikoff Laboratory
Hanover, NH 03755, U.S.A.

ABSTRACT

Some applications of simulation require that the model state be advanced in simulation time faster than the wall-clock time advances as the simulation executes. This *faster than real-time* requirement is crucial, for instance, when a simulation is used as part of a real-time control system, working through the consequences of contemplated control actions, in order to identify feasible (or even optimal) decisions. This paper considers the issue of faster than real-time simulation of very large communication networks, and how this is accomplished using our implementation (in C++) of the Scalable Simulation Framework (SSF). Our tool (called iSSF) uses hierarchical levels of abstraction, *and* parallelism, to achieve speedups of **nearly four orders of magnitude**, enabling real-time execution rates on large network models. We quantify the effects that choice of hierarchical abstraction has on the simulation time advance rate, and show empirically how changing the abstraction mix affects the execution rate on a large network example.

1 INTRODUCTION

Discrete-event simulation is a powerful computational paradigm that allows a modeler to explore the potential behavior of many kinds of discrete systems. Some applications of discrete-event simulation require that the models be evaluated very quickly. For instance, when simulation is used at the heart of an optimization solution, the faster a model can be evaluated the richer the solution space can be explored. Even more critical are applications where the discrete-event simulation is used in a real-time control system. Not only must the simulation advance the model state in simulation time as fast as wallclock time (in the same units), more often it must advance the model state at a rate significantly faster than wallclock time, such as in

cases where the consequences of multiple different control decisions must be computed and compared. An example of this is given in Ye, Kaur, Kalyanaraman, Kenneth, Vastola, and Yadav (2002), where real-time simulation is used to decide how to select inputs for the OSPF (Moy 1998) routing protocol.

Network defense is an area ripe for application of faster than real-time simulation. It is easy to imagine a control system that considers re-routing, partitioning, and/or quarantining decisions as evidence of a cyber-attack mounts. A simulation model can work through cost/benefit/risks assessments of considered actions, using potentially sophisticated metrics to find an effective response. However, faster than real-time simulation of *large* network models requires aggressive techniques both in modeling, and in execution strategy.

We will discuss simulator performance in terms of a "baseline" packet-oriented simulation. Here the majority of the workload involves ascribing delays to packets as they move through a network, the delays being functions of queuing at routers and processing through protocol stacks. Our unit of model activity then might be a "packet-event", which reflects either sending or receiving a packet (or both, if that occurs in the same computation). A packet sent between two hosts, across 4 routers, would account for at least 6 packet events from initial transmission to final receipt (one send-only event, four receive-followed-by-send events, and one final receive event). In a strictly packet oriented simulation a packet event is implemented within the simulation kernel as one discrete event, involving a computational action applied to the member of the event list with least future time-stamp. We consequently use as a baseline metric of simulator performance the rate at which the simulation kernel executes "kernel-events", on a large network model. We recognize that this figure depends on the problem size, insofar as the cost of a priority heap access depends on

the number of elements in the heap; we likewise recognize that it depends on the computational effort associated on average with each event executed. Nevertheless the concept provides a useful baseline that, within the context of a given simulation kernel, may be only slowly sensitive to problem size if the priority list mechanism is optimized.

Traffic intensity is a good measure of model "size"—the problem of simulating a 10K device network in faster than real time under very light traffic load is very different from that of simulating that same network under heavy load. The aggregate packet-event rate demanded of a workload mix, on a given topology, describes the rate (in simulation time) at which the network state is being modified. An implementation is faster than real time if those state modifications are made at a wallclock rate that exceeds the aggregate packet-event rate, using the same units of time. For example, consider a model with 500 traffic flows, where the offered load per flow is 10 packets/second, and an average flow crosses three routers. This implies an packet-event rate of 50 packets/sec/flow, for an aggregate packet-event rate of 25,000 packet-events/sec. This model can be run four times as fast as real-time on a simulation kernel capable of executing 100K kernel-events per second.

However, if we limit ourselves to pure packet-based representation on a purely serial simulation kernel, our ability to simulate networks faster than real-time is limited by the kernel-event rate of the serial simulation kernel. In order to handle larger models it is necessary to use more abstract model representations that can affect the model state with less effort per inherent packet-event than can a packet-oriented simulator, and it may be necessary to also use parallel execution.

