Integrated Simulation and Emulation Using Adaptive Time Dilation

Hee Won Lee, David Thuente, and Mihail L. Sichitiu
North Carolina State University
Raleigh, NC 27695
{hlee17, djthuent, mlsichit}@ncsu.edu

ABSTRACT

Simulation and emulation techniques are commonly used to evaluate the performance of complex networked systems. Simulation conveniently predicts the behavior of a complex networked system while usually requiring fewer simplifying model assumptions often necessary for theoretical analysis. In contrast, emulation does not need to re-implement the target real systems, so it may improve on the implementation efficiency of simulation while maintaining much of the realism of testbeds. A hybrid approach in which simulation nodes connect to emulation hosts can be used to combine the advantages of both approaches. In this paper, we propose integrating simulation with emulation using adaptive time dilation to evaluate system performance. If a simulator schedules its events in real time and the simulation time keeps up with the real time, then the hybrid system works very well and meets its deadlines. However, a heavily-loaded simulator can introduce significant simulation delays and thereby create situations where these delays impact the accuracy of the system. Our approach uses time dilation to reduce simulation delays and thus increasing the accuracy of the integrated simulation and emulation system. Our adaptive time dilation dynamically controls the time dilation factor to avoid system overloads for both the simulation and the emulation components and to improve the execution correctness of the hybrid system.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*; D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction*

Keywords

Simulation; Emulation; Virtualization; Time Dilation; ns-3; KVM

1. INTRODUCTION

Modern networks have evolved into highly complex systems that are difficult to debug and to evaluate their performance. Moreover, the network protocols and distributed applications currently in use are sometimes too complex to accurately model their behavior.

Simulation is a primary technique for evaluating the performance of networked systems. Simulation generally uses event-driven models to predict the behavior of complex networked systems while requiring fewer simplifying model assumptions than usually necessary for theoretical analysis. However, unless a simulation model accurately captures the behavior of the real system, the simulation results may be significantly different from those of real systems.

Emulation testbeds [9, 2] can directly use actual implementation code and thus avoid the verification and validation issues required for simulators. While emulation offers much of the realism of testbeds, it is often expensive to scale to a large number of emulated elements.

A hybrid approach using both simulation and emulation can take advantage of both approaches. While simulation is a powerful tool for evaluating large networks, emulation makes it possible to use real protocol implementations, real application code, and even real operating systems (OSs). For instance, the work in [30] integrates S3F [36], a scalable simulation framework, with network emulation OpenVZ [8].

Time dilation allows the passage of virtual time (i.e., time passage from the perspective of a virtual node) to proceed at a slower rate than real time by a specified factor, which is referred to as *time dilation factor* (TDF) [28]. When TDF > 1, time dilation creates, to the virtual machines, the illusion of increased performance [28].

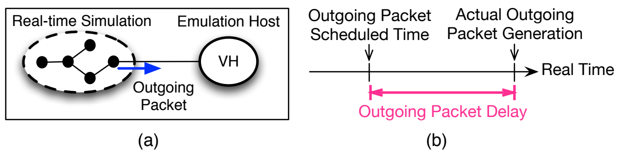


Figure 1: (a) Example topology with an integrated emulation and simulation system with the simulator sending one packet to the emulator. (b) If the simulator is overloaded, outgoing packets can be delayed with respect to their original scheduled times.

In this paper, we propose an approach integrating simulation with emulation using adaptive time dilation in order to keep the simulation and emulation appropriately synchronized and thereby improve accuracy. Usually, when a simulation node exchanges packets with an emulation host, the simulator schedules its events in real time under the assumption that simulation time passes faster than real time. However, when using real-time scheduling, the events' simulation time may fall behind the real time. Therefore, if an outgoing packet is generated from a simulation node with an emulation host as its destination, *outgoing packet delay* can be introduced, as shown in Fig. 1. When the simulator is heavily loaded with a large-scale network topology, outgoing packet delay can be significant and thereby reducing the accuracy of a hybrid simulation and emulation system. Time dilation can reduce the outgoing packet delay, since the simulator can process more events as the virtual time passage rate slows down.

For the emulation, time dilation can prevent CPU overload of the physical hosts (PHs), as time dilation allows virtual hosts (VHs) to perceive higher processing power and network capacity than in real time [27].

Many emulation approaches have used lightweight virtualization to increase scalability [44, 43, 14, 20]. These systems emulate part of the real code (e.g., the network protocol stack), but not an entire application or the OSs. In contrast, our approach uses full virtualization to create fully self-contained VHs and allows our system to emulate unmodified OSs.

Our adaptive time dilation mechanism dynamically changes the time passage rate of the simulator(s) and fully-virtualized hosts, while controlling physical system loads.

The remainder of the paper is organized as follows. In Section 2, we describe our adaptive time dilation approach that can reduce simulation delay in a hybrid simulation and emulation environment. Section 3 presents our system implementation. In Section 4, we discuss our TDF controller tuning and evaluate our integrated simulation and emulation system. Finally, related work in Section 5 is followed by our conclusion in Section 6.

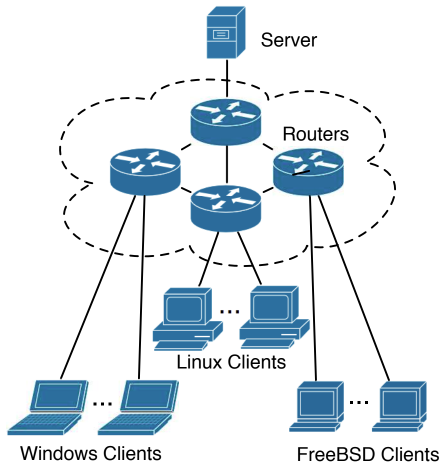
2. PROPOSED APPROACH

Our system emulates applications on unmodified OSs with each running as a *virtual host* (VH). Simulators are used to simulate networks that can exchange packets with the VHs.

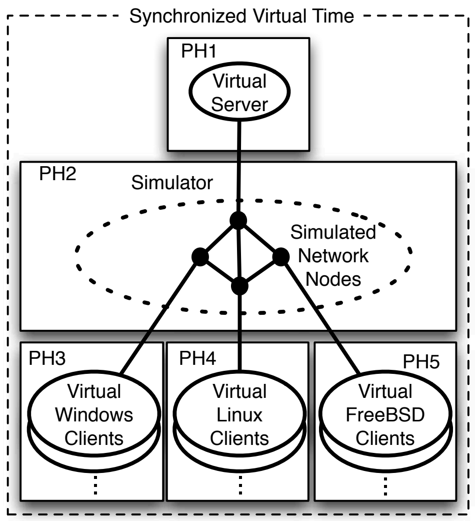
For scalability, multiple PHs can be used for mapping the VHs and simulators to host machines. Once all elements (i.e. VHs and simulators) are ready to start, our integrated simulation and emulation system proceeds at a variable time rate with precise synchronization between all VHs and the simulator.

Figure 2(a) depicts a sample network with Windows/Linux/FreeBSD clients connected to a server through several routers. Figure 2(b) shows a possible mapping of the elements in the real-world topology into virtual elements on five physical machines. The server and the clients can be emulated via a virtualizing technology (in our system we use KVM [10, 5]). The four routers and their links are modeled using a simulator (in our work ns-3 [7]).

Previous emulation systems [17, 20, 30, 14] avoid using full virtualization in order to use available resources efficiently because scalability of the emulated systems was considered more important than full isolation among VMs. In contrast, our approach uses full virtualization to emulate unmodified OSs and unmodified applications.



