Exploiting Event-Level Parallelism for Parallel Network Simulation on Multicore Systems

Shie-Yuan Wang, *Senior Member*, *IEEE*, Chih-Che Lin, *Member*, *IEEE*, Yan-Shiun Tzeng, Wen-Gao Huang, and Tin-Wei Ho

Abstract—This paper proposes a parallel simulation methodology to speed up network simulations on modern multicore systems. In this paper, we present the design and implementation of this approach and the performance speedups achieved under various network conditions. This methodology provides two unique and important advantages: 1) one can readily enjoy performance speedups without using an unfamiliar simulation language/library to rewrite his protocol module code for parallel simulations, and 2) one can conduct parallel simulations in the same way as when he conducts sequential simulations. We implemented this methodology and evaluated its performance speedups on the popular ns-2 network simulator. Our results show that this methodology is feasible and can provide satisfactory performance speedups under high event load conditions on wired networks.

Index Terms—Network simulation, parallel simulation, multicore system.

1 Introduction

Multicore computers have been ubiquitous in the current market. On such a computer, efficiently using the computing power of all cores (CPUs) to finish a task becomes important and challenging. (Note: in the rest of this paper, we use the conventional term "CPU" and "core" interchangeably when no ambiguity results.) As pointed out in [5], it is difficult for an application (including a network simulator) to automatically gain performance speedups on multicore systems because the application process can only be run on a single CPU at any given time. To gain performance speedups, an application program first needs to be made "multithreaded" so that its threads can be run on multiple CPUs simultaneously. However, turning an application program to be "multithreaded" is not trivial and does not necessarily achieve good performance speedups [5].

In the context of network simulations, parallel and distributed simulations can be categorized into two methodologies [7]—the conservative methodology and the optimistic methodology. Using either methodology, a simulation user needs to partition a simulated network into several portions and modify simulation code to perform simulation clock synchronization among these portions to avoid causality errors. The conservative methodology, although simpler to be implemented into the simulation code, usually results in very low performance speedups under tiny lookahead values [7]. On the other hand, the optimistic methodology, although potentially capable of achieving higher performance speedups than the conservative methodology, is very complicated and requires substantial modifications to the simulation code.

So far, parallel network simulations have not made a substantial impact on the network community. This may be attributed to the following reasons. First, parallelizing existing network simulators to achieve good performance speedups is difficult. In [8], Jones and Das attempted to parallelize the widely used open source network simulator ns-2, but could only support simple point-to-point links with static routing and UDP traffic. The supports for TCP connections, dynamic routing, and shared medium networks were not provided due to high complexities. In [16], Wu et al. attempted to parallelize a widely used commercial network simulator, called OPNET Modeler, but could only support simple UDP and IP protocols. The supports for TCP connections and other protocols were not provided due to extensive uses of global states, zero lookahead interactions, and pointer data structures in OPNET Modeler.

The second reason is that simulation users need to learn parallel simulation concepts and approaches to conduct parallel network simulations. Using parallel network simulation techniques to achieve good performance speedups is difficult for users without such trainings and knowledge. In [13] and [14], Riley et al. used a federated approach to interconnect multiple ns-2 simulation engines to simulate a network. In these proposals, each ns-2 simulation engine simulates a part of the whole network. The developed simulation platform is called Parallel/Distributed Network Simulator (PDNS). PDNS reduces the required modification to ns-2 simulation code. However, to use PDNS, a user needs to learn the modified ns-2 scripting language provided by PDNS and the concepts of parallel simulation, so that he can properly partition a simulated network into several portions and map them onto available CPUs for better performance.

Partitioning a large-scale network with a large number of nodes is a tedious task for the user. Worse yet, finding a load-balancing partition to achieve good performance

- *The authors are with the Department of Computer Science, National Chiao Tung University, Engineering Building No. 3, 1001 University Road, Hsinchu 300, Taiwan, Republic of China.*
  E-mail: {shieyuan, linjc, ytstseng, wghuang, anton}@cs.nctu.edu.tw.

Manuscript received 27 Sept. 2010; revised 5 June 2011; accepted 8 June 2011; published online 25 July 2011.
Recommended for acceptance by H. Jiang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-09-0563.
Digital Object Identifier no. 10.1109/TPDS.2011.215.

speedups is difficult and would be tricky, because dynamic traffic changes should be taken into consideration. For example, the amount of traffic that are generated and processed in each network portion should be balanced during simulation. However, such information would be difficult to obtain and estimate when a user plans the network partition.

These observations show a need of network simulation users: a methodology to enjoy performance speedups without changing the way they used to conduct a simulation (i.e., the way that they used to operate a sequential network simulator). Without the need of learning extra knowledge for parallel simulation, it can be expected that simulation users will be more willing to use parallel simulation techniques to speedup their simulations.

To this end, in this paper, we propose a parallel simulation methodology, called "Event-Level Parallelism (ELP)." The idea of ELP is analogous to that of the "Instruction-Level Parallelism (ILP)" approach employed in the computer architecture and compiler research. The ILP approach exploits instruction-level parallelism to achieve performance speedups over multiple CPUs without finding the parallelism inherent and latent in an application. In contrast to conventional parallel simulation approaches, which need to partition a network into several portions and perform simulation clock synchronization among them, the ELP approach does not require a user to partition a network. A user can run up his familiar network simulator, which has been slightly modified for using the ELP approach, as usual. During execution, the ELP-enabled parallel network simulator will automatically search for "safe events" [7] and dispatch them to its internal Worker Threads (WTs) for parallel execution over multiple CPUs.

