VIRTUAL TIME-AWARE VIRTUAL MACHINE SYSTEMS

A Dissertation
Presented to
The Academic Faculty

by

Srikanth B. Yoginath

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology
August 2014

VIRTUAL TIME-AWARE VIRTUAL MACHINE SYSTEMS

ACKNOWLEDGEMENTS

First and foremost, I am extremely grateful to have Dr. Kalyan Perumalla as my advisor at Oak Ridge National Laboratory (ORNL) and this thesis would not have been possible without his guidance and support. I am highly impressed with his expertise, enthusiasm and his ability to quickly glean the core of any given research problem. Under his supervision, I have been exposed to wide range of very interesting research projects and have learned a great deal by participating in them. I feel extremely fortunate and blessed to have had an opportunity to work as his research student.

I am also extremely grateful to Dr. Richard Fujimoto, for being my advisor at Georgia Tech. I am honored to work as his student and would like to express my deepest gratitude for his support throughout my PhD endeavor. I would also like to thank Dr. David Bader, Dr. George Riley and Dr. Umakishore Ramachandran for serving as my thesis committee members, and for their insightful comments.

I am extremely grateful to ORNL for providing me with an opportunity and the necessary financial support to pursue my PhD. At ORNL, I would like to thank Computational Sciences and Engineering Division directors Dr. Brian Worley and Dr. Shaun Gleason, and the Modeling and Simulation Group leaders Mr. David Hetrick and Dr. Robert Morris for their continuous encouragement and support. I would also like to thank Ms. Carolyn Ward at ORNL education assistance for making the logistic work related to my PhD effortless. I am also grateful for the timely funding of our research projects from Army Research Laboratory (ARL). I would like to thank Dr. Raju Namburu and Dr. Brian Henz from ARL for their support.

At Georgia Tech, I would like to thank CSE graduate program advisors Mr. Michael Terrell and Ms. Haley Mimi, who have been extremely helpful and supportive. I would also like to thank Ms. Holly Rush for helping me make necessary arrangements for my dissertation defense. I very much appreciate Ms. Lometa Mitchell and Ms. Phinisee Della for their help.

Finally, I would like to express my sincere gratitude to my wife, kids, parents and brothers for their continuous loving support throughout my PhD pursuit. Without their support and encouragement this work would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CI	Confidence Interval
CND	Constant Network Delay
CONS	Conservative synchronization scheme for PDES
CPU	Central Processing Unit
CSX	Credit Scheduler of Xen
DES	Discrete Event Simulation
DOM	Domain or VM
DOM0	Domain-0 or Control VM
DSB	Disease Spread Benchmark
GVT	Global Virtual Time
LLF	Least-LVT-First
LOC	Locality
LP	Logical Process
LVT	Local Virtual Time
NLP	Number of LPs
NMSG	Number of Messages/LP
NNC	NetWarp Network Control
NSX	NetWarp Scheduler for Xen
OPT	Optimistic synchronization scheme for PDES
OS	Operating System
PCPU	Physical CPU
PDES	Parallel Discrete Event Simulation
PSB	PHOLD Simulation Benchmark

PSX	PDES Scheduler for Xen
VCPU	Virtual CPU
VM	Virtual Machine
VND	Variable Network Delay
VTS	Virtual Time Systems

SUMMARY

Discrete dynamic system models that track, maintain, utilize, and evolve virtual time are referred to as *virtual time systems* (VTS). The realization of VTS using *virtual machine* (VM) technology offers several benefits including fidelity, scalability, interoperability, fault tolerance and load balancing. The usage of VTS with VMs appears in two ways: (a) VMs within VTS, and (b) VTS over VMs. The former is prevalent in high-fidelity cyber infrastructure simulations and cyber-physical system simulations, wherein VMs form a crucial component of VTS. The latter appears in the popular Cloud computing services, where VMs are offered as computing commodities and the VTS utilizes VMs as parallel execution platforms.

Prior to our work presented here, the simulation community using VM within VTS (specifically, cyber infrastructure simulations) had little awareness of the existence of a fundamental virtual time-ordering problem. The correctness problem was largely unnoticed and unaddressed because of the unrecognized effects of fair-share multiplexing of VMs to realize virtual time evolution of VMs within VTS. The dissertation research reported here demonstrated the latent incorrectness of existing methods, defined key correctness benchmarks, quantitatively measured the incorrectness, proposed and implemented novel algorithms to overcome incorrectness, and optimized the solutions to execute without a performance penalty. In fact our novel, correctness-enforcing design yields better runtime performance than the traditional (incorrect) methods.