(a) An example of a real-world topology



(b) Corresponding integrated simulation and emulation topology

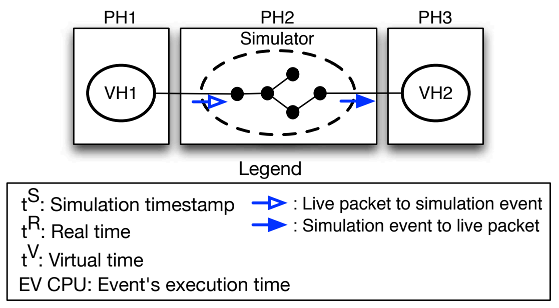
Figure 2: Overview of the proposed approach with (a) an example of a real world topology and (b) a possible mapping of the virtual elements to physical hosts

2.1 The Effects of Time Dilation on Simulation

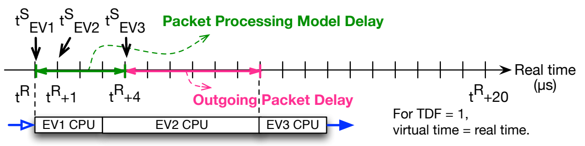
When the simulator schedules its events in real time, delays can be introduced due to excessive execution time. Time dilation can reduce these delays.

Consider the topology shown in Fig. 3(a). VH1 running on PH1 generates and sends a packet to VH2 running on PH3, and the packet passes through a simulator running on PH2. A packet that arrives at the simulator is transformed into a simulation event, which in turn triggers the generation of follow-up events. When the simulator completes the processing of all the events generated by the live packet injection, it then creates an outgoing live packet as appropriate.

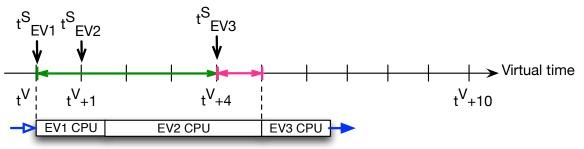
As the simulator uses real-time scheduling with a best-effort policy, an outgoing packet delay may occur due to


PH1
PH2
Simulator
PH3
VH1
VH2
Legend
t^S: Simulation timestamp
t^R: Real time
t^V: Virtual time
EV CPU: Event's execution time
: Live packet to simulation event
: Simulation event to live packet

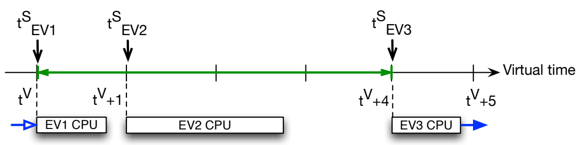
(a) Network topology


t^S_{EV1} t^S_{EV2} t^S_{EV3}
Packet Processing Model Delay
Real time (μs)
t^R t^R+1 t^R+4 t^R+20
Outgoing Packet Delay
For TDF = 1, virtual time = real time.
EV1 CPU
EV2 CPU
EV3 CPU

(b) No time dilation (TDF = 1)


t^S_{EV1} t^S_{EV2} t^S_{EV3}
Virtual time
t^V t^V+1 t^V+4 t^V+10
EV1 CPU
EV2 CPU
EV3 CPU

(c) Time dilation (TDF = 2)


t^S_{EV1} t^S_{EV2} t^S_{EV3}
Virtual time
t^V t^V+1 t^V+4 t^V+5
EV1 CPU
EV2 CPU
EV3 CPU

(d) Time dilation (TDF = 4)

Figure 3: The effect of reduction in outgoing packet delay by time dilation. (a) The simulator transforms a live packet into simulation events, processes the events, and creates an outgoing packet. (b) Real-time scheduler running in real time (TDF=1) introduces outgoing packet delay. (c) Time dilation, where virtual time passes at half the rate of real time (TDF=2), can reduce outgoing packet delay. (d) An even larger TDF=4 can completely eliminate outgoing packet delay.

the simulation events' execution time. Under a best-effort policy, if the timestamp of an event to be processed falls behind real time, the event is processed immediately provided processing resources are available.

For an illustrative explanation, assume that the simulator generates three events to handle an incoming live packet, as shown in Fig. 3(b) (in reality, in ns-3, about 13 events are generated for forwarding a packet through a simulated network node). When a live packet arrives at the simulator at t^R in real time, the simulator creates event 1 with timestamp t^S_{EV1}. Event 2 handles a propagation delay on a communication channel, and event 3 generates an outgoing live packet for VH2.

We refer to the time that it takes for the simulator to process a packet in the time unit of the simulator (i.e., the timestamp of ns-3) as *packet process model delay*. Hence, packet process model delay is t^S_{EV3} - t^S_{EV1} in Fig. 3. *Outgoing packet delay* is the time difference between event 3's scheduled timestamp (t^S_{EV3}) and the actual time at which the simulator starts to process event 3. Outgoing packet delay is a key metric for accuracy and we will explore adaptive time dilation to reduce it.

Assume that event 1 is scheduled by the simulator to finish after 1 μs. In the example in Fig. 3(b), however, the simulator takes 3 μs to process event 1. Event 1 triggers the follow-up events: events 2 and 3. Assume the execution of events 2 and 3 takes 7 μs and 3 μs respectively. Since event 3 is scheduled at time t^S_{EV3} (4 μs after t^S_{EV1}) in simulation time but is sent out after 10 μs from the start of t_{EV1}, an outgoing packet delay of 6 μs occurs in this case.

As shown in Fig. 3(c), when using a time dilation factor of two (TDF = 2), virtual time passes at half of the real-time rate. While the execution time of the events does not change, the simulator now runs in virtual time whose passage rate is $\frac{1}{TDF}$ ($= \frac{1}{2}$) with respect to real time. Event 1, 2, and 3 are then executed along the axis of virtual time (not real time as in Fig. 3(b)) at the scheduled timestamps t^S_{EV1}, t^S_{EV2}, and t^S_{EV3} respectively. Therefore, the outgoing packet delay is reduced to 2 μs in real time (= 1 μs in virtual time).

In Fig. 3(d), when TDF = 4, events 1 and 2 finish their computation before the next event's scheduled time. In this case the simulator has a chance to catch up with the real time events. Hence, outgoing packet delay is completely eliminated.

Therefore, outgoing packet delay can be reduced or completely eliminated by increasing the TDF. When the simulator is heavily loaded with many incoming packets or simulation events, the simulator may not be able to process all events in real time, thus increasing outgoing packet delay and degrading throughput.

2.2 The Effects of Time Dilation on Emulation

If virtual time passes at a slower rate, physical resources appear faster to virtual nodes [27], as the CPU can execute more instructions per unit of virtual time. Hence, a PH can support more VHs, or heavier traffic generators can be executed in a VH without a degradation of the emulated performance.

In addition, when virtual time slows down and a VH generates traffic, the virtual nodes (e.g., simulators and the other VHs) receive the traffic at a slower rate in real time. Consequently, time dilation reduces the workload of physical systems running virtual nodes at the cost of an increase in emulation time.

2.3 Virtual Time

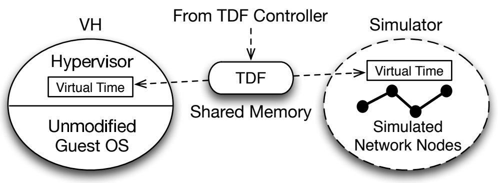


Figure 4: Synchronization of virtual elements

Virtual elements, including VHs and simulators, have their own time generators usually using the real time clock of their PH as a reference. All virtual elements running on our system are instead synchronized to a virtual clock with a common time passage rate by sharing a TDF value as shown in Fig. 4.

The ratio between the virtual time passage rate and real time is $\frac{1}{TDF}$. Hence, given the real time t^R, virtual time t^V can be obtained by:

$$t^V = t^V_{start}(n) + \frac{t^R - t^R_{start}(n)}{TDF(n)}, \tag{1}$$

where $t^V_{start}(n)$ is the value of starting point of n^{th} TDF change (epoch) in virtual time and $t^R_{start}(n)$ is the value of starting point of n^{th} TDF change in real time.

