International Journal of High Performance Computing Applications
http://hpc.sagepub.com/

High-performance emulation of heterogeneous systems using adaptive time dilation

Hee Won Lee, Mihail L Sichitiu and David Thuente

International Journal of High Performance Computing Applications published online 17 October 2014
DOI: 10.1177/1094342014554789

The online version of this article can be found at:
http://hpc.sagepub.com/content/early/2014/10/16/1094342014554789

Published by:
SAGE
http://www.sagepublications.com

Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–18
© The Author(s) 2014
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342014554789
hpc.sagepub.com
SAGE

High-performance emulation of heterogeneous systems using adaptive time dilation

Hee Won Lee, Mihail L Sichitiu and David Thuente

Abstract
Building a testbed for evaluating the performance of large-scale heterogeneous systems can be costly and inefficient. Emulation is often used to evaluate the performance of a system in a controlled environment. Time dilation allows virtual machines (VMs) to emulate higher performance than that of their physical machine. We present an approach using adaptive time dilation to emulate large-scale distributed systems composed of heterogeneous machines and Operating Systems (OSs). In our implementation, VMs are globally synchronized. To evaluate our system, distributed VMs running Linux, Windows, FreeBSD, and Junos are emulated on general-purpose physical machines.

Keywords
Emulation, time dilation, virtual time, virtualization, virtual machines, testbeds, heterogeneous systems, distributed applications

1 Introduction

Distributed systems rely on the collaboration of a large number of nodes. Due to complex interactions, their performance without prototypes is difficult to evaluate. Diverse techniques, such as theoretical analysis, simulations, testbed implementations, and emulations, have been used for evaluating their performance.

Theoretical analysis provides elegant solutions for relatively simple systems (Kleinrock, 1975; Bertsekas and Gallager, 1992). As long as underlying assumptions are reasonably satisfied, the results of the analysis often offer exceptional insights that cannot be obtained through any other methods. However, theoretical analysis frequently requires simplifying assumptions, which may make meaningful analysis extremely difficult (Bertsekas and Gallager, 1992). Compared to theoretical analysis, simulations allow for reasonably convenient testing, with relatively complex scenarios, while avoiding overly simplistic assumptions. However, the accuracy of the simulation results is heavily dependent on the accuracy of the models employed in the simulation. Subtle changes of parameter values in a simulation model may easily render many simulation results invalid. Furthermore, the vast majority of simulation experiments are not repeatable (Pawlikowski et al., 2002), and the results of each simulation package for a common scenario can be very different from one another or from real testbed results (Lucio et al., 2003). Although testbed measurements offer the most accurate and realistic results, building a testbed is often too expensive for large systems.

Emulation has been used often over the past decade because emulation avoids building costly real testbeds and is more realistic than simulation. Furthermore, to emulate the system there is no need to code a model for the system. Several large testbeds such as PlanetLab[1] (Chun et al., 2003), Emulab,[2] DETERlab,[3] and ModelNet,[4] which use emulation techniques or significantly depend on them, have been widely used by researchers.

Emulation enables one physical host (PH) to emulate multiple virtual hosts (VHs). An unmodified Operating System (OS) can run on a hypervisor, and unmodified applications, in turn, can be executed on the unmodified OS. Emulation based on VHs is useful to make efficient use of computing resources because VHs can be deployed to maximize the overall utilization of physical

North Carolina State University, Raleigh, NC, USA

Corresponding author:
Hee Won Lee, Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA.
Email: hlee17@ncsu.edu

resources. However, the results of emulation are compromised if PHs run out of resources.

In order to create the illusion of increased resources in an emulation system based on VHs, an approach to scaling CPU, network, and disk resources was proposed in DieCast (Gupta et al., 2008). Time scaling, which allows virtual time to pass at a rate different from real time, has also been studied with different methodologies by several research groups (Bergstrom et al., 2006; Grau et al., 2008; Weingärtner et al., 2008). *Time dilation* is a technique to slow the passage of virtual time (from the perspective of a VH) by a specified factor, which is referred to as *time dilation factor* (TDF) (Gupta et al., 2005). With time dilation, physical resources appear TDF times faster (i.e. higher performance is emulated). In VMs, one second of virtual time passes for every TDF seconds of real time. Therefore, time dilation enables empirical evaluation at CPU speeds that are not currently available from production hardware and larger system emulation on fewer PHs.

DieCast uses a constant TDF, which is suboptimal for systems with dynamic loads. In contrast, network emulation using adaptive virtual time (Grau et al., 2009) dynamically adjusts the clock rate to current system loads. The approach in Grau et al. (2009) uses the concept of epoch-based virtual time (Grau et al., 2008) where the experiment is divided in epochs, each of which has a constant TDF. Another paper (Bridges et al., 2012) also recognizes dynamic time dilation as a key component of future high-performance computing systems.

The emulation in Grau et al. (2009) uses OpenVZ[5] (Zheng and Nicol, 2011), an OS-level virtualization, allowing scalability to a few thousand nodes. Therefore, applications that can be tested in Grau et al. (2009) are limited to a single OS. In this paper, we propose an emulation system able to test heterogeneous systems consisting of distributed nodes running different OSs. Many Internet applications and services are supported in different OSs, especially Windows, Mac OS X (based on FreeBSD), and Linux, and are interconnected by routers, which are operated by yet different OSs such as the Cisco IOS and Juniper Junos. Our proposed emulation system can be used to test systems comprising heterogeneous OSs. To emulate heterogeneous OSs, we use Kernel-based Virtual Machine (KVM)[6] which supports full virtualization. Dynamically adapting the scaling factor to system loads enables our system to accommodate a large number of virtual machines (VMs) at scale with limited resources. To allow for different types of unmodified OSs we use a hypervisor-level abstraction for virtual time.

In our evaluation, a video streaming service was chosen to create realistic and scalable system loads on each OS. This choice allows us to test an application involving multiple operating systems. Streaming traffic has the desirable property that it can linearly increase system loads over multiple PHs.

The remainder of the paper is organized as follows. In Section 2, we first discuss requirements for testing heterogeneous distributed systems. Section 3 presents our approach for virtual time for the dynamic time dilation and fast synchronization algorithm. Section 4 introduces our implementation for time adaptation. In Section 5, we present the evaluation of heterogeneous systems. Finally, related work in Section 6 is followed by our conclusions in Section 7.

2 Emulating heterogeneous systems

In order to build a virtual testbed for heterogeneous systems such that diverse OSs can be used, VMs must be fully isolated. To allow VMs to run unmodified OSs, the control of time dilation must be performed at the level of the hypervisor.

2.1 Full virtualization

Previous emulation systems avoided using full virtualization in order to use available resources efficiently because scalability of emulated systems was more important than complete isolation among VMs. For example, PlanetLab (Chun et al., 2003) is based on OS-level virtualization that uses Linux-VServer,[7] which creates many independent containers under a common Linux kernel. OpenVZ, which also adopts OS-level virtualization, was employed to create virtual nodes in EMULAB, DeterLab, NETplace (Grau et al., 2010), NETbalance (Grau et al., 2011), and the other network emulation studies (Grau et al., 2008, 2009; Zheng and Nicol, 2011). Lightweight virtual nodes thus have been widely used for scalable network emulations, but have the limitation that they all support a single type of OS. When emulating heterogeneous systems that include diverse OSs, however, full virtualization is required. Each VM should have its own hardware resources including CPUs, memory, file systems, network devices, etc., such that the VMs can be completely isolated from one another.

VMware ESX,[8] Oracle VirtualBox,[9] KVM, and Xen hypervisor[10] are widely deployed hypervisors that support full virtualization. VMware ESX is under proprietary license and thus modifications of the hypervisor are limited. A common choice of hypervisor for most open platforms is Xen. However, for high-performance computing environments, the KVM hypervisor is recommended as the optimal choice (Younge et al., 2011). Xen's performance lags considerably behind either KVM or VirtualBox (Younge et al., 2011). Moreover, in comparison with Xen, the main advantage of KVM is that each guest runs as a separate process within a host OS. This allows a user to manage and control the

VM inside the host through many OS facilities such as shared memory, interprocess communications (IPCs), signals, etc. Hence, when controlling VMs, KVM does not depend exclusively on the interfaces offered by the hypervisor.

2.2 Abstraction layer of time control

A general-purpose computer offers several hardware time sources such as programmable interval timer (PIT), advanced programmable interrupt controller (APIC), advanced configuration and power interface (ACPI) timer, real-time clock (RTC), high precision event timer (HPET), and time-stamp counter (TSC). An OS that runs on a physical machine commonly keeps track of time by counting interrupts from the hardware timer or reading the time-stamp counter, RDTSC.

While booting, an OS finds all the clock sources available and uses one of them. The preferred clock source is TSC since it is a precise and reliable clock source, but if it is not available, HPET is the second-best option. In Linux systems, for example, TSC is chosen as a primary clock source and if TSC is unavailable or becomes unstable, HPET is used instead. In the absence of TSC and HPET, other options include the ACPI power management timer (ACPI_PM), PIT, and RTC. Since PIT and RTC have low resolution, they are the least preferred; for example, Linux 2.6.32 ranks the available clock sources as TSC, HPET, and ACPI_PM.

Virtual machines similarly provide their guest OSs with virtual timers: virtual PIT, virtual RTC, virtual ACPI timer, virtual HPET, virtual TSC, etc. Modifying virtual timers allows the manipulation of virtual time, which may proceed at a rate different from real-world time. We refer to the scaled time in the guest OS as *virtual time*.

It is possible to scale time in VHs at different abstraction levels. At the lowest level it is possible to modify the guest OS kernel of a VH, but this method is not sustainable as OS kernel updates may be required frequently. The method requires implementation for each OS if different OSs are used in the system. Scaling virtual timers in the hypervisor is applicable for systems with heterogeneous OSs because guest OS kernels do not have to be modified. The most efficient method is, however, to control the hypervisor time generator, which is the time source for all virtual timers. Implementing virtual time at the hypervisor layer has two important advantages: unmodified heterogeneous OSs can be used in the system, and no additional application programs for virtual time have to be installed on the VHs.

KVM has a kernel component that enables VHs to operate at a near-native speed. The KVM kernel component also allows VHs to directly use the TSC, APIC, or PIT from their physical machine. Therefore, when the kernel component is activated, if a VH uses TSC, APIC, or PIT as its time source, the virtual time generated at the hypervisor layer is not used in the VH. Hence, when using the KVM kernel component, the kernel component of PIT and APIC should be disabled for VHs that uses PIT and APIC as their time sources respectively. KVM has an option (`-no-kvm-pit`) that disables the KVM kernel mode PIT and redirects the physical PIT to the virtual PIT controlled by the hypervisor. KVM also has another option (`-no-kvm-irqchip`) that disables the KVM kernel mode PIC/IOAPIC/LAPIC and redirects the physical APIC to the virtual APIC. On the other hand, the kernel component itself should be deactivated for VHs that use TSC, but the performance will be significantly degraded. To operate our emulation system at a near-native performance, we switch all time sources of VHs to HPET, ACPI_PM, or PIT if they use TSC, as shown in Table 1.

Table 1. Each OS's clock source changes to HPET or PIT for virtual time.

