A Virtual Time System for Virtualization-Based Network Emulations and Simulations

Yuhao Zheng[1] zheng7@illinois.edu

David M. Nicol[2] dmnicol@illinois.edu

Dong Jin[2] dongjin2@illinois.edu

Naoki Tanaka[1] tanaka5@illinois.edu

[1]Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA

[2]Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA

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. To enhance temporal fidelity, we designed a virtual time system for virtualization-based network emulations and simulations, such that virtual machines perceive time as if they were running concurrently in physical world. Our time virtualization is not exact: there exist temporal errors primarily due to scheduler timeslice, which is tunable in our system. We then study the tradeoff between temporal fidelity and execution speed under different lengths of timeslices, both experimentally and analytically. We demonstrate that our virtual time system is flexible such that it can achieve different level of temporal accuracy at the cost of different execution speed.

Keywords—network emulation, virtual time, virtual machines

1. Introduction

The research community has developed many techniques for studying diverse communication networks. Evaluation based on any methodologies *other than* actual measurements on actual networks raises questions of fidelity, owing to necessary simplifications in representing behavior. An effective way to accurately model the behavior of software is to actually run the software (Ahrenholz et al, 2008; Sobeih et al, 2006; Vahdat et al, 2002; White et al, 2002) using virtualization techniques, which partition physical resources into different Virtual Environments (VEs) (Walters, 1999; Barham et al, 2003). 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 the temporal appearance of running concurrently on different physical machines. This idea is not completely unique, and related approaches have been developed for the Xen (Gupta et al, 2008; Biswas et al 2009) system, while our initial work more focuses on OpenVZ (OpenVZ, 2011). OpenVZ and Xen are very different, and the approaches are different: Xen is a heavy-weight system whose VEs (or "domains" in Xen) contain both operating systems (OSes) and applications, while all VEs under OpenVZ (called "containers" in OpenVZ parlance) use and share the host operating system. To the best of our knowledge, we are the first to introduce virtual time to OpenVZ. We are interested in OpenVZ because it scales better than Xen, as OpenVZ emulation can easily manage many more VEs than Xen can. With the great scalability of OpenVZ, we are able to emulate large-scale networks, such as enterprise networks or power grid control networks.

We implement our virtual time system by slightly modifying the virtualization kernels. For instance, the OpenVZ modifications measure the time spent in bursts of execution, stop a container on expiration of its timeslice, and gives the host 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 wallclock time.

Our time virtualization is not exact: there exist temporal errors primarily due to scheduling granularities, which are tunable in our system. For example, by using 1ms scheduler timeslice, 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. Further reducing the scheduler timeslice can bring smaller temporal error, but at the cost of more frequent context switches and hence slower execution speed. For instance, emulation runs 45% slower when we reduce the timeslice from 1ms to 30μs in OpenVZ. This illustrates the tradeoff between behavioral accuracy and execution speed (Nicol, 2006), a tradeoff we quantify analytically in this paper.

We measure the overhead of our virtual time implementation over OpenVZ and find it to be as low as 3%. Due to the light

weight of OpenVZ, our system is high scalable: we are able run 320 VEs in a single commodity server machine. In addition, our method is more efficient than the time virtualization proposed for Xen (Gupta et al, 2008). 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 time in 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 virtual clocks can jump over idle periods. We show that the virtual time can actually advance faster than the real-time clock, depending on the number of containers and their applications.

The rest of this paper is organized as follows. Section 2 reviews related work. Sections 3 explains our system architecture at a high level, while Section 4 provides detailed implementations. Section 5 evaluates our systems and gives our experimental results. Section 6 concludes the whole paper and identifies future work.

2. 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.

2.1. 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 (NS-2, 2011), ns-3 (NS-3, 2011), J-Sim (Sobeih et al, 2006), and OPNET (OPNET, 2011) 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 (Chun et al, 2003), ModelNet (Vahdat et al, 2002), and Emulab (White et al, 2002) 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 (Ahrenholz et al, 2008), ns-2 (NS-2, 2011), J-Sim (Sobeih et al, 2006), and ns-3 (NS-3, 2011). 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.