Similarly, the VTS execution over VM platforms such as Cloud computing services incurs large performance degradation, which was not known until our research uncovered the fundamental mismatch between the scheduling needs of VTS execution and those of traditional parallel workloads. Consequently, we designed a novel VTS-aware hypervisor scheduler and showed significant performance gains in VTS execution over VM platforms. Prior to our work, the performance concern of VTS over VM was largely unaddressed due to the absence of an understanding of execution policy mismatch between VMs and VTS applications. VTS follows virtual-time order execution whereas the conventional VM execution follows fair-share policy. Our research quantitatively uncovered the exact cause of poor performance of VTS in VM platforms. Moreover, we proposed and implemented a novel virtual time-aware execution methodology that relieves the degradation and provides over an order of magnitude faster execution than the traditional virtual time-unaware execution.

CHAPTER 1

INTRODUCTION

1.1 Virtual Machine (VM)

Starting from early 1960s, the concept of virtualization has been realized at many levels of the computer systems architecture, and continues to receive considerable attention even today, enabling novel computing solutions such as Cloud computing and computing as a service.

At the operating system (OS) level, a type of virtualization called "OS-level virtualization" can be realized, which is distinct from the hardware virtualization technology in vogue today. The OS-level virtualization technology enables multiple isolated execution environments *within* a single OS kernel. This instance or a container works as a stand-alone server, which can be shutdown, rebooted, provide root-access, instance-specific users, isolated memory, network (IP) address, processes, file systems, etc., but does not support different kernel from different operating systems to run at the same time. The concept of a virtual *machine* is completely absent in such virtualization. OpenVZ [2], FreeBSD Jail [3] and Solaris Zones/Containers [4] are a few well-known distributions that use OS-level virtualization.

At the user level, the applications can be run in a virtual execution environment, thus making the application highly portable across wide range of operating systems. The Java Virtual Machine (JVM) [1] is a popular example of such application-level virtualization. The concept of "library virtualization" also falls in this category, such as the Wine system that provided a subset of Win32 API as a library to allowed windows desktop applications to be executed in Linux environment.

At the hardware-level, two methodologies are used for virtualization, namely, *full-virtualization* and *para-virtualization*. Although the two types are similar in functionality, they differ in the underlying means used to realize virtualization. With the support for virtualization at the hardware-level from the processor vendors eased up the realizing certain characteristics of the hypervisor, their distinctions have continued to remain. Both these approaches run on the top of the hardware by pushing the OS above them and address thus address similar challenges. Both these approaches make use of highly configurable virtual machines comprising virtual peripheral I/O components and hence provide an isolated virtual machine environment for every guest OS hosted. Para-virtualization differs from full-virtualization in requiring the modification of guest OS kernel, while the full-virtualization can host OS without any modifications. VMware ESX Server [5] hypervisor was principally designed to support full-virtualization, the Xen [6][7] hypervisor started with the concept of para-virtualization but currently supports full virtualization also and Microsoft Hyper-V hypervisor also supports full-virtualization. In the entire thesis the virtualization at the hardware-level is discussed and used.

1.1.1 Virtual Machines History

The concept of Virtual Machines (VM) existed even before Intel released their first microprocessor 4004 in 1971. As early as the 1970s [8], there is mention of virtual machine CP-67 on IBM 360/67, a mainframe model, and the formal requirements are discussed for the third generation computer systems. Nevertheless, the significance of virtualization as a practical consumer technology was not realized till the end of the 20th century, at time by which information technology (IT) services had effectively permeated through almost all industry and business sectors.

Any modern computing machine comprises of physical hardware such as processor, memory, hard-disk, network-cards, etc., along with low-level software called *drivers* to interact with the hardware peripherals and these software elements are generally bundled up with the operating system (OS). Thus, in all the current systems lacking VM the OS is tightly coupled with the physical hardware. The VM with all its virtual peripherals running the same OS is loosely coupled to the physical hardware in such a way that it can detach from the executing physical hardware and migrate to another even *during* its execution. Further, the loose coupling also helps to host multiple instances of VMs of varying configurations, running same OS or different OSs, to run concurrently by sharing resources of their physical hardware host. Hence, the basic requirement in realizing a VM is to decouple the OS from the physical hardware and make it interact with its virtual counterpart, and also to be able to multiplex many virtual peripherals on to limited physical peripherals.