In our approach, the simulators use virtual time t^V instead of real time t^R for their real-time scheduler, while VHs use virtual time t^V generated by the modified hypervisors such that unmodified guest OSs can be used in the VHs.

Since each simulator keeps track of virtual time based on the TDF stored in shared memory, the simulator readjusts the virtual time rate whenever the TDF is changed. For VHs' virtual time, we control the hypervisor time by changing the interrupt frequency of the hypervisor according to the value of TDF. Since the guest OS of the hypervisor creates its own time based on the interrupt frequency, the guest's time passage rate changes according to TDF. Since all virtual elements use a common TDF (i.e., the system TDF), their virtual times are all synchronized.

2.4 Adaptive Time Dilation

Our approach is to control the time passage rate such that outgoing packet delay is reduced or maintained at a low level. Time passage rate control is also used to prevent VHs from overloading their PHs.

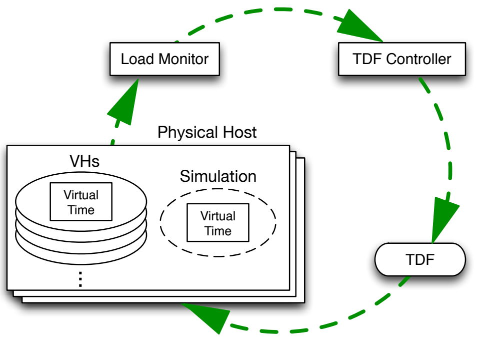


Figure 5: Virtual time control mechanism using TDF to control the system load

Our virtual time control mechanism is shown in Fig. 5. For each PH, a load monitor is periodically checking the CPU loads. The TDF controller then computes a new TDF based on its PH's CPU loads and broadcasts the TDF to all the controllers running on the other PHs. The computed TDF from each controller is the minimum TDF value that is required in a PH to prevent simulators and VHs from introducing outgoing packet delay created by PH overloads.

When a TDF controller receives the other PHs' TDF messages, it updates the current running TDF with the *maximum* of all values received from all PHs. Using the maximum of the TDF values guarantees that no virtual elements are overloaded (even if some of the PHs may be underloaded).

3. SYSTEM IMPLEMENTATION

Our proposed system uses ns-3, which is a widely used discrete-event network simulator. We chose ns-3 because its real-time scheduler uses CPU resources efficiently by using sleep-waiting and busy-waiting; however, our system works independent of the simulator choice.

For VHs, there are several hypervisors that support full virtualization (e.g. KVM [10, 5], Xen [13], and VirtualBox [12]). Our system uses the KVM hypervisor, which was recommended [42] as the optimal choice for high performance computing environments; however, our approach is equally applicable to other hypervisors.

In this section, we first present our system architecture and then our virtual time implementation. Lastly, we define TDF_{load}, the TDF used for system load control.

3.1 System Architecture

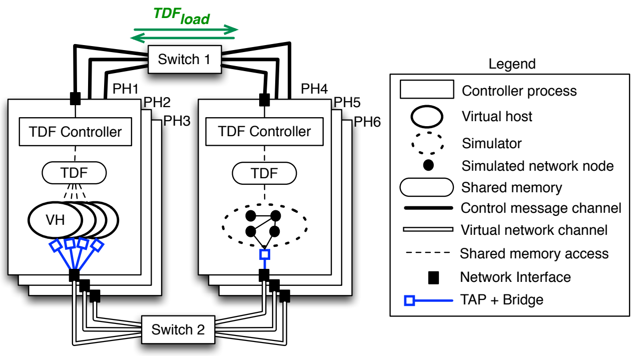


Figure 6: System architecture

Fig. 6 depicts our system architecture that synchronizes VHs and simulators distributed over PHs with a dynamic TDF. The TDF controller on each PH monitors system loads, computes TDF_{load}, and periodically broadcasts the TDF_{load} to all the controllers running on the other PHs. The system TDF is defined as the maximum of the TDF_{load} of all the PHs:

$$TDF_{system} = max(TDF_{load,PH1}, TDF_{load,PH2}, ...), \tag{2}$$

where TDF_{load,PH_i} is the minimum TDF required to maintain the system loads below a target level in PH_i.

The TDF controller stores the system TDF value in the shared memory of each PH, and the VHs use this TDF value to control the progress of their virtual clocks. Since VHs distributed over multiple PHs use a common TDF value, they are all synchronized.

Virtual elements such as VHs and simulators are connected through TAP [11] interfaces and bridges; TAP is a virtual network kernel device that simulates a link layer handling Ethernet frames. For example, as illustrated in Fig. 6, a simulator creates one or more TAP interfaces, which can be bridged to a real network interface to communicate with VHs running in other PHs.

The TDF control messages use a physically-isolated control channel through switch 1, while virtual elements (i.e., VHs and simulators) use a virtual network channel through switch 2 to communicate with each other. The isolated control channel assures robust TDF control operations and eliminates control traffic from affecting the virtual network traffic.

3.2 Virtual Time Implementation

We modify the real time scheduler of ns-3 to use virtual time instead of real time. For VHs, the hypervisors also use virtual time.

The ns-3 real-time scheduler obtains real time by calling `GetRealtime()`, a method of the `WallClockSynchronizer` class [7]. We replace `gettimeofday()`, a POSIX system call to retrieve the PH's real time in `GetRealTime()`, with our function `get_virtualtime()`. Our function that computes virtual time, `get_virtualtime()`, is an implementation of (1), where t^R is obtained by `gettimeofday()` and TDF is retrieved from shared memory. The KVM hypervisor also uses our function `get_virtualtime()` to obtain its virtual time.

In short, the ns-3 simulator and the KVM hypervisor both replace `gettimeofday()` with our function `get_virtualtime()` to obtain the time and hence all virtual elements' time passes at a rate of $\frac{1}{TDF}$ with respect to real time.

3.3 TDF for System Load Control

When the load monitor obtains a new CPU load, the TDF controller computes a new TDF, TDF_{load}. A desirable TDF control property is that the computed TDF adapts rapidly to current loads, while the frequency of TDF changes is minimized for system stability. To meet these conflicting requirements, the TDF controller uses three parameters: α for the exponential moving average (EMA), *Gain*, and *Insensitivity* that balance the TDF responsiveness with system stability as seen below.

EMA can be used to prevent rapid changes in the current load ($Load_{current}$) from changing TDF_{load} too rapidly. The EMA value of a system load for a monitoring interval, denoted by $Load_{EMA}(n)$, is computed as:

$$\begin{aligned} Load_{EMA}(n) = (1-\alpha)\cdot Load_{EMA}(n-1) \\ +\alpha \cdot Load_{current}. \end{aligned} \tag{3}$$

Gain determines how rapidly TDF_{load} will adapt to system loads. If $Load_{EMA}(n)$ is greater than a target CPU load ($Load_{target}$), TDF_{load} increases, and vice versa as well. The magnitude of the increase or decrease is directly proportional to the value of the *Gain*.

Insensitivity is used to minimize the number of TDF changes until there are significant deviations of $Load_{EMA}$ from a target CPU load $Load_{target}$.

In summary, $TDF_{load}(n)$ is given by:

$$\begin{aligned} TDF_{load}(n) = TDF_{load}(n-1) \\ + Gain \cdot sgn(Load_{EMA}(n) - Load_{target}) \\ \cdot \left| 2 \cdot \frac{Load_{EMA}(n) - Load_{target}}{Load_{target}} \right|^{Insensitivity}, \end{aligned} \tag{4}$$

where $sgn(x)$ is the sign of x, and $|x|$ is the absolute value of x.

