Large-Scale TCP Models Using Optimistic Parallel Simulation

Garrett Yaun and
Christopher D. Carothers

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, U.S.A.
{yaung,chrisc}@cs.rpi.edu

Shivkumar Kalyanaraman

Department of Electrical and
Computer Systems Engineering
Rensselaer Polytechnic Institute
Troy, NY 12180, U.S.A.
shivkuma@ecse.rpi.edu

Abstract

Internet data traffic is doubling each year, yet bandwidth does not appear to be growing as fast as expected and thus short falls in available bandwidth, particularly at the "last mile" may result. To address these bandwidth allocation and congestion problems, researchers are proposing new overlay networks that provide a high quality of service and a near lossless guarantee. However, the central question raised by these new services is what impact will they have in the large? To address these and other network engineering research questions, high-performance simulation tools are required. However, to date, optimistic techniques have been viewed as operating outside of the performance envelope for Internet protocols, such as TCP, OSPF and BGP.

In this paper, we dispel those views and demonstrate that optimistic protocols are able to efficiently simulate large-scale TCP scenarios for realistic, network topologies using a single Hyper-Threaded computing system costing less than $7,000 USD. For our real-world topology, we use the core AT&T US network. Our optimistic simulator yields extremely high efficiency and many of our performance runs produce **zero rollbacks**. Our compact modeling framework reduces the amount of memory required per TCP connection and thus our memory overhead per connection for one of our largest experimental network topologies was 2.6 KB. That value was comprised of all events used to model TCP packets, TCP connection state and routing information.

1 Introduction

The predominate technique used to analyze Internet protocol behavior is packet-level, discrete-event simulation. Here, networking researchers are interested in examining how routing protocols like OSPF and BGP have on quality service guarantees and measures [11] as well as other large-scale network operation and engineering problems. Because the computational requirements of this problem are immense, network designers require tools that can efficiently model a network with potentially millions of nodes and data streams. These tools will enable better network configurations and more efficient, accurate management of capacity.

To date, optimistic techniques, such as Time Warp [14], have been viewed as operating outside performance range for Internet protocol models such as TCP, OSPF and BGP. The reason most often cited are state-saving overheads [24, 23]. These overheads not only impede the performance of the model but also limits the scale of the model because of increased memory consumption. Other critiques of optimistic methods are associated with inconsistent states due to the inherent risk involved in optimistic processing [18].

In this paper, we dispel those views and demonstrate that optimistic protocols are able to efficiently simulate over million node TCP scenarios for realistic, network topologies. In addition to efficient execution, we observed that Time Warp executes with increased stability as model size increases and handles short delays in high-bandwidth links exceptionally well. Last, from the developers point-of-view, the issue of inconsistent states as a consequence of full optimistic processing was not observed. As we moved our TCP model from sequential to parallel execution, the only real developer overhead was in the creation of reverse execution event handlers. Special error handling considerations for the forward code path were not required.

The innovations for achieving this level of scalability are two fold. First, to the undo operation as part of optimistic processing, we employ *reverse computation* [2]. Here, the event computations are developed such that they can be reverse processed as opposed to state saving event computations as events process forward. This approach has been shown to have negligible impact on forward execution and substantially reduces the memory requirements of the optimistic parallel models [2, 30]

The second innovation is our *compact implementation* modeling approach. Here we demonstrate a TCP model compactly implemented atop a parallel discrete-event platform. The object hierarchy for a TCP connection is kept extremely lean and compressed into a single contiguous logical process (LP) state vector. Similarly, the event data is compressed to a minimum for the feature set of the protocol model. This approach enables a single TCP connection state to only occupy 320 bytes total (both sender and receiver) and 64 bytes per each packet-event.

The end result of these innovations is that we are able to simulate million node network topologies using commercial off-the-shelf multiprocessor systems costing less than $7,000 USD, and consumes less than 1.4 GB of RAM in total.

In the next section, we describe our TCP model and its implementation on an optimistic parallel simulation engine called ROSS.

2 TCP Model

2.1 TCP Overview

The Internet relies on the TCP/IP protocol suite combined with router mechanisms to perform the necessary traffic management functions. TCP provides reliable transport using a end-to-end window-based control strategy [12]. TCP design is guided by the "end-to-end" principle which suggests that "functions placed at the lower levels may be redundant or of little value when compared to the cost of providing them at the lower level" As a consequence, TCP provides several critical functions (reliability, congestion control, session/connection management) because layer four is where these functions can be completely and correctly implemented.

While TCP provides multiplexing/de-multiplexing and error detection using means similar to UDP (e.g.: port numbers, checksum), one fundamental difference between them lies is the fact that TCP is connection oriented and reliable. The connection oriented nature of TCP implies that before a host can start sending data to another host, it has to first setup a connection using a 3-way reliable handshaking mechanism.

The functions of reliability and congestion control are coupled in TCP. The reliability process in TCP works as follows:

When TCP sends the segment, it maintains a timer and waits for the receiver to send a acknowledgment on the receipt of the packet. If an acknowledgment is not received at the sender before its timer expires (i.e. a timeout event), the segment is retransmitted. Another way in which TCP can detect losses during transmission is through duplicate acknowledgments. Duplicate acknowledgments arise due to the cumulative acknowledgment mechanism of TCP wherein if segments are received out of order, TCP sends a acknowledgment for the next byte of data that it is expecting. Duplicate acknowledgments refer to those segments that re-acknowledge a segment for which the sender has already received an earlier acknowledgment. If the TCP sender receives three duplicate acknowledgments for the same data, it assumes that a packet loss has occurred. In this case the sender now retransmits the missing segment without waiting for its timer to expire. This mode of loss recovery is called "fast retransmit".

TCP's flow and congestion control mechanisms work as follows: TCP uses a window that limits the number of packets in flight, (i.e. unacknowledged). TCP's congestion control works by modulating this window as a function of the congestion that it estimates. TCP starts with a window size of one segment. As the source receives acknowledgments, it increase the window size by one segment per acknowledgment received ("slow start"), until a packet is lost, or the receiver window (flow control) limit is hit. After this event it decreases its window by a multiplicative factor (one half) and uses the variable `ss_thresh` to denote its current estimate of the network bandwidth-delay product. Beyond `ss_thresh` the window size follows a linear increase. This procedure of additive increase/multiplicative decrease (AIMD) allows TCP to operate in an efficient and fair manner [5].