OS	Clock source by default	Clock source for virtual time	Comment
Linux	TSC	HPET (or ACPI_PM)	–
FreeBSD	HPET	HPET	–
Windows XP	PIT	PIT	–
Junos	TSC	PIT	HPET unavailable

3 Adaptive time dilation

Since a physical machine can accommodate a multitude of VMs, it is possible to build a testing environment for a large-scale system with limited physical resources. However, the number of VMs is limited for any physical machine, due to limited CPU resources. This limitation can be overcome by the introduction of time dilation, which enables VMs to perceive improved performance.

A fixed TDF can result in system underutilization. Adapting time dilation to system loads allows the system to effectively utilize its resources.

3.1 Slice-based time dilation

For controlling virtual time for VMs, two distinct approaches have been used in SliceTime (Weingärtner et al., 2011) and DieCast (Gupta et al., 2008). The basic mechanism in SliceTime (Weingärtner et al., 2011) is to alternately suspend and resume the entire system in order to connect VMs to discrete event simulations

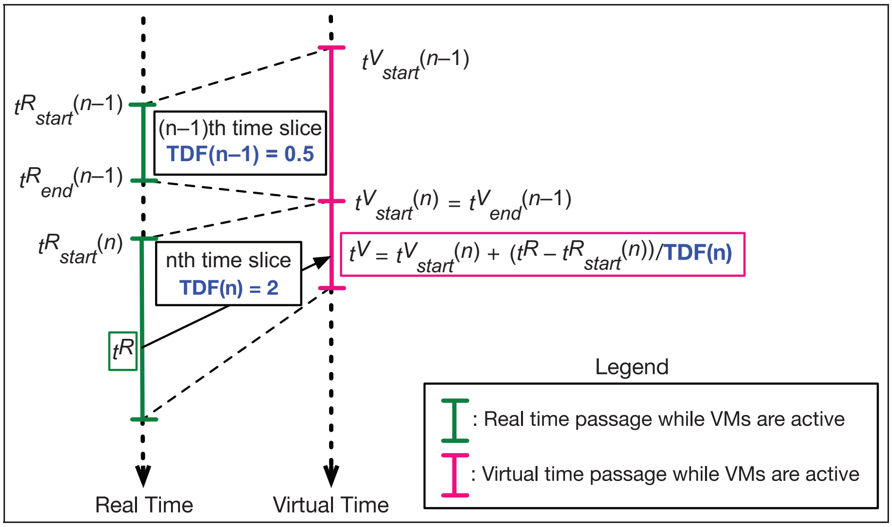

$t^V_{start}(n-1)$
$t^R_{start}(n-1)$
(n–1)th time slice
TDF(n–1) = 0.5
$t^R_{end}(n-1)$
$t^V_{start}(n) = t^V_{end}(n-1)$
$t^R_{start}(n)$
nth time slice
TDF(n) = 2
$t^V = t^V_{start}(n) + (t^R - t^R_{start}(n))/TDF(n)$
t^R
Legend
: Real time passage while VMs are active
: Virtual time passage while VMs are active
Real Time
Virtual Time

Figure 1. Illustration of slice-based time dilation.

(Law and Kelton, 1999) that may lag behind in time under heavy system loads (Weingärtner et al., 2008). Virtual time stops as VMs are suspended and then restarts as VMs are resumed. For example, if active time slices are the same size as inactive ones, virtual time passes at half the pace of real-world time. The other approach (introduced in DieCast; Gupta et al., 2008) is to control how fast virtual time proceeds by using a TDF, which is the ratio of the rate at which time proceeds in the physical world to the OS's perception of time (Gupta et al., 2005). If TDF is greater than one, then virtual time passes slower than real time, and vice versa. Both approaches, as introduced in Weingärtner et al. (2008, 2011) and Gupta et al. (2005, 2008), are static in that the relative ratio between real and virtual time is fixed for the life of VMs. On the other hand, NETplace (Grau et al., 2010) and NETbalance (Grau et al., 2011) use epoch-based virtual time (Grau et al., 2009) for implementing a dynamic TDF. By setting TDF to infinity, Grau et al. (2009) effectively suspend the advance of virtual time; however, since the VMs continue to run, the results of the emulation may be unrealistic (as the VMs perceive infinite CPU speed at infinite TDF).

For dynamic time dilation, we propose *slice-based time dilation*, where virtual time is dynamically dilated and, additionally, as virtual time stops, VMs freeze. Our approach uses both synchronized emulation (Weingärtner et al., 2008) and time-warped emulation (Gupta et al., 2005). In the slice-based time dilation, we refer to a *time slice* as an interval during which VMs are active. In the $(n-1)$ th time slice of Figure 1, real time starts at $t^R_{start}(n-1)$, while virtual time starts at $t^V_{start}(n-1)$. Once the emulation system resumes for the $(n-1)$ th time slice, virtual time starts and the VMs become active. When the emulation system is suspended, virtual time stops at $t^V_{end}(n-1)$, freezing VMs. At this moment, real time is at $t^R_{end}(n-1)$ and virtual time does not proceed until the system resumes. As the system begins for the nth time slice, real time starts at $t^R_{start}(n)$, but virtual time resumes at the previous point when virtual time is suspended in the $(n-1)$ th time slice. Consequently, $t^V_{start}(n)$ has the same value as $t^V_{end}(n-1)$.

The value of TDF remains constant in a time slice (i.e. it can change only during a suspension period of the system). Virtual time may proceed at a different rate in each time slice depending on the TDF, whose value is $TDF(n)$ in the nth time slice. In the $(n-1)$ th time slice of Figure 1, virtual time passes faster than real time, whereas it passes slower in the n th time slice (i.e. $TDF(n-1)$ is less than one, and $TDF(n)$ is greater than one).

Since the value of the starting point of the nth time slice in virtual time equals the sum of the lengths of all the previous time slices,

$$t^V_{start}(n) = \sum_{i=0}^{n-1} \Delta t^V(i) \tag{1}$$

Then given the real time t_R, virtual time t_V can be obtained by

$$t^V = \sum_{i=0}^{n-1} \Delta t^V(i) + \frac{t^R - t^R_{start}(n)}{TDF(n)} \tag{2}$$

3.2 Adaptation of dynamic time dilation to system loads

Time dilation allows VHs to perceive an increase in the performance offered by the physical machine. If the value of TDF is larger than one, a VH perceives its VM as if it runs on a faster machine. When a PH is overloaded, time dilation can alleviate the heavy load.

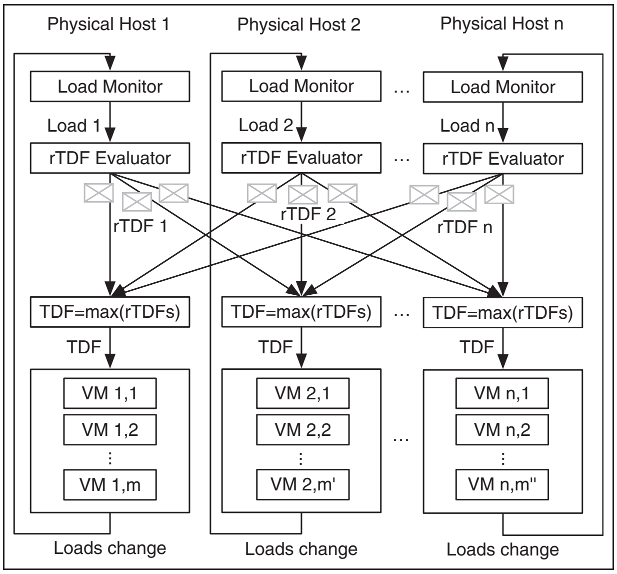


Figure 2. TDF controller.

Therefore, system load changes should be controlled by corresponding TDF alterations. Since the system load of a PH changes dynamically, the value of TDF should be evaluated frequently to allow rapid adaptation to system loads.

If VHs loads overload a PH, the TDF should increase to provide VHs with the perception of an improved physical machine, and vice versa. The value of TDF which is required in a PH to prevent operation delays in VHs caused by their PH overload is referred to as required TDF (rTDF). At every moment, each PH will have their own rTDF, potentially with different rTDFs for each PH.

In order to synchronize all VHs in the system, virtual time should pass at the same rate in every VH. For this, all VHs in the system should use a common system TDF whose value can adapt to system loads. System loads change the system TDF, and the changed TDF in turn affects system loads, as shown in Figure 2. The *TDF controller* dynamically adapts TDF to system loads. Each PH first monitors system loads, which in turn are used to generate their own rTDF. Each PH's rTDF message is then broadcast to the other PHs in each synchronization interval. The maximum value among all the rTDFs is selected as a new system TDF value, because choosing the greatest rTDF guarantees that VHs which runs on the most heavily loaded physical machine will operate without lagging due to CPU resources. TDF changes in turn causes load changes in the VHs. These load changes are monitored by each PH in the next monitoring interval.

When a PH evaluates its rTDF, the multi-core CPU should be taken into consideration because the unbalanced usage of CPU cores may occur. For example, when 100% of a CPU core is utilized and the other cores are idle, the operations of a VH that uses the most heavily used core can lag behind while the other VHs operate without delay. To make certain that all VHs in a PH operate without delay, the maximum value among all CPU core loads should be taken as the system load.

On one hand, TDF should adapt to current loads as rapidly as possible; on the other hand, the frequency of TDF change should be minimized. In other words, significant changes of system loads should be swiftly reflected in the TDF, but their minor oscillations may well be ignored. For emulation accuracy, frequent large changes of the TDF should be avoided, as a large change may not be applied perfectly simultaneously to all VHs, leading to a slight timing error in the emulated components.

To resolve these conflicting requirements, the TDF controller uses three parameters: α for the exponential moving average (EMA), *Gain*, and *Insensitivity*.

The EMA prevents a change in the current load ($Load_{current}$) from changing rTDF too rapidly. The EMA value of a system load in a monitoring interval, denoted by $Load(n)$, is computed as

$$Load(n) = (1 - \alpha) \cdot Load(n-1) + \alpha \cdot Load_{current} \tag{3}$$

Gain determines how rapidly rTDF will adapt to system loads. If $Load(n)$ is greater than a target load ($Load_{target}$), rTDF rises, and vice versa. The magnitude of the rising or falling is directly proportional to *Gain*.

Insensitivity minimizes the TDF change unless there is a larger deviation of $Load(n)$ from $Load_{target}$. The required TDF, $rTDF(n)$, is given by

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

where $sgn(x)$ is the sign of $\times$ (1 or $-$ 1), and $|x|$ is the absolute value of x. The second term in (4) has 2 as a scaling factor. A load of 50% is a typical target load. We want the change in the rTDF to be slow when $Load(n)$ is near the $Load_{target}$ but rapid when it is far from the $Load_{target}$. We also want rTDF to be approximately linear with respect to *Gain* when $Load(n)$ is around 75%. The scaling factor 2 in (4) accomplishes these goals.

Control theory could be brought to bear to design a controller for maintaining the CPU load on target; however, the model of the plant is not only not linear, but also depends on the applications running in the VHs. Therefore, we will be using a non-linear controller that we will show to work well for a large range of loads.