This paper describes how our implementation of the SSF interface (called iSSF) can meet faster than real-time challenges, using a combination of model abstraction and parallelism. We develop simple analytic models that help guide choices of abstraction to achieve faster than real-time goals, and demonstrate its faster than real-time capabilities on a large network model.

2 SSF

The Scalable Simulation Framework (SSF) defines an API for simulation kernels capable of high-performance, large-scale system simulation. SSF has definitions in Java and in C++, with multiple implementations in both languages. The API is simple, and defines five base classes. The **Entity** class serves as a container for model state variables, computational processes, and communication endpoints. The **Process** class defines a computational thread, which suspends and reactivates in a process-oriented fashion. A **Process** may suspend waiting for input on a named communication channel (or set of channels), it may suspend for a prescribed epoch of simulation time. The **inChannel** and **outChannel**

classes define communication endpoints; instances of the **Event** class are sent to instances of named **outChannel** objects, which appear (after a user-defined simulation delay) through one or more instances of **inChannel** objects. The full API may be obtained at www.ssfnet.org.

SSF defines a low level view of common capabilities of a simulation kernel. Users can (and do) develop models directly using this API, however it is also possible to define frameworks which provide domain-specific abstractions, and implement those abstractions in common libraries which interface with the SSF kernel while a user will not. Such frameworks have been developed for domains such as ad-hoc wireless communication networks, and high-performance computer architecture. SSFNet is a framework specialized for simulation of wireline communication networks. Modularity and object-orientation are characteristics that support scalable simulations. SSFNet uses the Domain Modeling Language (DML) to describe and configure network simulations. DML is simply a list of attribute-value pairs, defined recursively. A "keyword" (attribute) which will be recognized by a domain-specific parser is followed either by a string which associates a value with that keyword, or by a list (demarked by left and right brackets) of attribute-value pairs, which is itself considered to be a list structured value. So a network can be described as a list of devices, each device may have lists of attributes such as identity and capabilities, hosts have lists of traffic descriptions, each traffic description is a list of flows, and each flow has attributes that describe such things as the source and destination (normally in coordinates that reference DML structure—these are converted by SSFNet into automatically assigned IP addresses), input flow rate, schedules of input rate changes (in the case of UDP). Within DML the level of abstraction associated with a flow can also be identified with a few simple attributes. The default attributes describe a flow (UDP or TCP) as a packet-oriented flow, which behaves from end-to-end exactly as packet-oriented flows do in many different simulators. One non-default attribute, when included, indicates that the traffic is to be represented as a fluid. Internally this means the flow will be represented with a set of piece-wise constant rate functions. To determine how many bytes of traffic flow pass a certain point in the network over simulation time interval $[a, b]$, one integrates the rate function observed at that point, over time epoch $[a, b]$. As discussed in Kesidis, Singh, Cheung, and Kwok (1996), Nicol, Goldsby, and Johnson (1999), and Nicol (2001), this representation can be computationally more efficient than a pure packet oriented approach. Discrete events occur only when a flow's rate has to change, e.g. because of queuing or loss. The computational efficiencies occur when the rate does not change much, for there is only one event per rate change. In the case of TCP this occurs after the flow has gotten into congestion avoidance mode and the congestion window is large.

Our experience with fluid TCP (Nicol 2001, Nicol and Yan 2003) indicates that under typical conditions a fluid approach may reduce the computational workload by an order of magnitude. Perhaps we should say *only* an order of magnitude, because our attention to detail in the TCP mechanics costs work, at the very least, at the beginning and end of each TCP round. The workload reduction with UDP flows can be more significant, at least for long-lived flows. However, in congestion conditions it is still possible to get many rate change events from a few flows. Section 3 discusses this phenomenon in more detail.

A third level of abstraction is suitable for the aggregate description of many many background flows. These are flows which, in the aggregate and at the appropriate time-scale, aren't notably affected by flows we might choose to represent more exactly. However, as the low-resolution flows represent the bulk of the traffic, they should have a definite affect on higher resolution flows. This effect can be captured at routers, by the simple mechanism of subtracting available bandwidth on links carrying low-resolution traffic. So, rather than simulate 1000 packet-oriented flows across a link, we might account for the bandwidth consumption of 990 of them–on average and possibly with some synthetically introduced variation—by periodically changing the bandwidth available to the 10 high-resolution flows, by an amount designed to reflect the bandwidth consumption of the 990 flows in the background.