2.2. 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 (VMware, 2011) and QEMU (Bellard, 2005), 2) paravirtualization such as Xen (Barham et al, 2003) and UML (UML, 2011), and 3) Operating System (OS) level virtualization such as OpenVZ (OpenVZ, 2011) and Virtuozzo (Virtuozzo, 2011). 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 (Padala et al, 2007), 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 one million VM run on the Thunderbird Cluster, with 250 VMs each physical server (Mayo et al, 2009). 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.

2.3. Virtual time system

Recent efforts have been made to improve temporal accuracy using Xen paravirtualization. DieCast (Gupta et al, 2008), VAN (Biswas et al, 2009) and SVEET (Erazo et al, 2009) 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 (Grau et al, 2008) 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 of whether it is faster or slower than real time.

Our approach also bears similarity to that of the LAPSE (Dickens et al, 1994) 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 rou-

tines 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).

3. System Architecture

We begin by providing the architecture of our OpenVZ implementation on a single machine, and then explain the notion of our virtual time system.

3.1. Overall system architecture

The architecture of our single-machine OpenVZ implementation is illustrated in Figure 1. For a given experiment a number of guest VEs are created, each of which represents a physical machine in the scenario being emulated. Each VE has its own virtual clock and virtual network interface, acting like a real physical machine. Applications that run natively on Linux run in VEs without any modifications. The whole emulation progress is controlled by the Simulation/Control application (Sim/Control for short hereinafter), which runs on VE0 (the host VE). This is accomplished by modifying the OpenVZ system, such that Sim/Control can fully take control of the emulation through some special APIs. The main responsibility of Sim/Control is to maintain temporal fidelity. It not only controls the running sequence of all VEs, but also captures and delivers network packets at proper time, as explained correspondingly as follows.

The sequence of applications run on different VEs is controlled by the Sim/Control. Sim/Control communicates with the OpenVZ layer through the APIs to control VE execution so as to maintain temporal fidelity. For instance, a VE with virtual clock running too far ahead is blocked until other VEs catch up, to prevent causal violation. Sim/Control monitors all the virtual clocks, and so knows when to signal OpenVZ that the blocked VE may run again. The virtual clock of a VE is advancing when it is executing on the CPUs. When a VE is not running or being blocked, its virtual clock will not advance, unless Sim/Control sets it forward explicitly, and this is particularly useful to jump over idle waiting.

Sim/Control also 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). In our design, all the packets sent by VEs are received by Sim/Control, and Sim/Control knows the sending timestamps of all packets so that it can determine whether and when these packets will show up at their destinations. Packet delivery time is predicted by the network simulator module of Sim/Control,

and it depends on the network scenarios. As Sim/Control knows the virtual times of all VEs, it knows when to deliver a packet to a destination VE, such that the VE will not see the packet before it ought to. A blocked VE can be released only when the Sim/Control can ensure no packets will arrive within a small future period, in regards to the virtual time of that VE. Our naive implementation requires tight synchronization among all VEs. However, with look ahead in conservative parallel discrete event simulation (PDES) (Fujimoto, 1990; Nicol, 1993), VEs may have relatively loose synchronization and therefore performance can be improved. We leave this exploring look ahead as future work.