Popek and Goldberg [8] formally define and illustrate the requirements for VM. They aptly define VM as an efficient, isolated duplicate of a real machine. They elaborate on the concept of VM in the context of a Virtual Machine Monitor (VMM) also commonly known as a hypervisor. VM is the environment created by the VMM such that the software-based duplicate of real machine can function in isolation. They specified three characteristics that the VMM should possess to realize a VM. First, the VMM is expected to provide for programs an environment that is essentially *identical* to the physical machine; secondly, programs that are executed in such an environment experience some decrease in speed; thirdly, the VMM is in complete control of the system resources.

1.1.2 Virtualization of the X86 Architecture

Although virtualization technology existed for long time, it was not until the virtualization of the 32-bit x86 architecture by VMware for Linux in 1999 that its true power was realized by the industry and elsewhere. By then, the fourth generation microprocessor-based computer systems had become an integral part of almost every industry in one form or another. The OS that was working as the lowest level software providing an easy interface to develop, deploy and use a wide range of applications, had essentially grown the market for the computer systems. Further, this positive expanse of computing systems gave way to several practical issues such as inefficient use of resources, interoperability, reliability, security, OS migration, and so on. It was discovered by the pioneers of x86 virtualization that the flexibility afforded by virtual machine monitors could solve, simply and elegantly, a number of hard system software problems by innovating at the hypervisor layer below operating systems [9].

To add the requisite level of indirection to the computer hardware, new mechanisms were used to trap and emulate direct execution code for application software and dynamic binary translation for system software. In the course of time, via steady improvements to hardware technologies, such as carefully reworking the segment protection mechanism, the system code was eventually made to run at near-native speeds (comparable to non-virtualized execution), at least as far as the operating system mechanisms are concerned. The approach taken was "top-down addressing virtualization" of a single OS. To virtualize the highly diverse I/O peripherals in x86 systems, multiplexing and emulated components were used. By multiplexing, the physical hardware was shared in space and time across multiple VMs. Via emulation techniques, hardware-simulation software was used to support a wide range of VMs with different, largely unmodified, operating systems. For the BIOS, the VMM loaded into VM's ROM a copy of VMware BIOS (licensed from Phoenix Technologies) to serve as the x86 platform-specific firmware that first initializes hardware and then loads system software from the peripherals. The evolution of x86 architecture virtualization is well documented in terms of the efforts, inherent problems, challenges and solution approaches [9].

From that point on, there has been a dramatic increase in the research, application and utilization of virtual machines for various innovative purposes. Similarly, various techniques of virtualization have been proposed and realized at many levels in the computer system, with a wide variety of benefits.

1.1.3 Current Trends in Virtualization Technology

Currently, the expanse of the applications based on virtualization technology spans from a platform as small as a single user desktop to a massive system as huge as a commercial data center such as that of Amazon. On the smaller end, virtualization allows desktop users to concurrently host multiple OS on the same hardware. On a larger scale, the fault tolerance and economical-hosting possibilities of virtualization can be exploited because a VM has the capability to move from one hypervisor to another while running and also move from one storage device to another. Such agility enables automatic load balancing on a cluster of hypervisor servers, and, in combination with low-latency uptime, this technology addresses the fault-tolerance. Fault tolerant execution in case of hardware failure allows the hosted servers to automatically restart a VM on another host of a cluster.

A very attractive product for both business operators and users alike arose from tapping the virtualization technology through the Internet services, which famously came to be known as *Cloud computing*. The Infrastructure-as-a-Service (IAAS), Platform-as-a-Service (PAAS), Software-as-a-Service (SAAS) and Network-as-a-Service (NAAS) are prominent among the types of services offered currently by Cloud computing service vendors [10]. IAAS offers a cluster of user-specified hypervisors along with additional resources such as VM disk-image library, firewalls, load-balancers, IP-addresses, and so on. The PAAS, one level higher than IAAS, allows users to choose VMs running certain types of OS, with selected types of support for databases, programming languages, and so on. The SAAS, one level higher over PAAS, provides users the choice of different types of applications that they would like to use (e.g., Microsoft Office). With NAAS, the users are provided options for VPN-enabled networked systems with varying interconnect options, such as bandwidth, essentially creating a private cloud within a commercial, shared cloud.