When the load monitor obtains a new current load, $Load_{current}$, the TDF controller computes a new TDF, TDF_{load} using (3) and (4). The TDF controller computes the new TDF_{load} depending on the configuration of the control parameters α, *Gain*, and *Insensitivity* that reflect different priorities of the system such as responsiveness versus stability.

4. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the integrated simulation and emulation system.

4.1 Experimental Setup

For the evaluation we used three identical physical hosts (PHs): each PH is a Dell PowerEdge R210 with two 1 Gigabit Ethernet interfaces, which are connected to two separate 1 Gbps switches. The first interface is used for exchanging TDF control messages, and the second interface is used for emulating links between virtual nodes, i.e., virtual elements on different PHs communicate through the second interface.

We use KVM (qemu-kvm-0.13.0) for full virtualization and ns-3 (ns-3.12.1) for simulation. Ubuntu Linux (ubuntu-10.04-server-amd64) is used for both PHs and VH guest OSs.

4.2 TDF Controller Tuning

The goal of the TDF controller is to adapt the system TDF to CPU loads in the PHs, while minimizing the number of TDF changes. We determine the parameters α in (3) and *Gain*, and *Insensitivity* in (4) for a responsive, yet stable TDF_{load} control.

The current TDF is controlled to maintain the current PH CPU load $Load(i)$ close to a target CPU load TDF_{load}. The tracking error between the current load $Load(i)$ and the target load $Load_{target}$ can be quantified as:

$$C_1 = \frac{\sum_{i=1}^{N}(Load_{target} - Load(i))^2}{N}, \tag{5}$$

where the summation is taken over a measurement period with N samples. Similarly, the change in TDF_{load} can be measured by:

$$C_2 = \sum_{i=1}^{N-1}\left(\frac{TDF_{load}(i+1) - TDF_{load}(i)}{t(i+1)-t(i)}\right)^2, \tag{6}$$

where the summation is taken over the same measurement period of N.

The normalized value of C_1, denoted by $\overline{C}_1$, is obtained by dividing C_1 by the average of C_1 values over a measurement period (60 seconds in our experiments). Similarly, $\overline{C}_2$ is the normalized value of C_2.

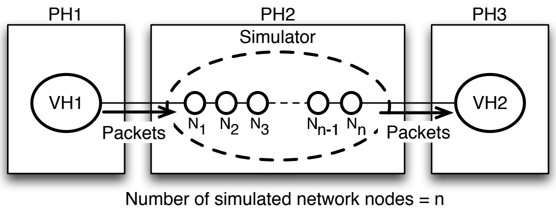
While we consider two objectives (low tracking error corresponding to a low value of C_1 and infrequent TDF_{load} changes corresponding to a low value of C_2), we favor infrequent TDF_{load} changes over better tracking and thus we define the total cost as:

$$C = \overline{C}_1 + w\overline{C}_2, \tag{7}$$

where w, the relative weight assigned to $\overline{C}_2$, is greater than one. Experimentally we find that the values of w between 5 and 20 results in relatively infrequent TDF_{load} changes; therefore, we choose $w = 10$. Our TDF controller operates as designed (i.e., the rapid adaptation of TDF to system loads and the minimization of the number of TDF changes) when $\alpha \le \frac{1}{8}$, *Gain* of $1 \sim 3$, and *Insensitivity* of $4 \sim 10$. Under these ranges of α, *Gain*, and *Insensitivity*, the total cost C is minimized in our experiments. For all the experiments we use $\alpha = \frac{1}{64}$, $Gain = 1$ and $Insensitivity = 10$, as the PH running the simulator changes the CPU load rapidly. We also use the target CPU load $Load_{target} = 60\%$, at which VHs are able to generate traffic without packet losses.

Finally, we use a TDF control interval of 10 ms (which is also a CPU load checking interval), as a smaller TDF control interval starts to affect system loads.

4.3 Evaluation Topology



(a)

Figure 7: Evaluation Topology

When the simulator schedules its events in real time, outgoing packet delay can be introduced due to the simulator execution time. In order to measure outgoing packet delay, we construct an evaluation topology, as shown in Fig. 7. This topology is used for the performance evaluation of our approach.

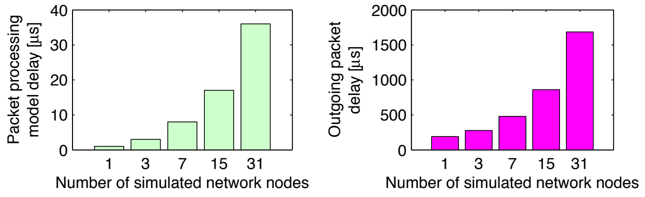
The simulator running in PH2 creates n simulated network nodes, each of which simulates a network node (i.e., a host or router) with the Internet protocol stack and two CSMA network devices. We use the network devices to connect simulated network nodes through CSMA channel models in ns-3. The CSMA channel model has two configurable parameters: data rate and delay for modeling transmission and propagation delay respectively. The number of simulated network nodes, n, does not include the ghost nodes for VHs.

VH1 running on PH1 generates packets and sends them toward VH2 running on PH3. The packets passes through simulated network nodes on PH2.

4.4 Outgoing Packet Delay

We discuss the effect of time dilation on outgoing packet delay, and then investigate how network nodes' processing delays can affect outgoing packet delay.

4.4.1 *The Effect of Time Dilation*



(a) (b)

Figure 8: The simulator measures (a) packet processing model delay and (b) outgoing packet delay, while processing a UDP packet in the topology of Fig. 7, where the CSMA channel model delay is 0 μs.

In the topology of Fig. 7, a CSMA channel model connects VH1 to the first simulated network node (N_1), and another CSMA channel model connects the last simulated network node (N_n) to VH2. The data rate and delay of the CSMA channel model are 1 Gbps and 0 μs respectively. VH1 generates 200 146-byte UDP packets (the UDP payload size = 100 bytes) at a constant bit rate of one packet per second. Figures 8 (a) and (b) show the average packet processing model delay and the average outgoing packet delay respectively, both measured in the simulator.

When a packet arrives at the simulator, the packet is transformed into a series of events. The real-time scheduler processes all the events. The ns-3 simulator computes packet processing model delay. The packet process model delay is the time that it takes for the simulator to process a packet in the time units of the simulator and is shown in Fig. 8(a). As the number of simulated network nodes increases, the packet processing model delay increases proportionally because a packet passes through more simulated network nodes and CSMA channels.

Outgoing packet delay is the delay between the instant when the simulator generates a packet towards VH2 and the scheduled time of that packet. As the number of simulated network nodes increases, outgoing packet delay also increases, as shown in Fig. 8(b). As the real-time scheduler in ns-3 simulates the behavior of a physical layer (i.e., the CSMA channel in our experiments), unless we use a hardware-based simulator such as a FPGA-based channel simulator [18], it is difficult or impossible for the simulation to keep pace with real time, because the execution time that it takes to process events for the physical layer behaviors (e.g., transmission delay, propagation delay, inter-frame gaps, etc.) is much larger than the scheduled time. As each network node in the simulation introduces additional delays, the simulator falls further and further behind real time as a packet progresses through the simulated network nodes. Hence, as the number of simulated network nodes and CSMA channels increase, outgoing packet delay also increases almost proportionally.

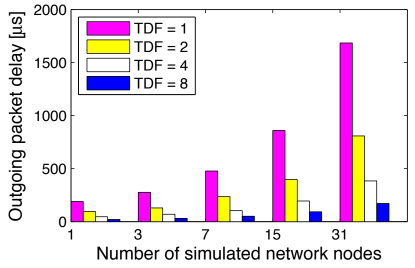


Figure 9: Increasing TDF decreases outgoing packet delay.

