A Virtual Time System for OpenVZ-Based Network Emulations

Yuhao Zheng
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
zheng7@illinois.edu

David M. Nicol
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
dmnicol@illinois.edu

Abstract*—Simulation and emulation are commonly used to study the behavior of communication networks, owing to the cost and complexity of exploring new ideas on actual networks. Emulations executing real code have high *functional* fidelity, but may not have high *temporal* fidelity because virtual machines usually use their host's clock. A host serializes the execution of multiple virtual machines, and time-stamps on their interactions reflect this serialization. In this paper we improve temporal fidelity of the OS level virtualization system OpenVZ by giving each virtual machine its own virtual clock. The key idea is to slightly modify the OpenVZ and OpenVZ schedulers so as to measure the time used by virtual machines in computation (as the basis for virtual execution time) and have Linux return virtual times to virtual machines, but ordinary wall clock time to other processes. Our system simulates the functional and temporal behavior of the communication network between emulated processes, and controls advancement of virtual time throughout the system. We evaluate our system against a baseline of actual wireless network measurements, and observe high temporal accuracy. Moreover, we show that the implementation overhead of our system is as low as 3%. Our results show that it is possible to have a network simulator driven by real workloads that gives its emulated hosts temporal accuracy.

***Keywords*—network emulation, virtual time, virtual machines**

I. Introduction

The research community has developed many techniques for studying diverse communication networks. Evaluation based on any methodology *other than* actual measurements on actual networks raises questions of fidelity, owing to necessarily simplifications in representing behavior. An effective way to accurately model the behavior of software is to actually run the software [1][5][9][10], by virtualizing the computing platform, partitioning physical resources into different Virtual Environments (VEs), on which we can run unmodified application code [19][22]. However, such emulations typically virtualize execution but not time. The software managing VEs takes its notion of time from the host system's clock, which means that time-stamped actions taken by virtual environments whose execution is multi-tasked on a host reflect the host's serialization. This is deleterious from the point of view of presenting traffic to a network *simulator* which operates in virtual time. Ideally each VE would have its own virtual clock, so that time-stamped accesses to the network would appear to be concurrent rather than serialized.

In this paper, we present a virtual time system that gives virtualized applications running under OpenVZ [24] the temporal appearance of running concurrently on different physical machines. This idea is not completely unique, related approaches have been developed for the Xen [11][12] system. Xen and OpenVZ are very different, and so are the approaches for virtualizing time. Xen is a heavy-weight system whose VEs contain both operating system and application. Correspondingly Xen can simultaneously manage VEs running different operating systems. By contrast, all VEs under OpenVZ (called "containers" in OpenVZ parlance) use and share the host operating system. In Xen virtualization starts at the operating system whereas in OpenVZ virtualization starts at the application. There are tradeoffs of course, we are interested in OpenVZ because it scales better than Xen, as OpenVZ emulation can easily manage many more VEs than can Xen. We believe we are first to introduce virtual time to OpenVZ; by doing so we are able to construct large scale models that run real application code, with rather more temporal accuracy than would be enjoyed without our modifications.

We implement our virtual time system by slightly modifying the OpenVZ and Linux kernels. The OpenVZ modifications measure the time spent in bursts of execution, stop a container on any action that touches the network, and gives one container (the network emulator) control over the scheduling of all the other containers to ensure proper ordering of events in virtual time. Modifications to the Linux kernel are needed to trap interactions by containers with system calls related to time, e.g., if a container calls `gettimeofday()`, the system should return the container's virtual time rather than the kernel's wall clock time – but calls by processes other than OpenVZ's ought to see the kernel's unmodified clock time.

Our time virtualization is not exact. However, comparison with experiments that use real time-stamped data measured on a wireless network reveal temporal errors on the order of 1ms – which is not large for this application. We also measure the overhead of our system's instrumentation and find it to be as low as 3%. In addition, our method is more efficient than the time virtualization proposed for Xen [11]. That technique simply scales real time by a constant factor, and gives each VM a constant sized slice of virtualized time, regardless of whether any application activity is happening. Necessarily, Xen VEs virtualized in time this way can only advance more slowly in virtual time than the real-time clock advances. Our approach is less tied to real time, and in principle can actually advance in virtual time faster than the real-time clock, depending on the number of containers and their applications.

The rest of this paper is organized as follow. Section II reviews related work. Sections III explain our system architecture at a high level, while Section IV provides detailed implementations. Section V evaluates our systems and gives our experimental results. Section VI concludes the whole paper and identifies future work.

II. Related Work

Related work falls into the following three categories: 1) network simulation and emulation, 2) virtualization technique and 3) virtual time systems. They are discussed one by one as follows.

A. *Network simulation and emulation*

Network simulation and network emulation are two common techniques to validate new or existing networking designs. Simulation tools, such as ns-2 [3], ns-3 [4], J-Sim [5], and OPNET [6] typically run on one or more computers, and abstract the system and protocols into simulation models in order to predict user-concerned performance metrics. As network simulation does not involve real devices and live networks, it generally cannot capture device or hardware related characteristics.

In contrast, network emulations such as PlanetLab [8], ModelNet [9], and Emulab [10] either involve dedicated testbed or connection to real networks. Emulation promises a more realistic alternative to simulation, but is limited by hardware capacity, as these emulations need to run in real time, because the network runs in real time. Some systems combine or support both simulation and emulation, such as CORE [1], ns-2 [3], J-Sim [5], and ns-3 [4]. Our system is most similar to CORE (which also uses OpenVZ), as both of them run unmodified code and emulate the network protocol stack through virtualization, and simulate the links that connect them together. However, CORE has no notion of virtual time.

B. *Virtualization technique*

Virtualization divides the resources of a computer into multiple separated Virtual Environments (VEs). Virtualization has become increasingly popular as computing hardware is now capable enough of driving multiple VEs concurrently, while providing acceptable performance to each. There are different levels of virtualization: 1) virtual machines such as VMware [20] and QEMU [21], 2) paravirtualization such as Xen [22] and UML [23], and 3) Operating System (OS) level virtualization such as OpenVZ [24] and Virtuozzo [25]. Virtual machine offers the greatest flexibility, but with the highest level of overhead, as it virtualizes hardware, e.g., disks. Paravirtualization is faster as it does not virtualize hardware, but every VE has its own full blown operating system. OS level virtualization is the lightest weight technique among these [18], utilizing the same operating system kernel (and kernel state) for every VE. The problem domain we are building this system to support involves numerous lightweight applications, and so our focus is on the most scalable of these approaches. The potential for lightweight virtualization was demonstrated by Sandia National Lab who demonstrated a one million VM run on the Thunderbird Cluster, with 250 VMs each physical server [2]. While the virtualization techniques used are similar to those of the OpenVZ system we have modified, the Sandia system has neither a network simulator between communicating VMs, nor a virtual time mechanism such as we propose.