Except for IAAS, the other services are essentially oblivious to the physical hardware they are executing on. Based on the reasonable assumption of the unlikeliness of 100% resource utilization from all clients at all times, the Cloud operators multiplex VMs on limited resources and hence are able provide easy accessibility to large compute resources at compellingly competitive prices.

1.1.4 The Xen Hypervisor

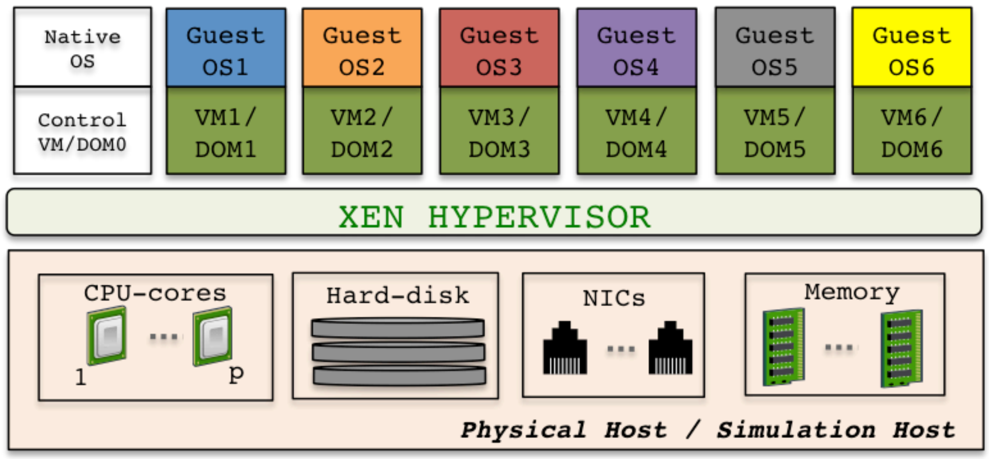


Figure 1 Xen Hypervisor

The Xen hypervisor is a popular open source industry standard for virtualization, supporting a wide range of architectures including x86, x86-64, IA64, and ARM, and guest OS types including Windows®, Linux®, Solaris® and various versions of BSD OS. Figure 1 shows a schematic of guests running on the Xen hypervisor. Xen refers to VMs as Guest Domains or DOMs. Each DOM is identified by its DOM-ID. The first DOM, "DOM0," affords special hardware privileges. Each DOM has its own set of virtual devices, including virtual multi-processors called virtual CPUs (VCPUs). System administration tasks such as suspension, resumption, and migration of DOMs are managed via DOM0.

Xen differs from VMware ESX Server in that the hypervisor only virtualizes and manages the CPU and memory resources, and delegates I/O operations, including the device drivers, to a privileged virtual machine called DOM0. Microsoft's Hyper-V shares the same architecture as Xen in which the Windows root partition replaces Xen's

DOM0. The VMware's ESX server uses *vmkernel* along with *VMM*. The *vmkernel* is responsible for global resource management as well as I/O operations; in particular, the *vmkernel* natively runs performance-critical device drivers [9].

1.2 Virtual Time Systems (VTS)

1.2.1 Background

A representation of a physical system, or more aptly a System Under Investigation (SUI), which is abstracted in order to understand, reason or engineer the subtle characteristics of the SUI, is called a model. Simulation is an imitative representation of functioning of one system (SUI) by means of the functioning of the model (which is typically a computer program). Together, modeling and simulation approaches have been applied to address wide range issues arising in various disciplines of science and engineering.

The system models can be broadly classified as deterministic and stochastic models. The deterministic models are characterized by the absence of random variable components while the stochastic models are characterized by their presence. For example, stochastic computer network simulation models based on queuing models use a variety of probability distributions for network packet-generation time, packet-processing time, and so on. An example of a deterministic system model is digital circuit simulation, where the output is deterministic for a given set of inputs.