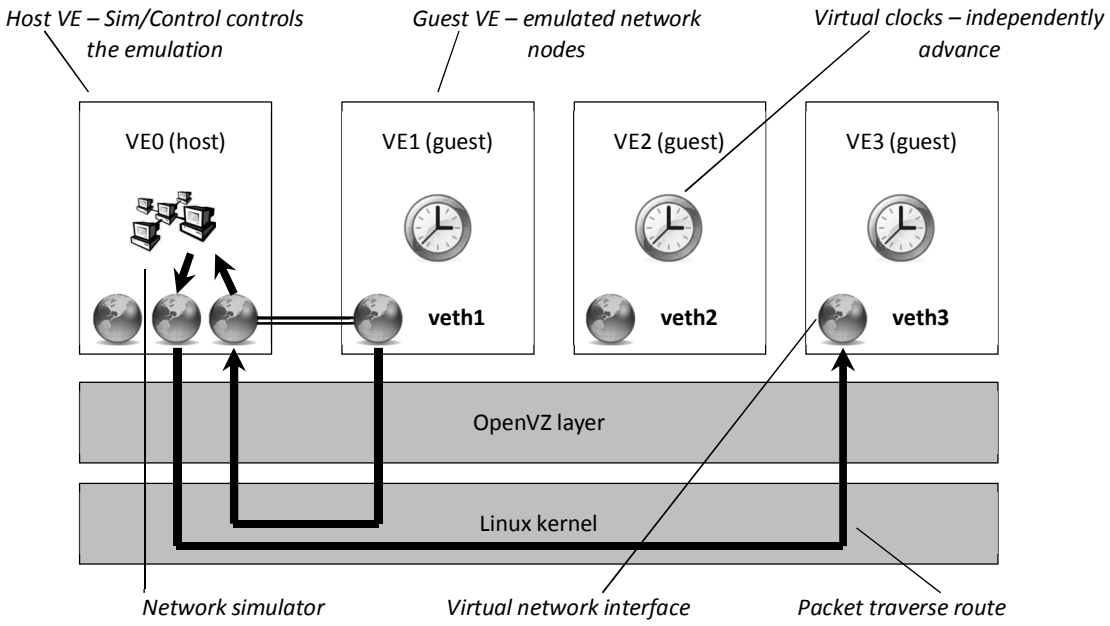

Figure 1 System architecture of OpenVZ implementation on single machine

3.2. Notion of virtual time

In order to make VEs perceive virtual time as they were running concurrently on different physical machines, we first need to understand the way how time advances in physical world. From operating system's point of view, a process can either have CPU resources and be running, or be blocked and waiting for I/O (Benvenuti, 2005) (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. When a VE is having the CPU and is running, its virtual clock keeps advancing in the same speed as the wallclock. This is the same as real world. On the other hand, when the VE is waiting for an I/O request, and such I/O request should be a simulated one (e.g. network requests), Sim/Control needs to make the VE perceives time in the same way as real world. Specifically, while waiting for I/O, the VE is suspended and therefore

its virtual clock is not advancing. Instead, Sim/Control captures the I/O request, simulates it, and returns it to the VE. Then the Sim/Control adds the simulated I/O time to the VE's virtual clock, and release the VE to let it run again. Consequently, the VE perceives virtual time as if it were running in real world. Such notion of virtual time is shown in Figure 2.

Note that simulated I/O time is normally irrelevant to the wallclock time. The actual (wallclock) time it takes to simulate an I/O request can be either faster or slower than real (wallclock), depending on the model complexity and the simulator.

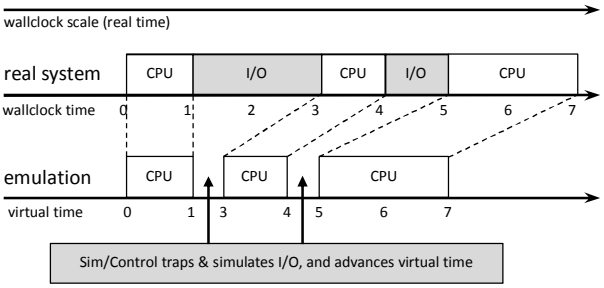

Figure 2 Wall clock time advancement vs. virtual time advancement

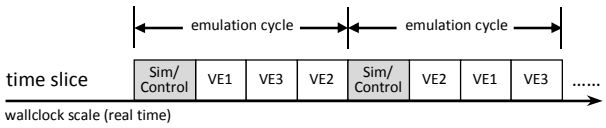

Figure 3 Scheduling of VEs and Sim/Control (example of 3 VEs)

4. Implementation

In this section, we present implementation details of our virtual time system, and analyze the effects of variable timeslices.

4.1. Timeslice-based implementation

To accomplish the virtual time notion mentioned above, we employ a timeslice-based implementation. We modify the OpenVZ scheduler so that Sim/Control governs execution of the VEs. By calling certain APIs, Sim/Control explicitly gives timeslices to certain VEs; a VE may run only through this mechanism. The typical scheduling sequence of emulation is shown in Figure 3, 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 timeslice, Sim/Control does the following: 1) collects all events made by VEs in the previous cycle (such as sent packets), 2) pushes all due events to VEs (such as packet deliveries, and timer expiration signals), and finally 3) decides which VEs can have timeslices in the current cycle. 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 the same emulation cycle are logically independent, so their execution order does not matter. Furthermore, we can utilize parallelism by assigning different VE timeslices on different processors.