C. *Virtual time system*

Recent efforts have been made to improve temporal accuracy using Xen paravirtualization. DieCast [11], VAN [12] and SVEET [13] modify the Xen hypervisor to translate real time into a slowed down virtual time, running at a slower but constant rate, and they call such mechanism time dilation. At a sufficiently coarse time-scale this makes it appear as though VEs are running concurrently. Other Xen-based implementations like Time Jails [14] enable dynamic hardware allocation in order to achieve higher utilization. Our approach also tries to maximize hardware utilization and keep emulation runtime short. Unlike the mechanism of time dilation, we try to advance virtual clock as fast as possible, regardless it is faster or slower than real time.

Our approach also bears similarity to that of the LAPSE [17] system. LAPSE simulated the behavior of a message-passing code running on a large number of parallel processors, by using fewer physical processors to run the application nodes and simulate the network. In LAPSE, application code is directly executed on the processors, measuring execution time by means of instrumented assembly code that counted the number of instructions executed; application calls to message-passing routines are trapped and simulated by the simulator process. The simulator process provides virtual time to the processors such that the application perceives time as if it were running on a larger number of processors. Key differences between our system and LAPSE are that we are able to measure execution time directly, and provide a framework for simulating any communication network of interest (LAPSE simulates only the switching network of the Intel Paragon).

III. System Architecture

We begin by providing an introduction of OpenVZ system, and then explain the architecture of our system.

A. *Overview of OpenVZ*

OpenVZ provides container-based virtualization for Linux [24]. It enables multiple isolated execution environments (called *Virtual Environments*, VEs, or *containers*) within a single OS kernel. It provides better performance and scalability as compared with other virtualization technologies. Figure 1 shows the architecture of OpenVZ. A virtual environment looks like a separate physical machine. It has its own process tree starting from the `init` process, its own file system, users and groups, network interfaces with IP addresses, etc. Multiple VEs coexist within a single physical machine, and they not only share the physical resources but also share the same OS kernel. All VEs have to use the same version of the same kernel.

A VE is different from a real OS. A VE uses fewer resources. For example, a newly created Ubuntu VE can have fewer than 10 processes. A VE has limited function compare with a real machine, e.g., it is prohibited from loading or unloading kernel modules inside a VE. Finally, the Linux host operating system provides all kernel services to every VE; the operating system is shared.

OpenVZ offers two types of virtual network interfaces to the VEs, one called a virtual network device (or `venet` in OpenVZ parlance) and the other called a virtual Ethernet device (or `veth` in OpenVZ parlance) [24]. A virtual network device has lower overhead, but with limited functionality, serving simply as a point-to-point connection between a VE and the host OS. It does not have a MAC address, has no ARP protocol support, no bridge support, and no possibility to assign an IP address inside the VE. By contrast, a virtual Ethernet device has slightly higher (but still very low) overhead, but it behaves like an Ethernet device. A virtual Ethernet device consists of a pair of network devices in the Linux system, one inside the VE and one in the host OS. Such two devices are connected via Ethernet tunnel: a packet goes into one device will come out from the other side.

The OpenVZ CPU scheduler has two levels. The first level scheduler determines which VE to give time slice to, according the VE's CPU priority and limit settings. The second level scheduler is a standard Linux scheduler, which decides which process within a VE to run, according to the process priorities.

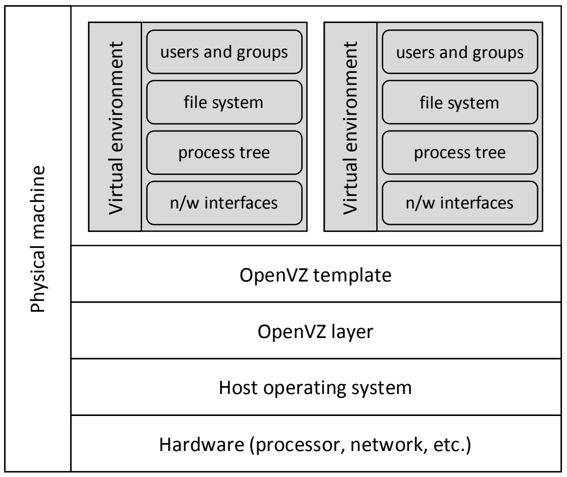


Figure 1 Architecture of OpenVZ

B. System designs

The architecture of our OpenVZ-based emulation system is illustrated by Figure 2. For a given experiment a number of VEs are created, each of which represents a physical machine in the scenario being emulated. Applications that run natively on Linux run in VEs without any modification. The sequencing of applications run on different VEs is controlled by the Simulation/Control application, which runs on host OS (or `VE0` in OpenVZ parlance). Sim/Control communicates to the OpenVZ layer to control VE execution so as to maintain temporal fidelity. For instance, a VE that is blocked on a socket read ought to be released when data arrives on that socket. Sim/Control knows when the data arrives, and so knows when to signal OpenVZ that the blocked VE may run again.

Under unmodified OpenVZ all VEs share the *wallclock* of the host computer (accessed through the shared host operating system). In our virtual time system, each VE has its own *virtual clock* (denoted as `vclk`$_i$ in Figure 2), while the host OS still uses the wallclock (denoted as `wclk`). Different virtual clocks advance separately, but all of them are controlled by the network simulator via the virtual time kernel module (`V/T module` in the figure).

Sim/Control captures packets sent by VEs and delivers them to destination VEs at the proper time ("proper time" being a function of what happens as the network is simulated to carry those packets). Sim/Control also controls the virtual times of VEs, advancing their virtual times as a function of their execution, but also blocking a VE from running, in order to prevent causal violation. For example, if a packet should arrive at a VE at virtual time *t*, but the virtual time of that VE is already *t*+1, a causal violation occurs because the application has missed the packet and may behave differently than expected. Sim/Control is responsible for stopping this VE at virtual time *t*, until the packet arrives. This is accomplished by modifying the scheduler, as we will describe in Section IV.

Sim/Control consists of two cooperating subsystems: 1) network subsystem (denoted as `N/W` in Figure 2) and 2) virtual time subsystem (denoted as `V/T`). For instance, when a packet sent by VE1 to VE2, it is captured by Sim/Control, which has to know the virtual sending timestamp of that packet in order to know when it entered the network. After the simulator determines the virtual arrival time of the packet at VE2, the simulator must ensure that VE2 has advanced far enough in simulation time to receive that packet, or that VE2 is blocked waiting for a packet, and so needs to be released to run.