The various flavors of TCP (TCP Tahoe, Reno, SACK) differ primarily in the details of the congestion control algorithms, though TCP SACK also proposes an efficient selective retransmit procedure for reliability. In TCP Tahoe, when a packet is lost, it is detected through the fast retransmit procedure, but the window is set to a value of one and TCP initiates slow start after this. TCP Reno attempts to use the stream of duplicate acknowledgments to infer the correct delivery of future segments, especially for the case of occasional packet loss. It is designed to offer 1/2 RTT of quiet time, followed by transmission of new packets until the acknowledgment for the original lost packet arrives. Unfortunately Reno often times out when a burst of packets in a window are lost. TCP NewReno fixes this problem by limiting TCP's window reduction to at most during a single congestion epoch. TCP SACK enhances NewReno by adding a selective retransmit procedure where the source can pinpoint blocks of missing data at receivers and can optimize its retransmission. All versions of TCP would timeout if the window sizes are small (e.g.: small files) and the transfer encounters a packet loss. All versions of TCP implement Jacobson's RTT estimation algorithm (that sets the timeout to the mean RTT plus four times the mean deviation of RTT, rounded up to the nearest multiple of the timer-granularity (e.g.: 500 ms)). A comparative simulation analysis of these versions of TCP was done by Fall and Floyd[8].

2.2 ROSS: Optimistic Parallel Simulation Engine

ROSS is an acronym for Rensselaer's Optimistic Simulation System. It is a parallel discrete-event simulator that executes on shared-memory multiprocessor systems. ROSS is geared for running large-scale simulation models.

To achieve good parallel performance, ROSS uses a technique called Reverse Computation. Here, the roll back mechanism in the optimistic simulator is realized not by classic state-saving, but by enabling events to be reverse processed. Thus, as models are developed for parallel execution, both the forward and reverse execution code must be written.

The key property that Reverse Computation exploits is that a majority of the operations that modify the state variables are "constructive" in nature. That is, the undo operation for such operations requires no history. Only the most current values of the variables are required to undo the operation. For example, operators such as $++$, $--$, $+=$, $-=$, $*=$ and $/=$ belong to this category. Note, that the $*=$ and $/=$ operators require special treatment in the case of multiply or divide by zero, and overflow/underflow conditions. More complex operations such as *circular shift* (*swap* being a special case), and certain classes of random number generation also belong here.

Operations of the form $a = b$, modulo and bitwise computations that result in the loss of data, are termed to be *destructive*. Typically these operations can only be restored using conventional state-saving techniques. However, we observe that many of these destructive operations are a consequence of the arrival of data contained within the event being processed.

For example, in our TCP model, the last-sent time records the time stamp of the last packet forwarded on a router LP. We use the *swap* operation to make this operation reversible. More detail on precisely how the TCP model was made reversible is given in Section 2.6.

The ROSS API is kept very simple and lean. Developed in `ANSI C`, the API is based on logical process or LP model. Here, an LP represents a physical object in the model such as a host or router in the case of network simulation. To model packets traveling through the network, LPs will schedule time stamped events messages. Services are provided to allocate and schedule messages between LPs. A random number generator library is provided based on L'Ecuyer's Combined Linear Congruential Generator [15]. Each LP by default is given a single seed set. All memory is directly managed by the simulation engine. Fossil collection and global virtual time computations are driven by the availability of free event memory. Their frequencies are controlled with tuning parameters and start-up memory allocation. The event-list priority queue can be configured to a Calendar Queue [1], a binary heap or splay tree. The heap is used in all experiments presented here.

To reduce fossil collection overheads, ROSS introduces kernel processes (KPs). A KP contains the statistics and process event-list for an collection of LPs. With KPs there are fewer event-list to search through during fossil collection, thus improving performance, particularly when the number of LPs is large. For the experiments presented here we typically allocate 4 to 8 KPs irrespective of the number of LPs. KPs are similar to DaSSF timelines [19] and USSF clusters [26]

For more information on ROSS and Reverse Computation we refer the interested reader to the ROSS User's Guide [4]

2.3 ROSSNet TCP Host Functionality

Our implementation follows the TCP Tahoe specification. Below are the specific capabilities of the ROSSNet TCP session on a single host.

- **Logs:** The system has the ability to log sequence numbers, and congestion control window information. This information was used in our validation study. For performance runs, logging was disabled.

- **Receiver side:** Data is acknowledged when received. If the received packet's sequence number is NOT equal-to AND is greater-than the expected sequence number, it is stored in the receive buffer. Next, an acknowledgment is sent for the wanted packet (duplicate acknowledgment). When a packet with the expected sequence number is received, the next appropriate acknowledgment is sent according to the receive buffer's contents.

- **Sender side:** The sender will be in slow-start until the congestion window is greater than the slow-start threshold. After that, congestion avoidance is started. If 3 duplicate acknowledgments are observed by the sender, then fast retransmission is performed (see below). If the acknowledgment sequence number is greater then the lowest unacknowledged sequence number, the sender assumes that a gap was filled and sends the appropriate packet.

- **Fast retransmission:** When 3 duplicate acknowledgments are observed, fast retransmission is started. Here, the slow-start threshold is set to half the minimum congestion window size or the maximum of the receive window. If this value is less than two times the maximum segment size, the slow start threshold is reset to that value. The congestion window is set to maximum segment size.

- **Slow start** In slow start, two packets are sent for every acknowledgment. Here, the congestion window grows by one maximum segment size every acknowledgment.

- **Congestion avoidance** The window grows by one maximum segment size every window worth of acknowledgments. Here, one packet per acknowledgment is normally sent and two packets are sent every congestion window worth acknowledgments.

- **Round trip time (Rtt)**.The Rtt is measured one segment at a time. When sending a packet and Rtt is not being measured, a new measure is initiated. When retransmitting, cancel the current Rtt measurement if ongoing. The Rtt measurement process is complete upon receiving the first acknowledgment that covers the Rtt packet which is being measured.