It is very important to see that due to practical limitations, a VE timeslice cannot be of arbitrary length, it must be a multiple of the time between hardware timer interrupts. For example, the timer interrupts of Linux 2.6 kernel on modern machines can be as large as 1000Hz, in which case the timeslice must be integer number milliseconds. Although we can raise the frequency of timer interrupts and make the timeslice granularity smaller, it still cannot be an arbitrary value. Once the scheduler gives a timeslice to a VE, there is no way to stop the VE from consuming up the whole timeslice before the VE yields. Furthermore, the Sim/Control cannot push events (i.e. network packets) to the VE while the VE is running, simply because Sim/Control does not have the CPU. This means Sim/Control is sometimes unable to accurately deliver an event to a VE, and this introduces temporal errors. We next analyze such temporal errors.

4.2. Error analysis

We analyze the temporal accuracy of the timeslice-based mechanism mentioned above, by considering 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 waken up, pre-empting Thread A. As a result, this application can always obtain the exact time of packet arrival (ignoring the noise introduced by the OS and other processes). This is illustrated in Figure 4.

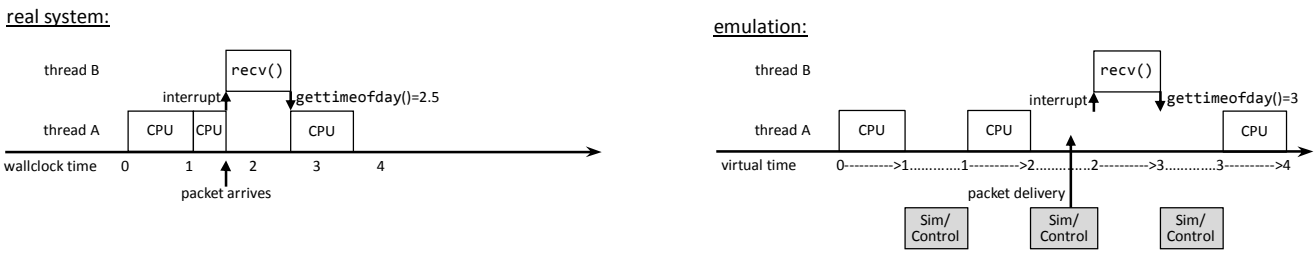

Figure 4 Error in virtual time of packet delivery

Now consider the same application running on our virtual time system, assuming the scheduler granularity is 1ms. When there are no incoming packets, Thread B is blocked, but Thread A is always ready. After Sim/Control releases the VE, 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* the timeslice the application is given to. The comparison is shown in Figure 4.

We summarize the above case as an instance of an *interrupt-like* event. The key problem is the situation 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 timeslice. As explained by the next subsection, our system can reduce this error by reducing the length of the timeslice, but this of course increases the overhead by increasing context switches (Tanenbaum, 2007). The tradeoff between behavioral accuracy and execution speed is a common tradeoff in simulation (Nicol, 2006).

4.3. Variable Timeslice

Our virtual time system can reduce the temporal error by using smaller timeslices. But as explained above, the minimal allowed timeslice is subject to the frequency of hardware timer interrupts. To achieve smaller timeslices, we need to raise the frequency of timer interrupts, i.e. the HZ value in Linux kernel. For example, by raising the HZ value from 1000 to 4000, the smallest allowed timeslice is 250μs, rather than 1ms. Actually, with HZ=4000, any timeslice length of n*250μs is allowed, where n is an integer. However, changing the HZ value has some side effects to the Linux kernel, which must be dealt with for the kernel to work properly. Such side effects are mainly due to counter overflows or integer operation overflows (Andrews, 2003), as some Linux kernel developers did not anticipate that the HZ value might be set so large.