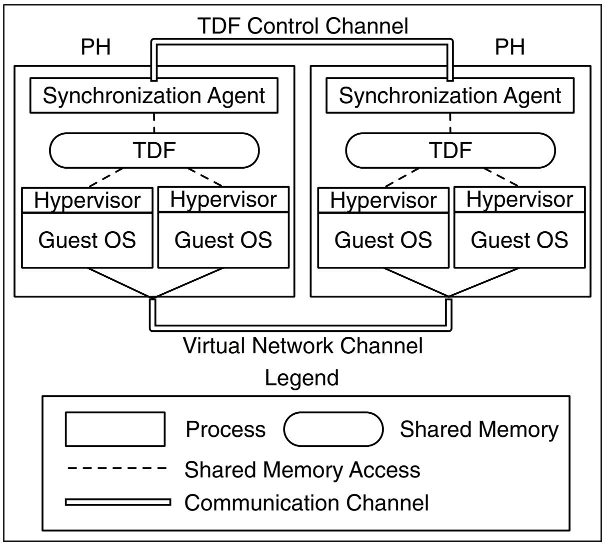


Figure 3. The synchronization agent on each PH determines a current TDF and disseminates it to all VHs on that PH.

4 System architecture and implementation

Synchronization agents exchange control messages (containing rTDFs) to synchronize VHs distributed over different PHs. Each synchronization agent determines the system TDF based on received rTDFs, and if the TDF is changed, each VH immediately applies it. The TDF controller of the synchronization agent is tuned so that the rTDF rapidly adapts to system loads, while avoiding the unnecessary minor changes.

4.1 Synchronization

The current TDFs in VHs must be synchronized whether they are colocated in a single PH, or are deployed over multiple PHs. VHs that run on the same PH are locally synchronized through IPCs based on shared memory and semaphores, while VHs that are distributed over different PHs are globally synchronized by using User Datagram Protocol (UDP) packets. For local synchronization, hypervisors on the same PH share a common TDF stored in shared memory, and control their local virtual time based on this TDF value. rTDF values computed by each PH using (4) are broadcast via UDP synchronization messages. Synchronization agents are used to compute the current TDF stored in the shared memory and exchange UDP messages.

Figure 3 shows our synchronization method. A synchronization agent periodically generates an rTDF value and broadcasts it to the other PHs. When receiving synchronization messages, each synchronization agent computes a new TDF, which is the maximum of all recently received rTDFs. Thus, all PHs will have a common TDF value upon receiving UDP packets from the other PHs. Virtual time then proceeds at the same rate with a common TDF value in all VHs. To ensure timely dissemination of rTDFs, the synchronization agent periodically generates a new message in a synchronization interval.

VHs running on different PHs can be synchronized with the granularity of the synchronization interval. So, for example, when testing on our emulation system packets' round-trip time, whose measurement on a real testbed is less than the synchronization interval, the interval can appear to be long. As the synchronization interval decreases, on the other hand, the control messages increase system loads. It is desirable to minimize system loads generated by control messages while maintaining system responsiveness. Hence, a synchronization interval should be carefully chosen, depending on measurement granularity.

Synchronization agents exchange synchronization messages through TDF control channels, and virtual nodes communicate with one another through virtual network channels, as depicted in Figure 3. The isolation of control messages from virtual network traffic helps to minimize synchronization delay that network traffic congestion could cause.

Since UDP packets are used for synchronization messages, their loss may cause synchronization delay, but synchronization will be rapidly recovered in an interval by the following synchronization message. We deploy our PHs on one LAN and synchronization messages have their own control channels, so there is only a small chance of synchronization delay. Even if a packet loss were to occur, since all PHs are in a common broadcast domain, they will usually lose the packet at the same time, so all PHs remained synchronized. TCP would likely have a very large overhead (as it is necessarily unicast vs the UDP broadcast, and requires acknowledgments), and would introduce additional delays.

4.2 Synchronization agent

Our synchronization agent has three threads, as depicted in Figure 4. The first thread is used to monitor the system load, compute the rTDF, and periodically broadcast the rTDF to the other PHs. Each PH broadcasts their own rTDF. The second thread handles incoming synchronization messages from the synchronization agents on the other PHs. When a synchronization agent receives a synchronization message, it updates the rTDF for the source host of the message. If the maximum value of rTDFs is different from the current TDF, the TDF change process is triggered in the third thread. To change the current TDF, the synchronization agent first suspends the operation of all VHs, renews TDF in the shared memory, and resumes VHs. All VMs then operate at a new TDF. In order for the VHs to use virtual time controlled by the

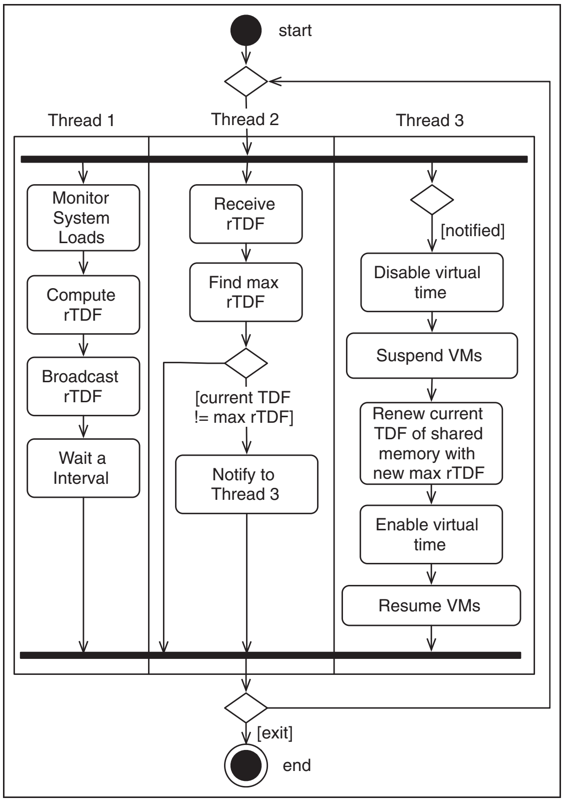


Figure 4. Activity diagram of synchronization agent.

synchronization agent, hypervisors must use the TDF of the synchronization agent.

4.3 System tuning

The goal of the dynamic TDF controller of the synchronization agent is to rapidly adapt a TDF value to system loads while avoiding unnecessary changes in TDF. To achieve this goal we must determine appropriate parameter values for α, *Gain*, and *Insensitivity*.

To tune the system controller, we apply to a VH a square load pattern that alternates between heavy and low CPU load (generated by repeatedly computing random numbers) at different duty cycles. The VH directly offers a certain load to its physical machine. This is the measured PH system load used to compute the rTDF, which is broadcast to the other PHs. For the purpose of system tuning, we use a single VH on the PH, so rTDF = TDF.

System performance can be judged by changes in the TDF and CPU loads. Minimizing unneeded TDF variation minimizes changes in TDF across hosts. The degree of TDF oscillation on any PH can be measured by the summation of the square of the derivatives:

$$C_1 = \sum_{i=1}^{N-1} \left(\frac{TDF(i+1) - TDF(i)}{t(i+1) - t(i)} \right)^2 \tag{5}$$

where the summation is computed over a measurement period with N samples. We refer to C_1 as *TDF instability*.

System overload can be avoided when the current PH CPU load *Load*(i) are close to a target load $Load_{target}$ (50% in our experiments). The oscillation degree of PH loads around a target can be quantified by

$$C_2 = \frac{\sum_{i=1}^{N} (Load_{target} - Load(i))^2}{N} \tag{6}$$

where the summation is computed over the same measurement period of N. We refer to C_2 as *load disturbance attenuation*. A *normalized TDF instability* $\bar{C}_1$ is obtained by dividing a TDF instability by the mean value of all TDF instabilities over a measurement period (i.e. 60 s in our experiments), and a *normalized load disturbance attenuation* $\bar{C}_2$ is computed by dividing a load disturbance attenuation by the mean value of all load disturbance attenuations over a measurement period.

The normalized C_1 and C_2 are components of cost which are used to minimize TDF changes as system loads vary. Infrequent and small TDF changes will have less effect on many OS system calls associated with the time passage rate. Some system load fluctuation should be tolerated. Hence, TDF instability is given more weight than load disturbance attenuation and we define total cost as

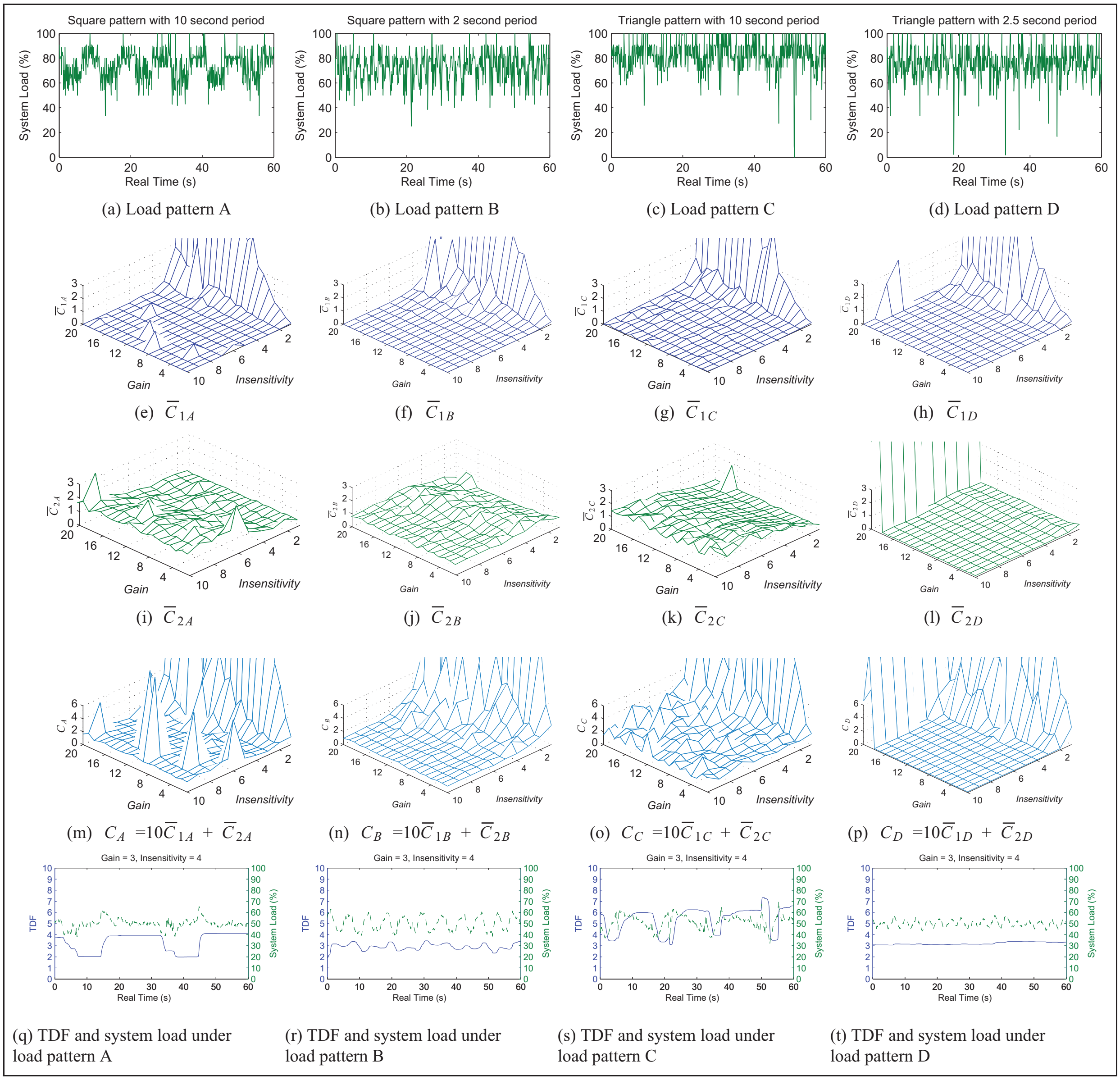
$$C = w_I \bar{C}_1 + \bar{C}_2 \tag{7}$$

where $w_I \geq 1$ denotes the weight assigned to TDF instability.