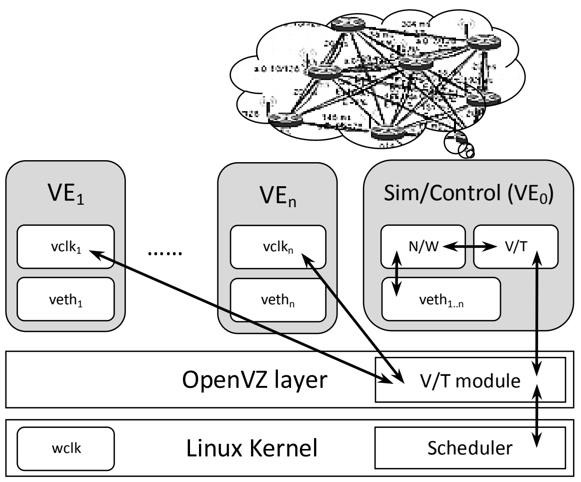


Figure 2 Architecture of network emulation with virtual time

C. Sim/Control

The Sim/Control process captures VE packets, simulates their travel within the imagined network, and delivers them to their destinations. Packet capture is accomplished using the OpenVZ virtual Ethernet device (`veth`). When an application within a VE sends a packet via its `veth` interface, the packet appears at `veth` in the host OS due to the virtual Ethernet tunneling. Sim/Control monitors all `veth` interfaces to capture all packets sent by all VEs. Similarly, when it wants to deliver a packet to a VE, it just simply sends the packet to the corresponding `veth` interface. The packet tunnels to the VE's corresponding `veth` interface, where the application receives it. The packet travel route is shown in Figure 3.

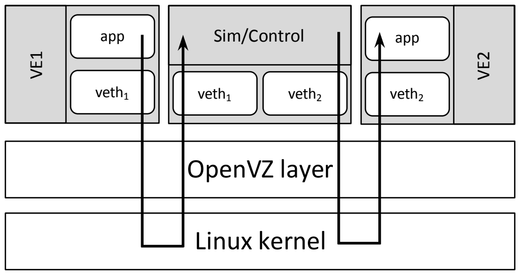


Figure 3 Packet traverse route in emulation

Sim/Control needs to cooperate with the virtual time subsystem when VEs are either sending packets or receiving packets. For example, in real system, blocking socket sends (e.g., `sendto()`) are returned after the packets have been taken care of the underlying OS. Correspondingly, in emulation, the process should perceive comparable amount of elapsed time after the call returns. This is done by trapping those system calls, suspending the VE, have Sim/Control figure out the time at which the packet is taken care, and then return control to the VE, at the corresponding virtual time. Similarly, when an application is blocked waiting to receive a packet, it should be unblocked at the virtual time of the packet's arrival. The comparison between real network operations and emulated ones is shown in Figure 4. As long as the processes perceive comparable elapsed time after system calls are returned and the network simulation gives high enough fidelity for the system measures of interest, this approach is viable. Our approach to integrating the network simulation allows us to include any one of a number of physical layer models. Detailed technical issues are discussed in Section IV.

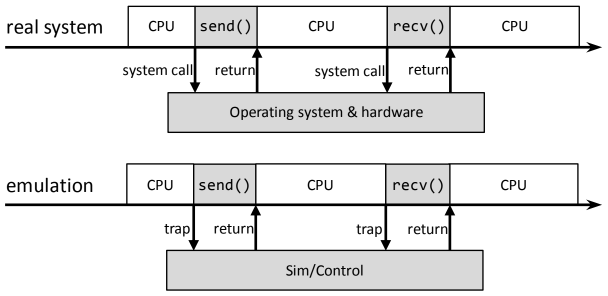


Figure 4 Real network operations vs. simulated network operations

D. Virtual time subsystem

The responsibility of the virtual time subsystem includes advancing virtual clocks of VEs and controlling the execution of VEs. From the operating system's point of view, a process can either have CPU resources and be running, or be blocked and waiting for I/O [30] (ignoring a "ready" state, which rarely exists when there are few processes and ample resources). The wall clock continues to advance regardless of the process state. Correspondingly, in our system, the virtual time of a VE advances in two ways. At the point a VE becomes suspended, the elapsed wallclock time during its execution burst is added to the virtual clock. This is shown in Figure 5.

The situation is different when the application within a VE interacts with the I/O system. Our modified OpenVZ kernel traps the I/O calls, Sim/Control determines the I/O delay and adds that to the VE's virtual clock, and then returns the I/O request to unblock the process. As shown in Figure 5, when I/O delay is accurately simulated, the virtual clock will have the same value as wall clock, and therefore the application perceives the same elapsed time. However, the real elapsed time depends on the time spent on emulating such I/O, which depends on the model and the communication load.

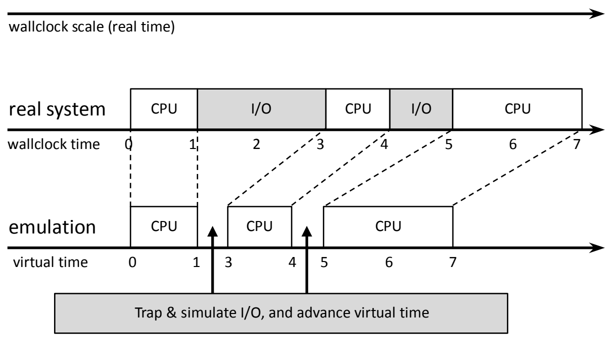


Figure 5 Wall clock time advancement vs. virtual time advancement

It is sometimes necessary to block a running VE in order to prevent casual violation. An example is when an application queries for incoming I/Os, e.g. a non-blocking socket call using `select()` [32]. Even though there may be no pending packets at that *wallclock moment*, it is possible still for a packet to be delivered with a virtual arrival time that is no greater than the virtual time *t* of the `select()` call, because the virtual clock the sending VE may be less than *t*. Therefore, when an application makes a non-blocking socket receive call at virtual time *t*, our system suspends it until we can ensure no packets can arrive with time-stamps less than or equal to *t*. On a blocking call we need to ensure that the right packet unblocks the VE, and so the same logic applies – before the VE is released at time *t*, we ensure that any packet with time stamp *t* or small has already been delivered. Implementation details are given in Section IV.

IV. IMPLEMENTATION

We next present implementation details of our virtual time system, and discuss some related issues.