We can decrease the outgoing packet delay from the ns-3 simulator by slowing the virtual time. As time proceeds at a slower rate (i.e., TDF increases), the outgoing packet delay is reduced almost linearly. For instance, as shown in Fig. 9, when the simulator runs 31 nodes in real time, there is an outgoing packet delay of 1685 μs. When TDF increases to 2, 4, and 8, the outgoing packet delay decreases to 808, 384, and 171 μs respectively.

4.4.2 *Consideration of Network Node's Processing Delay*

The ns-3 network simulator does not include a model for the network node's processing delay. The work in [32] proposes a methodology to capture processing delay models from communication software running on real devices.

A network node's processing delay can significantly affect the total packet delay. Hence, for our evaluation we measure a physical node's processing delay.

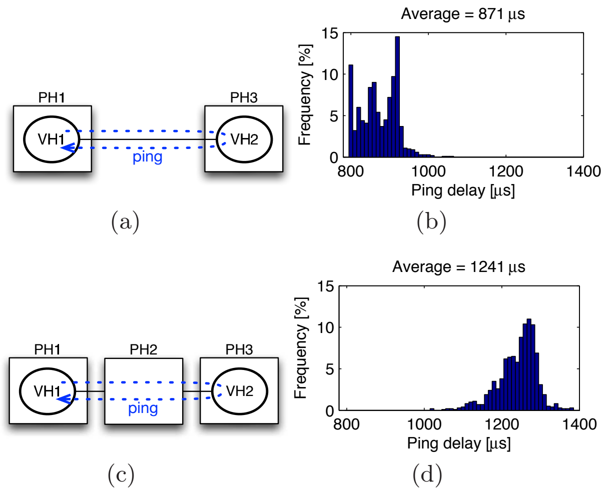


Figure 10: Experiments for measuring the processing delay of a PH: ping delay distribution without (a), (b) and with (c), (d) a PH in the path

In order to measure a node's processing delay, we conduct delay measurements on two different network topologies illustrated in Fig. 10. We first measure a ping delay between two VHs without a PH between VHs, as shown in Fig. 10(a). We send a 64-byte ICMP packet every second. Figure 10(b) shows a delay distribution for a total of 1000 ICMP packets. The average ping delay of 1000 ICMP packets is 871 μs.

We then measure another ping delay on the configuration shown in Fig. 10(c). In this configuration, VH1 sends the same ICMP packets to VH2 through a real physical machine (PH2). The average delay of 1000 ICMP packets on this configuration is 1241 μs, and the distribution is shown in Fig. 10(d).

Hence, the average processing delay of a real node (PH2) is $\frac{1241-871}{2} = 185\mu s$. In our performance evaluation, we use $185\mu s$ for simulating the nodes' processing delay.

In order to model a node's processing delay in simulation, we use an additional 185 μs of CSMA channel model delay. Since a packet is transmitted from simulated network node 1 (N_1) to simulated network node n (N_n) in the topology of Fig. 7, we add the CSMA channel delay on each node's outgoing link; i.e., for simulated network node N_1, the delay addition is placed on N_1's right-side CSMA channel. Hence, packet processing model delay linearly increases by ~ 185 μs per simulated network node. For instance, for 31 simulated network nodes, the packet processing model delay is 185 $\mu s\times$ 31 nodes = 5735 μs while the other delays such as inter-frame gaps and transmission delays are negligible by comparison.

While packet processing model delay increases approximately linearly with the number of simulated network nodes

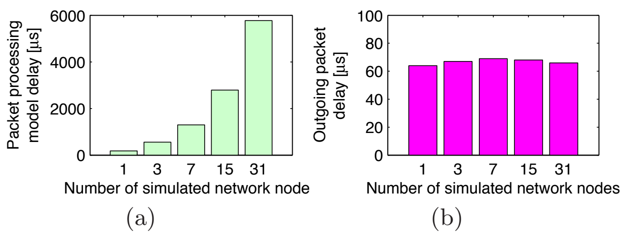
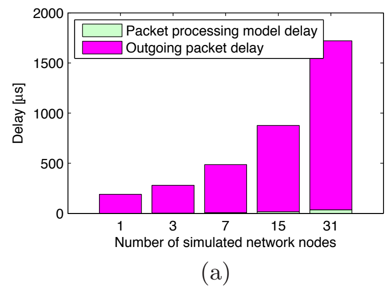


Figure 11: The simulator measures (a) packet processing model delay and (b) outgoing packet delay, while processing a 146-byte UDP packet in the topology of Fig. 7, where the CSMA channel model delay is 185 μs.

in Fig. 11(a), outgoing packet delays are almost constant, as shown in Fig. 11(b). When CSMA channel delay is sufficiently large (185 μs) for the simulation to keep pace with real time, the outgoing packet delay is expected to be completely removed, but a delay of approximately 60 μs is still present: after the ns-3 real-time scheduler performs a sleep and spin wait for a CSMA channel model delay of 185 μs, the `CsmaNetDevice` of the ns-3 simulator receives a packet and then sends it to `TapBridge` for a real packet generation. This process takes approximately 60 μs. Since outgoing packet delay occurs only at the last node (i.e., N_n), it remains almost constant, independent of the number of simulated network nodes.



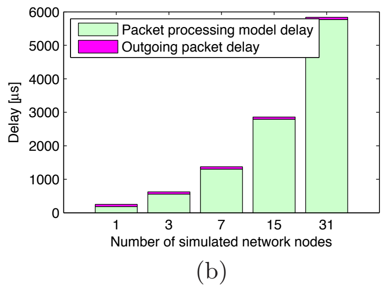


Figure 12: Comparison of the packet processing model delay and the outgoing packet delay when CSMA channel model delay is (a) 0 μs and (b) 185 μs

For comparison, Fig. 12 juxtaposes the packet processing model delay and the outgoing packet delay shown in Fig. 8 (for CSMA channel delay = 0 μs) and Fig. 11 (for CSMA channel delay = 185 μs). Recall that outgoing packet delay is our key metric for accuracy. When comparing Fig. 12 (a) and (b), as CSMA channel delay of 185 μs is included for nodes' processing delay, the outgoing packet delay is signif-

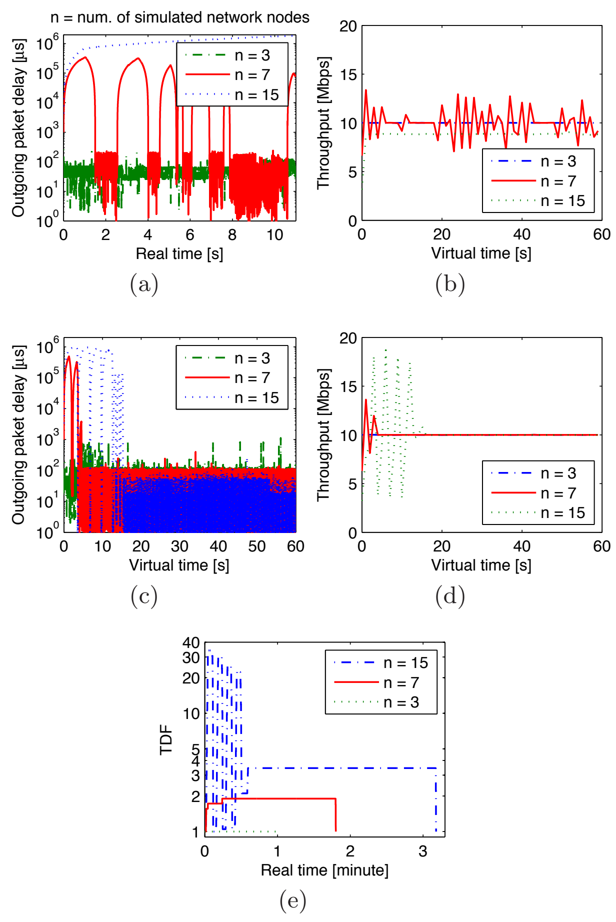


Figure 13: Without time dilation, as the number of simulated network nodes increases, outgoing packet delay significantly increases in (a), and throughput is degraded in (b). When using adaptive time dilation, outgoing packet delay is reduced to $\sim$ 100 μs in (c), and throughout reaches the packet generation rate (10 Mbps) in (d). Our TDF controller increases TDF for a larger number of simulated network nodes in (e).

icantly reduced. Even though the real-time scheduler simulates a physical layer, if the nodes' processing delay is captured in the model (through the CSMA channel delay in our setup), the simulator is able to catch up with real time, resulting in a significant reduction in the delay introduced by the simulator.