The weight w_I balances the effects of TDF changes and load response. We choose $w_I = 10$ since the values of *Gain* and *Insensitivity* provide responsive systems with modest TDF changes. Experiments showed that w_I can vary considerably around 10 without changing near-optimal system parameters.

Our goal is to find parameters *Gain, Insensitivity*, and α that can quickly respond to load changes, and that move the current load to the target load while still keeping the TDF changes relatively small and infrequent; in other words, minimize the cost function C in (7). As shown in Figure 2, our TDF controller is a closed-loop control system where a system load changes rTDF, and the changed rTDF in turn affects the system load in (4). Hence, it is extremely complex to find a minimized total cost C, so we rely on measurement rather than introducing a model where the expected value of a linear combination of *Gain, Insensitivity*, and α is minimized under a randomized experiment plan.

In order to find near-optimal parameter combinations, we analyze how system stability is related to *Gain* and *Insensitivity*, as shown in Figure 5. The value of α for EMA is set to 0.125 so that the oscillations caused



(a) Load pattern A

(b) Load pattern B

(c) Load pattern C

(d) Load pattern D

(e) $\bar{C}_{1A}$

(f) $\bar{C}_{1B}$

(g) $\bar{C}_{1C}$

(h) $\bar{C}_{1D}$

(i) $\bar{C}_{2A}$

(j) $\bar{C}_{2B}$

(k) $\bar{C}_{2C}$

(l) $\bar{C}_{2D}$

(m) $C_A = 10\bar{C}_{1A} + \bar{C}_{2A}$

(n) $C_B = 10\bar{C}_{1B} + \bar{C}_{2B}$

(o) $C_C = 10\bar{C}_{1C} + \bar{C}_{2C}$

(p) $C_D = 10\bar{C}_{1D} + \bar{C}_{2D}$

(q) TDF and system load under load pattern A

(r) TDF and system load under load pattern B

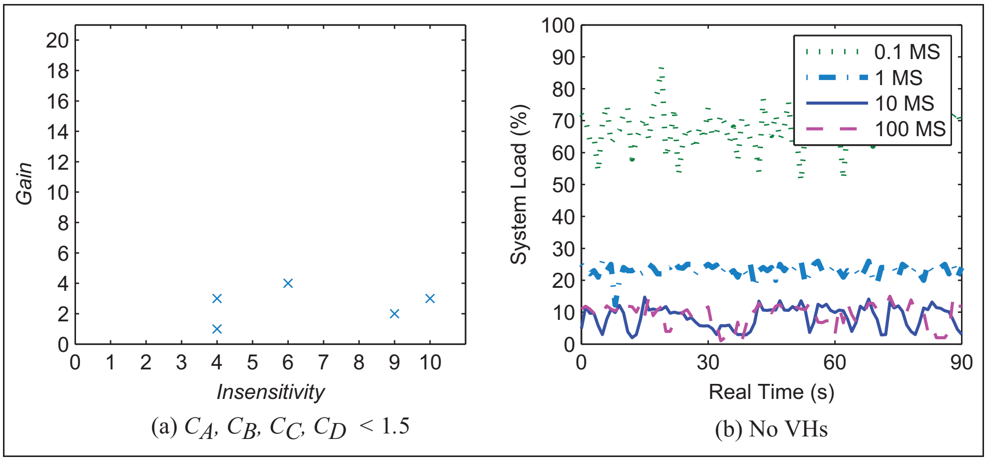
(s) TDF and system load under load pattern C

(t) TDF and system load under load pattern D

Figure 5. As four patterns of system loads A, B, C, D ((a) to (d)) are offered, the TDF change drives system loads; TDF change is measured by normalized TDF instabilities $\bar{C}_{1A}, \bar{C}_{1B}, \bar{C}_{1C}, \bar{C}_{1D}$ ((e) to (h)) and load change is measured by normalized load disturbance attenuations $\bar{C}_{2A}, \bar{C}_{2B}, \bar{C}_{2C}, \bar{C}_{2D}$ ((i), to (l)); total costs C_A,C_B,C_C,C_D ((m) to (p)) are used to find *Gain* and *Insensitivity*; a profile of TDF control over system loads for each system load pattern is shown in (q) to (t), when *Gain* = 3 and *Insensitivity* = 4. For all figures α = 0.125.

by frequent minor variations of loads are removed. The choice $\alpha = 0.125$ is analyzed at the end of this section. First of all, square system-load patterns with 10 s period, shown in Figure 5(a), are applied. TDF instability and load disturbance attenuation depend upon *Gain* and *Insensitivity*. As shown in Figure 5(e), TDF instability is very high when *Insensitivity* < 2 regardless of *Gain*. Otherwise, the effect of *Insensitivity* is negligible when *Insensitivity* $> \sim 3$. The normalized load disturbance attenuation gradually increases as *Insensitivity* grows regardless of *Gain*, as shown in Figure 5(i). For load scenario A, $\bar{C}_{1A}$ is minimized when *Insensitivity* > 3, whereas $\bar{C}_{2A}$ is minimized for smaller values of *Insensitivity*. Finally, the total cost ($C_A = 10\bar{C}_{1A} + \bar{C}_{2A}$) is depicted in Figure 5(m). Our goal is to find the combinations of *Gain* and *Insensitivity* that approximately minimize C_A.

To accommodate diverse forms of system loads, we repeat the same analysis for additional system load patterns: a square pattern with 2 s period, a triangle pattern with 10 s period, and a triangle pattern with 2.5 s period. These patterns are shown in



(a) C_A, C_B, C_C, C_D < 1.5

(b) No VHs

Figure 6. (a) Candidates of *Gain* and *Insensitivity*, and (b) the effect of synchronization message intervals on system loads.

Figure 5(b), (c), and (d). Using the same methodology as for load A, we obtain the total costs C_B, C_C, C_D for loads B, C, D as shown in Figure 5(n), (o), and (p).

Our goal is to find *Gain* and *Insensitivity* that simultaneously minimize C_A, C_B, C_C, and C_D. When a threshold of 1.5 is set for total costs, that is, $C_A < 1.5$, $C_B < 1.5$, $C_C < 1.5$, and $C_D < 1.5$, the combinations of (1, 4), (3, 4), (4, 6), (2, 9), and (3, 10) are found, as shown in Figure 6(a).

Even though it is possible to use any combination found in Figure 6(a) in term of system stability, we choose *Gain* = 3 and *Insensitivity* = 4, because the combination not only satisfies the cost threshold, but also has rapid responsiveness to load changes (larger *Gain* and less *Insensitivity*). Higher *Insensitivity* decreases TDF changes, but adapts slower to system loads.

System loads caused by synchronization messages should be minimized, subject to responsiveness, so that VHs can maximize the use of their PH's computing resources. When the synchronization agent runs only in a PH without creating any VHs, the system load increases as a synchronization message interval decreases. This can be seen in Figure 6(b). However, if the synchronization interval 10 ms, the synchronization messages do not significantly affect system loads. Hence, we choose an update period of 10 ms as a reasonable compromise.

System stability depends on α of EMA as well as *Gain* and *Insensitivity*. Smaller α prevents system loads' large variances from changing TDF too rapidly. As shown in Figure 7(a), (b), and (c), when *Insensitivity* = 1 and *Gain* increases from 1 to 3, TDF instability (C_{1A}) increases significantly with α. As seen in Figure 7(c), (d), (e), and (f), as α decreases and *Insensitivity* increases from 1 to 4 where *Gain* is held at 3, TDF instability rapidly decreases. When *Insensitivity* increases to 3 and α decreases to 0.125 in Figure 7(e), both C_{1A} and C_{2A} become stable. When *Gain*, *Insensitivity* ≥ 3, $\alpha = 0.125$ stabilizes the system as seen in the pattern of Figure 7(a) to (f). Values of $\alpha < 0.125$ (i.e. $\alpha = 0.0625, 0.0312$) also stabilize the system, but decrease responsiveness to system load changes, as shown in Figure 8.

In sum, by using the combination of *Gain* = 3, *Insensitivity* = 4, $\alpha = 0.125$, and a synchronization interval of 10 ms, the system can stably operate with rapid TDF response to dynamic system loads, while minimizing TDF changes caused by frequent minor load oscillations.

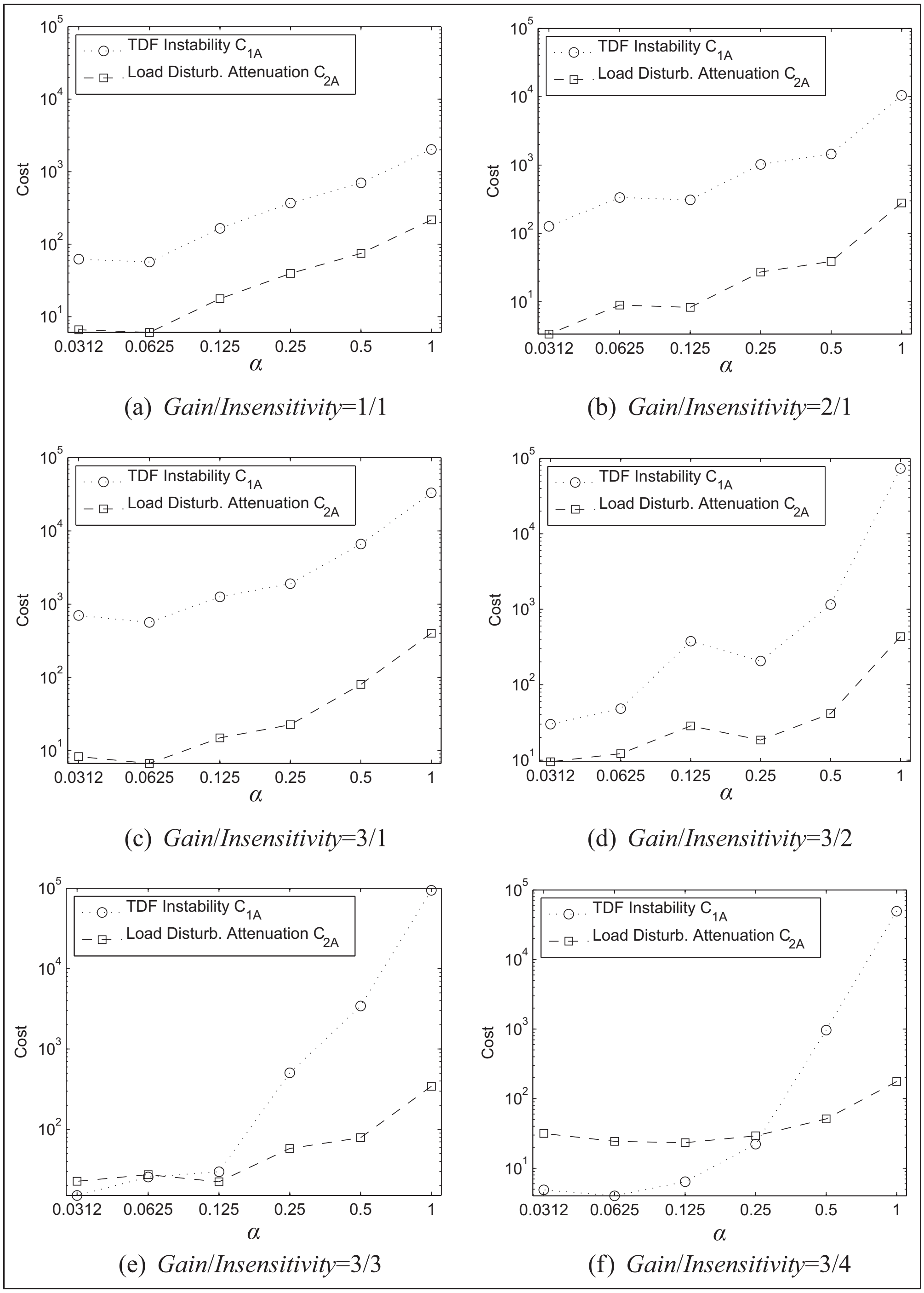
With *Gain* = 3, *Insensitivity* = 4, $\alpha = 0.125$, and a synchronization interval of 10 ms, a profile of TDF control over system loads for each system load pattern is shown in Figure 5(q) through (t). Under load pattern A (a square pattern with 10 s), the system loads remain near-constant on both high and low loads. During these intervals, the TDF also remains constant, as seen in Figure 5(q). If system loads change from low to high loads, TDF rapidly increases and results in reducing system loads, and vice versa. For the other load patterns, the TDF is controlled in the same fashion. Figure 5(s) (for a triangle pattern with 10 s period) also shows that the TDF rapidly follows system load changes. However, under load pattern D (a triangle pattern with 2.5 s period), the TDF's alteration does not follow the loads'; this test verifies that minor and rapid load oscillations do not change TDF.