- **Round trip timeout (Rto)**. We approximate Rto using a weighted average of the past values of Rto and Rtt. We are currently implementing Jacobson's tick-based algorithm for computing round trip time, which provides more of a dampen Rto computation by including the deviation its measure [12].

2.4 ROSSNet TCP Model Data Structures

In the implementation of the TCP model there are three main data structures. The message, which is the data packet, is sent from host to host via the forwarding plane. The routers LP state maintains the queuing information along with the dropped statistics. Finally the host LP's data structure keeps track of the transferring of data.

A message contains the source and destination address. These addresses are used for forwarding. The message also has the length of the data being transferred which is used to calculate the transfer times at the routers. The acknowledgment number is also included for the sender to observe which packets have been received. The sequence number is another variable which indicates which chuck of data is being transferred.

Now, in our model the actual data transferred is irrelevant and therefore it was not modeled. However in the case that the application was running on top of TCP, such as the Border Gateway Protocol (BGP), such data is required for the correctness of the simulation. We are currently examining solution to this issue.

Now, the router model's state is kept small by exploiting the fact that most of the information is read-only and does not change for the static routing scenarios described in this paper. Inside each router, only queuing information is kept along with a dropped count statistics.

There is a global adjacency list which contains link information. This information is used by the All-Pairs-Shortest-Path algorithm to generate the set global routing tables (one or each router). Each table is initialized during simulation setup and consists only of the next hop/link number for all routers in the network.

Given the link number the router can directly lookup of the next hop's IP address in its entry of the adjacency list. The adjacency list has an entry for each router and each entry contains all the adjacencies for that router. Along with the router neighbor's address, it contains the speed, buffer size, and link delay for that neighbor.

The host has the same data structures for both the sender and receiver sides of the TCP connection. There is also a global adjacency list for the host, however there is only one adjacency per host. In our model, a host is not multi-homed and can only be connected to one router. There is also a read-only global array which contains the sender or receiver host status, and size of the network transfer (which is usually a file of infinite size). The maximum segment size and the advertised window size were also implemented as global variables to cut down on memory requirements.

The receiver contains a "next expected sequence" variable and a buffer for out-of-order sequence numbers. On the sender side of a connection the following variables are used to complete our TCP model implementation: the round trip timeout (Rto), the measured round trip time (Rtt), the sequence number that is being use to measure the Rtt, the next sequence number, the unacknowledged packet sequence number, the congestion control window (cnwd), the slow-start threshold, and the duplicate acknowledgment count.

For all experiments reported here, the Rto is initialized to 3 second at the beginning of a transfer, along with the slow start threshold being initialized to 65,536. The maximum congestion window size is set to 32 packets, however this value is easily modified. The host, in addition to the variables needed for TCP, has variables for statistic collection. Each host keeps track of the number of packets sent and received, the number of timeouts that occur and its measurement of the transfer's throughput.

2.5 Compressing Router State

As previously indicated, our router design at this point is assumed to have a fixed, static routes. By leveraging this assumption, we set out to reduce the router table state.

Now, a problem encountered with real Internet topologies, such as the AT&T network, is they tend not to have an well defined structure for the purpose of imposing a space-efficient address mapping scheme. Ideally, one would like to impose some hierarchical address mapping scheme on the topology for the purposes of compressing the routing tables. From the point-of-view the model, such a compression will not lead to an incorrect simulation of the network so long as flow paths remain the same from the real network to the simulated network. Currently, we are implementing such a scheme.

Our implementation of the routing table just contains the next hop's link number. **Here, the maximum number of links per routers is 67. Therefore the routing table could be represented in a byte per entry instead of an full integer size address**. In our simulation we have an entry in the routing table for each router. If we had to have an entry for each host, the routing tables would be extremely large. The hosts were addressed in such a way that the router they are connected to can be inferred and therefore a routing table of only routers is acceptable. In the case that it cannot be inferred, we could have a global table of hosts and the routers that they are connected to. This one table is a lot smaller than having a routing table in each router with every host. We note that some topologies are such that a routing table is not needed, such as a hypercube. In these topologies the next hop can be inferred based on current router and the destination.

```
ack = SV->unack;
SV->unack = M->ack + g_mss;
/* other operations */
M->ack = ack;
```

Figure 1: LP state to message data swap example for acknowledgment process.

Next, we assume that routers implement a drop-tail queuing policy. Because of this, routers need not keep a queue of packets to be sent. Instead, the routers schedule packets based on the service rate based on bytes per seconds and the timestamp of last sent packet. As an example of how this works, let assume we have a buffer size of 2 packets, a service time of 2.0 time units per packet and 4 packets arrive at the following times: 1.0, 2.0, 3.0 and 3.0. Clearly, the last packet will be dropped, but lets see how we can implement this without queuing them. If we keep track of the last send time, we see that the packet at 1.0 will be scheduled at 3.0, following 5.0 and 7.0. Thus, when the last packet arrives, the last sent time is 7.0. If we subtract arrival time of last packet, 3.0 from the last sent time of 7.0, this says there are 4.0 time units worth of data to be sent, which dividing by the service time, yields there are currently 2 packets in the queue. Thus, this packet will be dropped. We are currently examining how this approach could be extended to other queuing policies as well as correct operation under dynamic routing scenarios.

2.6 Reverse Computation

The TCP model uses both reverse computation and incremental state saving. However, as opposed to using a logging structure [10], we reuse data space contained within the event that is currently be processed. This not only reduces the complexity of our simulation engine by not having to perform complex memory management on logs, but also increasing data locality. Because we know that the page of memory a message is located will be accessed as a consequence of normal event process. Thus, memory overheads (space and time) will not increase by having to swap data between an LP and existing message data.

As an example of how the swap operation is used in our TCP model, consider the processing of an acknowledgment event, as shown in Figure 1.