4.5 Evaluation of Adaptive Time Dilation

When the simulator is heavily loaded due to network traffic, outgoing packet delay can increase. In this section, we show how adaptive time dilation can reduce outgoing packet delay under several traffic scenarios. For all the scenarios, we use a data rate 10 Mbps and we model the 185 μs of node processing delay by using the CSMA channel model.

4.5.1 UDP Traffic Scenario

In this section we test UDP traffic on our evaluation topology (Fig. 7). VH1 sends towards VH2 1046-byte UDP packets (the UDP payload size = 1000 bytes) at 10 Mbps.

The simulator can be overloaded by a large number of simulated network nodes or a large traffic load. Under the influx of 10-Mbps UDP packets, with a single simulated network node (n = 1), the simulator can almost process all the events in time, i.e., the outgoing packet delay is approximately 60 μs in our experiments. We do not show this result to avoid cluttering Fig. 13(a). However, as the number of simulated network nodes increases, the outgoing packet delay significantly increases as shown in Fig. 13(a). When n = 7, peaks and troughs occur, as the simulator is repeatedly overloaded and underloaded by the traffic loads. When n = 15, the outgoing packet delay continues to increase since the simulator is not able to process all the events in time. As the number of simulated network nodes increases, throughput is also degraded, as shown in Fig. 13(b).

As the TDF controller dynamically changes TDF according to CPU loads, outgoing packet delay is maintained at approximately 100 μs regardless of the number of simulated network nodes. When the number of simulated network nodes is 15, outgoing packet delay is even lower due to the effect of time dilation, as shown in Fig. 13(c). Figure 13(c) also shows that as the number of simulated network nodes increases, it takes more time for the simulation to reach a steady state. With adaptive time dilation, throughput is approximately the same as the offered load (i.e., 10 Mbps), as shown in Fig. 13(d).

As the number of simulated network nodes increases, our TDF controller increases the TDF to reduce simulation loads and, consequently, simulation run time increases, as shown in Fig. 13(e).

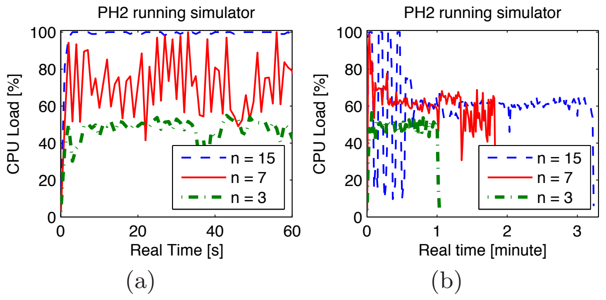


Figure 14: CPU loads (a) without and (b) with adaptive time dilation.

Our TDF controller controls CPU loads at $Load_{target}$ = 60%, as shown in Fig. 14. As the number of simulated network nodes increases, the PH2's CPU load also increases, as shown in Fig. 14(a). However, when using adaptive time dilation, even if the number of simulated network nodes increases, the CPU loads are controlled near the target load 60%. The CPU loads of PH1 and PH3 are much lower than 60%, meaning that in our system at an equal traffic load, the simulator has a much higher CPU load than the VHs.

4.5.2 TCP Traffic Scenario

We generate TCP traffic on our evaluation topology (Fig. 7). We run an iperf [4] client on VH1 and an iperf server on VH2, while changing the number of simulated network nodes on PH2.

Without time dilation (TDF = 1), as the number of simulated network nodes increases, outgoing packet delay significantly fluctuates, as shown in Fig. 15(a) . However, when

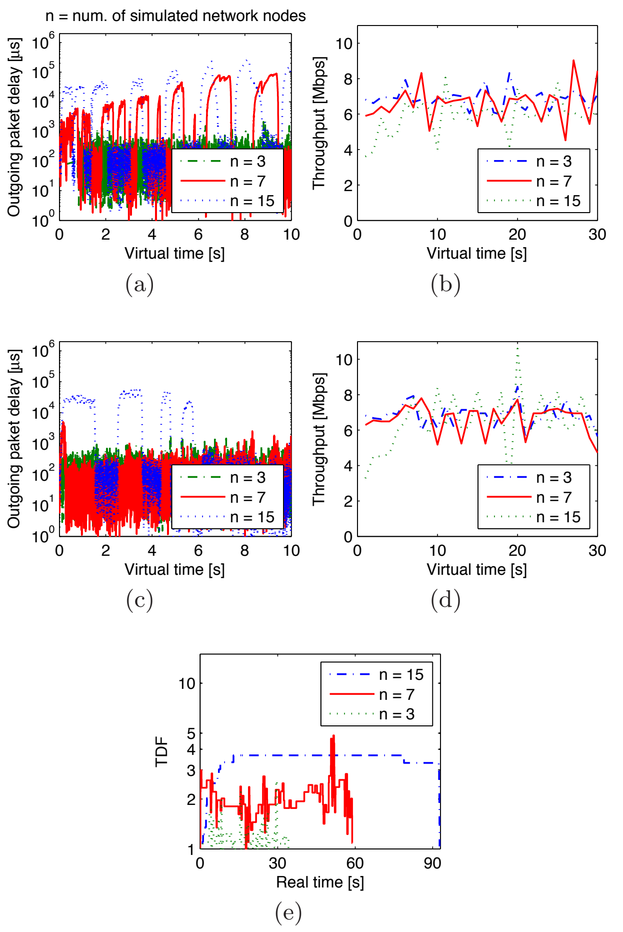


Figure 15: Without time dilation, as the number of simulated network nodes increases, outgoing packet delay significantly fluctuates in (a), and the corresponding throughput is shown in (b). When using adaptive time dilation, outgoing packet delay is reduced to approximately 200 μs in (c), and the corresponding throughput is shown in (d). Our TDF controller increases TDF for larger number of simulated network nodes in (e).

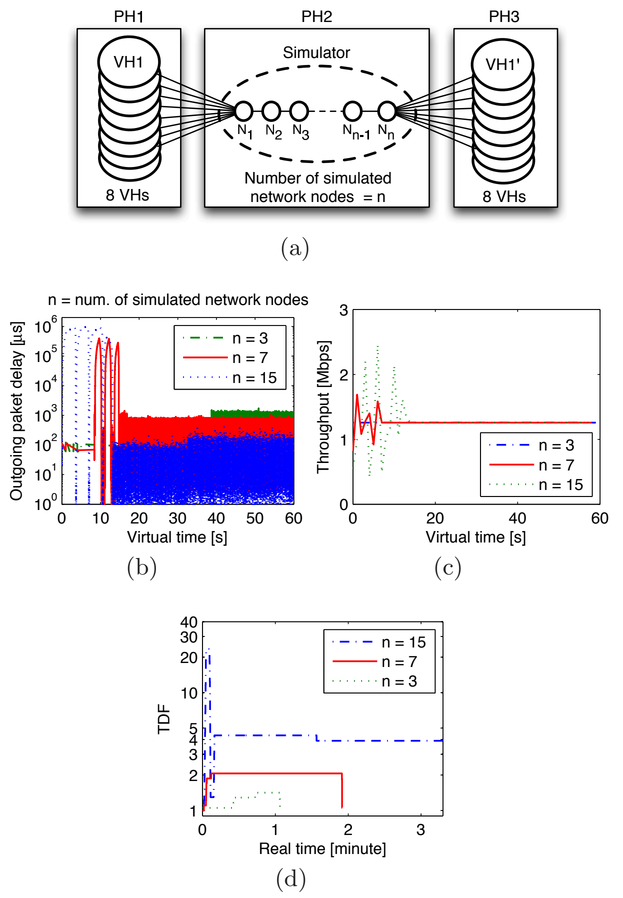


Figure 16: Each VH on PH1 sends 1.25-Mbps UDP packets to the corresponding VH on PH3 through the simulator in (a). With adaptive time dilation, outgoing packet delay, throughput and TDF are shown in (b), (c), and (d) respectively.

using adaptive time dilation, the outgoing packet delay is controlled to about 200 μs when the number of simulated network nodes is $n = 3$, 7 and 15. Throughput variance in Fig. 15(d) is slightly reduced by adaptive time dilation, as compared with Fig. 15(b).