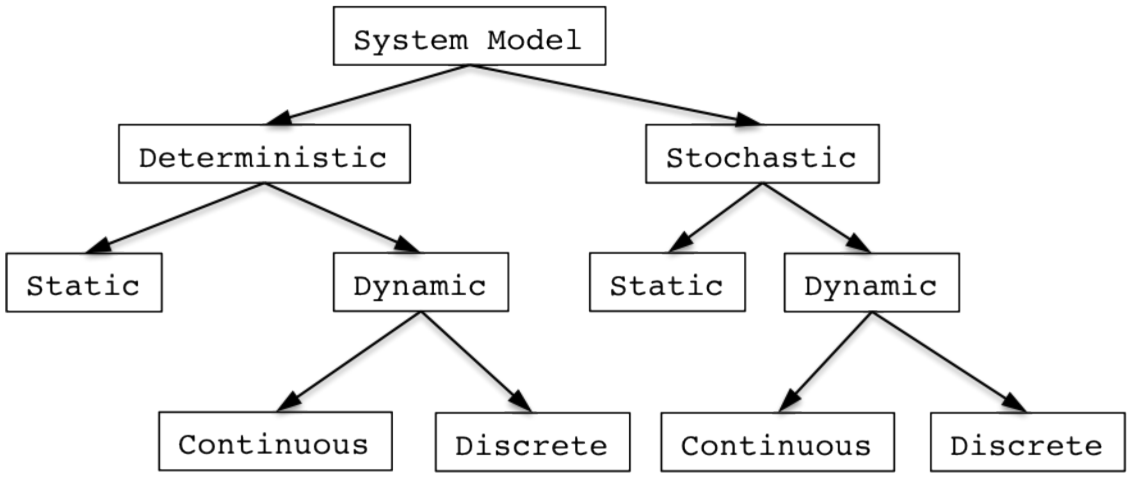


Figure 2 System model classification

Each of these system models (deterministic and stochastic) is further classified into static and dynamic, based on the models' dependence on the temporal component. In static models, the temporal component is absent or is insignificant, while the dynamic models are designed to imitate the evolution of the SUI over time. Monte Carlo simulations are an example of static stochastic system models and linear programming is an example of static deterministic system models. Regardless of being deterministic or stochastic, the time component is tightly coupled with the dynamic systems and these system models associate the characteristic change of the system model state to their time component. To differentiate the time component of a system model from the wall-clock time or real time of the computer executing the model, we refer to the time of the system model as the *virtual time*. All system models that maintain and/or use virtual time for simulation are referred to as *Virtual Time System* (VTS).

VTS models can be classified into two as (a) continuous and (b) discrete. In the continuous system model, the state of the system model is perceived to change continuously with the *virtual time*, while in the discrete system model, the system model is perceived to change the state at discrete points of *virtual time*. Although the principles discussed are generic and could be applicable to other models, we restrict our discussions to the discrete VTS models in this thesis.

Discrete event simulations are an example of the discrete VTS model. Discrete event simulation of an SUI emulates the behavior of the SUI over time in terms of events and state changes. An event is a discrete point in time that corresponds to an instant in the evolution of the SUI. A state of the simulation model is a snapshot of the evolving SUI at a given instant of time. Generally, an event is associated with a change in the state.

In discrete event simulations, the modeler defines the different states and events of the SUI to be simulated during the modeling phase. This definition of states and events varies with the specific goals of the simulation exercise. Thus, a discrete event simulation comprises a set of states and events. To ensure that the transition across the model states realistically emulates the SUI transitions over real time, the events generated by the model must be executed in virtual time-order.

There are primarily three different worldviews for realizing discrete event simulations: (a) activity scanning approach, (b) event oriented approach, (c) process oriented approach. Conceptually, the worldviews address the same virtual time-ordered execution issue via different execution techniques. Nevertheless, each worldview has its advantages and disadvantages in realizing the discrete-event simulations [11].

Regardless of the worldviews, each event in the model would contain a specific virtual time that is used for queuing the event in a Virtual Time-ordered Priority-Queue (VTPQ). During processing of events, additional future events will be queued in VTPQ.

Hence, a discrete-event simulation essentially involves continuously processing the events from a VTPQ.