A swap operation is performed between the message's acknowledgment value $M- > ack$ and the unacknowledged sequence number contained within the LP's state $SV- > unack$. This is done to effectively state-save the unacknowl-

```
Forward:

M->dest = SV->cwnd;
SV->cwnd = 1;

Reverse:

SV->cwnd = M->dest;
```

Figure 2: Swap of the congestion window in a timeout.

```
Forward:

while(SV->out_of_order[cur_var]){
  M->RC.dup_count++;
  SV->out_of_order[cur_var] = 0;
  cur_var++;
}

Reverse:

for(i = M->RC.dup_count; i > 0 ; i--) {
  SV->out_of_order[cur_var] = 1;
  cur_var--;
}
```

Figure 3: Forward and reverse code for the out-of-order buffer.

edged sequence number prior to is being overwritten with new acknowledge sequence number. The message acknowledgment value can be recreated by subtracting `g_mss` from unacknowledged sequence number. We also use the message destination to swap congestion window `cwnd` since the packet is already at the destination. Figure 2 shows the forward and reverse code for this swap.

Moreover, the receiver's state and the out of order buffer could take a lot of space to state save. The reason for this is that the whole window worth of packets could be acknowledged with the correct sequence number. This would mean that the buffer would be completely emptied, so the whole buffer state would need to be saved. In our implementation the buffer has a one or a zero depending if the packet is in the buffer. The buffer is circular starting with the next expected sequence number. The only time the buffer changes state dramatically is when a sequence number fills in a gap. The acknowledgment is sent for next missing sequence number. All buffer locations would be set to zero up to the next gap. Because the buffer locations are all consecutive, we are able count how many were set to zero and record that number as seen in Figure 3. The reverse event handling code uses the packet which was acknowledged and the count to revert the previous values from zero back to one.

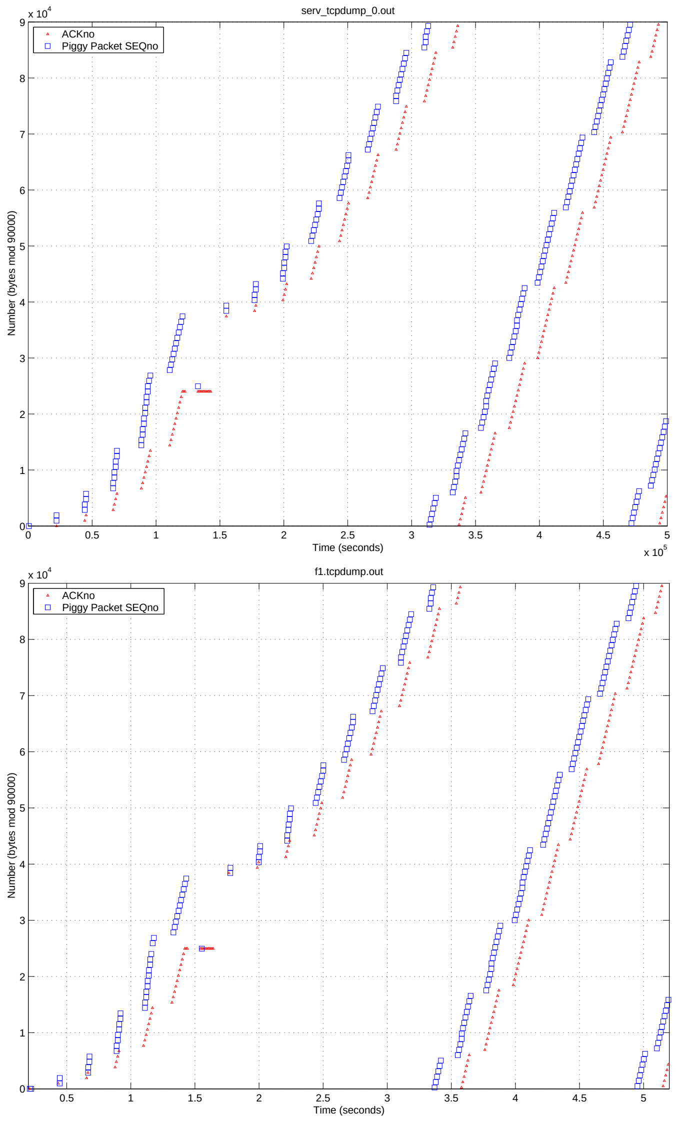

Figure 4: Comparison of SSFNet and ROSSNet TCP models based on sequence number for TCP Tahoe fast retransmission behavior. Top panel is ROSSNet and bottom panel is SSFNet.

2.7 ROSSNet TCP Model Validation

SSFNet [7] has a set of validation test which shows the basic behavior of TCP. Because of space limitations, we are only showing TCP Tahoe's behavior with congestion avoidance and fast retransmission.

As can be seen from Figures 4 our implementation with respect to sequence number behavior performs very similar to SSFNet. The packet drop happens at similar times and so does the fast retransmission.

3 Performance Study

3.1 Hyper-Threaded Computing Platform

Our experiments were conducted on a dual Hyper-Threaded Pentium-4 Xeon processor system running at 2.8 GHz. *Hyper-threading* is Intel's name for a simultaneous multithreaded architecture (SMT) [16]. SMT supports the co-scheduling of many threads or processes to fill-up unused instruction slots in the pipeline caused by control or data hazards. Because the system knows that there can be no control or data hazards between threads, all threads or processes that are ready to execute can be simultaneously scheduled. In the case of threads that share data, mutual exclusion is guarded by locks. Consequently, the underlying architecture need not know about shared variables or how they are used at the program level. Additionally, because the threads assigned to the same physical processor share the same cache, there is no additional hardware needed to support a cache-coherency mechanism.

Intel's Hyper-Threaded architecture supports two instruction streams per processor core [13]. From the OS scheduling point-of-view, each physical processor appears as if there are two distinct processors. Under this mode of operation, an application must be threaded to take advantage of the additional instruction streams. The dual-processor configuration behaves as if it was a quad processor system. Because of multiple instruction streams per processor, we denote instruction stream (IS) count instead of processor count in our performance study to avoid confusing the issue between physical processor counts and virtual processors or separate instruction streams.

The total amount of physical RAM is 6 GB. The operating system is Linux, version 2.4.18 configured with the 64 GB RAM patch. Here, each process or group of threads (globally sharing data) is limited to a 32 bit address space, where the upper 1 GB is reserved for the Linux kernel. Thus, an application is limited to 3 GB for all code and data (both heap and stack space and thread control data structures).