In this paper, we show that the ELP approach generates correct simulation results and achieves satisfactory performance speedups under high event load conditions on wired networks. With its easy-to-use property and good performance speedups on wired networks, simulation users will be more willing to use the ELP approach for parallel simulations on modern multicore systems.

The remainder of this paper is organized as follows. In Section 2, we describe related work in the field of parallel network simulations. In Section 3, we present the design and architecture of the ELP approach. In Section 4, we explain how to examine whether an event can affect another event in the ELP approach. In Section 5, we present the performance speedup results of the ELP approach under various network conditions using the popular ns-2 network simulator. The comparison between the results derived from the analytical model and those obtained from real-life experiments are also presented. Finally, we conclude the paper in Section 6.

2 Related Work

The parallelization of network simulations has been studied for a long time. The work reported in [8], [16], [13], and [14], and their differences are already discussed in Section 1. The PDNS proposed in [13] and [14] has been obsolete and not available for being evaluated on the state-of-the-art operating systems. In [18], Zeng et al. proposed GloMoSim, a library for parallel simulation of large-scale wireless networks based on the conservative approach. The library was developed using a C-based parallel simulation language called PARSEC [1]. In [4], Cowie et al. presented the Scalable Simulation Framework (SSF) and its parallel version DaSSF written in C++ or Java. SSF/DaSSF defines five core classes and provides the SSF Application Programming Interface for users to develop their protocol modules. In [2], Bhatt et al. presented Ted/GTW. Ted is a small language expressing a natural modeling framework that is transparently mapped onto a parallel simulation kernel, the Georgia Tech Time Warp (GTW) [6].

These previous approaches are scalable when the lookahead value is large. However, when the lookahead value is tiny, the achieved performance speedups can be very poor, making the speed of a parallel network simulation much lower than that of its corresponding sequential network simulation. In addition, some of them require a simulation user to develop simulation codes "from scratch" so that the simulation codes can be tailored for parallel execution. A simulation user needs to learn the language or library to develop his protocol modules, partition the simulated network into Logical Processes (LPs), and wisely map LPs to CPUs to achieve good performance speedups. Using these approaches, one cannot use existing popular network simulators such as ns-2 and OPNET Modeler. Regarding the issue of the poor parallelism among LPs when the lookahead values are tiny, Lin et al. proposed an algorithm to explore the parallelism inside an LP in [9].

In [11], Park and Fujimoto presented the Aurora approach for web-based distributed parallel simulation. Aurora is designed for high throughput computing where the goal is to harness available computing cycles from a large number of machines. Its primary goal is not to achieve high performance speedups because it is not suitable for every type of simulation. Due to the high cost of communication among machines, this approach is best suited for applications where a significant amount of computation can be handed to a client machine for execution. In Aurora, the parallel simulation program is composed of a collection of LPs and adopts a master/worker architecture. In Aurora, LPs and their associated data structures and states are clustered into "work units" and the master dispatches these work units to available client machines (workers) for parallel execution. Similar to our proposed ELP approach, Aurora also adopts a centralized conservative synchronization mechanism to determine what events in an LP (i.e., work unit) are safe to be processed on a client machine.

The implementation of Aurora differs from that of the ELP approach. However, if the Aurora approach is implemented on a multicore machine rather than on a distributed multimachine environment, its operations are semantically similar to those of the ELP approach. For example, ELP that differentiates packet arrival events from local computation events is equivalent to Aurora that defines individual network nodes as LPs and schedules them according to the earliest timestamp of events within each LP. That way, events (messages) between LPs correspond to packet arrival events, and events internal to each LP to local computation events. Although our proposed ELP approach can be semantically similar to Aurora, in [11] Park and Fujimoto did not address how to turn the multimachine version of Aurora into an efficient multicore version. Our proposed ELP approach serves an example implementation that exploits multithread architecture to efficiently realize a parallel simulation engine combining the centralized event list design and multiple LPs.

In the literature, several previous works also use a central event list for parallel execution of safe events in [3] and [12]. However, the degrees of parallel event execution in these proposals are very low as compared with our ELP approach. In [3], only the events with the same timestamp can be executed in parallel. As a result, the achieved performance speedups are very low. In [12], a window-based scheme, called the Moving Time Window (MTW), is used to find safe events that can be executed in parallel. In MTW, the event execution units are allowed to simultaneously execute events with timestamps falling into a calculated time window $[T_{ref}, T_{ref} + dL]$, where T_{ref} denotes the timestamp of the earliest unprocessed event and dL denotes a known time interval during which all events in the system are guaranteed to be independent of each other. For example, it is known that an event representing an outgoing packet sent by node i to node j at time T will not affect other events on node j until the transmission time of this packet (denoted as Tx) plus the signal propagation delay over the link (i, j) (denoted as D_{ij}) has elapsed (i.e., after $T + Tx + D_{ij}$). To find the dL of a simulated network, the minimum of all $Tx + D_{ij}$, where i and j represent any possible index of links, is used as the dL in MTW. Because the found dL is mostly small (which is the minimum of all possible $Tx + D_{ij}$), the number of safe events found in MTW is also small, resulting in low performance speedups.