There are a variety of ways one might express this low resolution model. For the purposes of comparison against flows of other resolutions we adopted a means whereby an individual flow is named in the DML input as being fluid, and also *insensitive*. The DML file contains a schedule of rate changes injected by a host into the network. Consider the snippet below:

```
traffic [
  pattern [
    # src host (NHI coordinates)
    server 10:0:0
    schedule [    # one @ update
      time 0      # update time
      target [    # one @ dest
      dest 11:1:0 # dest host
      rate 300    # flow (Mbps)
      ]
      target [ ... ]     # other dest
    ]
    # other update scheds
    schedule [ ... ]
  ]
]
```

Working from the inside-out, a target describes a destination and a flow rate to that destination. The destination

and rate are the target's attributes. A schedule is a list of targets, which indicates a start-time for the flow rates described by the targets. A pattern identifies (in DML coordinates) a host, and a list of schedules associated with that host. A traffic list describes a list of patterns. There is normally one traffic list in a network. schedule lists implicitly describe UDP flows. Therefore all we need to do to indicate that a flow is fluid, and insensitive, is to add to a target description attributes such as

```
target [     # one @ dest
dest 11:1:0 # dest host
rate 300     # flow (Mbps)
fluid on
insensitive on
]
```

3 MIXED LEVEL SIMULATION

In SSFNet different levels of traffic abstraction meet in the router. SSFNet routers contain forwarding tables which support the forwarding of any legal packet address the simulation might generate. Explicit packets, and those fluid flows with the sensitive attribute all describe their destinations explicitly. The router models the influence that packets have on the sensitive fluid flows, and the influence that the sensitive flows have on packets. This two-way influence is captured through computed latency and loss characteristics.

SSFNet models a router's buffer space as being associated with an output port, and models the link serving that port and the traffic enqueued at that port as a FCFS fluid buffer. In this model there is an available bandwidth rate μ, a set of time-dependent input flow rates $\lambda_1(t), \ldots, \lambda_n(t)$, and a set of time-dependent output flow rates $\rho_1(t), \ldots, \rho_n(t)$. $\lambda_i(t)$ describes how quickly bits associated with flow i arrive at the buffer at time t, and $\rho_i(t)$ describes how quickly bits for flow i are being sent out at time t. A fluid buffer has a capacity C, and a time-dependent level $L(t)$, with $0 \leq L(t) \leq C$ for all t. No congestion exists when $L(t) = 0$ and $\sum_i \lambda_i(t) \leq \mu$, in which case $\rho_i(t) = \lambda_i(t)$ for all i. If $L(t) > 0$ at time t, then $\sum_i \rho_i(t) = \mu$. The bandwidth $q_i\mu = \rho_i(t)$ allocated to flow i at this instant is such that the fraction q_i of the available total is the fraction of total input flow contributed by flow i at some instant in the past. To see how this works, imagine that at time t_0- (the instant just before t_0) $L(t_0-) = 0$, and that at t_0 one or more of the flows increase the arrival rate so much that the aggregate arrival rate exceeds μ. A backlog begins to build, and so long as the buffer level is less than C and no input flow rate changes, a volume of fluid representing a particular mix of input flow rates is defined. During this period the fraction q_i of the bandwidth allocated to flow i is precisely the fraction of total input flow contributed by

flow i. Now the rate for some input may change, which means that a new volume with a different description of input flow mixes begins to build. Because the buffer is modeling FCFS processor sharing, none of that volume is output before all of the earlier output is served. This process can continue, with a potentially large number of input-mix volumes being enqueued. If the input rates drive the buffer level up to capacity C then loss takes place. An important point to remember is that in this scheme one change in an input flow rate when the buffer is congested can cause *every* flow's output rate to change, at the point one input-mix volume is completely served and a new one (defined by the input rate change) starts to be served. This is the source of the so-called ripple effect (Kesidis, Singh, Cheung, and Kwok 1996, Nicol, Goldsby, and Johnson 1999), and is one for which we provide a solution. We observe that flows traveling from one router to another typically have a latency delay between transmission and full receipt of the packet. This delay represents a period of time after the packet is sent where the receiving entity is unaware and unaffected by it. This means that during this period of insensitivity it is possible to revise flow rate changes that are "on the way", between the time they were sent and the time they affect the recipient. We do this in a way that bounds the number of fluid events that traverse a link per flow, while preserving the total traffic delivered by the fluid. This is illustrated in Figure 1. Before smoothing we have a situation that a fluid change event will reach the receiver at time t_1, and will change the rate to r_1. There is another rate change for that flow scheduled, at time t_2, which will cause the implementation of rate r_2. Yet another rate change is scheduled for time t_3. Duration $[t_1, t_3]$ is within the latency of the link between sender and receiver. Over that epoch a total of $r_1(t_2 - t_1) + r_2(t_3 - t_1)$ fluid will flow. This means that we can remove the rate change event scheduled for time t_2, and change the rate to be reported to the receiver at time t_1 in such a way that by time t_3 exactly the same amount of fluid will have been delivered. The precise rate to report at time t_1 to effect this change is $r_4 = (r_1(t_2 - t_1) + r_2(t_3 - t_2))/(t_3 - t_1)$.