A. Implementation architecture

As shown in Figure 2, virtual time management needs kernel support, therefore modification to OpenVZ kernel is necessary. We try to keep the modifications simple. The kernel implements only some primitive operations, while Sim/Control calls these operations to control sequencing of

VE execution. Sim/Control runs at user level on the host OS (`VE0`), and communicates with the kernel through new system calls we have implemented. We have chosen system calls to be the communication channel between user space and kernel because of its high efficiency. We placed Sim/Control in user space in order to keep the kernel safe, but the frequent communication between user and kernel raises the question of overheads. Section V.E discusses our measurement of this, found in our experiments to be small.

We first provide the kernel modifications in Subsection B, and then present the design of Sim/Control in Subsection III.C. Finally, we discuss the upper bound of error in Subsection D.

B. *Modifications to OpenVZ kernel*

OpenVZ scheduler. Scheduling VEs properly ensures the correctness of the emulation. To support this we modified the OpenVZ scheduler so that Sim/Control governs execution of the VEs. By making system calls to the kernel, Sim/Control explicitly gives time slices to certain VEs; a VE may run only through this mechanism.

The typical scheduling sequence of emulation is shown in Figure 6, showing how Sim/Control and VEs take turns running. We refer Sim/Control time slice together with all the subsequent VE time slices before the next Sim/Control time slice as one *emulation cycle*. At the beginning of a cycle, all VEs are blocked. In its time slice Sim/Control pushes all due events to VEs (such as packet deliveries, details in Subsection III.C), and then decides which VEs can have time slices and notifies the kernel that they may run. Causal constraints may leave any given VE blocked, but Sim/Control will always advance far enough in virtual time so that at least one VE is scheduled to run in the emulation cycle. VE executions within an emulation cycle are logically independent of each other, and so their execution order does not matter.

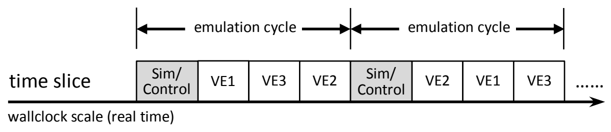


Figure 6 Scheduling of VEs and Sim/Control (example of 3 VEs)

An executing VE is suspended when either it uses up the time slice or when it makes a system call that is trapped (typically one that interacts with the communication system). Such system calls includes network send and receive calls, as discussed in Section III.C. Once a VE makes such special calls, it is blocked immediately and cannot run any more within this emulation cycle. After a VE stops, the actual time it executed will be added to its virtual clock.

This discussion summarizes *Rule #1: A VE can run only after Sim/Control has released it, and will then suspend when either its time slice is consumed, or it executes a trapped system call that interacts with the I/O subsystem.*

Trap network related system calls. We first discuss socket send calls, both blocking and non-blocking. As pointed out by Section III.C, blocking socket sends should be returned after a virtual time equivalent to the time required to transmit the packet. In unmodified OpenVZ, such system calls are returned almost immediately, as the virtual Ethernet device handles packets at an extremely high speed. However, the time elapsed in real systems depends on the underlying physical layer. It can be in the order of microseconds for some gigabit Ethernet [39], but can be as large as several milliseconds for some wireless networks. We modified OpenVZ to give Sim/Control the responsibility of returning those system calls. The VE is suspended at the point of the call, and its current virtual time is updated. Since the packet was presented to the virtual Ethernet interface it tunnels almost immediately to be reflected at its corresponding interface in Sim/Control. Therefore at the beginning of the next emulation cycle Sim/Control observes the packet and marks its send time as the virtual time of its suspended source. Once the packet departure has been fully simulated (and this may take some number of emulation cycles, depending on the network model and the traffic load on the simulator), Sim/Control will know to release the suspended sender. We likewise suspend a VE that makes a non-blocking send, but just to obtain the packet's send time. In this case Sim/Control immediately releases the sender to run again in the following emulation cycle.

Now consider socket receive calls. As discussed in Section III.D, an application that makes a socket receive call (either blocking or not) is suspended *before* it looks at the receive buffer in order to ensure that the state of the receive buffer is correct for that virtual time. The VE must wait at least until the next cycle, at which point Sim/Control will either release it to run, or not, depending on whether there is a threat of violating causality constraints. Once the VE is released to run again it looks at the receive buffer and responds to the previous socket receive call with normal semantics (possibly blocking again immediately, if it is a blocking receive that finds no packet in the buffer).

This discussion summarizes *Rule #2: a VE is always suspended upon making a network related system call.*

Other kernel modifications. Other kernel modifications are also necessary. This includes trapping `gettimeofday()` system calls and returning virtual times to the application, and basing kernel timers on virtual time. The implementations are entirely straightforward and need no further comment.

C. *VE Scheduling Control*

Sim/Control runs in user-space in the host OS. With the support of the kernel, it only needs to maintain a simple logic to control VE execution. The algorithm used is described in Algorithm 1, and it is a simple variation of a conservative parallel discrete event simulation (PDES) synchronization method [27][28]. Sim/Control maintains its own virtual clock `current_time`. Conceptually, in every time slice of an emulation cycle, Sim/Control does the following one by one: (1) buffers packets sent by VEs during the last cycle (line 10-15), (2) simulates the network and pushes due events to VEs (line 17-23), (3) decides which VEs can run in the next cycle (line 25-31), (4) advances virtual clocks if no VEs can run (line 33-38), and finally (5) yields the processor to let VEs run (line 41-42). In step (3), an event is considered "due" if its virtual time is no greater than the virtual time of the VE to which it is pushed. We will comment more on this in our section on error analysis.

As shown in Algorithm 1, the Sim/Control calls the network simulator program `nw_sim` to simulate the network (line 17 & 35); this function call needs some explanation. The first parameter is the current status of the network (stored in the `VE` data structure), a status that changes as the simulation executes. The second and third parameters indicate the desired start time and end time of simulation. The fourth parameter is a flag, explained later. The outcomes of `nw_sim` are the events to be returned to VEs (both finished transmissions and packet receptions) within the desired time interval, and they are stored back into the network status `VE`. With this information, the Sim/Control knows which system calls to return and which packets to deliver within a single time slice (line 20-21).