As shown in Fig. 13(e), for seven simulated network nodes, TDF is stably maintained at $TDF \approx 2$ for UDP traffic. However, as shown in Fig. 15(e), TDF changes are frequent for TCP traffic, since TCP produces bursty traffic. On the other hand, since TCP gradually increases traffic over time, for 15 simulated network nodes, there are no TDF oscillations as seen for UDP traffic (compare Fig. 13(e) with Fig. 15(e)).

4.5.3 *Multiple VHs Scenario*

In this section, we create a scenario where multiple VHs run on a PH, as shown in Fig. 16(a). We run 8 VHs in PH1 and 8 VHs in PH3. VH1 running on PH1 sends UDP packets at 1.25 Mbps (constant bit rate) towards VH1′ running on PH3. The size of each UDP packet is 1046 bytes. VH2 on PH1 sends UDP packets at the same rate towards VH2′ on PH3, and so on. Since the 8 VHs on PH1 generate a total of 10 Mbps (= 1.25 Mbps × 8 VHs), the simulator processes the same amount of traffic as in the previous UDP scenario discussed in Section 4.5.1, where a single VH on PH1 generated 10-Mbps UDP packets.

As shown in Fig. 16(b), the outgoing packet delay has slightly increased compared to Fig. 13(c). While the same amount of traffic (i.e., 10 Mbps) passes through the simulator in total, multiple VHs can create more variance in traffic patterns. Hence, the simulator may sometimes process a closely bunched series of packets such that the simulator can often be busier, thus slightly increasing outgoing packet delay. As the number of simulated network nodes increases, time dilation decreases outgoing packet delay. As shown in Fig. 16(c), despite a slight increase in the outgoing packet delay, throughput remains the same as in Fig. 13(d). On the other hand, the TDF changes for the multiple VHs scenario in Fig. 16(d) are similar to those for the UDP traffic scenario in Fig. 13(e).

5. RELATED WORK

The work in this paper builds on a large body of related work from many different areas, including time dilation, hybrid simulation and emulation systems, real time emulation systems and large-scale emulation systems.

Time control: The approach in SliceTime [41] is to alternately suspend and resume the entire system in order to connect VMs to discrete event simulations [33] that may lag behind in time under heavy system loads [40]. DieCast [27] uses time dilation, which allows virtual time to pass slower than real time such that physical resources appear to virtual nodes to be faster [28]. The Open Network Emulator uses a temporal model referred to as *relativistic time* and employs a lightweight virtualization framework called *Weaves* to enhance scalability [17, 20]. *Weaves* emulates multiple instances of an application or protocol stack inside a single OS process [35]. The approach in [43, 14] uses a timeslice-based scheduler; as the scheduler gives a timeslice to a VE (Virtual Environment), the VE consumes the timeslice and stops its operation.

The approaches in [40, 41, 28, 27] are static in that the relative ratio between real and virtual time is fixed for the life of the VMs. NETplace [23] and NETbalance [24] use epoch-based virtual time [22] for implementing a dynamic time dilation. While the dynamic time dilation in [22] uses a threshold-based load control mechanism, our TDF controller maintains system loads at a target level. Our TDF control mechanism towards a target system load can perform more accurately for a hybrid simulation and emulation environment in that real-time simulation is sensitive to system loads.

In parallel discrete event simulation, conservative algorithms use lookahead for time synchronization [37]. While these algorithms send null messages to compute lookahead, our approach does not exchange null messages. Instead, our adaptive time dilation approach sends TDF messages, which may change the time passage rate of virtual elements. In the algorithm of [37], time proceeds for an amount of lookahead and stops, but in our approach, once TDF changes, virtual time continues to proceed at the rate of $\frac{1}{TDF}$.

Hybrid simulation and emulation: As mentioned in Introduction, the work in [30] integrates S3F [36], a scalable simulation framework, with the OpenVZ [8]-based network emulation. Each VE is synchronized with the simulation clock in S3F. The S3F simulation engine controls VEs' execution to preserve the causal relationship of the whole network scenario [30].

ROSENET [26, 25] is a network emulation approach that uses a remote high-fidelity simulation along with a low-fidelity emulator serving a locally executing real-time application to improve scalability and accuracy.

WHYNET [45] consists of a wireless network emulator (TWINE [44]), simulated radio devices, and physical testbeds. Physical elements includes 801.11-based networks, sensor networks, and SDR/MIMO radio platforms. TWINE [44] embeds the simulated physical and MAC layers into the operating system for scalability.

The approach in [38] incorporates a physical layer emulator for OFDM-based IEEE 802.11 communications into the ns-3 simulator. The work in [21] simulates heterogeneous systems by using a discrete-event simulator, TOSSIM, that implements the lowest layer of components in the TinyOS API, and EmSIM that provides a real code simulation capability.

Real-time emulation: CORE (Common Open Research Emulator) is a real-time network emulator that emulates the network stack of routers or hosts through virtualization, and that simulates the links which connect them together [15]. CORE employs the lightweight virtualization to allow over a hundred virtual machines to run on a single emulation server. Wireless channel emulators use hardware to simulate wireless channel propagation in real time [39, 31, 18]. RAMON (Rapid-Mobility Network emulator) is a software/hardware emulator that allows the ns-2 simulator to interact with hardware components including access points (APs), attenuators, laptops, and smart phones, while emulating mobility in wireless networks [29]. MNE (Mobile Network Emulator) simulates the mobility of wireless nodes [34]. Since MNE operates in real time, it uses simplistic propagation models that do not require significant amounts of processing.

Large-scale emulation: Several large testbeds partially employ or significantly depend on emulation techniques. PlanetLab [9] is a distributed overlay network designed to evaluate planetary-scale network services, allowing multiple services to run concurrently, each in its own *slice* [16, 19]. Emulab [2] offers integrated access to emulated PC nodes, an 802.11 a/b/g testbed, and universal software defined radios (USRP devices). Emulab can also be expanded into PlanetLab testbeds, allowing for live Internet experimentation. GENI [3] provides researchers across the country with collaborative environments on which new network architectures and their implementations can be tested, while supporting scalable experimentation on shared and heterogeneous infrastructure. DETERlab [1] supports experimentation on next-generation cyber security technologies, and uses the Emulab cluster testbed software to control and manage a pool of PCs. ModelNet [6] emulates the delays, losses, and throughput of packets traveling between different application instances.