It happens that we cannot implement this smoothing technique if we adhere strictly to the SSF API. Under that API, when an **Event** is written to an **outChannel** at time t, a bit-for-bit copy of that **Event** is delivered to a receipient at some time in the future (the delay between the send time and receive time is the sum of a delay specified
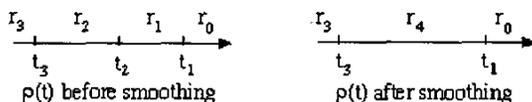

Figure 1: Smoothing Fluid Events Over a Link Latency Time

when the **outChannel** is created, a delay specified when an **outChannel** is mapped to an **inChannel**, and a delay specified when the **Event** written to the **outChannel**. The key point is that the content of that **Event** is determined at the instant it is sent. In the example of Figure 1 all of the rate changes shown are in transition between their send and receive times. We wish to delay the rate reported to the recipient until the receive time, t_1. Our implementation of SSF contains some API extensions, including one that handles cases like this. The extension is called an *appointment outChannel*. When an appointment **outChannel** is constructed, a call-back function is identified. Essentially, when an **Event** is written to the **outChannel**, a call to the callback function is scheduled at the time the **Event** is to be received. The callback function returns the data content of the **Event**, which is what is delivered to the receiving entity.

The dynamics described above give sensitive fluid flows latency due to queueing, and loss (precise description of the loss calculations are given in (Nicol and Yan 2003). SSFNet routers cause flows described by packets to affect senstive fluid flows by describing the aggregate packet flows as an virtual fluid flow that feeds into the normal fluid computations, and so represent to the fluid system the claims on bandwidth made by the packet flows. The packet flows are handled separately, in such a way that when a packet arrives its final departure time (queueing plus transmission) is computed as a function of the known unserved packets in front of it, and the backlog of actual fluid flows at the time of arrival. Details, again, are in (Nicol and Yan 2003). In this way the sensitive fluid flows affect the packet flow behaviors.

The insensitive fluid flows are easily augmented into this framework. If $A(t)$ describes the sum of the arrival rates of insensitive flows to the router at time t, then the available bandwidth for the sensitive fluid flows and the packet flows is $\mu - A(t)$. $A(t)$ changes at time-scales much larger than the time-scale of packet arrivals and sensitive fluid flows input rates. Thus, for long periods of time, the effects of insensitive flows on the rest of the system can be represented simply, but with no significant computational effort.

To summarize then, SSFNet provides three levels of traffic model abstractions, and conducts simulations with all three levels of abstraction, concurrently. Packet oriented flows involve computational work every time a packet arrives to and departs from a device. Network sensitive fluid flows can provide some aggregation of a fluid in time, over epochs when the transfer rate behavior of a flow is constant. The computational work occurs when somewhere in the network a flow's rate changes. Such changes may be caused by queuing at a router, or even loss of traffic at a router. When a fluid's rate changes a description of that change percolates downstream, triggering computation

at every device that receives the rate change descriptor. The most aggressive form of aggregation is of network insensitive flows. The effect of these on the other flows is manifested by simply claiming the bandwidth they consume from the links over which they are mapped. Updates to these flows are infrequent relative to the time-scale of packet and sensitive fluid flows.

4 MODEL

We now develop a simple analytic model that explains how simulation execution workload is affected by the mix of abstraction levels. In this model we concentrate on the *state-event* rate in simulation time. A state-event is a computational activity associated with one event on the simulation kernel's event list, an activity which may change the simulation model's state. This is to be distinguished from the *model-event* rate, which we define to be the state-event rate from a model in which every flow is a packet flow. We develop formulae to describe the aggregate state-event rate. When that rate is less than a simulation kernel's kernel-event rate (the rate at which the kernel is able to implement these state changes in *wallclock* time), the simulation engine is executing the model in faster than real-time.