In contrast, our ELP approach uses the path lookahead values of paths among nodes to check whether two events are safe for execution to each other. Note that, although the path lookahead notion has been present in the literature, there has been no work that exploits it to find parallelism at the event level with a central event list. Our ELP approach is the first work that combines the central event list architecture with the path lookahead notion to find the event-level parallelism. Our simulation results show that such a combination can achieve satisfactory performance speedups in wired network simulations without the need to modify existing simulation code or partition the simulated network topology.

Xiao et al. [17] and Simmonds et al. [15] proposed the Critical Channel Traversing (CCT) algorithm for conservative distributed simulation systems with low granularity on shared-memory multiprocessor computers. The CCT algorithm still needs to partition a simulated topology into several partitions. Events generated in a partition can be executed by several LPs, which are dynamically mapped onto available CPUs. In the CCT algorithm, a channel is defined as a unidirectional connection between two LPs. By properly arranging the execution priority of events on each channel, the cache locality for events to be processed can be increased, leading to more cache hits in each CPU and thus resulting in a better simulation performance.

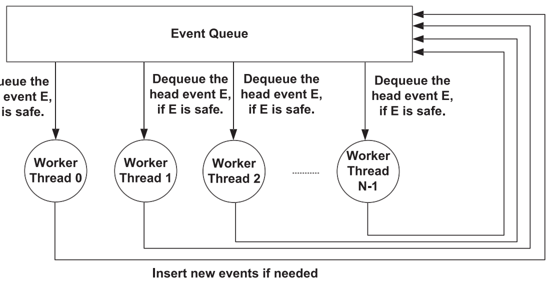
Event Queue
Dequeue the head event E, if E is safe.
Dequeue the head event E, if E is safe.
Dequeue the head event E, if E is safe.
Dequeue the head event E, if E is safe.
Worker Thread 0
Worker Thread 1
Worker Thread 2
Worker Thread N-1
Insert new events if needed

Fig. 1. The architecture of a parallel network simulator using the ELP approach.

Since the CCT algorithm still needs to partition a simulated network, the designs proposed in [17] and [15] are different from the ELP approach.

3 The ELP Approach

Fig. 1 shows the thread architecture of a parallel network simulator using the ELP approach. In the ELP thread architecture, multiple WTs are created in the simulation program and they run in parallel to collaboratively complete a simulation. When a simulation is started, the main (first) WT first initializes a path lookahead table covering all simulated nodes. This table is used for determining whether an event is safe to be executed and will be explained in Section 4. It then creates other $(N-1)$ WTs, where N is the total number of WTs used in a simulation. The value of N is specified by the user and is usually set to the number of cores available on the system. During simulation, each created WT executes Algorithm 1, which is explained below.

Algorithm 1. The Operation of WT i

```
 1: $Q_{run}$ := { e | e is an event to be simulated and sorted by
    its timestamp}
 2: event($i$) = NULL
 3: while $Q_{run} \neq \emptyset$ do
 4:    Contend for the write lock of $Q_{run}$.
       (If WT $i$ does not win the write lock of $Q_{run}$, it will
       sleep until some WT releases the write lock of $Q_{run}$.
       WT $i$ should repeat this lock contending process
       until it wins the write lock of $Q_{run}$.)
 5:    $e_{head} \leftarrow$ get_head($Q_{run}$)
 6:    se_flag := true
 7:    for $j := 0$ to $|WT| - 1$ do
 8:       if $j \neq i$ and event(j) $\neq$ NULL then
 9:          if is_safe(event(j), $e_{head}$) = false then
10:             se_flag = false
11:             break
12:          end if
13:       end if
14:    end for
15:    if se_flag = true then
16:       $e_{head} \leftarrow$ dequeue ($Q_{run}$)
17:       event($i$) = $e_{head}$
18:       Release the write lock of $Q_{run}$
19:       Execute event($i$) (If WT $i$ needs to generate
          another event and insert it into $Q_{run}$, WT $i$ should
          contend for the write lock of $Q_{run}$ again.)
```

```
20:     else
21:         $\mathrm{event}(i) = \mathrm{NULL}$
22:         Release the write lock of $\mathrm{Q}_{\mathrm{run}}$
23:     end if
24: end while
```

Initially, every WT sets its local event as NULL and contends for the access to the global event queue (denoted as $\mathrm{Q}_{\mathrm{run}}$). Only after winning the access to $\mathrm{Q}_{\mathrm{run}}$, can a WT proceed to do its task. WTs that fail to win the access to $\mathrm{Q}_{\mathrm{run}}$ should put themselves into the sleep state until the winning WT releases the lock for $\mathrm{Q}_{\mathrm{run}}$. At that time, they will be wakened up and contend for the access to $\mathrm{Q}_{\mathrm{run}}$ again.

An event currently possessed (i.e., currently executed or to be executed) by a WT j is denoted as event(j). The is_safe() function checks the safe event relationships between e_{head} and event(j), for all possible j in the system. If e_{head} and the events possessed by other WTs can be executed in parallel without generating causality errors, the is_safe() function returns *true* as its output. Otherwise, it returns *false*. Its details are explained in Section 4.

Note that, in the is_safe() function, WT i need not acquire the read lock to event(j). This is because in our ELP architecture a WT j erases its own event(j) only when it wins the access to $\mathrm{Q}_{\mathrm{run}}$ again. Because the access to $\mathrm{Q}_{\mathrm{run}}$ is exclusive, when WT i is checking the safety relationship among events, no other WTs will change the ELP-related information of their (possessed) events.