Higher HZ frequency and smaller timeslice has overhead, which comes from at least two sources: 1) more frequent timer interrupt handlings, and 2) more frequent context switches. We model the extra time consumed by each timer interrupt as T_{int}, and model that consumed by each context switch as T_{CS}. Let TS be the length of a scheduler timeslice. Then the ratio of time spent actually working during a timeslice to the length of that timeslice (i.e., system efficiency) is

$$\rho = \frac{TS - T_{CS} - k \cdot T_{int}}{TS} = 1 - T_{CS} \cdot \frac{HZ}{k} - HZ \cdot T_{int}$$

where $TS = k/HZ$, making k the total number of timer interrupts within one timeslice of length TS. Observe that for fixed HZ, system efficiency is an increasing concave function of k (i.e., increasing TS), which suggests (and will be verified) that increasing TS from very small values will have the largest positive impact on efficiency, after which efficiency approaches an asymptote of $1-HZ*T_{int}$. As TS increases, accuracy decreases linearly. All of this means that for a given accuracy constraint, e.g., no error greater than E, we seek to maximize the expression above subject to $k/HZ \leq E$. By concavity, this occurs when k=1, and HZ=1/E. Section 5.6 provides experimental validation which shows that the above formula fits our measurements on real hardware quite well.

5. Evaluation

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

5.1. Experimental framework

In order to validate the emulated results, we compare them with that we obtained in (Zheng and Vaidya, 2008) within the Illinois Wireless Wind Tunnel (iWWT). The iWWT (Vaidya et al, 2005) 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 (Zheng and Vaidya, 2008) 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 Core 2 Duo T7200 CPU and 2GB RAM (if not otherwise specified), while we used Soekris Engineering net4521 (Soekris, 2011) 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 those on 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 spent on I/O is dominant while the time spent on CPU is negligible.

In the experiment inside the chamber, the wireless nodes were operating on 802.11a (IEEE, 1999). 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 the bit-error-rate model to stochastically sample lost packets.

5.2. Validating bandwidth and delay – single-threaded application

We start with the simplest scenario with two wireless nodes and one wireless link, and we use single-threaded receiver applications. 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. Then we run the same scenario on our testbed. To study the temporal accuracy under different loads, we replicate the single wireless link for many times. Replicated links are only used to saturate the emulator, and they are independent (no interference) so that they are expected to behave identically. The comparison between real trace and emulated results is shown in Figure 5 and Figure 6.

As shown in Figure 5, regardless of the number of replicated links, the emulated results under our virtual time system are almost identical to the real trace, except for some random packet losses. The temporal error is within 1msec, and it is not large for this application. As retransmission is a random event, it is entirely reasonable that our 802.11a model cannot predict retransmis-

sion for the exact packet as real trace. In fact, it is mainly the channel model that brings such difference but not the virtual time system itself.

On the other hand, Figure 6 shows an increasing error while the number of replicated links increases. Such error is due to physically serial executions of multiple VEs, but the VEs still read the wallclock time without eliminating the interference of other VEs. The error scales up when the number of VEs increases, as more VEs introduce longer delay. 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 occurs because the emulator cannot run fast enough to catch up with real time.

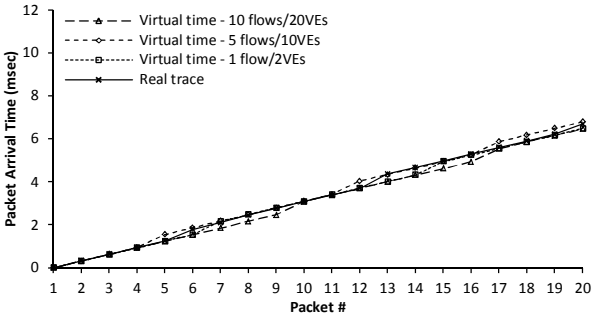

Figure 5 Packet arrival time, single-threaded application, *with* virtual time

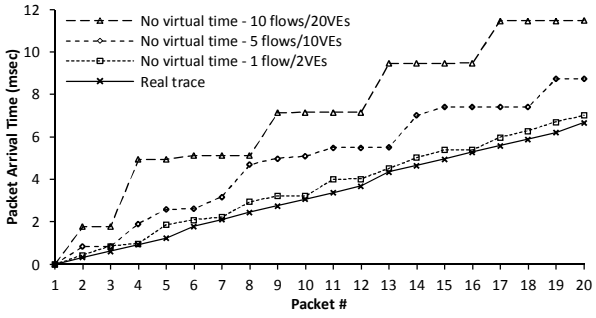

Figure 6 Packet arrival time, single-threaded application, without virtual time

5.3. Validating bandwidth and delay – multi-threaded application

We next test a multi-threaded receiver application under different length of timeslices. As a single-threaded receiver application will block and wait for incoming packets, its virtual clock will not advance while blocking. Therefore, the length of timeslice does not matter for the single-threaded case. However, consider a multi-threaded receiver application as defined in Section 4.2, the receiver application will use up the whole timeslice because of the computation thread.