Algorithm 1 Logic for Controlling VE Execution

```
01 for each VE i do
02   init_ve_status(VE[i])
03 end for
04 current_time ← 0
05
06 while true do
07   wait_until_all_ves_stop()
08   last_time ← current_time
09   current_time ← min_i(VE[i].time)
10   for each VE i do
11     for each packet p sent by VE[i] do
12       p.send_time ← VE[i].time
13       buffer_packet(VE[i], p)
14     end for
15   end for
16
17   nw_sim(VE, last_time, current_time, false)
18   while true do
19     for each VE i do
20       return_due_send_calls(VE[i])
21       deliver_due_packets(VE[i])
22       fire_due_timers(VE[i])
23     end for
24
25     for each VE i do
26       VE[i].can_run ← VE[i].ready and
27           VE[i].time < current_time + δ
28     end for
29     if at least one VE can run then
30       break
31     end if
32
33     next_time ← min_j(virtual_timers[j].exp)
34     current_time ←
35         nw_sim(VE, current_time, next_time, true)
36     for each VE i do
37       VE[i].time ← max(VE[i].time, current_time)
38     end for
39   end while
40
41   release_VEs()
42   sched_yield()
43 end while
```

The fourth parameter of `nw_sim` interface is a flag which tells the simulator whether to stop when such return event *first occurs*. When the flag of `nw_sim` is set to false, network is simulated for the given time interval (e.g. line 17) and returns the virtual end-time of its execution period. When the flag is set to true, the network simulator finds the exact time of the next return event (e.g. line 35), stops and returns that time. When the emulator detects that no VEs can run, it advances its current virtual clock to the point when next event happens. Such event can be either a packet transmission or a kernel timer expiration, whichever happens first (line 33-35).

Finally, there is a tunable parameter δ in line 27 used to control how tightly virtual clocks of different VE are synchronized. In the following section we show how an application running multiple threads in a VE can have different behavior in our system that it does in a real system. The smaller δ is, the smaller the potential error of that difference can be. Our experiments use a value of 1msec, which is the minimum possible value in our current platform.

D. Error Analysis

According to Algorithm 1, an event is delivered to a VE *no earlier* than it should be, in the sense that if an event time is t, the VE notices it at a time u that is at least as large as t. Since we suspend a VE at all points where it interacts with the network, if the VE is single-threaded it will be insensitive to the difference between t and u because its behavior is not affected by the arrival at t. This is not the case however if an application is multi-threaded, as we show below.

Consider an application which consists of two threads. Thread A does CPU intensive computation, while Thread B repeatedly receives packets using blocking socket call `recvfrom()`, and calls `gettimeofday()` immediately after it receives a packet. In a Linux system, when there is an incoming packet, Thread B will be immediately be run, pre-empting Thread A. As a result, this application can get the exact time of packet arrival. This is illustrated in Figure 7.

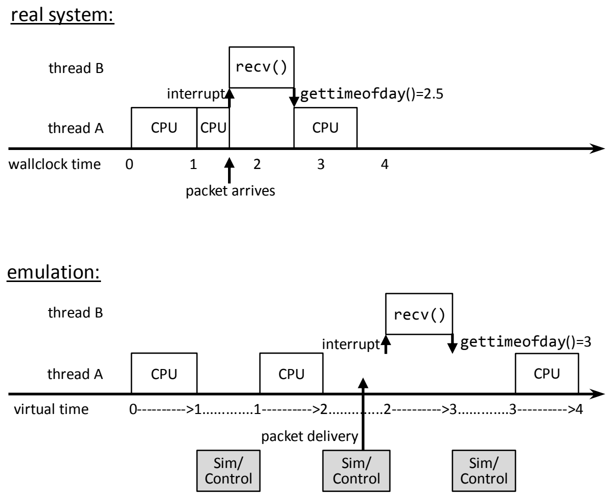


Figure 7 Error in virtual time of packet delivery

Now consider the same application running on our virtual time system. Whenever Thread B makes the socket receive call, the *whole application* – both threads – is blocked by Rule #2. After Sim/Control releases the application to run again, Thread A will take off. However, once the application gets the CPU, it uses the entire time slice because Thread A keeps doing computation. Meanwhile Sim/Control is waiting its turn and so packets are *not* delivered to wake up Thread B until that turn comes around, after the actual delivery time. In this case, the error in that delivery time can be *as most as large* the time slice the application is given to. The comparison is shown in Figure 7.

We summarize the above case as an instance of an *interrupt-like* event. The key problem are situations where in the real system a process is interrupted immediately, whereas in the emulation the interrupting event is not recognized until Sim/Control gets a time-slice. We can reduce this error by reducing the length of the time-slice, but this of course increases the overhead by increasing context switching [30]. The tradeoff between behavioral accuracy and execution speed is a common tradeoff in simulation [29]. We leave exploring such tradeoff as future work.

V. EVALUATION

This section provides our experimental results that demonstrate the performance of our virtual time system.

A. *Experimental framework*

In order to validate the emulated results, we compare them with that we obtained in [36] within the Illinois Wireless Wind Tunnel (iWWT). The iWWT [35] is built to minimize the impact of environment in wireless experiments. It is an electromagnetic anechoic chamber whose shielding prevents external radio sources from entering the chamber; and whose inner wall is lined with electromagnetically absorbing materials, which reflect minimal energy. This gives us a simulated "free space" inside the chamber, which is ideal for conducting wireless network experiments.

We run the same application as we used in [36] within our emulator. We notice the hardware difference between the machine on which we run our emulator, and the devices we used to collected data inside the chamber. Specifically, our emulator is running on a Lenovo T60 laptop with Intel T7200 CPU and 2GB RAM (if not otherwise specified), while we used Soekris Engineering net4521 [37] with mini-PCI wireless adapters plugged in as wireless nodes inside the iWWT. Due to the difference in processors, applications running on the Soekris box should observe longer elapsed times than the Lenovo laptop. However, this should not bring large error into the results, as the applications we run are I/O bound ones, i.e. the time spend on I/O is dominant while the time spend on CPU is negligible.

In the experiment inside the chamber, the wireless nodes were operating on 802.11a [38]. Correspondingly, we have an 802.11a physical layer model in our Sim/Control, which predicts packet losses and delay. Our 802.11a model implements CSMA/CA, and it uses bit-error-rate model to stochastically sample lost packets.

B. *Validating bandwidth, delay and jitter – one-link scenario*

We start with the simplest scenario with two wireless nodes and one wireless link. Packets are sent to the receiver as fast as possible using 54Mbps data rate under 802.11a, and the receiver records the timestamp of each received packet. The comparison between real trace and emulated results is shown in Figure 8 and Figure 9. We study the delay rather than throughput because the sender is sending packets of constant size, and therefore accurate delay implies accurate throughput, but not vice versa. The experiment actually persists for 10sec in real time, but we only show 20 packets for conciseness.