Using this design, the local event of WT j read by WT i is, therefore, either the one that WT j is currently executed or the one that WT j has finished executing. However, in either case, the correctness of the simulation results generated by the ELP approach is not affected. This is because the ELP approach employs the safe event rule set (explained in Section 4) to control concurrency of event execution. Thus, the events that have been processed or is being processed by WT j are always safe to the event that is going to be processed by WT i.

Using this peer thread architecture design, at any point of time it is guaranteed that at least one WT will be able to find a safe event to execute. This is because, like in a sequential simulation, the event with the smallest timestamp in the whole simulated network (i.e., in $\mathrm{Q}_{\mathrm{run}}$) must be a safe event. This property enables the ELP approach to quickly find a safe event to execute even under tiny lookahead values.

We implemented our ELP approach in the ns-2 network simulator (version 2.31). The modifications to ns-2 can be divided into four parts. The first part is adding two fields into the event structure to store the IDs of the source node and the destination node for an event. These two fields are used to compute the path lookahead values of an event. The second part is expanding the tcl scripts provided by ns-2 such that, when a protocol module is initialized, its node ID will be properly assigned. The third part is to provide a facility to automatically fill in the source and destination node IDs of an event when it is inserted into the event list (i.e., $\mathrm{Q}_{\mathrm{run}}$). This can be accomplished by expanding the APIs used by a protocol module to insert an event into $\mathrm{Q}_{\mathrm{run}}$.

The final part is the modification to the event scheduler (in the common/scheduler.cc file in its package). The modified *Scheduler::run()* function, which is the first function invoked by ns-2 when it starts to run a simulation, first initializes 1) the path lookahead table for all simulated nodes, 2) the global data structures for storing the information of each WT, and 3) the $mutex$ lock of $\mathrm{Q}_{\mathrm{run}}$. Creating the path lookahead table requires the link delay information, which can be obtained from the simulation case description file (i.e., the tcl file used by ns-2).

This run() function then creates $(N-1)$ WTs, where N is the number of specified WTs. The main thread and other WTs then, respectively, run a while loop that implements Algorithm 1. We use the heap scheduler, which has been provided by ns-2, to implement the function of $\mathrm{Q}_{\mathrm{run}}$. Our proposed ELP approach hides the modifications to the network simulator from users and protocol designers. Thus, they can use the ELP-enabled network simulator in the same way as using a sequential network simulator and benefit from the speedup brought by the ELP approach without extra efforts.

Note that, the ELP approach is only applicable to those event-driven network simulators where the states of each simulated node's protocols are separately maintained. For example, most protocols of ns-2 do not use global variables to maintain their operations. However, for those network simulators without this property, the ELP approach may not generate the same simulation results as the sequential approach does. In this case, the ELP approach is applicable only when the order of updating shared global variables does not affect the observed simulation results.

Another limitation of the ELP approach is that, because all WTs are run on the same machine, the ELP approach can generate good speedup performance only when the amount of memory consumed by the simulation program does not exceed the amount of the main memory installed on the machine. Fortunately, as the semiconductor technology advances, the high-end work station can be installed with up to 32-GB main memory nowadays, greatly increasing the scalability of the ELP approach. The ELP approach is not designed for running huge-scale simulations with the required amount of memory exceeding that can be afforded by a single machine. In such cases, the existing parallel simulation techniques using multiple machines are more preferred.

4 THE "SAFE" EVENT RELATIONSHIP

In this section, we present the safe event rules used in the is_safe() function to check whether an event is safe to be executed. The safety of executing events in parallel is based on the notion of "lookahead." Its detailed explanations can be found in [7] and is briefly explained here.

Consider that a sequential network simulator has advanced its simulation clock to simulation time T, which is the timestamp of the next unprocessed (i.e., the first) event e_{next} in the event queue, and is going to execute that event. Without loss of generality, assume that there is a constraint that a new event must be scheduled at least L units of simulation time into the future, then it can be guaranteed that all new events scheduled during the execution of e_{next} must have a timestamp larger than or equal to $(T + L)$.

With this property, e_{next} and any other event in the event queue with a timestamp less than $(T + L)$ can be safely executed in parallel without generating causality errors. In this example, a pair of e_{next} and an event with a timestamp less than $(T + L)$ are called "safe events" and L is the lookahead value for e_{next}. In the context of network simulation, a lookahead value L for an event e that represents a packet transmitted over a link l can be easily derived as follows:

$$L = D + TxTime, \tag{1}$$

where D is the signal propagation delay over the link l and $TxTime$ is the transmission time of the packet represented by e over link l. D is a fixed value and $TxTime$ is the packet length divided by the bandwidth of link l. When the packet represented by e is transmitted over link l, it requires the time amount L to arrive at the other end of l. (L includes the time required by the network interface on the remote node to receive all bits carried by this packet.)

Consider the case that the current simulation time is t_1, an event e_1 is being executed at the source node of a link, and during the execution of e_1 a packet is generated by e_1 and transmitted to the destination node of the link. Assume that there is an event e_2 in the event queue with a timestamp of t_2 larger than t_1 and it is to be executed on the destination node of the link. If $t_1 + D + TxTime > t_2$, e_1 cannot affect e_2, meaning that e_1 and e_2 are safe events and can be executed in parallel. On the other hand, if $t_1 + D + TxTime \leq t_2$, e_1 can affect e_2 because, before e_2 is executed, a packet generated by e_1 may have arrived at the node that will execute e_2 and changed the node's internal states. In this condition, e_1 and e_2 should be executed sequentially according to their timestamps to avoid causality errors.