5 Performance Evaluation

The proposed emulation system is tested by creating distributed VMs and generating streaming traffic, and the heterogeneity and scalability of the system are evaluated.

5.1 Experimental setup

Our emulation system is built on five general-purpose servers (Dell PowerEdge R210). Each PH has four



(a) *Gain/Insensitivity*=1/1

(b) *Gain/Insensitivity*=2/1

(c) *Gain/Insensitivity*=3/1

(d) *Gain/Insensitivity*=3/2

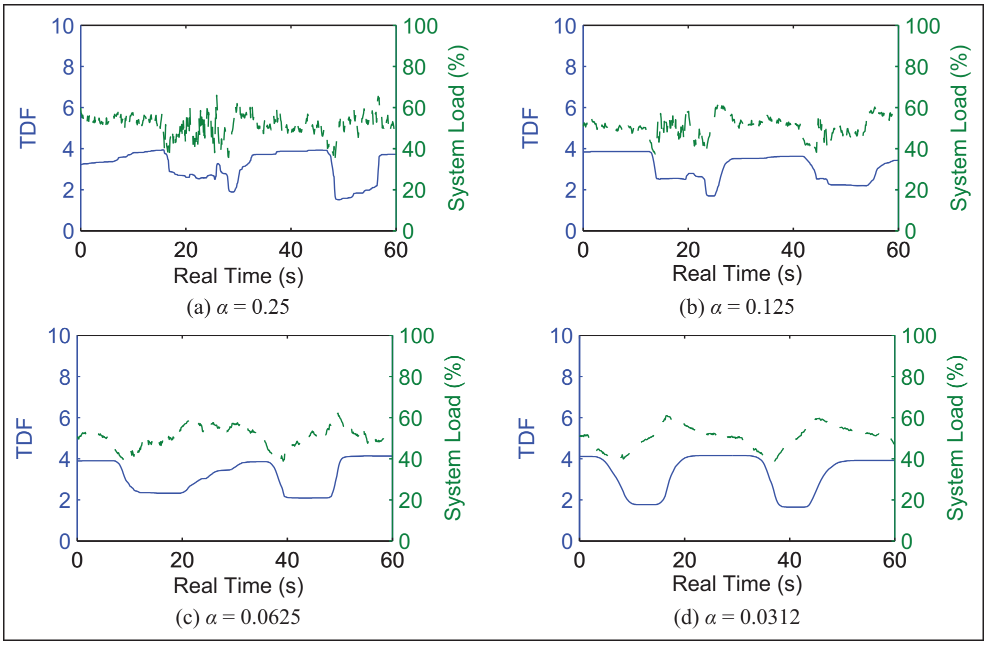
(e) *Gain/Insensitivity*=3/3

(f) *Gain/Insensitivity*=3/4

Figure 7. Effect of exponential moving average coefficient α on our control system performance

CPU cores with hyper-threading. Therefore, each host OS recognizes eight CPU cores. In our heterogeneity evaluation, we initially create eight VHs in a PH such that the load of each VH does not greatly affect the other VHs that share a PH. Eight or fewer VHs will have a modest impact on the performance of each other, while more than eight VHs may significantly affect the VHs' performance. In order to test our system's scalability, we also load more than eight VHs in a PH during subsequent evaluations.

Each physical machine has two Gigabit Ethernet interfaces connected to different switches. The first interface is used for exchanging synchronization messages and the second interface is used for building a virtual network. VHs in different PHs communicate through the second interface.



(a) $\alpha = 0.25$

(b) $\alpha = 0.125$

(c) $\alpha = 0.0625$

(d) $\alpha = 0.0312$

Figure 8. TDF and system loads for α = 0.25, 0.125, 0.0625, 0.0312 when *Gain* = 3 and *Insensitivity* = 4 under load pattern A.

Table 2. Software packages used in our emulation system.

Linux (PH)	ubuntu-10.04-desktop-amd64	Unmodified
Linux (VH)	ubuntu-10.04-server-amd64	Unmodified
FreeBSD	FreeBSD-9.0-RELEASE-i386	Unmodified
Windows	Windows XP SP3	Unmodified
Junos	jinstall-12.1R1.9	Unmodified
Hypervisor	qemu-kvm-0.13.0	Modified: 307 LOC
Sync. Agent	Written from scratch	Created: 2196 LOC

We employ four different OSs: Linux, FreeBSD, Windows XP, and Junos,[11] to create the VHs. For virtualization, we use and modify the hypervisor of KVM. KVM's timer is modified for virtual time, and code for accessing shared memory (where the TDF is stored) is added. Table 2 shows details of each software package, including modified lines of code (LOC).

While booting, each OS by default selects a clock source: Linux (TSC), FreeBSD (HPET), Windows XP (PIT), and Junos (TSC). We change the time sources of Linux and Junos to HPET and PIT respectively, to run VHs at a near-native speed (i.e. to activate the KVM kernel component); Junos can only use TSC and PIT.

We want to be able to have virtual time proceed at a slower rate (as we have seen when the load is heavy) or at a faster rate when the load is light. The first case corresponds to a TDF $>$ 1 and the second case corresponds to a TDF $<$ 1. If an experiment generates light system loads, then it can have TDF $<$ 1 and virtual time proceeds faster than real time. Thus an experiment with TDF $<$ 1 will run on a slower machine but that is advantageous since it will complete in less real time.

5.2 Emulation accuracy

For testing emulation accuracy, we measure the inter-ping interval as follows. We set VH1, running on PH1, to send an Internet Control Message Protocol (ICMP) echo request packet towards VH2, running on PH2, every second in virtual time, as depicted in Figure 9(a). That is, an application running in VH1 sends an ICMP packet and sleeps for one virtual second. On waking up, the application sends the next ICMP packet to VH2.

In order to evaluate our system's emulation time accuracy under dynamic time dilation, we change the TDF every 100 ms such that the TDF values create a sinusoidal wave between 1 and 100, as seen in Figure 9(b). The inter-ping intervals are maintained at approximately 1 s with less than 2 ms' variation in virtual time, as shown in Figure 9(c). When the TDF changes every 10 ms and the interval of sinusoidally changing TDF further decreases as seen in Figure 9(d), the inter-ping intervals are still kept around one virtual second with similar variation, as shown in Figure 9(e).

5.3 Heterogeneity

We use a virtual network as shown in Figure 10 to evaluate the heterogeneity of the proposed emulation system. The virtual topology has heterogeneous VHs running Linux, FreeBSD, Windows XP, and Junos (Juniper network operating system). The Linux server on PH1 streams video traffic to clients: FreeBSD, Windows, and Linux on PH3, PH4, and PH5 respectively. Four virtual Juniper routers on PH2 connect the server and the clients running the Open Shortest Path

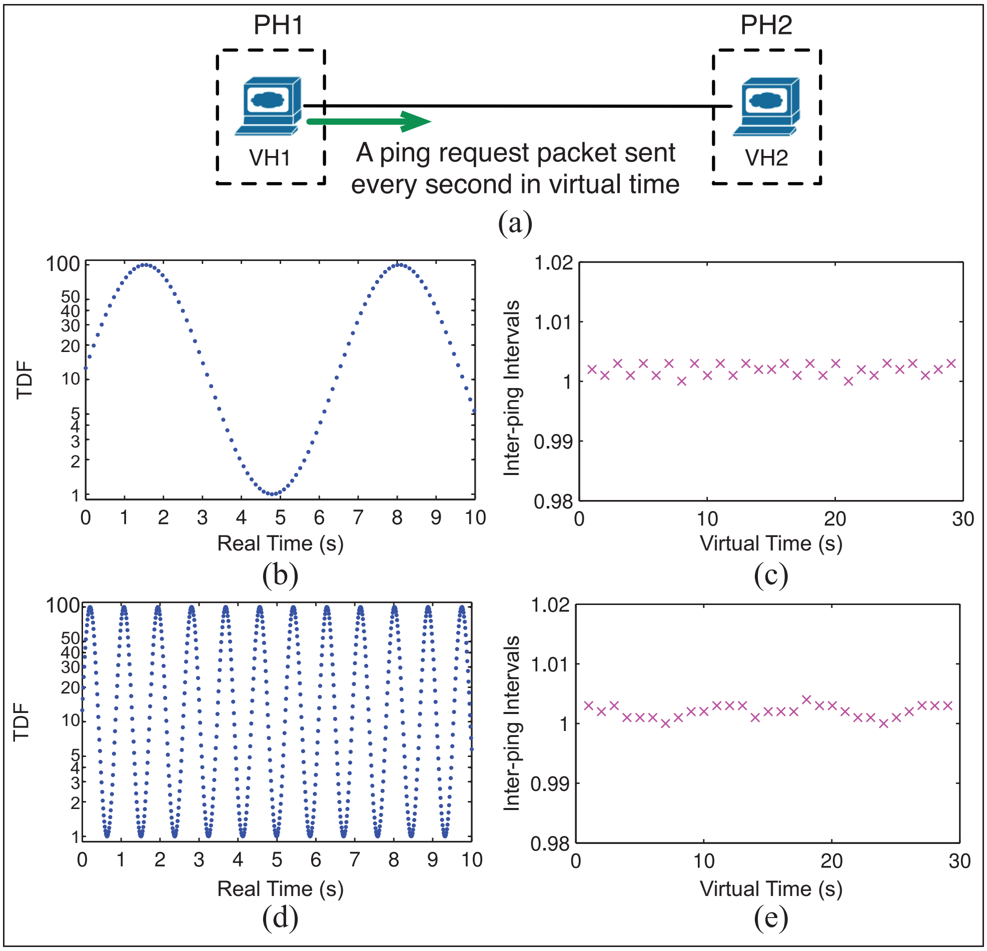


Figure 9. In a virtual network topology (a), while TDF sinusoidally changes with a different interval as in (b) and (d), every ICMP echo packet is sent from VH1 to VH2 after 1 s sleep in virtual time. Independent of TDF changes, the inter-ping intervals are maintained at about one virtual second in (c) and (e).