As shown in Figure 8, the emulated result (the 1-flow line) under our virtual time system is almost identical to the real trace, with the error within 1msec. The only difference is due to random retransmissions, which are caused by low SINR. In 802.11a, when the receiver cannot correctly decode the data frame, it will not send an ACK frame. In this case, the sender will have to retransmit that data frame, and this will approximately double the transmission time of this single packet compare with no retransmission. From the figure, we are able to tell when a retransmission happens by examining inter-packet arrival time. As retransmission is a random event, it is reasonable that our 802.11a model cannot predict retransmission for the exact packet as real trace. In general, the model predicts comparable retransmission rate.

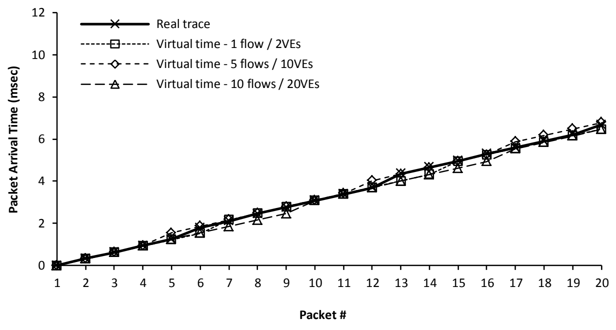


Figure 8 Packet arrival time, one link scenario, with virtual time

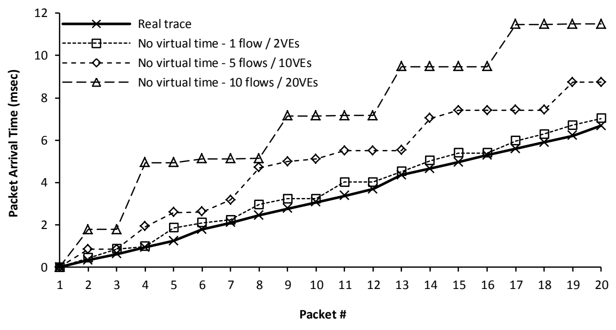


Figure 9 Packet arrival time, one link scenario, without virtual time

To demonstrate the accuracy of our system under different loads, the previous one-link scenario is replicated several times, with each replication emulated simultaneously. However, replicated links are only used to saturate the emulator, so they are independent and will not interfere with each other. This represents the scenario in which we have multiple non-interfering wireless links; each flow has the same behavior as a single flow (modulo differences in random number seeds.) In Figure 8, the 5-flow line and the 10-flow line show the results of a single flow out of the 5 or 10 simultaneous ones. Regardless of the number of flows, the behavior of each flow is identical to 1-flow case, as expected.

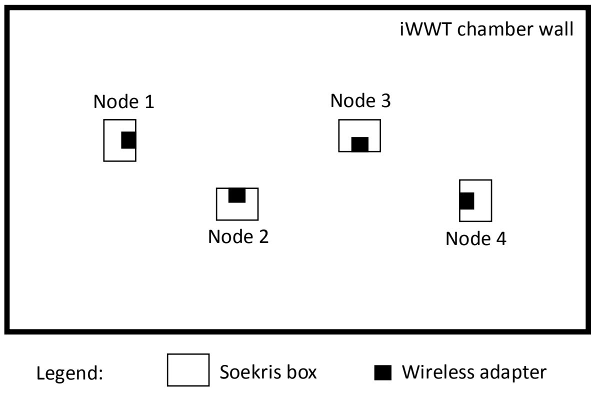
For comparison, we run the same experiment under the system without virtual time, and the results are shown in Figure 9. The 1-flow result is has only slightly larger error than that with virtual time, being caused by scheduling delay. This occurs when a VE cannot get the CPU and execute exactly on time, including (1) when the network emulator should deliver a packet to destination VE, and (2) when the destination VE should process an incoming packet. In fact, such scheduling delay also exists in virtual time implementation, but VEs will not perceive delay because their virtual clocks do not advance meanwhile. The error of not having virtual time here is as small as 1msec, but it will scale up when the number of VEs increases, as more VEs may introduce longer delay. As shown by the 5-flow line in Figure 9, the error can be as large as several milliseconds. Worse still, when the offer load is too high (e.g. the 10-flow line) for the emulator to process in real time, the error accumulates and becomes larger and larger. This is occurs because the emulator cannot run fast enough to catch up with real time.

C. *Validating bandwidth, delay and jitter – two-link scenario*

We next validate the scenario with four wireless nodes and two conflicting wireless links. Space limitations inside the iWWT prohibit study of more complex scenarios. Figure 10 shows the experimental setup. Although we only have four wireless nodes, we have six different combinations by changing source/destination pair and data rates. For example, Scenario 3 and Scenario 4 are those of heterogeneous data rate.

We use the iperf link test application [34] to test both links, and we are interested in both bandwidth and jitter (delay variation). We configure the emulator to simulate the above scenario. However, by examining the real data, we found that the link with Node 2 as sender usually has a higher throughput, regardless of its receiver. We conclude this is due to the hardware, although the four wireless adapters are of the same manufacture and same model. We capture such hardware characteristic by increasing the antenna gain of Node 2 in our network simulator. Higher antenna gain results in higher SINR, lower bit error rate (BER), lower retransmission rate, and finally higher throughput.

The results are shown in Figure 11 and Figure 12. The former shows comparison of throughput and the latter shows that of jitter. We observe a very accurate emulated throughput, with error less than 3% in most cases. On the other hand, the jitter obtained from emulated platform has a larger error, which is within 10% in most cases but which sometimes is as large as 20%. By comparing timestamps generated by the simulator and that perceived by the application, we found such error is due to the inaccuracy of the 802.11a model, *not* the virtual time system itself. As discussed before, accurate delay implies accurate throughput but not vice versa. The results here demonstrate that jitter is more difficult to model than throughput.