In the ELP approach, events are divided into two types: 1) a packet arrival event and 2) a local computation event. A packet arrival event is used to simulate a packet arriving at a simulated network node. When such an event is executed at its timestamp, the internal state (e.g., the buffer occupancy level) of the receiving node may be changed to reflect the fact that a packet has been received. Suppose that a packet arrival event represents a packet sent from node i to node j, then in the ELP approach the source and destination node ID fields of its event structure are set to i and j, respectively.

On the other hand, a local computation event is used to simulate the local computation behavior of a simulated network node (e.g., the expiration of a local timer that triggers the execution of its associated function). Such an event only modifies the internal state of the node where the event is executed and does not transmit a packet to another node. For a local computation event to be executed on node i, both the source and destination node ID fields of its event structure are set to i in the ELP approach.

Note that in a simulated node's protocol stack, a packet arrival event for a specific destination node can only be scheduled at the bottom physical layer of the source node, which models the operations of the network interface at the source node of the link. Consider the triggering of a "hello message" timer, which is commonly used in a routing protocol to send out a HELLO packet to inform other nodes of the aliveness of this sending node. In this example, although the upper-layer routing protocol schedules a packet *transmission* event (not an arrival event), before this event reaches the bottom physical layer, when it passes along all protocol modules in the protocol stack, it is still considered a local computation event because the execution of this event only affects the internal state of the local node.

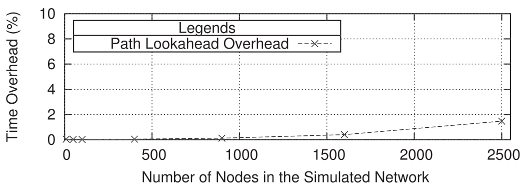

Fig. 2. The time overhead used for building the path lookahead table.

A more generic discussion about the safe event rule set used by the ELP approach is presented in Section 1 of the provided supplementary data, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.215.

5 Performance Evaluation

We evaluate the performance of the ELP approach using the widely used ns-2 network simulator [10], which is open source and widely used in the network research community. We modified the event scheduler of ns-2 to make it multithreaded and ELP-capable. In the following, we denote the N-thread ELP version of the ns-2 network simulator as "N-WT" for brevity. For example, 1-WT ns-2 denotes the modified ELP-capable ns-2 using only one WT. The simulation results generated by 1-WT ns-2 are the same as those generated by the original single-thread ns-2.

Before comparing their simulation speedups, we first validated the correctness of simulation results generated by the ELP approach. We used many different simulation cases to do the validations. For each of these cases, we compared the packet log files generated by 1-WT ns-2 and 8-WT ns-2. The packet log file records the timestamps of the transmission and reception events of every packet generated during a simulation. Events occurring on the same time T are sorted in the nondecreasing order based on the source node IDs. After carefully checking, we found that the packet log files generated by 1-WT ns-2 and 8-WT are exactly the same. This evidences that the ELP approach is feasible and valid.

Although the building of the path lookahead table in the ELP approach may require some time, it is insignificant to the total simulation time under most conditions. For example, when the simulated network load is heavy, the number of simulated network nodes is small, or the total simulated time is large, the overhead of building the path lookahead table is insignificant. We plotted the proportion of the total simulation time used by 1-WT ns-2 for building the path lookahead table in Fig. 2, where 1-WT ns-2 is used to simulate $N \times N$ grid networks and the simulated time is 180 seconds only. One can see that such time overheads are not significant. This means that the performance of 1-WT ns-2 is very close to that of the original ns-2 in most

TABLE 1
The Used Parameter Settings

Parameter Name	Value
ns-2 Version	2.31
Machine Model	Tyan GT20-B7002
CPU Model	Intel Xeon E5520 Quad Core 2.27GHZ X2
Main Memory	4 GBytes
Simulated Grid Network Topologies	3x3, 4x4, 5x5, 6x6, 7x7, 8x8, 9x9, 10x10, 20x20, 30x30, 40x40, 50x50*
Number of WTs	1, 2, 3, 4, 5, 6, 7, 8
Simulated Time (s)	180
Traffic Type	CBR UDP
Interface Output Queue Length (pkts)	50
Event Computation Empty Loop Count (k)	0, 1, 10, 20, 30, 50, 60, 70, 80, 90, 100, (3 on sending nodes, 60 on receiving nodes)*
Link Bandwidth (Mbps)	1, 10*, 100
Link Delay (ms)	1, 10*, 50, 100, 200
Packet Generation Interval (ms)	0.05, 0.5*, 5, 50, 500, 5000
Payload Length (Byte)	100, 500, 1000, 1400*

conditions. Thus, for simplicity the performance of 1-WT ns-2 is used as the baseline performance in this study.

The parameter settings used in our simulations are listed in Table 1. For each parameter, we varied its value to study its effects on the simulation performance. The default value used for a parameter when it is not explicitly specified is indicated by a star sign (*). Two performance metrics are used to study the performance of the ELP approach: 1) speedup and 2) P_{sa}. The speedup is defined as the time required by 1-WT ns-2 network simulator to complete the simulation case divided by that required by an N-WT ELP-capable ns-2 network simulator to complete the same simulation case. The formal definition of speedup can be found in Section 2 of the supplementary data, which can be found on the Computer Society Digital Library. The P_{sa} is defined as the probability for a WT to find a safe event to execute.