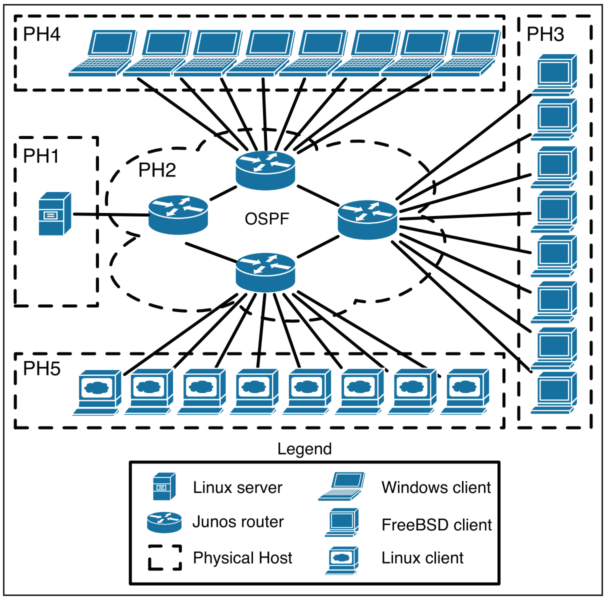
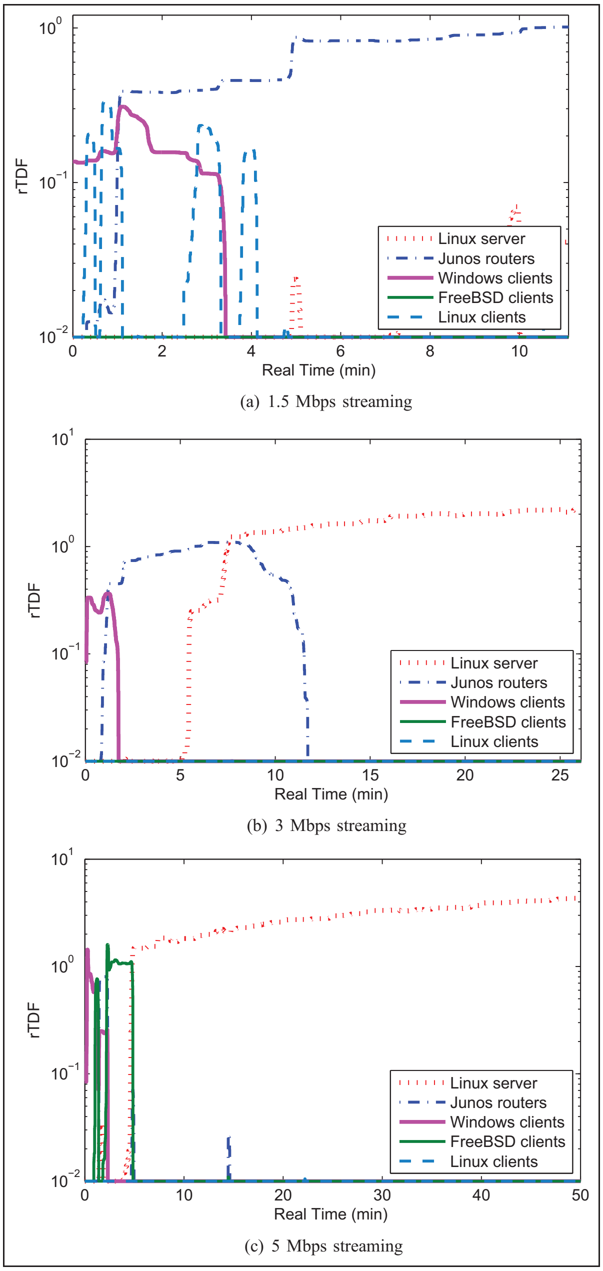


Figure 10. A streaming service topology is created for evaluating heterogeneity.

First (OSPF) protocol. While our system supports different OSs on the same PH, we enforce the separation of OSs for simplicity and cleaner system evaluation.

System evaluation is done using a video streaming service similar to the ones offered by Netflix[12] and Hulu.[13] The video streaming service loads are approximately constant in the long term (they do vary slightly in the short term), and hence, scale linearly with the number of clients. In our evaluation, we increase the number of streaming clients up to 24 every minute, thus increasing the load on all systems.

To model the video streaming service, the virtual Linux server running on PH1 generates UDP streams at three data rates: 1.5 Mbps (corresponding to SD quality), 3 Mbps (DVD quality), and 5 Mbps (HD quality). The packets are forwarded to the clients by the Juniper virtual routers. Upon receiving a packet each client emulates video decoding by computing a random number a thousand times. In addition to modeling a video streaming service, we also test our emulation system using a real application, the VLC media player,[14] in Section 5.5.



(a) 1.5 Mbps streaming

(b) 3 Mbps streaming

(c) 5 Mbps streaming

Figure 11. rTDF dynamically changes under system loads generated by streaming traffic; each streaming duration is 24 min in virtual time.

To evaluate the scaling of TDF with the offered load we add one client every virtual minute (first a Windows client, then a FreeBSD client, then finally a Linux client) until there are 24 clients total (eight of each type). The minimum system TDF is set at 0.01 (i.e. virtual time passes 100 times faster than real time). The target CPU load is set at 50%.

Figure 11 shows the rTDF for each of the five PHs for each of the three considered traffic loads. In Figure 11(a) at light loads the PH with the Windows clients experiences the largest CPU loads which drives the system TDF from 0.01 to about 0.1. As the number of clients increases, that is, the traffic forwarded by the routers increases, the load on PH2 (of Junos routers) increases faster than that of PH4 (of Windows clients), further driving the system TDF up to one. When the Linux server generates 3 Mbps flows per connection (results shown in Figure 11(b)), the PH generating the largest rTDF changes from the Windows clients (PH4), to the routing PH (PH2) and eventually to the Linux server (PH1). The final TDF at full load, that is, 24 3 Mbps streams, is slightly larger than two. Finally, when the largest streams (5 Mbps, shown in Figure 11(c)) are offered, the Linux server quickly becomes the system that drives the overall emulation TDF.

The VHs are completely unaware of TDF changes as they change over one order of magnitude. Figure 12(a), (c), and (e) shows the total amount of data received by the FreeBSD clients while the Linux server generates the packet streams of 1.5 Mbps, 3 Mbps, or 5 Mbps to its clients.

While TDF is dynamically changing, as shown in Figure 12(b), (d), and (f), each slope (which represents a data reception rate) in Figure 12(a), (c), and (e) remains nearly constant at 1.5 Mbps, 3 Mbps, and 5 Mbps respectively. This demonstrates that the VH is truly isolated from changes in TDF.

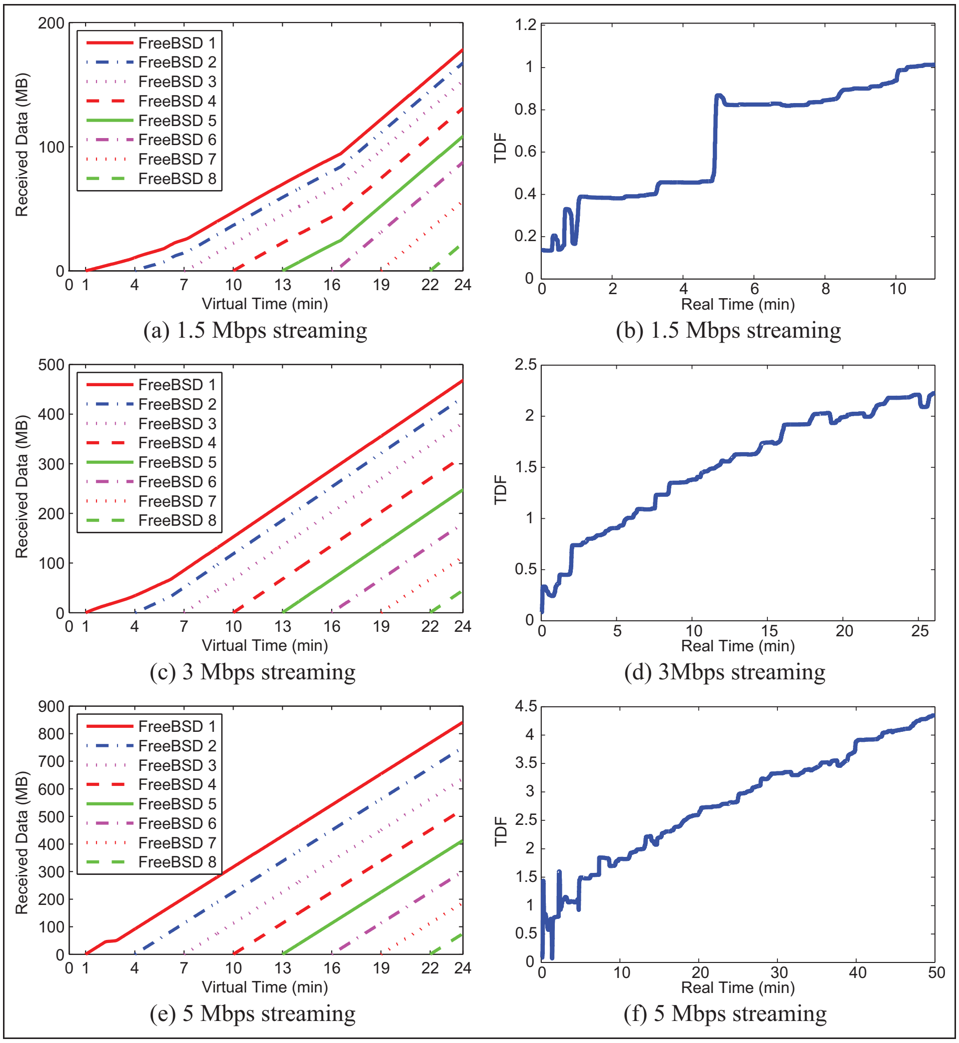
5.4 Scalability

The virtual network topology shown in Figure 13 is used to evaluate the scalability of our emulation system. The Linux server in PH1 generates a stream of traffic through a Junos router in PH2 to FreeBSD clients in PH3. PH1 creates system loads by generating streaming packets, PH2, by forwarding packets, and PH3, by receiving the packets and emulating video decoding. Each PH generates rTDFs based on the system loads.