Scenario ID	Link 1		Link 2	
	src→dst	data rate	src→dst	data rate
1	2→1	6Mbps	3→4	6Mbps
2	2→4	6Mbps	3→1	6Mbps
3	2→1	54Mbps	3→4	6Mbps
4	2→4	54Mbps	3→1	6Mbps
5	2→1	54Mbps	3→4	54Mbps
6	2→4	54Mbps	3→1	54Mbps

Figure 10 Two-link experiment setup inside the iWWT

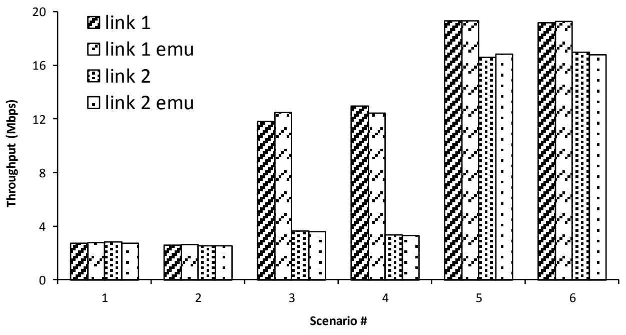


Figure 11 Two-link scenarios – throughput

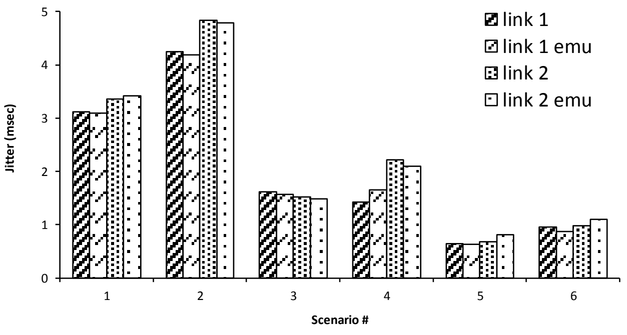


Figure 12 Two-link scenarios – jitter

D. Emulation runtime and scalability

We tested the execution speed of our system, by modifying the number of simultaneous network flows. We reuse the previous one-link scenario, in which the sender transmits packets to the receiver for 10sec using 54Mbps data rate. As in Subsection B, we replicate this link by several times, but only in order to saturate the emulator.

The emulation runtime under different loads is shown in Figure 13. When there is only one link, the runtime is only less than 2sec in real time. Compared with 10sec elapsed in virtual time, emulation runs faster because in this case the emulated I/O is faster than real I/O. As the number of flows increases, the runtime increases linearly as well. When there are more than 7 flows, the emulation runs slower than real time because of large volume of traffic. In addition, we plot the CPU usage percentage, and we find that our system can achieve high CPU usage when there is enough load. It cannot achieve 100% CPU usage because our Lenovo laptop is using dual-core CPU. As explained in Section IV.C, before the emulator process starts its time slice, it will wait until all VEs stop running. Different VEs can run on different CPUs simultaneously, but the emulator process runs on only one CPU, and meanwhile the other CPU is idle. As shown in Figure 13, increased number of flows results in higher CPU utilization. This is because as the number of flows increases and so does the number of VEs, the fraction of time during which the CPUs are saturated by VE execution also increases.

To demonstrate the scalability of our system, we tested our system on a Dell PowerEdge 2900 server with dual Intel Xeon E5405 and 16GB RAM. We rerun the above scenario but with 160 flows and 320 VEs, which finishes in 307sec (compared with < 2 sec. for 1 flow). Our current system can only run on a single machine, but with its distributed version as a future work, we will be able to emulate more network nodes. Nevertheless, we found that implementation with OpenVZ yields high VE density (320 VEs per physical machine), thanks to the light weight of OpenVZ.

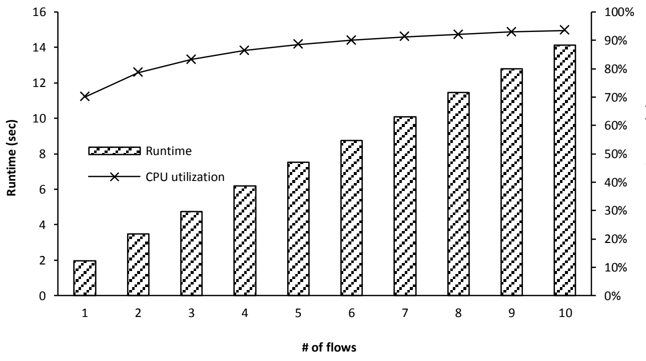


Figure 13 Emulation runtime

E. Implementation overhead

We are concerned about the implementation overhead of our system, and we measure it in the following way. We slightly modify the user-level emulator application, making it run in real time instead of virtual time so that it can run on original Linux platform. When running on original Linux, we use the `snull` virtual Ethernet tunnel module [33] to replace the virtual Ethernet devices created OpenVZ, so that both `iperf` and the emulator can be configured in the same way.

We reuse the scenario of the previous subsection, but we only use 5 flows so that our Lenovo laptop is capable to run in real time. When running in real time, either on original Linux or on original OpenVZ, the runtime of emulation is exactly 10sec. Instead of using the elapsed time of wallclock, we use the total CPU time for comparison. For instant, if the average CPU utilization is 60% during the 10-second period, the total CPU time is 6sec.

	Original Linux	Original OpenVZ	Virtual time system
Total CPU time	6.345 sec	6.478 sec	6.654 sec
Time in %	100.0%	102.1%	104.9%

Figure 14 Implementation overhead

The comparison among Linux, OpenVZ, and our virtual time system is shown in Figure 14. For the scenario we consider, we obverse 2% overhead of OpenVZ compared with original Linux. This matches the claim on OpenVZ project website, which says OS-level virtualization only has 1%-3% overhead [24]. In addition, we observe another 3% overhead of implementing virtual time, based on the original OpenVZ system. We find that such overhead is due to both frequent system calls and context switches. Implementing the emulator in kernel space might help on reducing such overhead, but we prefer to keep the kernel safe and simple. The observed 3% overhead is low, and considering the performance of OS-level virtualization, we conclude that our system is highly scalable.

Finally, we notice that such implementation overhead on OpenVZ is competitively low compared with other virtualization techniques. For instance, QEMU without a kernel acceleration module is 2X to 4X slower [21]. We are currently developing a virtual time system for QEMU as well, in order to support time virtualization to applications running on a larger number of operating systems.

VI. Conclusions and Future Work

We have implemented a virtual time system which allows unmodified application to run on different virtual environments (VEs). Although multiple VEs coexist on a single physical machine, they perceive virtual time as if they were running independently and concurrently. Unlike previous work based on Xen paravirtualization, our implementation is based on OpenVZ OS-level virtualization, which offers better performance and scalability (at the price of less flexibility). In addition, our system can achieve high utilization of physical resources, making emulation runtime as short as possible. Our implementation has only 3% overhead compared with OpenVZ, and 5% compared with native Linux. This indicates that our system is efficient and scalable.

Through evaluation, we found the accuracy of virtual time can be within 1msec, at least if an accurate network simulator is used. It is indeed the model that introduces the error, rather than the virtual time system itself. While our system can currently simulate limited type of hardware, future work can model more hardware configuration, e.g. multicore CPU, other I/O devices such as disk. It is also worth exploring the tradeoff between behavioral accuracy and execution speed in the domain of emulation using unmodified compiled code.