3.2 TCP Configuration

For all experiments, each TCP connection maintain a consistent configuration. The transfer size was infinite, leading to the transfers running for the duration of the simulation. The maximum segment size was set to 960 bytes and the total size of all headers was 40 bytes. The packet size was 1,000 bytes which was consistent with the size used in SSFnet's validation tests. The Initial sequence number was initialized to zero and the slow start thresh was 65,536.

We did, however, have a window size of 16 for the one million host case which differed from the default window size was 32. We set the window to 16, hoping to cut down on the number of packets in the system. Later we found out that changing the window size did not have an impact due to the limits on the bandwidth in the system. The congested links force the TCP window to shrink so that the slow-start threshold became small and the upper bounds on the window size is never encountered. A similar behavior has been recently noted in [20].

All clients and servers were connected in the way that the first half of hosts randomly connected to the second half of hosts. There was a distinct client-server pair for each TCP connection in the simulation. Because of the random nature of connections, there was a high percentage of "long-haul" links that result in a large the number of remote events scheduled between threads.

3.3 Synthetic Topology Experiments

The synthetic topography was fully connected at the top and had 4 levels. A router at one level had N lower level routers or hosts connected. The number of nodes was equal to $N^4 + N^3 + N^2 + N$. N was varied between, 4, 8, 16, and 32. The nodes were numbered in such a way that the next hop can be calculated based on the destination at each hop.

The bandwidth, delay and buffer size for the synthetic topology is as follows:

- 2.48 Gb/sec, a delay of 30 ms, and 3 MB buffer,
- 620 Mb/sec, a delay between 10 ms to 30 ms, and 750 KB buffer,
- 155 Mb/sec, a delay of 5 ms, 10ms and 30ms, and 200 KB buffer,
- 45 Mb/sec, a delay of 5 ms, and 60 KB buffer,
- 1.5 Mb/sec, a delay of 5 ms, and 20 KB buffer,
- 500 Kb per second, a delay of 5 ms, and 15 KB buffer

Here, we considered 3 bandwidth scenarios: (i) **high**, which has 2.48 Gb/sec for the top-level router link bandwidths, and each lower level in the network topology uses the next lower bandwidth yielding a host bandwidth of 45 Mb/sec, (ii) **medium**, which starts with 620 Mb/sec and goes down to 1.5 Mb/sec at the end host, and (iii) **low**, which starts with 155 Mb/sec and goes down to 500 Kb/sec at the end host. We note that these bandwidths and link delays are realistic relative to networks in practice.

Our test were run on 1, 2 and 4 instructions streams (IS). The synthetic topography was mapped with each core router and all its children mapped to the same processor.

Table 1 show the performance results for all synthetic topology scenarios across varying numbers of available instruction streams on the Hyper-Threaded system. For all configurations, we report an extremely high degree of efficiency. The lowest efficiency is 97.4% and to our surprise we observe a large number of zero rollback cases for 2 and 4 instruction streams resulting in 100% simulator efficiency. We observe that the amount of available work per instruction stream (IS) retards the rate of forward progress of the simulation, particularly as N grows and the bandwidth increases. Thus, remote messages arrive ahead of when they need to be processed resulting almost perfect simulator efficiency. This result holds despite inherently small lookahead which is a consequence of link delay and relatively large amount of remote schedule work, which ranges from 7% to 15%. Recall, our link delays range from as small as 5 ms at the low network levels to only about 30 ms at the top router level.

The observed speedup ranges between 1.2 and 1.6 on the dual Hyper-Threaded processor system. These speedups are very much in line with what one would expect, particularly given the memory size of the models at hand relative to the

N	Bandwidth	IS	EvRate	Effi c	% Remote	SU
4	500 Kb	1	441692	NA	NA	NA
4	500 Kb	2	535093	99.388	7.273	1.211
4	500 Kb	4	660693	97.411	14.308	1.495
4	1.5 Mb	1	386416	NA	NA	NA
4	1.5 Mb	2	440591	99.972	7.125	1.140
4	1.5 Mb	4	585270	99.408	14.195	1.516
4	45 Mb	1	402734	NA	NA	NA
4	45 Mb	2	440802	99.445	7.087	1.094
4	45 Mb	4	586010	99.508	14.312	1.612
8	500 Kb	1	210338	NA	NA	NA
8	500 Kb	2	270249	100	7.273	1.284
8	500 Kb	4	331451	99.793	10.746	1.575
8	1.5 Mb	1	177311	NA	NA	NA
8	1.5 Mb	2	237496	100	7.313	1.339
8	1.5 Mb	4	287240	99.993	10.823	1.619
8	45 Mb	1	176405	NA	NA	NA
8	45 Mb	2	221182	99.999	7.259	1.253
8	45 Mb	4	257677	99.996	10.758	1.460
16	500 Kb	1	128509	NA	NA	NA
16	500 Kb	2	172542	100	7.091	1.342
16	500 Kb	4	199282	99.987	10.600	1.550
16	1.5 Mb	1	100980	NA	NA	NA
16	1.5 Mb	2	137493	100	7.092	1.361
16	1.5 Mb	4	153454	99.998	10.626	1.519
16	45 Mb	1	99162	NA	NA	NA
16	45 Mb	2	117312	100	7.102	1.183
16	45 Mb	4	145628	99.999	10.648	1.468
32	500 Kb	1	80210	NA	NA	NA
32	500 Kb	2	108592	100	7.058	1.353
32	500 Kb	4	126284	100	10.586	1.57
32	1.5 Mb	1	75733	NA	NA	NA
32	1.5 Mb	2	90526	100	7.052	1.20

Table 1: Performance results measured in speedup (SU) for $N = 4, 8, 16, 32$ synthetic topology network for low (500 Kb), medium (1.5 Mb) and high (45 Mb) bandwidth scenarios on 1, 2 and 4 instruction streams (IS) on a dual Hyper-Threaded 2.8 GHz Pentium-4 Xeon. Efficiency is the net events processed (i.e., excludes rolled events) divided by the total number of events. Remote is the percentage of the total events processed sent between LPs mapped to different threads/instruction streams.

small level-2 cache. We note that we were unable to execute the $N = 32$, 45 Mb bandwidth case. This aspect and memory overheads are discussed in the paragraphs below.