To simplify the mathematics we suppose that there are N UDP total flows and no TCP flows. Each flow i is "on" long enough to generate n_i packets, is off for a time, and then repeats. The average period between successive on times is P_i. Of the N flows, we assume that N_p are packet-oriented, N_s are fluid and sensitive, and N_i are fluid and insensitive.

4.1 Packet Flows

Suppose that packet-oriented flow i crosses k_i routers. As we have seen already, every packet injected into the network from the source host gives rise to at least $k_i + 2$ state-events. The state-event rate associated with flow i is thus $n_i(k_i + 2)/P_i$. We see that the aggregate state-event rate of packet-oriented flows is

$$\Lambda_P = \sum_{i \in I_p} \frac{n_i(k_i + 2)}{P_i},$$

where I_p is the set of flow indices of packet flows.

4.2 Sensitive Fluid Flows

A sensitive fluid flow has rather a different effect on the state-event rate. As we have discussed, the arrival and processing of a fluid rate change event at a router may future rate-change events (at other routers) for other sensitive fluid flows passing through that same router port. Some of these may be smoothed away before being executed. Ones that
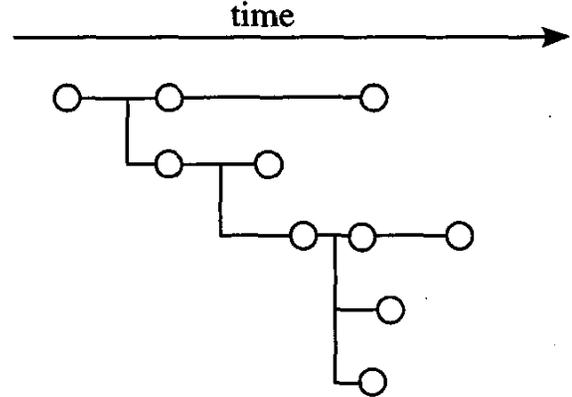

Figure 2: Branching Process Describes Fluid Rate Event Propagation

aren't smoothed can themselves causes additional events when they are processed. We model the total effect of a fluid rate change in terms of a construct called a branching process (Ross 1983). This is illustrated in Figure 2. Here we denote a fluid event with a circle, whose horizontal orientation reflects its position in simulation time. If the execution of a fluid event causes the scheduling and execution of new fluid events at other routers, a line is drawn from the originating event to the subsequent events. At the leftmost side we have a single event whose execution creates a tree of subsequent executions.

Every time the host of a sensitive fluid changes the input rate of that flow, the rate change event describing this change will propagate from the source, through every router on the path, to the destination. Each time this event is received en-route, it serves as root to a branching process of events, as described above. To enumerate the number of state-events executed as a result of the host changing the input rate, we need to count the number of state-events in each of the trees rooted in the flow-change event as it traverses its path.

In order to estimate the number of events in a branching process tree, consider the effects of a *foundational* rate change event at a router, that is, one of the rate change events that propagate directly along a flow's path to effect a rate change at the source. If the output port is not congested, the foundational rate change passes through without causing rate changes to any other flows. If the foundational rate change event is processed at a congested port, then potentially all of the flows through that port with non-zero flow rates will eventually have rate-change events scheduled as a result. Of these, some may be smoothed away. Of the changes remaining we name all those for flows other than the one for the causing event *consequential*. Each consequential event may likewise cause multiple consequential events, but for these we do count the event associated with the flow of the initiating event.