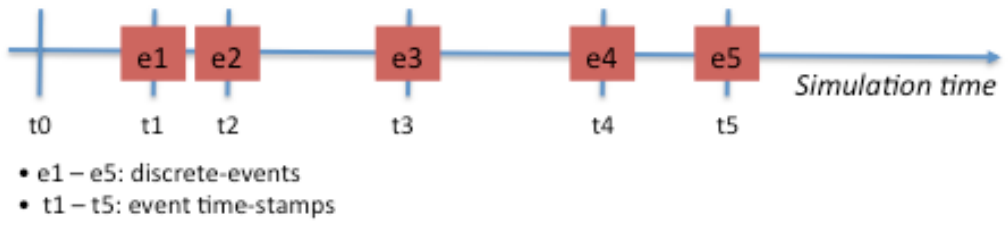


Figure 3 Illustration of timeline and events in discrete-event simulation

1.2.2 Parallel Discrete Event Simulation (PDES)

PDES involves execution of several serial sequential DES systems in parallel. Usually the SUI to be simulated is spatially divided into a set of similar partitions and a DES is used to simulate each such partition. The DES process in PDES is generally referred to as a Logical Process (LP). Figure 4 depicts the parallel execution (involving 2 LPs) of the sequential discrete-event simulation model shown in Figure 3.

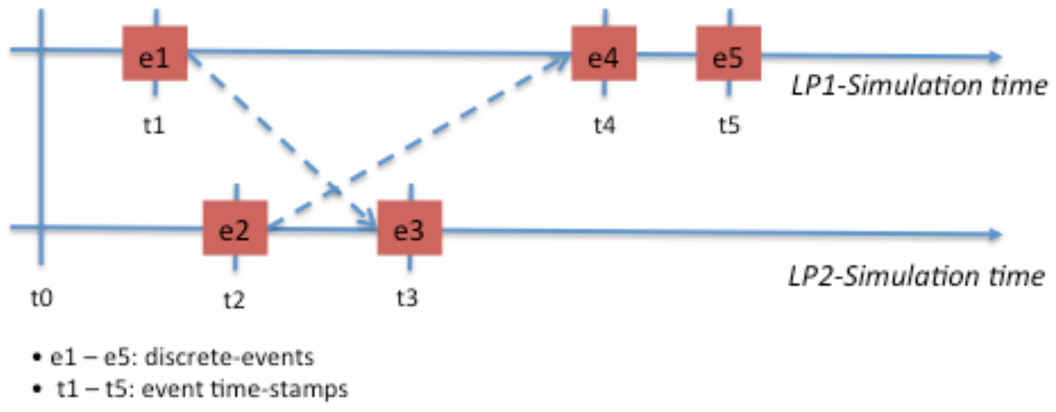


Figure 4 Parallel execution of discrete-event simulation

As shown in Figure 4, each LP maintains its own simulation time line, and LPs have the capability to schedule events on their peers' time lines.

A verification measure, wherein the application running on the PDES framework yields exactly same results as its sequential counterpart, is termed as the *correctness* of parallel execution. Note that, with parallel execution, we lose the notion of a single simulation time-line, and hence, to ensure *correctness* in execution, it becomes necessary to enforce some form of a synchronization mechanism across LPs. Two distinct synchronization mechanisms are popularly used in the PDES community namely, (a) conservative synchronization [12] [13], and (b) optimistic synchronization [14].

Conservative synchronization approaches enforce a strict rule on each LP such that any attempt to breach the virtual-time-ordered processing of events is thwarted. In contrast, the optimistic synchronization approaches allow the LPs to (temporarily and transitorily) breach virtual-time ordered processing or commit execution errors, and provide mechanisms to revert back committed errors, if any, to eventually ensure *correctness*. A large body of research work spanning past three decades has elaborately addressed various facets of the synchronization issues in parallel discrete event simulations and they are very well documented [12].

1.2.2.1 Lookahead

The amount of parallelism in PDES is directly dependent on the number of events that can be concurrently processed by each processor. In sequential DES, the processing of events generates more events for the future. The virtual time of any future event is essentially a positive-time leap from the current virtual time. The time period between these events (current and next virtual time) is specific to the SUI being modeled;

furthermore, within a given SUI being modeled, it could span a wide range values. The decomposition of timelines of DES into PDES results in a set of LPs that can schedule events to themselves and/or to other LPs at a virtual time in future. The minimum virtual time-period between the sending time and receiving time of any event is termed as the *lookahead* of the LP.

This *lookahead* parameter plays an important role in the performance of PDES execution. The larger the lookahead value, the greater the concurrency.