6. CONCLUSION

In this paper, we proposed an approach integrating simulation with emulation by using adaptive time dilation. VHs and simulators are synchronized to a common virtual time passage rate. When the simulator schedules its events in real time, event processing can easily overload the simulator, as the simulation load increases. Our TDF controller dynamically changes a virtual time rate such that outgoing packet delay is minimized in the simulator. Without using virtual time in our evaluation, the simulator becomes heavily loaded as TCP and UDP traffic passes through it, and thus increases outgoing packet delay. Our adaptive time dilation approach dynamically changes TDF such that outgoing packet delay is minimized. Since our integrated simulation and emulation approach uses full virtualization, unmodified OSs and applications can be used in the system.

7. REFERENCES

[1] DeterLab. `http://www.isi.deterlab.net`.
[2] Emulab. `http://www.emulab.net`.
[3] GENI Project. `http://www.geni.net`.
[4] Iperf. `http://iperf.sourceforge.net`.
[5] KVM. `http://www.linux-kvm.org`.

[6] ModelNet. `http://modelnet.ucsd.edu`.

[7] ns-3. `http://www.nsnam.org`.

[8] OpenVZ. `http://wiki.openvz.org`.

[9] PlanetLab. `http://www.planet-lab.org`.

[10] QEMU. `http://wiki.qemu.org`.

[11] Universal TUN/TAP Device Driver. `http://vtun.sourceforge.net/tun`.

[12] VirtualBox. `http://www.virtualbox.org`.

[13] Xen. `http://www.xenproject.org`.

[14] Y Zheng, D M Nicol, D Jin1 and N Tanaka, A virtual time system for virtualization-based network emulations and simulations, Journal of Simulation, 1 June 2012.

[15] Ahrenholz, J., Danilov, C., Henderson, T., and Kim, J. CORE: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE* (Nov. 2008), pp. 1–7.

[16] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. Operating system support for planetary-scale network services. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1* (Berkeley, CA, USA, 2004), USENIX Association, pp. 19–19.

[17] Bergstrom, C., Varadarajan, S., and Back, G. The distributed open network emulator: Using relativistic time for distributed scalable simulation. In *Principles of Advanced and Distributed Simulation, 2006. PADS 2006. 20th Workshop on* (2006), pp. 19–28.

[18] Borries, K., Judd, G., Stancil, D., and Steenkiste, P. FPGA-based channel simulator for a wireless network emulator. In *Vehicular Technology Conference, 2009. VTC Spring 2009. IEEE 69th* (April 2009), pp. 1–5.

[19] Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev. 33* (July 2003), 3–12.

[20] Duggirala, V., and Varadarajan, S. Open network emulator: A parallel direct code execution network simulator. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation* (Washington, DC, USA, 2012), PADS '12, IEEE Computer Society, pp. 101–110.

[21] Girod, L., Stathopoulos, T., Ramanathan, N., Elson, J., Estrin, D., Osterweil, E., and Schoellhammer, T. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2004), SenSys '04, ACM, pp. 201–213.

[22] Grau, A., Herrmann, K., and Rothermel, K. Efficient and scalable network emulation using adaptive virtual time. In *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on* (Aug. 2009), pp. 1–6.

[23] Grau, A., Herrmann, K., and Rothermel, K. NETplace: Efficient runtime minimization of network emulation experiments. In *Performance Evaluation of Computer and Telecommunication Systems (SPECTS), 2010 International Symposium on* (July 2010), pp. 265–272.

[24] Grau, A., Herrmann, K., and Rothermel, K. NETbalance: Reducing the runtime of network emulation using live migration. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on* (Aug. 2011), pp. 1–6.

[25] Gu, Y., and Fujimoto, R. Applying parallel and distributed simulation to remote network emulation. In *Simulation Conference, 2007 Winter* (Dec 2007), pp. 1328–1336.

[26] Gu, Y., and Fujimoto, R. Performance evaluation of the rosenet network emulation system. In *Distributed Simulation and Real-Time Applications, 2007. DS-RT 2007. 11th IEEE International Symposium* (Oct 2007), pp. 276–283.

[27] Gupta, D., Vishwanath, K. V., and Vahdat, A. DieCast: Testing distributed systems with an accurate scale model. In *Proc. of NSDI* (2008), pp. 407–421.

[28] Gupta, D., Yocum, K., Mcnett, M., Snoeren, A. C., Vahdat, A., and Voelker, G. M. To infinity and beyond: time warped network emulation. In *In ACM Symposium on Operating Systems Principles* (2005).

[29] Hernandez, E., and Helal, A. RAMON: rapid-mobility network emulator. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on* (Nov. 2002), pp. 809–817.

[30] Jin, D., Zheng, Y., Zhu, H., Nicol, D. M., and Winterrowd, L. Virtual time integration of emulation and parallel simulation. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation* (Washington, DC, USA, 2012), PADS '12, IEEE Computer Society, pp. 201–210.

[31] Kahrs, M., and Zimmer, C. Digital signal processing in a real-time propagation simulator. *Instrumentation and Measurement, IEEE Transactions on 55*, 1 (Feb. 2006), 197–205.

[32] Kristiansen, S., Plagemann, T., and Goebel, V. Modeling communication software execution for accurate simulation of distributed systems. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (New York, NY, USA, 2013), SIGSIM-PADS '13, ACM, pp. 67–78.

[33] Law, A. M., and Kelton, D. M. *Simulation Modeling and Analysis*, 3rd ed. McGraw-Hill Higher Education, 1999.

[34] Macker, J., Chao, W., and Weston, J. A low-cost, ip-based mobile network emulator (MNE). In *Military Communications Conference, 2003. MILCOM 2003. IEEE* (Oct. 2003), vol. 1, pp. 481–486.

[35] Mukherjee, J., and Varadarajan, S. Weaves: A framework for reconfigurable programming. *International Journal of Parallel Programming 33* (2005), 279–305. 10.1007/s10766-005-3591-5.

[36] Nicol, D., Jin, D., and Zheng, Y. S3f: The scalable simulation framework revisited. In *Simulation Conference (WSC), Proceedings of the 2011 Winter* (2011), pp. 3283–3294.

[37] Nicol, D. M. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM 40*, 2 (Apr. 1993), 304–333.

[38] Papanastasiou, S., Mittag, J., Strom, E., and Hartenstein, H. Bridging the gap between physical layer emulation and network simulation. In *Wireless Communications and Networking Conference (WCNC), 2010 IEEE* (2010), pp. 1–6.

[39] Picol, S., Zaharia, G., Houzet, D., and El Zein, G. Hardware simulator for MIMO radio channels: Design and features of the digital block. In *Vehicular Technology Conference, 2008. VTC 2008-Fall. IEEE 68th* (Sept. 2008), pp. 1–5.

[40] Weingärtner, E., Schmidt, F., Heer, T., and Wehrle, K. Synchronized network emulation: matching prototypes with complex simulations. *SIGMETRICS Perform. Eval. Rev. 36* (August 2008), 58–63.

[41] Weingärtner, E., Schmidt, F., Lehn, H. V., Heer, T., and Wehrle, K. SliceTime: a platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 19–19.

[42] Younge, A., Henschel, R., Brown, J., von Laszewski, G., Qiu, J., and Fox, G. Analysis of virtualization technologies for high performance computing environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on* (July 2011), pp. 9–16.

[43] Zheng, Y., and Nicol, D. A virtual time system for openvz-based network emulations. In *Principles of Advanced and Distributed Simulation (PADS), 2011 IEEE Workshop on* (2011), pp. 1–10.

[44] Zhou, J., Ji, Z., and Bagrodia, R. Twine: A hybrid emulation testbed for wireless networks and applications. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings* (2006), pp. 1–13.

[45] Zhou, J., Ji, Z., Varshney, M., Xu, Z., Yang, Y., Marina, M., and Bagrodia, R. Whynet: A hybrid testbed for large-scale, heterogeneous and adaptive wireless networks. In *Proceedings of the 1st International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization* (New York, NY, USA, 2006), WiNTECH '06, ACM, pp. 111–112.