In order to evaluate scalability, we increase the VHs (FreeBSD clients) in PH3, so the rTDF from PH3 drives the system TDF. The Linux server in PH1 sends a UDP packet stream to all VHs created in PH3. As the number of VHs increases in PH3, PH3's system loads increase accordingly.

As more VHs receive a 3 Mbps stream, PH3 creates larger rTDF values as shown in Figure 14(a). When eight VHs are added to the existing eight VHs, the rTDF increases from 1.7 to 3.2, and when eight more VHs are added (total 24 VHs), the rTDF climbs to 6.4. The reason the rTDF increases more from 16 to 24 VHs (3.2) than from 8 to 16 VHs (1.5) is because as more VHs are packed into a PH, context switching and sharing I/O resources require more computing time.

When the server hosts 16, 24, and 32 VHs each generating a 1.5 Mbps stream (total 24, 36, and 48 Mbps respectively), the rTDF values are shown in Figure 14(b). Compared to the 3 Mbps streaming traffic



(a) 1.5 Mbps streaming

(b) 1.5 Mbps streaming

(c) 3 Mbps streaming

(d) 3Mbps streaming

(e) 5 Mbps streaming

(f) 5 Mbps streaming

Figure 12. The data reception rate in virtual time, that is, the slope in (a), (c), (e), remains near-constant and the value is close to each traffic generation rate (1.5, 3, 5 Mbps), even while TDF is dynamically changing over one order of magnitude; a stream is added every 3 min in virtual time.

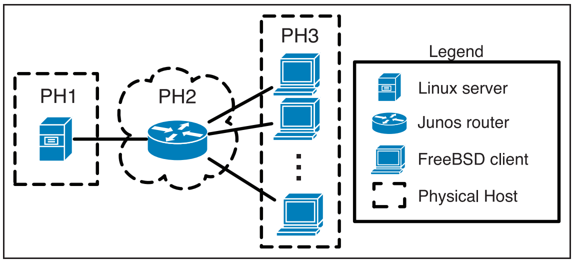


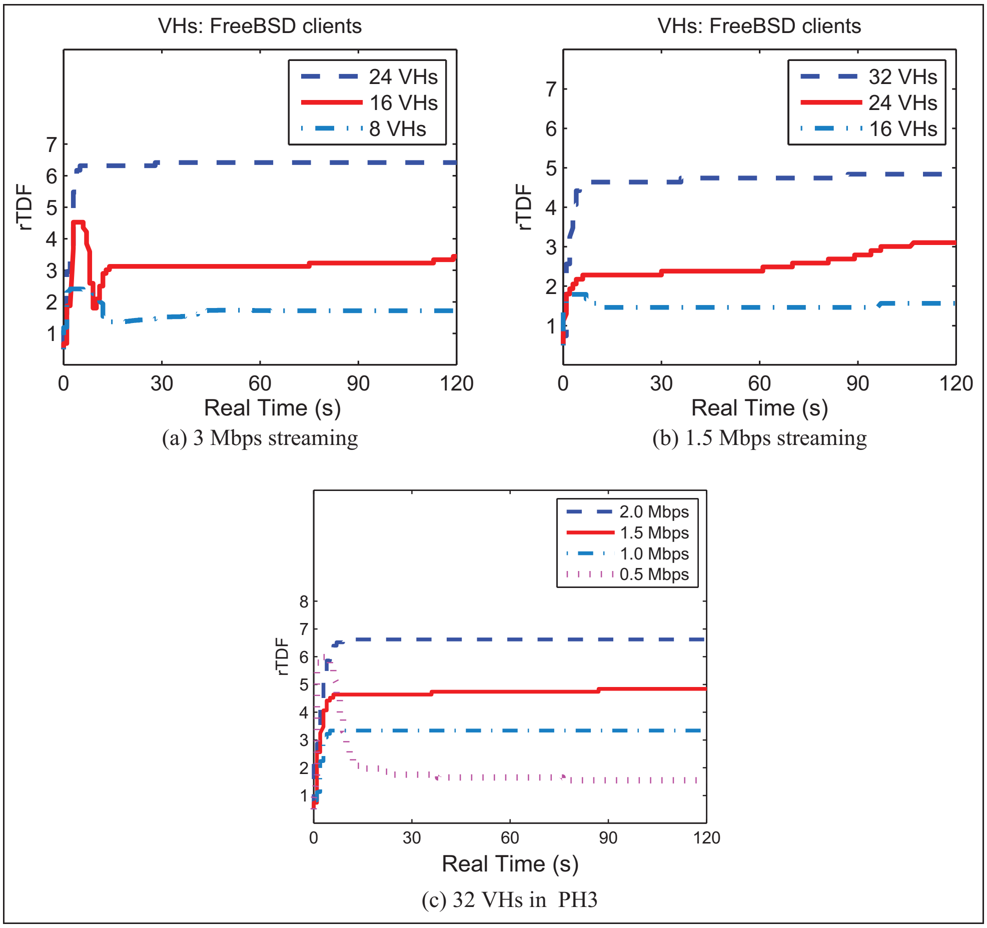
Figure 13. A virtual network topology for evaluating scalability.

shown in Figure 14(a), the 1.5 Mbps streaming traffic requires a smaller rTDF for the same number of VHs. Even for 32 VHs, the rTDF is approximately 4.8, which is smaller than rTDF $\simeq$ 6.4 for 3 Mbps 24 VHs. As shown in Figure 14(c), when we load 32 VHs with 0.5 Mbps of traffic and then increment the traffic load by 0.5 Mbps, the rTDF increases approximately linearly with a slightly larger increase for a larger number of VHs.

5.5 Real-world application: VLC media player

We test a real application, the VLC media player from VideoLan, on our emulation system. VLC is an open-source media player that runs on diverse OSs (e.g. Linux, FreeBSD, Windows, Mac OS X, etc.), and the player can operate as a streaming server or a streaming client. For our test, a VLC media player operates as a streaming server in a virtual Linux server on PH1 and streams a video file to eight virtual Linux clients on PH2, each running a VLC media player operating as a streaming client, as depicted in Figure 15(a).

We evaluate our emulation system using four sample video files compressed with different codecs and resolutions. A higher video quality requires a larger bitrate,



(a) 3 Mbps streaming

(b) 1.5 Mbps streaming

(c) 32 VHs in PH3

Figure 14. Each VH (FreeBSD client) receives a stream of 3 Mbps in (a) or 1.5 Mbps in (b). As the number of VHs increases on PH3, their rTDF increases accordingly. As the Linux server generates more streaming traffic (0.5, 1, 1.5, and 2 Mbps), the PH accommodating 32 VHs increases its rTDF as shown (c).

Table 3. Specification of sample video files.

File	Resolution	Codec	Video Bitrate
video1	480p	Xvid	1 Mbps
video2	720p	H.264	3 Mbps
video3	720p	H.264	6 Mbps
video4	1024p	H.264	10 Mbps

as seen in Table 3. When a single VLC streaming server broadcasts UDP streams to eight VLC streaming clients using the sample video files, PH2, running the VLC streaming clients, is heavily loaded (approaches 100%). Hence, the system load of PH2 drives the system TDF. Since higher-quality video streams require more CPU resources for their decompression, the system TDF increases as a video bitrate increases, as shown in Figure 15.

6 Related work

Time dilation. DieCast (Gupta et al., 2008) scales CPU cycles, network communication characteristics, and disk I/O, providing the illusion that each VM matches a machine from the original service in terms of available computing resources and communication behavior to remote service nodes. The behavior of DieCast therefore matches that of the original service at a fraction of the physical resources. The main limitation of DieCast is that the scaling factor cannot be changed during runtime.

When real systems interact with a simulated network, the simulation executes in real time, but there is no simple solution if the simulation lags behind in time. Failing to deliver packets in a timely manner generates corrupted results such as highly increased network latency or jitter. Instead of real systems, *synchronized network emulation* uses VHs in order to be able to synchronize their execution behavior with the network simulation (Weingärtner et al., 2008). Time in VHs proceeds only when the synchronization component allocates the next time slice; otherwise, time in VHs remains stalled. In contrast, when connecting a simulated network to VHs, the approach in Lee et al. (2014) uses time dilation to reduce simulation delay that occurs due to the heavily loaded simulator.

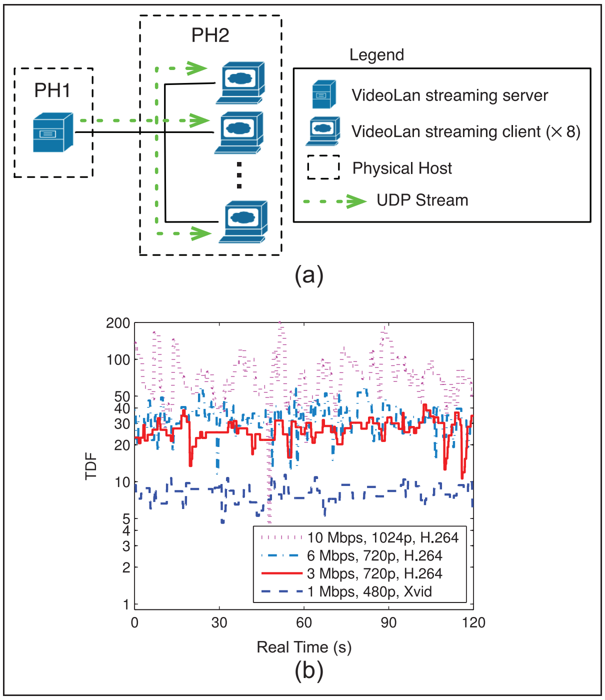


Figure 15. (a) A virtual network topology for evaluating the VLC media player that runs on our emulation system; (b) Higher video bitrates imply larger TDF.

The Distributed Open Network Emulator, or dONE, employs a temporal model called *relativistic time*, which is conceptually the same technique as time dilation, reconciling the real-time nature of direct code execution with the event-driven nature of simulation models (Bergstrom et al., 2006). In addition, dONE creates VHs using a composition framework called *Weaves*, which provides the ability to emulate multiple instances of an application or protocol stack inside a single OS process (Mukherjee and Varadarajan, 2005). In comparison with synchronized network emulation that runs unmodified OSs as well as unmodified applications and protocol stacks, dONE allows a VH to execute only applications and protocol stacks without modification. The design goal of dONE that uses relativistic time on the Weaves framework is to enhance scalability in building hybrid network emulation/simulation environment.

The MicroGrid virtualizes network resources using the MicroGrid network emulator (MaSSF), and compute resources using the MicroGrid CPU controller (Xia et al., 2004). MaSSF uses a real-time scheduler, but when the emulated system is too large to be emulated on available hardware, the scheduler runs in a scaled-down mode. The MicroGrid CPU controller also supports a scaled-down mode to emulate VMs that are faster than available physical resources.

Compared to a constant clock rate that results in suboptimal experiment runtime, network emulation using adaptive virtual time (Grau et al., 2009) dynamically adjusts the clock rate to system loads. The approach in Grau et al. (2009) uses the concept of epoch-based virtual time (Grau et al., 2008) where the experiment is divided into epochs, each of which has a constant TDF. TDF adaptation process is based on threshold-based adaptive sampling of system loads (Grau et al., 2009).