1.2.2.2 Lower Bound on Incoming Time Stamps (LBTS)

With the *lookahead* guarantee on the minimum value of timestamps generated at any given moment, the LPs essentially guarantee a specified virtual-time period in between the sequence of events they might schedule on other LPs. However, this guarantee can only be used for event processing by an LP when the (lower bound of) the virtual times of unprocessed events of its peer LPs are known. This globally minimum virtual time, called LBTS, is computed by the PDES algorithms to guarantee that no LP will schedule events with a virtual time lower than the LBTS. To segregate its local events as *safe* or *unsafe*, each LP uses the LBTS value, thereby avoiding violating the *correctness* criterion. Thus, the computation of successively larger LBTS values dynamically is essential for conservative synchronization scheme to function in parallel.

1.2.2.3 Global Virtual Time (GVT)

The optimistic synchronization schemes ensure the *correctness* in the processing of events in virtual time order in the distributed environment by rolling back out-of-order events. Optimistic synchronization involves addressing issues such as: (a) the memory limitation issues that arise from state-saving needs for rollbacks (b) handling of events that cannot be rolled back but are necessary, such as the I/O operations (c) ensuring the regular global progress of simulation so that a straggler event in the future would not revert back to the start of simulation, and (d) detecting simulation termination.

GVT, a property of an instantaneous global snapshot of the distributed system in real time, is utilized in addressing all the aforementioned issues. GVT has many equivalent definitions. Informally, it is the smallest "unprocessed" time stamp among all the *virtual clock*s in the distributed system including transient messages. In other words, it is the minimum among all the virtual times of all the scheduled-but-not-yet-executed events in the system. It serves as a lower bound on the virtual times to which any process can ever again rollback. It allows committing results of the events that cannot be reverted.

1.2.3 PDES Models

In PDES, the model is divided into distinct independent virtual timelines referred to as Logical Processes (LPs). Each LP typically encapsulates a set of state variables of a modeled entity. The timelines of LPs within and across processors are kept synchronized by the PDES engine. The μsik [16] parallel/distributed simulation kernel built upon a micro-kernel architecture is used for most of our experimentations. It is an optimized implementation of a parallel discrete event simulation interface that provides the option of conservative as well as optimistic synchronization. The synchronization protocols are the blocking-based variant of the set of highly scalable global virtual time (GVT) algorithms recently tested at very large scale on supercomputers [17]. The algorithms are implemented using the blocking collective call of Message Passing Interface (MPI), namely, the `MPI_Allreduce()` call, inside the iterations of the GVT algorithm to account for all transient messages. This is a Mattern-style [18] epoch-based framework that colors messages and uses reduction-based counting of transient messages. The parallel implementation of μsik [16] has previously been reported to be competitive with sequential execution, with a high efficiency.

Three applications namely, PHOLD [19] (a synthetic PDES application generally used for performance evaluation), Disease Spread Simulation [20] and SCATTER [21][22] (reverse-computation-based vehicular traffic PDES application) are used in our performance studies.

1.2.3.1 PHOLD

This is a widely used synthetic benchmark for performance evaluation by the PDES research community. This PDES application randomly exchanges a fixed population of events among the LPs. The μsik implementation of PHOLD allows exercising a wide variety of options in its execution. In all of our PHOLD benchmarks, we use a lookahead value of 1.0 and exercise combinations of the following parameters for performance evaluation.

- Synchronization: optimistic (OPT) or conservative (CONS)
- Number of LPs per Federate (NLP) [for example: 10 NLP = 10 LPs/federate]
- Number of messages per LP (NMSG) [for example: 10 NMSG = 10 messages/LP]
- Destination locality of the LP generated message (LOC) specifies the percentages of local and remote events. Values of 50, 90 and 100 suggest respectively that 50%, 90% and 100% of the messages generated by an LP are local to its federate. Hence, a value of 50% for LOC involves more LP message exchanges across the network and results in increased network traffic; a value of 90% results in a reasonable amount of

inter-federate event traffic, and, 100% suggests an embarrassingly parallel PDES application involving little inter-federate interaction except for GVT computations.

1.2.3.2 Disease Spread Simulation

As a second application, we use an epidemiological disease spread model that defines a discrete event model for the propagation of a disease in a population of individuals in groups called locations and aggregates of locations called regions. Each region is mapped to a Federate. Multiple locations are contained in each region. Each location is housed within an LP. Multiple individuals are instantiated at each location, and they not only interact with individuals within the same location but also periodically (conforming to an individual-specific time distribution function) move from one location to another. Similar to PHOLD, the number of individuals per location is varied (e.g., 1000 individuals/location), and the number of locations per region is also varied (e.g., 10 locations/region).