ACKNOWLEDGEMENTS

This material is based upon worked supported under Dept. of Energy under Award Number DE-0E00000097, and under support from the Boeing Corporation.[1]

REFERENCES

[1] J. Ahrenholz, C. Danilov, T.R. Henderson, and J.H. Kim, “Core: a real-time network emulator”, MILCOM 2008, Nov. 2008.

[2] J. Mayo, R. Minnich, D. Rudish, R. Armstrong, “Approaches for scalable modeling and emulation of cyber systems: LDRD final report”, Sandia report, SAND2009-6068, Sandia National Lab, 2009.

[3] The Network Simulator - ns-2: http://www.isi.edu/nsnam/ns/

[4] The ns-3 project. http://www.nsnam.org/

[5] A. Sobeih, W.-P. Chen, J. Hou, L.-C. Kung, N. Li, H. Lim, H.-Y. Tyan, and H. Zhang, “J-Sim: a simulation and emulation environment for wireless sensor networks”, IEEE Wireless Communications Magazine, vol. 13, no. 4, pp. 104-119, Aug. 2006.

[6] OPNET Modeler: Scalable Network Simulation: http://www.opnet.com/solutions/network_rd/modeler.html

[7] P. Riley and G. Riley, “SPADES--A distributed agent simulation environment with software-in-the-loop execution”, Winter Simulation Conference Proceedings, pp. 817–825, 2003.

[8] B. Chun., D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, “Planetlab: an overlay testbed for broad-coverage services”, SIGCOMM Computer Communication Review, 2003.

[9] A. Vahdat, K. Yocum, K.Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, “Scalability and accuracy in a large-scale network emulator”, In Proceedings of the 5th Symposium on Operating Systems Design and Implementation, 2002.

[10] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, “An integrated experimental environment for distributed systems and networks”, In Proceedings of the 5th Symposium on Operating Systems Design and Implementation, 2002.

[11] D. Gupta, K. Vishwanath, and A. Vahdat, “DieCast: testing distributed systems with an accurate scale model”, In Proceedings of the 5th USENIX Symposium on Networked System Design and Implementation (NSDI'08), San Francisco, CA, USA, 2008.

[12] P. Biswas, C. Serban, A. Poylisher, J. Lee, S.-C. Mau, R. Chadha, C.-Y. Chiang, R. Orlando, K. Jakubowski, “An integrated testbed for virtual ad hoc networks”, Proceedings of TRIDENTCOM 2009, April 6-8, 2009.

[13] M. Erazo, Y. Li, and J. Liu, “SVEET! A scalable virtualized evaluation environment for TCP”, TridentCom'09, 2009.

[14] A. Grau, S. Maier, K. Herrmann, K. Rothermel, “Time Jails: a hybrid approach to scalable network emulation”, Workshop on Principles of Advanced and Distributed Simulation (PADS), pp. 7-14, 2008.

[15] E. Weingärtner, F. Schmidt, T. Heer, and K. Wehrle, “Synchronized network emulation: matching prototypes with complex simulations”, SIGMETRICS Perform. Eval. Rev., vol. 36, no. 2, pp. 58–63, 2008.

[16] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik, “SHRiNK: a method for scalable performance prediction and efficient network simulation”, In IEEE INFOCOM, 2003.

[17] P. Dickens, P. Heidelberger, and D. Nicol, “A distributed memory LAPSE: parallel simulation of message-passing programs”, In Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS ’94), pp. 32-38, Jul. 1994.

[18] P. Padala, X. Zhu, Z.Wang, S. Singhal, and K. Shin, “Performance evaluation of virtualization technologies for server consolidation”, Technical Report HPL-2007-59, HP Labs, Apr. 2007.

[19] B. Walters, “VMware virtual platform”, Linux journal, 63, Jul. 1999.

[20] VMware virtualization software: http://www.vmware.com/

[21] F. Bellard. “QEMU, a fast and portable dynamic translator”, Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 41–46, 2005.

[22] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization”, In Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP19), pages 164–177. ACM Press, 2003.

[23] The User-Mode Linux Kernel: http://user-mode-linux.sourceforge.net/

[24] OpenVZ: a container-based virtualization for Linux. Project website available at http://wiki.openvz.org/Main_Page

[25] Virtuozzo Containers: http://www.parallels.com/products/pvc46/

[26] G. Bhanage, I. Seskar, Y. Zhang, and D. Raychaudhuri, “Evaluation of OpenVZ based wireless testbed virtualization”, Technical Report, WINLAB-TR-331, Rutgers University, 2008.

[27] R. Fujimoto, “Parallel discrete event simulation”, Communication of the ACM, vol. 33, no. 10, pp. 30-53, 1990.

[28] D. Nicol, “The cost of conservative synchronization in parallel discrete event simulations”, Journal of the ACM, vol. 40, pp. 304-333, 1993.

[29] D. Nicol, “Tradeoffs between model abstraction, execution speed, and behavioral accuracy”, European Modeling and Simulation Symposium, 2006.

[30] A. Tanenbaum, Modern Operating Systems, Third Edition, Prentice Hall, Dec. 2007.

[31] D. Bovet and M. Cesati, Understanding Linux Kernel, Third Edition, O'Reilly Media, Nov. 2005.

[32] C. Benvenuti, Understanding Linux Network Internals, O’Reilly Media, Dec. 2005.

[33] J. Corbet, A. Rubini, and G. Kroah-Hartman. Linux Device Drivers, Thrid Edition. O’Reilly Media, Feb. 2005.

[34] Iperf link test application: http://iperf.sourceforge.net/

[35] N. Vaidya, J. Bernhard, V. Veeravalli, P. R. Kumar , R. Iyer, “Illinois Wireless Wind Tunnel: a testbed for experimental evaluation of wireless networks”, SIGCOMM’05 Workshops, 2005.

[36] Y. Zheng and N. Vaidya, “Repeatability of Illinois Wireless Wind Tunnel”, Technical Report, Wireless Networking Group, University of Illinois at Urbana-Champaign, May 2008.

[37] Soekris Engineering box: http://www.soekris.com/net4521.htm

[38] 802.11a-1999 High-speed Physical Layer in the 5 GHz band. IEEE standard, 1999.

[39] D. Jin, D. Nicol, and M. Caesar, “Efficient gigabit ethernet switch models for large-scale simulation”, Principles of Advanced and Distributed Simulation (PADS), May 2010.

[40] Y. Zheng and D. Nicol, “Validation of radio channel models using an anechoic chamber”, Principles of Advanced and Distributed Simulation (PADS), May 2010.

1 This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.