Figure 7 shows the packet arrival time of a multi-threaded receiver application under various timeslices. For best comparison, we modify the network simulator to eliminate the randomness of packet losses, i.e. packet losses and retransmissions now occur at exactly the same packet # of each run. As we can see from Figure 7, the larger the timeslice, the more the results deviate from the real trace. In 802.11a, the normal transmission time of a 1470B packet under 54Mbps is around 300μs. For timeslice larger than a packet transmission time (i.e. 400μs, 800μs, and 1600μs), we observe "packet stacking" at the receiver side. This is because the Sim/Control can only deliver packets to the VE in its own Sim/Control timeslices, and there is always more than one packet due at a time because of large VE timeslices.

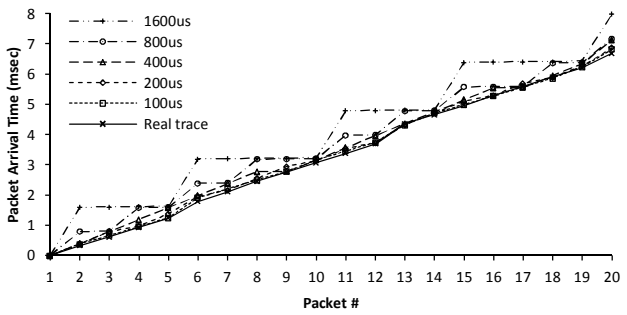

Figure 7 Packet arrival time, multi-threaded application, *with* virtual time

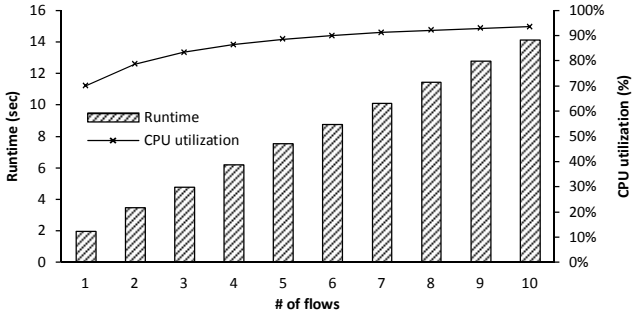

Figure 8 Emulation runtime and scalability

5.4. 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 5.2, 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 8. As the number of links increases, the runtime increases linearly. Here we show the "best effort" time advancement mechanism of our virtual time system: it will quickly jump over idle waiting rather than wasting CPU resources. When there are fewer than 6 links, the emulation runs faster than real time (10sec), because the machine is capable to handle the emulation that fast without sacrificing fidelity. 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 did not achieve 100% CPU utilization just because our simple channel model does not exploit multicores, and hence one CPU is idle during the Sim/Control timeslice (recall that our machine has 2 CPUs). We may achieve even higher CPU utilization with more complex scenario and the parallelism of S3F (Nicol et al, 2011), the next generation Scalable Simulation Framework (SSF) (SSF, 2011).

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 reuse 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), and this benefits from the light weight of OpenVZ.

5.5. Implementation overhead

We are concerned about the implementation overhead of our system, and we measure actual consumed CPU time among original Linux, OpenVZ Linux, and OpenVZ Linux with our virtual time modifications. As shown in Figure 9, we reuse the above

5-link scenario, and find that OpenVZ Linux introduces a 2.1% overhead to original Linux, while our virtual time modification brings another 2.8% overhead based on OpenVZ (with default 16ms OpenVZ scheduler timeslice). The observed ~3% overhead is low, and considering the performance of OS-level virtualization, we conclude that our system is highly scalable.