654

For simplicity we assume that all branching trees rooted in a consequential rate change are stochastically identical, meaning that each port has the same number N_{port} of sensitive flows mapped to it, and each such flow has the same probability p_{rc} of having a consequential rate change event scheduled for it as a result of a consequential rate change event being processed at that port. To quantify p_{rc}, let p_c be the probability that a flow rate change arriving at a port buffer finds it congested, let p_a be the probability that a flow is active (e.g., has non-zero flow rate) conditioned on the buffer being congested, and let $\bar{p}_s$ be the probability that a rate change is *not* smoothed as it transits a link, given that the port buffer is congested. Then the probability that an arbitrary sensitive flow has a rate change scheduled as a result of an arriving rate change is $p_{rc} = p_c \times p_a \times \bar{p}_s$. According to the theory of branching processes, the expected number of events in a branching tree rooted in a consequential event is finite if and only if the product $p_{rc}N_{port} < 1$. If this is the case then the average number of events in that tree is $1/(1 - p_{rc}N_{port})$. If $p_{rc}N_{port} \geq 1$ then in this model, the explosion of rate change events triggered by the root of the tree does not die away. There is a limit to this, we have proven elsewhere that our smoothing technique ensures that no more than 2 rate change events cross a link per link latency time. Nevertheless the point is made—to avoid rate change event explosion, the average number of rate-change events caused by a consequential rate change event needs to be less than 1.

The average number of events in a branching process rooted in a foundational event is

$$1 + \frac{p_{rc}(N_{port} - 1)}{1 - p_{rc}N_{port}}.$$

The fraction's numerator is the average number of events caused by the foundational event, *other than one on that same flow*. This is multiplied times the average size of a tree rooted in a consequential event. The expected total number of events related to a rate change for flow i at the host is thus

$$k_i \times \left(1 + \frac{p_{rc}(N_{port} - 1)}{1 - p_{rc}N_{port}}\right) + 2$$

recalling that k_i is the number of routers on the path (and we add 2 to account for the events at source and destination hosts). Recalling that expected simulation duration between successive "on" periods for this flow is P_i, then the rate at which this flow generates rate change events is, accounting for one "start" and one "stop" event per period

$$\frac{k_i \times \left(1 + \frac{p_{rc}(N_{port}-1)}{1-p_{rc}N_{port}}\right) + 2}{P_i}.$$

This equation ignores branching trees that are introduced when a port buffer transitions between the "filling" state, the "full" state, and the "uncongested" state.

The overall rate at which sensitive fluid flows generate events in the simulation is

$$\Lambda_S = \sum_{i \in I_s} \frac{k_i \times \left(1 + \frac{p_{rc}(N_{port}-1)}{1-p_{rc}N_{port}}\right) + 2}{P_i}$$

where I_s is the set of flow indices of sensitive fluid flows.

5 INSENSITIVE FLUID FLOWS

We suppose that the aggregate insensitive fluid flow bandwidth demand at each router is updated simultaneously, every Δ units of simulation time. These changes can affect sensitive fluid flows. We can model that effect as if, simultaneously, one fluid input to every port changed rate. That is, every change of available bandwidth may trigger a branching tree of consequential events, through the sensitive fluid flows. As a result of the bandwidth changes, the probability p_c of a consequential event finding congestion at a port has changed. We therefore denote the new probability of a flow having a rate change event scheduled on it as p'_{rc}. If the model has M ports, then the rate at which changes to insensitive fluid flows cause events is

$$\Lambda_I = \frac{M \times \frac{1}{1-p'_{rc}N_{port}}}{\Delta}.$$

Note that this expression is independent of the number of insensitive fluid flows.

Figure 3 illustrates the power of abstraction to reduce execution costs. The scenario depicted scales up a model in which the characteristics given in Table 1 are held constant.
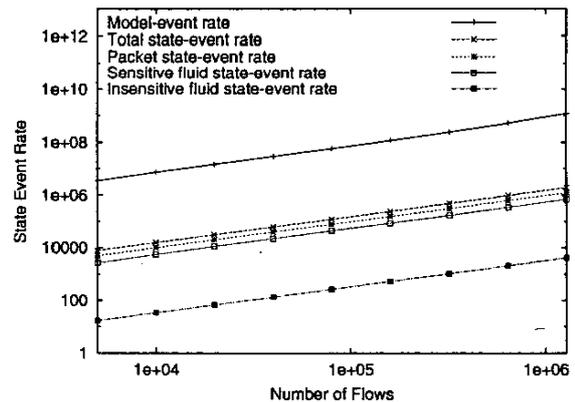

Figure 3: Breakdown of State-Event Rates Predicted by an Analytic Model, as a Function of Model Size, on a Log-Log Scale

Table 1: Model Characteristics

Hops per flow	5
Packets per flow per second	100
Sensitive flows / Packet flows	66.66
Insensitive flows / Sensitive flows	10
Sensitive flows / router port	50
p_{rc}	0.01
Δ	60 seconds
#flows / #routers	550.5