To obtain P_{sa} results, during simulation, for each WT we recorded how many times it entered Q_{run} trying to find a safe event and how many times it succeeded in finding a safe event. The P_{sa} value of a WT is obtained by dividing the latter number by the former number and the P_{sa} result of a simulation case is obtained by averaging the P_{sa} values of all WTs in the case. A larger P_{sa} value indicates that a WT has a higher probability to find a safe event to execute and thus the event-level parallelism can be easily exploited. In contrast, a smaller P_{sa} value indicates that a WT has a lower probability to find a safe event to execute and in this condition the event-level parallelism is hard to be exploited.

We created twelve grid wired network topologies, each containing $9 (3 \times 3), 16 (4 \times 4), \ldots, 2{,}500 (50 \times 50)$ nodes, respectively. In these grid networks, each node is connected to its nearest neighboring nodes. An $N \times N$ simulated grid network has N rows, each having N nodes. During simulation, a Constant-Bit-Rate (CBR) UDP flow is created to generate traffic on each row. The source node and destination node of a flow are, respectively, set to the leftmost node and the rightmost node on the row where the flow resides. Each point plotted in the presented figures is the average across 10 simulation runs using different random number seeds.

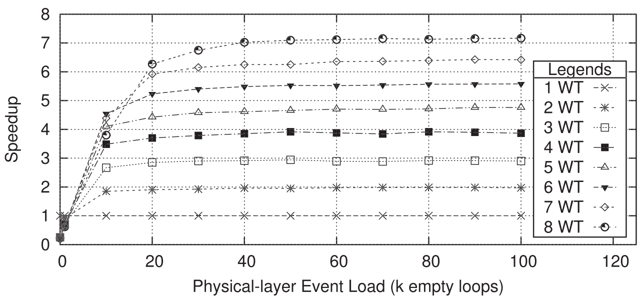

Fig. 3. UDP speedup over different event computation loads.

Fig. 3 shows the speedup results of the ELP approach over different event computation loads in the 10×10 simulated grid network with the packet transmission interval and the packet size set to 50 microseconds and 1,000 bytes, respectively. There are several findings about Fig. 3. First, when the physical-layer event computation load is heavy enough, the simulation speedup achieved by the ELP approach can approach N. For example, a speedup of 7.21 can be achieved when 8 WTs are set up. Several reasons can explain why the ELP approach cannot achieve a speedup of 8 on a 8-CPU system. In the 1-WT ns-2 code, some parts of code are not event-related and thus cannot be executed in parallel using the ELP approach. One example is the code related to enqueuing/dequeuing a packet into/from the event queue. These operations must be executed sequentially without intervention from other operations to ensure the consistency of the event queue. As a result, it is impossible to always achieve a speedup of N on a N-CPU system. In addition to the above reason, the locking and unlocking operations to shared global variable during event execution (e.g., the event ID acquisition in ns-2) account for other overheads that further decrease the speedup. The effects of locking/unlocking operations are discussed in Section 3.2 of the provided supplementary data, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.215.

The second finding is that, when the physical-layer event computation load goes down to less than 2K integer operations, the speedups of multi-WT ns-2 simulations become less than 1, meaning that in this situation their simulations are slower than the corresponding 1-WT ns-2 simulations. This phenomenon is expected and explained here. In the ELP approach, after finishing executing an event, each WT will try to find another safe event to execute. The process of finding a safe event includes:

1. locking the event queue;
2. comparing the source and destination node IDs of the head event in the event queue with the ones currently possessed by other WTs;
3. comparing the lookahead values of these events to determine whether they are safe events (if needed); and
4. unlocking the event queue.

These overheads,[1] although not much, occur each time when a WT tries to find a safe event. Therefore, when the event

1. From our measurements, the pthread library on Linux requires tens of nanoseconds to resolve the contention of a lock on a 1.6 GHZ CPU.

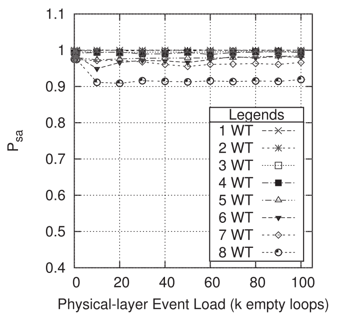

Fig. 4. UDP P_{sa} results over event computation loads.

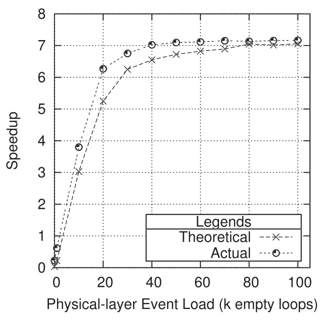

Fig. 5. Theoretical speedups and actual speedups.

computation load is tiny, these undesired time overheads become significant. The speedup results shown in Fig. 3 suggest that the ELP approach is more suitable for accurate heavy-load network simulations (e.g., the simulations that perform time-consuming physical-layer processings).