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

Figure 9 Implementation overhead

5.6. Tradeoff between fidelity and speed

Finally, we explore the tradeoff between accuracy and speed by tuning the scheduler granularity. As explained in Section 4.2, smaller scheduler granularity can reduce temporal error, but at the cost of more frequent context switches and hence slower execution speed. We change the scheduler timeslice in our OpenVZ implementation, and measure the maximum network traffic process rate in *real time*. Such process rate reflects the speed of the system: the higher the rate, the faster the emulation runs. The result is shown in Figure 10. We observe a 45% overhead when we reduce the timeslice from 1ms to 30μs.

On the other hand, as shown in Figure 11, we also find the analytical result we get from Section 4.3 fits the experimental results roughly, with the parameters being properly chosen (T_{int}=6μs, T_{CS}=8μs, and let throughput=ρ*970Mbps). As we can see from the analytical results, overhead increases convexity as TS gets small. When the timeslice is already small, any further reducing it will cause significant overhead, e.g. 60μs to 30μs. However, as we have seen already, for a given error constraint we can choose a timeslice that maximizes efficiency subject to that constraint.

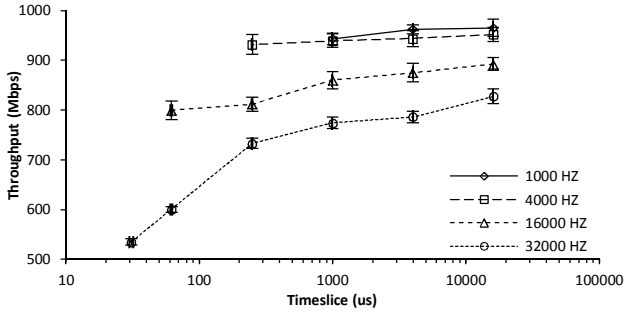

Figure 10 Emulation speed under various timeslice – *experimental* results

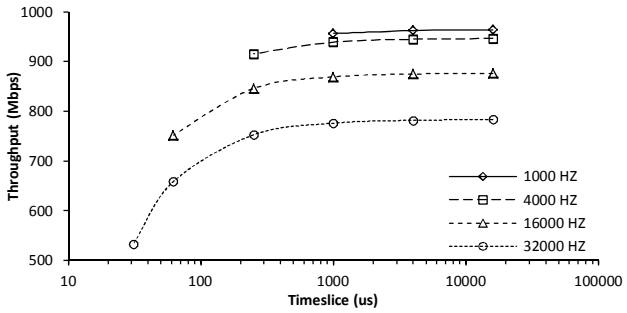

Figure 11 Emulation speed under various timeslice – *analytical* results

6. 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. Our implementation is based on OpenVZ OS-level virtualization, which offers the best performance and scalability (at the price of less flexibility), compared with other virtualization technologies like Xen and QEMU. 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 1ms, at least if an accurate network simulator is used. It might be indeed the simulator that introduces the error, rather than the virtual time system itself. If not from the simulator, the temporal error can be further reduced by having smaller scheduler timeslices, but at the cost of slower execution speed. This is the common tradeoff between behavioral accuracy and execution speed in simulation domain. We observe a 45% slower execution speed when we reduce the time slice from 1ms to 30μs.

While our current implementation focuses on OpenVZ on single server, future work includes making the emulation distributed. It is also worth porting our virtual time system to other virtualization platforms, such that more OS diversity can be embraced.

Acknowledgements

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

References

Ahrenholz J, Danilov C, Henderson T R, and Kim J H (2008). Core: a real-time network emulator. *Proceedings of the 2008 International conference for military communications* (MILCOM'08).

Andrews J (2003). Linux: Running At 10,000 HZ: http://kerneltrap.org/node/1766

Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, and Warfield A (2003). Xen and the art of virtualization. *Proceedings of the 9th ACM symposium on Operating Systems Principles* (SOSP19), pages 164–177. ACM Press, 2003.

Bellard F (2005). QEMU, a fast and portable dynamic translator. *Proceedings of the USENIX Annual Technical Conference*, FREENIX Track, pp. 41–46.

Benvenuti C (2005). *Understanding Linux Network Internals*, O'Reilly Media, Dec. 2005.

Bhanage G, Seskar I, Zhang Y, and Raychaudhuri D (2008). Evaluation of OpenVZ based wireless testbed virtualization. Technical Report, WINLAB-TR-331, Rutgers University, 2008.