The most important aspect of this configuration is the high degree of aggregation. There is a 200:3 ratio of sensitive flows to packet flows, and a 10:1 ratio of insensitive flows to sensitive flows. This reflects an application where one is interested in the detailed behavior of a small number of applications. Sensitive flows add variation to the bandwidth available to the application of interest. It is also important to notice that the branching trees of rate-changes associated with sensitive flows do not explode under these assumptions. Figure 3 is plotted on a log-log scale, and shows the very significant differences in state-event rates due to insensitive flows. The reduction of effort due to abstraction is three orders of magnitude. Despite the packet flows accounting for only 1/6 of a percent of all flows their workload accounts for 66% of the whole execution budget. This reduction supports faster than real-time simulation of very large models. Under this particular mix of workload to abstraction levels, a simulation engine capable of executing 2M state-events per second can simulate a model with over 1.3M flows faster than real-time. The figure of 2M state-events per second is one we have achieved using parallelism.

Figure 4 graphs the largest model size (in numbers of flows) which can be simulated faster than real-time, assuming the parameters of Table 1, except for the mix of abstractions. Throughout the experiments the packet oriented flows account for 0.14% of the flows, the remaining flows are partitioned among sensitive and insensitive fluid flows according to the independent variable. Thus we see that under the stated model assumptions, there is an order magnitude difference in the number of flows that can be simulated as fast as wallclock time, and that the number of flows that can be simulated that fast is quite large.

6 EXPERIMENTS

To demonstrate that faster than real-time simulation of very large models is more than a theoretical possibility, we report here on a set of experiments run on the iSSF system. These experiments use the same parametric model construction as that which underlies Figure 3. However, for these experiments the ratio of sensitive flows to packet flows is smaller, 100:3, and the distribution of fluid flows among the sensitive and insensitive classifications is an experimental variable. We use a distributed memory cluster computer to run the
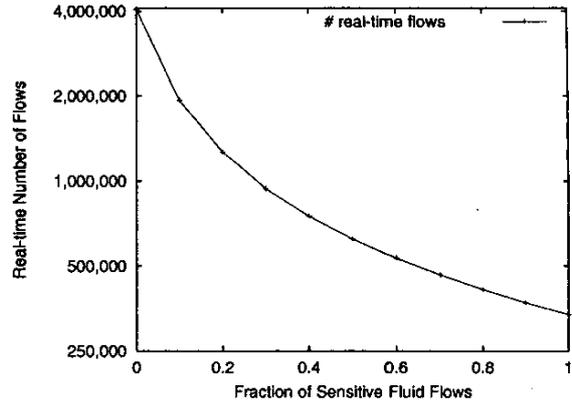

Figure 4: Largest Model Size Capable of Simulation in Real-Time, Predicted by the Analytic Model, as a Function of the Fraction of Fluid Flows that are Sensitive, Assuming a Simulation Engine Capable of Executing 2M State-Events per Second

experiments, using 20 CPUs on each experiment. Like Figure 3, Figure 5 uses the number of flows as the independent variable (understanding that a complete specification of the workload also needs specification of the average hops per flow—5—and the application traffic injection rate—100 packets per flow per second). The dependent variable is *slowdown*—the ratio of the execution time to the simulation time. We consider experiments which vary the percentage of sensitive flows from 0% to 50%.

Slowdown is a linear function of the number of flows, which just means that the execution time scales linearly in the number of flows. For any given mixture of insensitive and sensitive flows we look for the largest model size which can be executed faster than real-time. So, in the case when all fluid flows are insensitive, a model representing 850,000 flows can be executed faster than real-time; when 12.5% are sensitive then a model with something close to 450,000
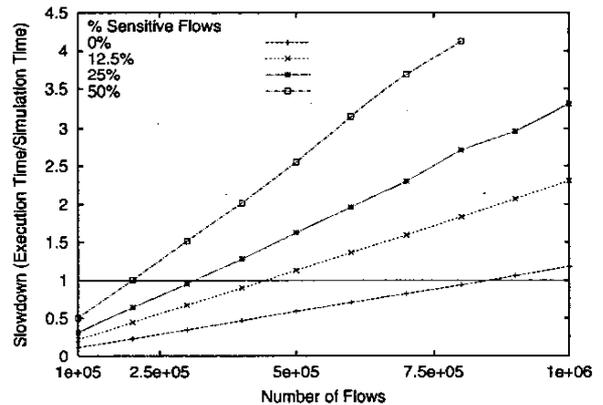

Figure 5: Slowdown as Function of Traffic Composition and Model Size. Faster than Real-Time Performance is Achieved at Sizes Where the Slowdown is Less than 1

flows can be executed faster than real-time. When the fluid flows are evenly divided between sensitive and insensitive, a model with 200,000 flows can be run faster than real-time.