The memory footprint of each model is shown as a function of nodes and bandwidth in Table 2. We report a steady increase in memory requirements and event-list size as bandwidth and the number of nodes in the network increase. The peak memory usage is almost 1.4 GB of RAM for the $N = 32$, 1.5 Mb bandwidth scenario. The amount of additional memory allocated for optimistic processing is 7,000 event buffers which is less than 1 MB. Thus, for 524,288 TCP connections,

N	Bandwidth	Max Ev-list Size	Mem Size
4	500 Kb	4,792	3 MB
4	1.5 Mb	5,376	3 MB
4	45 Mb	5,376	3 MB
8	500 Kb	45,759	11 MB
8	1.5 Mb	85,685	17 MB
8	45 Mb	86,016	17 MB
16	500 Kb	522,335	102 MB
16	1.5 Mb	1,217,929	202 MB
16	45 Mb	1,380,021	226 MB
32	500 Kb	5,273,847	1,132 MB
32	1.5 Mb	6,876,362	1,364 MB

Table 2: Memory requirements for $N = 4, 8, 16, 32$ synthetic topology network for low (500 Kb), medium (1.5 Mb) and high (45 Mb) bandwidth scenarios on 1, 2 and 4 instruction streams on a dual Hyper-Threaded 2.8 GHz Pentium-4 Xeon. Optimistic processing only required 7,000 more event buffers (140 bytes each) on average which is less 1 MB.

this model only consumes 2.6 KB per connection including event data. By comparison, Nicol [20] reports that Ns consumes 93 KB per connection, SSFNet (Java version) consumes 53 KB, JavaSim consumes 22 KB per connection and SSFNet (C++ version) consumes 18 KB for the "dumbbell" model which contains only two routers. Our topology is obviously different from the dumbbell model and thus these memory usage statistics are only qualitatively comparable.

Last, we find that there is an interplay in how the event population is effected by the network size, topology, bandwidth and buffer space. In examining the memory utilization results, we find that the maximum observed event population differs by only a moderate amount for 1.5 Mb versus 45 Mb case when $N = 16$ despite a rather significant change in network buffer capacity. However, we were unable to execute the 45 Mb scenario when $N = 32$ because it requires more than 17,000,000 events, which is the maximum we can allocate for that scenario without exceeding operating system limits (≈3 GB). This is because there are many more hosts at a high bandwidth, resulting in a higher utilization of the available buffer capacity. This case results in a 2.5 times increase in the amount of required memory. Consequently, model designers will have to perform some capacity analysis, since networks memory requirements may explode after passing some size, bandwidth or buffer capacity threshold, as happened here.

3.4 Hyper-Threaded vs. Multiprocessor System

In this series of experiments we compare a standard quad processor system to our dual, Hyper-Threaded system in order to better quantify our performance results relative to past processor technology. The network topology is the same as previous described with $N = 8$, thus there are 4,680 LPs in this simulation. We did however modify the TCP connections such that they are more locally centered. So, in total 87% of all TCP connections were within the same kernel process (KP).

We observe that the dual processor out performs the quad

SysConfi g	EvRate	% Effi c	% Remote	SU
1 IS, Hyper-Threaded	220098	NA	NA	NA
2 IS, Hyper-Threaded	313167	100	0.05	1.42
4 IS, Hyper-Threaded	375850	100	0.05	1.71
1 PE, Pentium-III	101333	NA	NA	NA
2 PE, Pentium-III	183778	100	0.05	1.81
4 PE, Pentium-III	324434	100	0.05	3.20

Table 3: Performance results measured in speedup (SU) for $N = 8$ synthetic topology network medium bandwidth on 1, 2 and 4 instruction streams (dual Hyper-Threaded 2.8 GHz Pentium-4 Xeon) vs. 1, 2 and 4 processors (quad, 500 MHz Pentium-III

processor system by 16%. The respective speedups relative to their own sequential performance are 3.2 for the quad processor and 1.7 for the dual Hyper-Threaded system.

Additionally, we observe 100% simulator efficiency for all parallel runs. We attribute this phenomenon to the low remote messages and large amount of work (event population) per unit of simulation time.

3.5 AT&T Topology

SysConfi g	EvRate	% Effi c	% Remote	SU
medium, 1 IS	138546	NA	NA	NA
medium, 2 IS	154989	99.947	52.030	1.12
medium, 4 IS	174400	99.005	78.205	1.25
large, 1 IS	127772	NA	NA	NA
large, 2 IS	143417	99.956	51.976	1.12
large, 4 IS	165197	99.697	78.008	1.29

Table 4: Performance results measured in speedup (SU) for AT&T network topology for medium (96,500 LPs) and large (266,160 LPs) on 1, 2 and 4 instruction streams (IS) on the dual Hyper-Threaded system.

The U.S. AT&T network topology contains 13,173 nodes (i.e., routers) and 38,164 links. What makes Internet topologies like the AT&T network both interesting and challenging from a modeling prospective is the spareness and power-law structure [28]. In the case of AT&T, there are less than 3 links on average. However, at the super core there is a high-degree connectivity. Typically, an Internet service provider's super core will be configured as a fully connected mesh. Consequently, backbone routers will have up to 67 connections to other routers, some of which are other backbone or super core routers and other links to region core routers. Once at the region core level, the number of links per router reduces and thus the connectivity between other region cores is spare. Most of the connectivity is dedicated to connecting local points of presence (PoPs).

In performing a breath-first-search of the AT&T topology, there are distinct eight levels. At the backbone, there are 414 routers. At each successive level yields the following router count : 4861, 5021, 1117, 118, 58, 6 and at the final level there are 5 nodes. There were a number of routers not directly reachable from within this network. Those routers are most likely transit routers going strictly between autonomous systems (AS). With the transit routers removed, our AT&T network scenario has 11670 routers. Link weights are derived based on the relative bandwidth of the link in comparison to other available links. In this configuration, routing is keep static, however we do have dynamic routing currently working on a light-weight OSPF model in which we plan to integrate with our TCP model in the very near future.