Biswas P, Serban C, Poylisher A, Lee J, Mau S-C, Chadha R, and Chiang C-Y (2009). An integrated testbed for virtual ad hoc networks. *Proceedings of the 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities Internet*, pp. 1-10.

Chun B, Culler D, Roscoe T, Bavier A, Peterson L, Wawrzoniak M, and Bowman M (2003). Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review, 33(3)*.

Dickens P, Heidelberger P, and Nicol D (1994). A distributed memory LAPSE: parallel simulation of message-passing programs. *Proceedings of the 8th Workshop on Parallel and Distributed Simulation* (PADS '94), pp. 32-38.

Erazo M, Li Y, and Liu J (2009). SVEET! A scalable virtualized evaluation environment for TCP. *Proceedings of the 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities* (TridentCom'09).

Fujimoto R (1990). Parallel discrete event simulation. *Communication of the ACM*, vol. 33, no. 10, pp. 30-53.

Grau A, Maier S, Herrmann K, Rothermel K (2008). Time Jails: a hybrid approach to scalable network emulation. *Proceedings of the 22th Workshop on Principles of Advanced and Distributed Simulation* (PADS'08), pp. 7-14.

Gupta D, Vishwanath K, and Vahdat A (2008). DieCast: testing distributed systems with an accurate scale model. *Proceedings of the 5th USENIX Symposium on Networked System Design and Implementation* (NSDI'08).

IEEE (1999). 802.11a-1999 High-speed Physical Layer in the 5 GHz band. IEEE standard.

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

Nicol D (1993). The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM*, vol. 40, pp. 304-333.

Nicol D (2006). Tradeoffs between model abstraction, execution speed, and behavioral accuracy. *European Modeling and Simulation Symposium*, 2006.

Nicol D, Jin D, and Zheng Y (2011). S3F: The Scalable Simulation Framework Revisited. *Proceedings of the 2011 Winter Simulation Conference* (WSC'11).

NS-2 (2011). The Network Simulator - ns-2: http://nsnam.isi.edu/nsnam/index.php/Main_Page

NS-3 (2011). The ns-3 project. http://www.nsnam.org/

OpenVZ (2011). OpenVZ: a container-based virtualization for Linux. Project website available at http://wiki.openvz.org/Main_Page

OPNET (2011). OPNET Modeler: Scalable Network Simulation: http://www.opnet.com/solutions/network_rd/modeler.html

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

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

Soekris (2011). Soekris Engineering box: http://www.soekris.com/

SSF (2011). Scalable Simulation Framework (SSF): http://www.ssfnet.org

Tanenbaum A (2007). *Modern Operating Systems*, Third Edition, Prentice Hall, Dec. 2007.

UML (2011). The User-Mode Linux Kernel: http://user-mode-linux.sourceforge.net/

Vahdat A, Yocum K, Walsh K, Mahadevan P, Kostić D, Chase J, and Becker D (2002). Scalability and accuracy in a large-scale network emulator. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (OSDI'02).

Vaidya N, Bernhard J, Veeravalli V, Kumar P R, Iyer R (2005). Illinois Wireless Wind Tunnel: a testbed for experimental evaluation of wireless networks. *Proceeding of the 2005 ACM SIGCOMM workshop on experimental approaches to wireless network design and analysis*, pages 64-69.

Virtuozzo (2011). Virtuozzo Containers: http://www.parallels.com/products/pvc46/

VMware (2011). VMware virtualization software: http://www.vmware.com/

Walters B (1999). VMware virtual platform. *Linux journal*, 63, Jul. 1999.

White B, Lepreau J, Stoller L, Ricci R, Guruprasad S, Newbold M, Hibler M, Barb C, and Joglekar A (2002). An integrated experimental environment for distributed systems and networks. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (OSDI'02).

Zheng Y and Nicol D (2010). Validation of radio channel models using an anechoic chamber, *Proceedings of the 24th Principles of Advanced and Distributed Simulation* (PADS'10).

Zheng Y and Nicol D (2011). A Virtual Time System for OpenVZ-Based Network Emulations. *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation* (PADS'11).

Zheng Y and Vaidya N (2008). Repeatability of Illinois Wireless Wind Tunnel. Technical Report, Wireless Networking Group, University of Illinois at Urbana-Champaign, May 2008.