7 CONCLUSIONS

The main point to be taken from this paper is that using model abstraction and parallelism, we are able to simulate very large models faster than real-time. On the models described in this paper, abstraction reduces the computational workload by a factor of approximately 400, while parallelism reduces execution time by a factor of 20. This gives an aggregate acceleration factor of **8000**, over the capabilities of a pure packet simulator using only one CPU.

Accuracy of results is of course an important consideration in the abstraction-tradeoff consideration. We have demonstrated elsewhere (Nicol and Yan 2003) that iSSF models which mix packet flows and sensitive fluid flows are very accurate; we are working on establishing accuracy results including insensitive flows as well.

ACKNOWLEDGMENTS

REFERENCES

Kesidis, G., A. Singh, D. Cheung, and W. Kwok. 1996, Nov.. Feasibility of fluid event-driven simulation for ATM networks. In *IEEE Globecom 1996*.

Moy, J. 1998. *OSPF: Anatomy of an internet routing protocol*. Addison-Wesley.

Nicol, D., M. Goldsby, and M. Johnson. 1999, Oct.. Fluid-based simulation of communication networks using SSF. In *Proceedings of the 1999 SCS European Simulation Conference*. Erlangen, Germany.

Nicol, D. M. 2001, December. Discrete-event fluid modeling of TCP. In *Proceedings of the 2001 Winter Simulation Conference*. Arlington, VA.

Nicol, D. M., and G. Yan. 2003. Discrete-event fluid modeling of TCP for background traffic. Submitted for publication.

Ross, H. 1983. *Stochastic processes*. New York: Wiley.

Ye, T., H. T. Kaur, S. Kalyanaraman, Kenneth, S. Vastola, and S. Yadav. 2002. Dynamic optimization of OSPF weights using online simulation.

AUTHOR BIOGRAPHIES

DAVID M. NICOL is Professor of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign, and member of the Coordinated Sciences Laboratory. He is co-author of the textbook *Discrete-Event Systems Simulation*, and served as Editor-in-Chief at ACM TOMACS from 1997-2003. He will serve as the General Chair of the Winter Simulation Conference in 2006. From 1996-2003 he was Professor of Computer Science at Dartmouth College, where he served as department chair, and at the Institute for Security Technology Studies served as Associate Director for Research and Development, and finally as Acting Director. From 1987-1996 he was on the faculty of the Computer Science department at the College of William and Mary; 1985-1987 he was a staff scientist at the Institute for Computer Applications in Science and Engineering. He has a B.A. in mathematics from Carleton College (1979), an M.S. (1983) and Ph.D. (1985) in computer science from the University of Virginia. His research interests are in high performance computing, performance analysis, simulation and modeling, and network security. He is a Fellow of the IEEE.

JASON LIU is a post-doctoral student at the Coordinated Sciences Laboratory, at the University of Illinois, Urbana-Champaign. His research focuses on parallel discrete-event simulation, performance modeling and simulation of computer systems and communication networks, and large-scale simulation of wireless ad hoc networks. He received B.A. in Computer Science from Beijing Polytechnic University in China in 1993, M.S. in Computer Science from College of William and Mary in 2000, and Ph.D. in Computer Science from Dartmouth College in 2003.

MICHAEL LILJENSTAM is a post-doctoral student at the Coordinated Sciences Laboratory, at the University of Illinois, Urbana-Champaign. His research interests include: large scale network modeling, security, routing, traffic analysis, and modeling and simulation of wireless networks. He received his M.Sc. and Ph.D. degrees in Computer Science from the Royal Institute of Technology (KTH), Stockholm, Sweden in 1993 and 2000, respectively.

GUANHUA YAN is a Ph.D. student in the Computer Science Department at Dartmouth College. His current research focus is on large-scale network modeling and simulation, and real-time network emulation. He received the B.E. in Computer Science at Huazhong University of Science & Technology in China in 1997, M.E. in Computer Science at Beijing University of Posts & Telecommunications in China in 2000.