The bandwidth, delay, and buffer size for the AT&T topology is as follows: **Level 0 router:** 9.92 Gb/sec with a delay randomly selected between 10 ms to 30 ms, and 12.4 MB buffer; **Level 1 router:** 2.48 Gb/sec, a delay selected randomly between 10 ms to 30 ms, and 3 MB buffer; **Level 2 router:** 620 Mb/sec with a delay selected randomly between 10 ms to 30 ms, and 750 KB buffer; **Level 3 router:** 155 Mb per second with a delay of 5 ms, and 200 KB buffer; **Level 4 router:** 45 Mb per second, a delay of 5 ms, and 60 KB buffer; **Level 5 router:** 1.5 Mb/sec, a delay of 5 ms, and 20 KB buffer; **Level 6 router:** 1.5 Mb per second, a delay of 5 ms, and 20 KB buffer; **Level 7 router:** 500 Kb per second, a delay of 5 ms, and 5 KB buffer; **link to all hosts:** 70 Kb per second, a delay of 5 ms, and 5 KB buffer.

Hosts are connected in the network at PoP level routers. These routers only have one link to another higher-level router. In the medium size configuration there where 10 hosts per PoP level router which totaled 96,500 nodes (hosts plus routers). In the large configuration there where 30 hosts per PoP totaling 266,160 LPs. In each configuration, the half the host establish a TCP session to a *randomly selected* receiving host. **We observe this configuration is almost pathological for a parallel network simulation because the amount of remote network traffic will be much greater than is typical in practice**. The amount of remote message traffic is much greater than the synthetic network topology because of the networks sparse structure. Our goal is to demonstrate simulator efficiency under high-stress workloads for realistic topologies.

We observe over 99% efficiency for the 2 and 4 IS runs as shown in Table 4, yet there is a substantial reduction in the overall obtain speedup. Here, we report speedups for the 4 IS cases of 1.25 for the medium size network and 1.29 for the large. We attribute this reduction to enormous amount of remote messages sent between instruction streams/processors. The AT&T network topology for a round-robin mapping results 50 to even almost 80% of the all processed events being remotely schedule. We hypothesize that behavior on the part of the model reduce memory locality and results in much higher cache miss rates. Consequently, all instruction streams are spending more time stalled waiting for memory requests to be satisfied. However, we note that more investigation is required to full understand this behavior.

The memory requirements for the AT&T scenario were 269 MB for the medium size network and 328 MB for the large size network, yielding a per TCP connection overhead of 2.8 KB and 1.3 KP respectively. The reason for the reduction per connection in moving from medium to large configuration is because the amount of network buffer space which effects the peak event population did not change, yet the number of connections went up by almost a factor of 3.

4 Related Work

Much of the current research in parallel simulation for network models is largely based on conservative algorithms. For example, PDNS [27] is parallel/distributed network simulator that leverages HLA-like technology to create a federation of Ns [21] simulators. GloMoSim [17], TaskKit [31] and SSFNet [7], all use Critical Channel Traversing (CCT) [31] as the primary synchronization mechanism. DaSSF employs a hybrid technique called *Composite Synchronization* [19], where both the asynchronous CCT algorithm and a barrier synchronization are combined to avoid channel scanning limitations associated CCT while at the same time reducing the frequency a global barrier must by applied.

Recent optimistic simulation systems for network models include TeD [22], which is a process-oriented framework for constructing high-fidelity telecommunication system models. Premore and Nicol [24] implement a TCP model in TeD, however no performance results are given. USSF [26] is an optimistic simulation system that dramatically reduces model runtime state by LP aggregation, and swapping LPs out of core. Additionally, USSF proposes to execute simulation unsynchronized using their NOTIME approach. Based on the results here, a NOTIME synchronization could prove beneficial for large-scale TCP models. Unger et. al. simulate a large-scale ATM network using an optimistic approach [29]. They report speedups ranging from 2 to 7 on 16 processors and indicate that optimistic outperforms a conservative protocol on 5 of the 7 tested ATM network scenarios.

5 Conclusions and Future Work

In this paper we present a new scalable TCP model. With the use of optimistic parallel simulation techniques coupled with reverse computation, speedups of 1.7 for a Hyper-Threaded dual processor system and 3.2 for a quad processor system are reported. These speedups were achieved with an insignificant amount of additional memory for optimistic processing (i.e., about 1 megabyte in practice).

The parallel TCP model proved to be extremely efficient with very few rollbacks observed. Parallel simulator efficiency ranged between 97 to 100% (i.e., zero rollbacks). This suggests that the model could be executed *unsynchronized* with a negligible amount of error.

The model was implemented as lean as possible which allowed for the million node topology to be executed. We observed model memory requirements between 1.3 KB to 2.8 KB per TCP connection depending on the network configuration (size, topology, bandwidth and buffer capacity).

Last, the Hyper-Threaded system was able to provide a low cost-performance ratio. What is even more interesting is that these systems blur the lines in terms of sequential versus parallel processing. Here, to obtain higher rates of performance from a single processor, one has to resort to executing the model in parallel. As this technology matures to even high clock rates, we anticipate single processors having many more instruction streams, which will provide an even greater opportunity for parallel simulation tools and techniques.

In the future, we will be working on the implementation of a faster event-list management to cut down on priority queue overheads. Also the implementation of TCP functionality such as delayed acknowledgment, ticks for round trip time calculation, and Reno capabilities are in the works. The concept of creating a hierarchical address mapping scheme from a random network topology as well as a better LP to processor mapping scheme to reduce remote events has also been a topic of discussion.

Additionally, as more optimistic models are developed we are learning how they interoperate and how network researchers would like to utilize them. The outcome from this research will be modular software architecture that does not add either memory or computational overheads as compared with its direct implementation counterpart. The architecture should allow for the creation of different applications using the transport protocol level (i.e., TCP), such as Border Gateway Protocol for both inter and intra domain routing and web traffic. In the modular model there should be the ability to turn on and off different layers within the overall protocol stack as well as particular features, such as the need to have data represented in the message. This flexibility will enable the model to be tuned for optimum performance within the constraints placed on its expected operating environment and required level of accuracy.