Note that, due to the peer thread architecture used by the ELP approach, when the event computation load is not heavy enough, it is easy to vary the number of activated WTs to achieve a simulation performance not worse than what 1-WT ns-2 can achieve. One approach is to first compare the simulation speedup achieved by the ELP approach using multiple WTs against that achieved by 1-WT ns-2 on a simulation case. If the speedup is less than 1, then one can simulate all remaining cases (e.g., with different random number seeds) using the ELP approach with only one WT activated. In the ELP approach, if it is invoked with only one WT activated, the four operations (described above) associated with an event execution will be automatically skipped during simulation as there is no need to do so. In such a situation, the ELP approach with only one WT activated can perform the same as 1-WT ns-2. Controlling the number of WTs to be activated in the ELP approach is simple. It can be specified as a command-line argument to the ELP simulation engine or let the ELP simulation engine dynamically measure the performance speedup and accordingly adjust the number of activated WTs during simulation. This advanced topic is out of the scope of this paper and is left as our future work.

Fig. 4 shows the relationships between P_{sa} and the event computation load. This phenomenon can be explained from two aspects. First, when the number of activated WTs increases, safe events that can be processed in parallel are consumed more quickly. As a result, the probability for a WT to find a safe event to execute is reduced. Second, when the event computation load increases from 0 to 2K, P_{sa} of 8-WT ns-2 slightly decreases. This is because, when the event computation load increases from a very light load to a moderate load, the WTs executing very-light-load events will often need to wait for the safe events generated by the WTs that are executing heavier load events. Therefore, P_{sa} of 8-WT ns-2 slightly decreases when the event computation load increases from 0 to 2K. However, when the event computation load further increases from 2K, each WT will now spend most of its time on executing heavy-load events. In such a condition, the probability of such waitings becomes stable, causing the P_{sa} value of 8-WT ns-2 to become stable as well.

To clearly formulate the performance of the ELP approach, we have built an analytical model to analyze its performance. Due to the space limitation, we will only briefly describe the final form of the model, leaving the derivation steps in Section 2 of the provided supplementary data, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.215.

Denote S_{a} as the set of all events that are generated and processed by all WTs during a simulation run. Without loss of generality, suppose that events in S_{a} have n different processing loads. Thus, S_{a} can be defined as

$$S_{\text{a}} = \cup_{i=1}^{n} S_i, \text{ for any } i, j, \quad S_i \cap S_j = \emptyset, \tag{2}$$

where each S_i is the set of events associated with a specific processing time T_i. Following this, the speedup of the ELP approach for a simulation case can be defined as follows:

$$\text{Speedup} = \left\{ \sum_{i=1}^{n} (|S_i| * T_i) \right\} \Bigg/ \left\{ \sum_{i=1}^{n} (|S_i| * T_i)/(N * P_{\text{sa}}) + T_{\text{o}} \right\}, \tag{3}$$

where N denotes the number of WTs used in simulation, P_{sa} denotes the probability that a WT finds a safe event, and T_{o} denotes the time amount taking into account all overheads incurred by the ELP approach.

We plotted the theoretical speedup of the 8-WT ns-2 simulator derived from our analytical model and the actual speedup in Fig. 5. The values of the parameters for the analytical model are obtained from real measurements as follows: the P_{sa} results have been plotted in Fig. 4; the event execution time results are measured and plotted in Fig. 6; and the time required for a WT to find safe events is measured and plotted in Fig. 7. The total number of

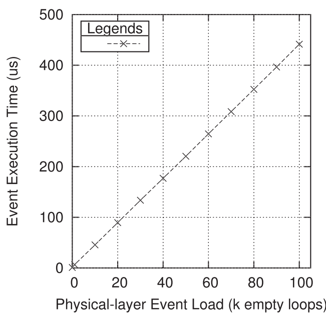

Fig. 6. Event execution time over event computation loads.

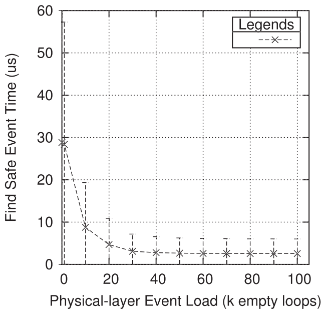

Fig. 7. Time for finding a safe event over event computation loads.

executed events in a simulation run in this series of experiments is 21,196,621. Fig. 5 shows that the theoretical speedup results derived from our analytical model match the actual speedup results quite well. However, there is some mismatch between these two results. This is because to obtain the event execution time results and the time for a WT to find safe events, logging the timestamps before/after executing several function calls is needed. Logging timestamps in a real system requires switches between the user space and the kernel space. This inevitably causes the switching overhead to be included in the logged time results. Therefore, the logged "find safe event" times are a bit higher than their actual values, which makes the theoretical speedup a bit lower than the actual speedup.

To further study the performance of our proposed ELP approach, we conducted a series of simulation experiments to study its performance under different network conditions. These extensive experiment results are presented in Section 3 of the provided supplementary data, which can be found on the Computer Society Digital Library.

6 Conclusions

In this paper, we propose a parallel simulation approach that exploits the event-level parallelism to enable easy-to-use and efficient parallel network simulations. The performance of the ELP approach is evaluated using the ns-2 network simulator. Our experiment results show that 1) in a simulated wired network abundant event-level parallelism can be exploited by multicore systems; and 2) the ELP approach can provide satisfactory performance speedups under high event load conditions on wired networks. Our future work is to design and implement a more efficient methodology to find safe events to further improve the performance of the ELP approach.