High-performance virtual machines. The work in Bridges et al. (2012) provides a high-level framework in terms of using virtualization in high-performance computing. The work in Cui et al. (2012) closes the gap between the real and emulated systems by using receiver-side optimizations (optimistic interrupts and cut-through forwarding) in real time. Finally, Ibrahim et al. (2011) considers optimizations for live migration of VMs in high-performance systems.

Real-time emulation. Common Open Research Emulator (CORE) deploys a real-time network emulator with a hybrid approach, emulating a network stack of routers or hosts through virtualization and simulating the links that connect them together (Ahrenholz et al., 2008). CORE uses the FreeBSD network stack virtualization provided by the VirtNet[15] project. The lightweight virtualization, where only part of the OS is made virtual, allows CORE to scale to over a hundred VMs running on a single emulation server. For wireless emulation, CORE focuses on realistic emulation of layer 3 and above, and does not model layer 1 and 2 of a wireless medium such as 802.11.

Wireless channel emulators use hardware to simulate wireless channel propagation in real time. The reason for using hardware is that emulation must be fast to operate in real time, flexible to capture many different environments, and accurate to ensure that the radio signals are not distorted. There are a number of such wireless emulators (Kahrs and Zimmer, 2006; Picol et al., 2008), including the Field-Programmable Gate Array (FPGA)-based channel simulator (Borries et al., 2009).

RAMON, a rapid-mobility network emulator, is a software/hardware emulator that mimics realistic characteristics of wireless networks (Hernandez and Helal, 2002). RAMON allows the ns-2 simulator to interact with actual hardware components including access points (APs), attenuators, laptops, and smart phones, while emulating mobility in wireless networks.

Mobile Network Emulator (MNE) focuses on simulating mobility of wireless nodes, each of which is represented by a single physical device such as a laptop computer (Macker et al., 2003). Each device has two interfaces: one is used as a mobile emulation control channel and the other works as an emulated wireless interface. Information about topology changes are transmitted through the control channel and each node then sets or removes rules of its own 'iptables' according to the control information. Since it operates in a real-time environment, MNE uses simplistic propagation models that do not require significant amounts of processing, so MNE is applicable mainly for testing the protocols at layer 3 and above.

Large testbed. Several large testbeds, which partially use emulation techniques or significantly depend on them, have been developed. PlanetLab is a geographically distributed overlay network designed to support the deployment and evaluation of planetary-scale network services (Bavier et al., 2004). PlanetLab enables multiple services to run concurrently, each in its own *slice* of PlanetLab (Chun et al., 2003). Emulab provides researchers with integrated access to emulated PC nodes, an 802.11 a/b/g testbed, and universal software defined radios (Universal Software Radio Peripheral devices). Additionally, Emulab can be expanded into PlanetLab testbeds, enabling live Internet experimentation. Global Environment for Network Innovations (GENI)[16] is an National Science Foundation (NSF) initiative that supports at-scale experimentation on shared and heterogeneous infrastructure. GENI provides researchers across the country with collaborative environments on which new network architectures and their implementations can be tested. DETERlab is a testbed designed to support research and development on next-generation cyber security technologies. DETERlab uses the Emulab cluster testbed software that allows for controlling and managing a pool of PC experimental nodes that are interconnected in a network topology for testing. ModelNet emulates packet delays/losses/throughput of packets flowing between different instances of applications.

7 Conclusion

In this paper we presented a new heterogeneous system emulator based on virtual time featuring an adaptive TDF. The system emulates VMs by using a slightly modified KVM virtualization, allowing for heterogeneous VHs to be integrated in the same emulation. The experimental evaluation shows that the system scales well to a large number of VHs even when only a few physical machines are used to host the virtual systems. With a 1 Gbps backplane, a synchronization interval of 10 ms, five PHs, and our synchronization packet size of 60 B, the control channel utilization is just 0.024% and it is not a serious limitation. In conclusion, the system can be expanded to a large PH cluster and can be used to evaluate the performance of large-scale heterogeneous systems without having to build large, expensive, and custom testbeds.

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Notes

1. See http://www.planet-lab.org.
2. See http://www.emulab.net.
3. See http://www.isi.deterlab.net.
4. See http://modelnet.ucsd.edu.
5. See http://wiki.openvz.org.
6. See http://www.linux-kvm.org.
7. See http://linux-vserver.org.
8. See http://www.vmware.com.
9. See http://www.virtualbox.org.
10. See http://www.xenproject.org.
11. See http://www.juniper.net/products/junos and http://juniper.cluepon.net/index.php/Olive.
12. See http://netflix.com.
13. See http://hulu.com.
14. See http://videolan.org.
15. The FreeBSD Network Stack Virtualization Project, http://imunes.tel.fer.hr/virtnet.
16. See http://www.geni.net.

References

Ahrenholz J, Danilov C, Henderson T, et al. (2008) CORE: A real-time network emulator. In: *Military communications conference*, pp. 1–7.

Bavier A, Bowman M, Chun B, et al. (2004) Operating system support for planetary-scale network services. In: *Proceedings of the 1st conference on networked systems design and implementation – volume 1*, Berkeley, CA.

Bergstrom C, Varadarajan S and Back G (2006) The Distributed Open Network Emulator: Using relativistic time for distributed scalable simulation. In: *20th workshop on principles of advanced and distributed simulation*, pp. 19–28.

Bertsekas DP and Gallager R (1992) *Data Networks*. 2nd edn. Englewood Cliffs, NJ: Prentice Hall.

Borries K, Judd G, Stancil D, et al. (2009) FPGA-based channel simulator for a wireless network emulator. In: *IEEE 69th vehicular technology conference*, pp. 1–5.

Bridges PG, Arnold D, Pedretti KT, et al. (2012) Virtual-machine-based emulation of future generation high-performance computing systems. *International Journal of High Performance Computing Applications* 26(2): 125–135.

Chun B, Culler D, Roscoe T, et al. (2003) PlanetLab: An overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review* 33: 3–12.

Cui Z, Xia L, Bridges PG, et al. (2012) Optimizing overlay-based virtual networking through optimistic interrupts and cut-through forwarding. In: *Proceedings of the international conference on high performance computing, networking, storage and analysis*, Los Alamitos, CA.

Grau A, Herrmann K and Rothermel K (2009) Efficient and scalable network emulation using adaptive virtual time. In: *Proceedings of the 18th international conference on computer communications and networks*, pp. 1–6.

Grau A, Herrmann K and Rothermel K (2010) NETplace: Efficient runtime minimization of network emulation experiments. In: *2010 international symposium on performance evaluation of computer and telecommunication systems*, pp. 265–272.

Grau A, Herrmann K and Rothermel K (2011) NETbalance: Reducing the runtime of network emulation using live migration. In: *Proceedings of the 20th international conference on computer communications and networks (ICCCN)*, pp. 1–6.

Grau A, Maier S, Herrmann K, et al. (2008) Time jails: A hybrid approach to scalable network emulation. In: *22nd workshop on principles of advanced and distributed simulation*, pp. 7–14.

Gupta D, Vishwanath KV and Vahdat A (2008) DieCast: Testing distributed systems with an accurate scale model. In: *Proceedings of NSDI*, pp. 407–421.

Gupta D, Yocum K, McNett M, et al. (2005) To infinity and beyond: Time-warped network emulation. In: *ACM symposium on operating systems principles*.

Hernandez E and Helal A (2002) RAMON: Rapid-mobility network emulator. In: *27th annual IEEE conference on local computer networks*, pp. 809–817.

Ibrahim KZ, Hofmeyr S, Iancu C, et al. (2011) Optimized pre-copy live migration for memory intensive applications. In: *Proceedings of the 2011 international conference for high performance computing, networking, storage and analysis*.

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

Kleinrock L (1975) *Queueing Systems Volume 1: Theory*. New York, NY: Wiley.

Law AM and Kelton DM (1999) *Simulation Modeling and Analysis*. 3rd edn. New York, NY: McGraw-Hill Higher Education.

Lee HW, Thuente D and Sichitiu ML (2014) Integrated simulation and emulation using adaptive time dilation. In: *Proceedings of the 2nd ACM SIGSIM/PADS conference on principles of advanced discrete simulation*, New York, NY, pp. 167–178.

Lucio GF, Paredes-Farrera M, Jammeh E, et al. (2003) OPNET modeler and ns-2: Comparing the accuracy of network simulators for packet-level analysis using a network testbed. In: *3rd WEAS international conference on simulation, modelling and optimization (ICOSMO)*, pp. 700–707.

Macker J, Chao W and Weston J (2003) A low-cost, IP-based mobile network emulator (MNE). In: *Military communications conference*, pp. 481–486.

Mukherjee J and Varadarajan S (2005) Weaves: A framework for reconfigurable programming. *International Journal of Parallel Programming* 33: 279–305.

Pawlikowski K, Jeong HDJ and Lee JSR (2002) On credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine* 40: 132–139.

Picol S, Zaharia G, Houzet D, et al. (2008) Hardware simulator for MIMO radio channels: Design and features of the digital block. In: *IEEE 68th vehicular technology conference*, pp. 1–5.

Weingärtner E, Schmidt F, Heer T, et al. (2008) Synchronized network emulation: Matching prototypes with complex simulations. *SIGMETRICS Performance Evaluation Review* 36: 58–63.

Weingärtner E, Schmidt F, Lehn HV, et al. (2011) SliceTime: A platform for scalable and accurate network emulation. In: *Proceedings of the 8th USENIX conference on networked systems design and implementation*, Berkeley, CA.

Xia H, Dail H, Casanova H, et al. (2004) The MicroGrid: Using online simulation to predict application performance in diverse grid network environments. In: *Proceedings of the second international workshop on challenges of large applications in distributed environments*, pp. 52–61.

Younge A, Henschel R, Brown J, et al. (2011) Analysis of virtualization technologies for high performance computing environments. In: *2011 IEEE international conference on cloud computing (CLOUD)*, pp. 9–16.

Zheng Y and Nicol DM (2011) A virtual time system for OpenVZ-based network emulations. In: *Proceedings of the 2011 IEEE workshop on principles of advanced and distributed simulation*, Washington, DC.

Author biographies

Hee Won Lee is a Ph.D. candidate in the Department of Computer Science at North Carolina State University. He received his Bachelor of Engineering from Korea University in 2002, and Master of Software Engineering from Carnegie Mellon University in 2005. He worked for KT Corporation as a research engineer in 2002-2009. He also worked for AT&T Labs Research as an intern during the 2014 summer. His research interests include networking and storage systems, distributed and cloud computing, high performance computing, and network simulation.

Mihail L. Sichitiu received his B.E. and an M.S. in Electrical Engineering from the Polytechnic University of Bucharest in 1995 and 1996 respectively. In May 2001, he received a Ph.D. degree in Electrical Engineering from the University of Notre Dame. He is currently employed as a professor in the Department of Electrical and Computer Engineering at North Carolina State University. His primary research interest is in Wireless Networking.

David J. Thuente received a Summa Cum Laude Honors B.S. degree in Mathematics from Loras College. He received his MS and Ph.D. degrees from the University of Kansas. He is currently Professor of Computer Science at North Carolina State University. He has done extensive consulting in sonobuoy signal processors and networking protocols and applications for Magnavox Electronic Systems Company. He has also done network consulting for Hughes Systems Company and Raytheon among others. He is a Professor Emeritus of Purdue University. His primary research area is networking.