6 Acknowledgments

This research is supported by an NSF CAREER Award CCR-0133488, the DARPA Network Modeling and Simulation program, contract #F30602-00-2-0537 and a AT&T University Relations Program Grant.

References

[1] R. Brown. Calendar Queues: A Fast *O(1)* Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM (CACM)*, volume 31, number 10, pages 1220–1227, October 1988.

[2] C. D. Carothers, K. Perumalla and R. M. Fujimoto. Efficient Parallel Simulation Using Reverse Computation *ACM Transactions on Modeling and Computer Simulation*, volume 9, number 3, pages 224–253, July 1999.

[3] C. D. Carothers, D. Bauer and S. Pearce. ROSS: A High-Performance, Low Memory, Modular Time Warp System, In *Proceedings of the 14th Workshop of Parallel on Distributed Simulation (PADS 2000)*, pages 53–60, May 2000.

[4] C. D. Carothers, D. Bauer and S. Pearce. ROSS: Rensselaer's Optimistic Simulation System User's Guide. Technical Report #02-12, Department of Computer Science, Rensselaer Polytechnic Institute, 2002, `http://www.cs.rpi.edu/tr/02-12.pdf`

[5] D. Chiu and R. Jain, Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks, Journal of Computer Networks and ISDN, Volume 17, Number 1, June 1989, pages 1–14.

[6] K. G. Coffman and A. M. Odlyzko Internet Growth: Is there a "Moore's Law" a Data Traffic? Preliminary Version, `http://www.research.att.com/~amo/doc/internet.moore.ps`

[7] J. Cowie, H. Liu, J. Liu, D. Nicol and A. Ogielski. Towards Realistic Million-Node Internet Simulations. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June, 1999, Las Vegas, Nevada.

[8] K. Fall and S. Floyd, Simulation-based Comparisons of Tahoe, Reno, and SACK TCP, Computer Communication Review, Volume 26, Number 3, July 1996, pages 5–21.

[9] R. M. Fujimoto. Parallel discrete-event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[10] F. Gomes, Optimizing incremental state-saving and restoration. Ph.D. thesis, Dept. of Computer Science, University of Calgary, 1996.

[11] D. Harrison, Edge-to-edge Control: A Congestion Control and Service Differentiation Architecture for the Internet, Ph.D. Dissertation, Computer Science Department, Rensselaer Polytechnic Institute, May 2002.

[12] V. Jacobson, Congestion avoidance and control, *Proceedings of the ACM SIGCOMM*, August 1988, pages 314–329.

[13] Intel. Pentium 4 and Xeon Processor Optimization Reference Manual, `http://developer.intel.com/design/pentium4/manuals/248966.htm`

[14] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[15] P. L'Ecuyer and T. H. Andres. "A Random Number Generator Based on the Combination of Four LCGs." *Mathematics and Computers in Simulation*, volume 44, pages 99–107, 1997.

[16] Jack L. Lo, Susan J. Eggers, Joel S. Emer, and Henry M. Levy, and Rebecca L. Stamm and Dean M. Tullsen. Converting Thread-Level Parallelism to Instruction Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, 15(3), pages 322–354, August 1997.

[17] R. A. Meyer, and R. L. Bagrodia Path Lookahead: a Data Flow View of PDES Models. In *In Proceedings of the* 13^{th} *Workshop on Parallel and Distributed Simulation (PADS '99)*, pages 12–19. May 1999.

[18] D. Nicol, and X. Liu. The Dark Side of Risk – What your mother never told you about Time Warp). In *Proceedings of the* 11^{th} *Workshop on Parallel and Distributed Simulation (PADS '97)*, pages 188–195, 1997.

[19] D. M. Nicol and J. Liu. Composite Synchronization in Parallel Discrete-Event Simulation, *IEEE Transactions on Parallel and Distributed Systems*, Volume 13, Number 5, May 2002.

[20] D. Nicol. Scalability of Network Simulators Revisited, In *Proceedings of the 2003 Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS '03)*, January, 2003.

[21] NS: Network Simulator – Home Page `http://www-mash.cs.berkeley.edu/ns/ns.html`

[22] K. Perumalla, A. Ogielski, and R. Fujimoto. TeD — A Language for Modeling Telecommunication Networks. In *Proceedings of ACM SIGMETRICS Performance Evaluation Review*, Vol. 25, No. 4, March 1998.

[23] A. Poplawski and D. M. Nicol. Nops: A Conservative Parallel Simulation Engine for TeD. *In Proceedings of the* 12^{th} *Workshop on Parallel and Distributed Simulation (PADS '98)*, volume 23, pages 180–187, May 1998.

[24] B. J. Premore and D. M. Nicol. Parallel Simulation of TCP/IP Using TeD, In *Proceedings of the 1997 Winter Simulation Conference*, pages 437-443, Atlanta, December 1997

[25] R. Ronngren and Rassul Ayani. A Comparative Study of Parallel and Sequential Priority Queue Algorithms *ACM Transactions on Modeling and Computer Simulation*, volume 7, number 2, pages 157–209, April 1997.

[26] D. M. Rao and P. A. Wilsey, "An Ultra-large Scale Simulation Framework", Journal of Parallel and Distributed Computing (in press)

[27] G. F. Riley, R. M. Fujimoto and M. H. Ammar. A Generic Framework for Parallelization of Network Simulations, In *Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 128–135, October 1999.

[28] N. Spring, R. Mahajan, and D. Wetherall Measuring ISP Topologies with Rocketfuel, In Proceedings of ACM SIGCOMM, August 2002

[29] B. Unger, Z. Xiao, J. Cleary, J-J Tsai and C. Williamson. Parallel Shared-Memory Simulator Performance for Large ATM Networks, *ACM Transactions on Modeling and Computer Simulation*, volume 10, number 4, pages 358–391, October 2000.

[30] G. Yaun, C. D Carothers, S. Adali and D. Spooner. "Optimistic Parallel Simulation of a Large-Scale View Storage System", In *Proceedings of 2001 Winter Simulation Conference (WSC'01)*, pages 1363–1371, December 2001.

[31] Z. Xiao, B. Unger, R. Simmonds and J. Cleary. Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation", In *Proceeding of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 20–28, 1999.