References

[1] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H.Y. Song, "Parsec: A Parallel Simulation Environment for Complex System," *Computer,* vol. 31, no. 10, pp. 77-85, Oct. 1998.
[2] S. Bhatt, R. Fujimoto, A. Ogielski, and K. Perumalla, "Parallel Simulation Techniques for Large-Scale Networks," *IEEE Comm. Magazine,* vol. 36, no. 8, pp. 42-47, Aug. 1998.
[3] I. Castilla, F. Garcia, and R. Aguilar, "Exploiting Concurrency in the Implementation of a Discrete Event Simulator," *Simulation Modelling Practice and Theory,* vol. 17, no. 5, pp. 850-870, May 2009.
[4] J. Cowie, D. Nicol, and A. Ogielski, "Modeling the Global Internet," *Computing in Science and Eng.,* vol. 1, no. 1, pp. 42-50, Jan. 1999.
[5] M. Creeger, "Multicore CPUs for the Masses," *ACM Queue,* vol. 3, no. 7, pp. 63-64, 2005.
[6] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinnette, "GTW: A Time Warp System for Shared Memory Multiprocessors," *Proc. Winter Simulation Conf.,* pp. 1332-1339, Dec. 1994.
[7] R.M. Fujimoto, *Parallel and Distributed Simulation Systems.* Wiley, 2000.
[8] K.G. Jones and S.R. Das, "Parallel Execution of a Sequential Network Simulator," *Proc. 23rd Conf. Winter Simulation,* pp. 418-424, Dec. 2000.
[9] S. Lin, X. Cheng, and J. Lv, "Micro-Synchronization in Conservative Parallel Network Simulation," *Proc. 22nd Workshop Principles of Advanced and Distributed Simulation (PADS '08),* pp. 195-202, June 2008.
[10] The ns-2 Network Simulator, http://www.isi.edu/nsnam/ns/, 2011.
[11] A. Park and R.M. Fujimoto, "Aurora: An Approach to High Throughput Parallel Simulation," *Proc. 20th Workshop Principles of Advanced and Distributed Simulation (PADS '06),* May 2006.
[12] C. Phillips and L. Cuthbert, "Concurrent Discrete Event-Driven Simulation Tools," *IEEE J. Selected Areas in Comm.,* vol. 9, no. 3, pp. 477-485, Apr. 1991.
[13] G. Riley, M.H. Ammar, R.M. Fujimoto, A. Park, K. PeruMalla, and D. Xu, "A Federated Approach to Distributed Network Simulation," *ACM Trans. Modeling and Computer Simulation,* vol. 14, no. 2, pp. 116-148, Apr. 2004.
[14] G. Riley, R.M. Fujimoto, and M.H. Ammar, "A Generic Framework for Parallelization of Network Simulations," *Proc. Seventh Int'l Symp. Modeling, Analysis and Simulation of Computer and Telecomm. Systems* pp. 128-135, Oct. 1999.
[15] R. Simmonds, C. Kiddle, and B. Unger, "Addressing Blocking and Scalability in Critical Channel Traversing," *Proc. 16th Workshop Parallel and Distributed Simulation,* pp. 15-22, May 2002.
[16] H. Wu, R.M. Fujimoto, and G. Riley, "Experiences Parallelizing a Commercial Network Simulator," *Proc. Winter Simulation Conf.,* pp. 1353-1360, Dec. 2001.
[17] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary, "Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation," *Proc. 13th Workshop Parallel and Distributed Simulation,* pp. 20-28, May 1999.
[18] X. Zeng, R. Bagrodia, and M. Gerla, "Glomosim: A Library for Parallel Simulation of Large-Scale Wireless Network," *Proc. 12th Workshop Parallel and Distributed Simulation (PADS '98),* pp. 154-161, May 1998.

Shie-Yuan Wang received the bachelor degree in computer science from National Taiwan Normal University in 1990 and the master degree in computer science from National Taiwan University in 1992. He also received the master and PhD degrees in computer science from Harvard University in 1997 and 1999, respectively. Currently, he is working as a professor of the Department of Computer Science at National Chiao Tung University, Taiwan. He authors a network simulator, called NCTUns, which is now is a tool widely used by people all over the world. His research interests include wireless networks, Internet technologies, network simulations, and operating systems. He is a senior member of the IEEE.

Chih-Che Lin received the BS and PhD degrees in computer science from National Chiao Tung University, Taiwan, in 2002 and 2010, respectively. He was a core team member of the NCTUns network simulator project during 2002 and 2010. Currently, he is working with the Industrial Technology Research Institute, Taiwan. His current research interests include wireless networking, wireless mesh networks, vehicular networks, intelligent transportation systems, and network simulation. He is a member of the IEEE.

Yan-Shiun Tzeng received the BS degree in communication engineering from Feng Chia University, Taiwan, in 2005 and the MS degree in network engineering from National Chiao Tung University, Taiwan, in 2007. He was a core team member of the NCTUns network simulator project during 2005 and 2007. He has been working with the Trend Micro Incorporated since 2007. His major research interests include networking, network security, and network simulation.

Wen-Gao Huang received the BS degree in computer science and information engineering from National Dong Hwa University, Taiwan, in 2006 and the MS degree in network engineering from National Chiao Tung University, Taiwan, in 2008. He was a core team member of the NCTUns network simulator project during 2006 and 2008. He has been working with the Institute for Information Industry, Taiwan, since 2008. His major research interests include wireless networking and network simulation.

Tin-Wei Ho received the BS and MS degrees in computer science from National Chiao Tung University, Taiwan, in 2007 and 2009, respectively. He was a core team member of the NCTUns network simulator project during 2007 and 2009. He has been working with the Gemintek Corporation, Taiwan, since 2009. His major research interests include wireless networking and network simulation.