1.2.3.3 SCATTER

This application is a discrete-event formulation and a parallel execution framework for vehicular traffic simulation. It uses the µsik library for parallel execution and is amenable to both conservative (CONS) and optimistic (OPT) synchronizations. A simulation scenario is set up by reading an input file that specifies the road network structure, number of lanes, speed limit, source nodes, sink nodes, vehicle generation rate, traffic light timings and other relevant information. Dijkstra’s shortest-path algorithm is used to direct a vehicle to its destination.

1.3 Problem Statement and Research Challenges

The relation of VTS with VMs, is two-fold: (a) VM within VTS, and (b) VTS over VM. Hence, the problem statement, the research challenges and subsequently the research contributions of this dissertation, also fall in two categories, as described in the next two sub-sections.

1.3.1 VM within VTS

The use of VMs within PDES (discrete VTS) appears prominently in the field of cyber-infrastructural and cyber-physical simulations, such as computer network simulation/emulations, where fidelity, portability, performance and scalability define the design criteria. Driven by the need to understand the dynamics of complex parallel/distributed network application behaviors whose software cannot be easily modeled without actually re-implementing them, network emulation platforms such as Emulab were designed. However, in all traditional emulation approaches, virtual time was intermixed with real time or wall-clock time, creating the possibility of pollution of the notion of time in general.

Even if the VM technologies are used on physical machines with multiple compute resources with each VM mapped to a single, distinct compute resource, the correctness of the simulation/emulation cannot be guaranteed because the scheduler that shares the physical compute-resources among virtual compute resources does not take idle time of VMs into consideration, and, in the simulation, we can rarely expect all the subsidiary parallel computing peripherals to be perfectly load-balanced. Even if this issue can be resolved by strictly pinning every physical compute-resource to a particular virtual compute-resource, a loss of performance in such case is unavoidable.

These limitations of any emulation platform is due to its dependence on real-time for determining the simulation time. Detaching the emulation's reliance on real-time resolves most of the above discussed problems but doing so involves addressing several challenges. The disassociation with real-time puts the burden on the modeler to maintain the simulation time. The following fundamental challenges will need to be addressed:

- How can the system take into account and maintain the simulation time?
- How can the execution ensure a global-simulation-time across all VMs?
- How can multiple VMs be handled in time-order with multi-core platforms?
- How can multiple VMs be scheduled to ensure correctness of simulation?
- How can multiple VMs of varying compute capacity (differing in number of virtual compute cores) be accommodated?
- How can the correctness of the simulation system be measured?
- How can the correctness of the simulation be demonstrated?
- How can the correctness issues in complex network simulations be convincingly identified?
- To what extent does the system performance get affected?
- How well does the system scale?

1.3.2 VTS over VMs

The rapidly changing landscape of computing due to the largely appealing economical and technological advantages of VM technologies is bound to impact PDES based simulations. While, the next generation computations will be realized and accounted for in terms of resource utilization of VMs, the PDES adaptability is still unexplored in this new execution environment that directly impacts its performance.

A large portion of prior work on PDES targeted high-performance computing platforms where dedicated compute units with high-bandwidth and low-latency networks are assumed. On the contrary, the VM platforms alter the traditional assumptions about the computing platform since the VM-based execution platforms share the compute resources and might migrate to another physical node during execution. While, in the HPC execution platform, the physical hardware is always precisely specified prior to execution, that might not be the case in VM-based execution environments. Furthermore, the physical hardware hosting the VM could change even *during* the run. Hence, one basic challenge is to evolve some rules of thumb or guidelines to the PDES user in order to efficiently utilize the VM-based execution platforms.

Despite the challenges that the VM execution environment poses, it also offers several new capabilities for a PDES user. One important aspect to be noted about VMs is that their execution parameters (such as VM-to-processor mapping) can be significantly altered quite efficiently during runtime. PDES applications are generally characterized by their highly dynamic and unbalanced computational loads on the LPs (and thereby on the processors). Hence, performance-tuning opportunities arise with VM-based execution platforms that can be flexibly controlled at runtime. In exploiting this flexibility of VMs, the challenge is to realize a VM-based execution mechanism for PDES applications that appropriately adapts the resource-sharing operations at runtime based on the needs of the application, without the need for any user-monitoring and intervention.

1.4 Research Contributions

1.4.1 VM within VTS

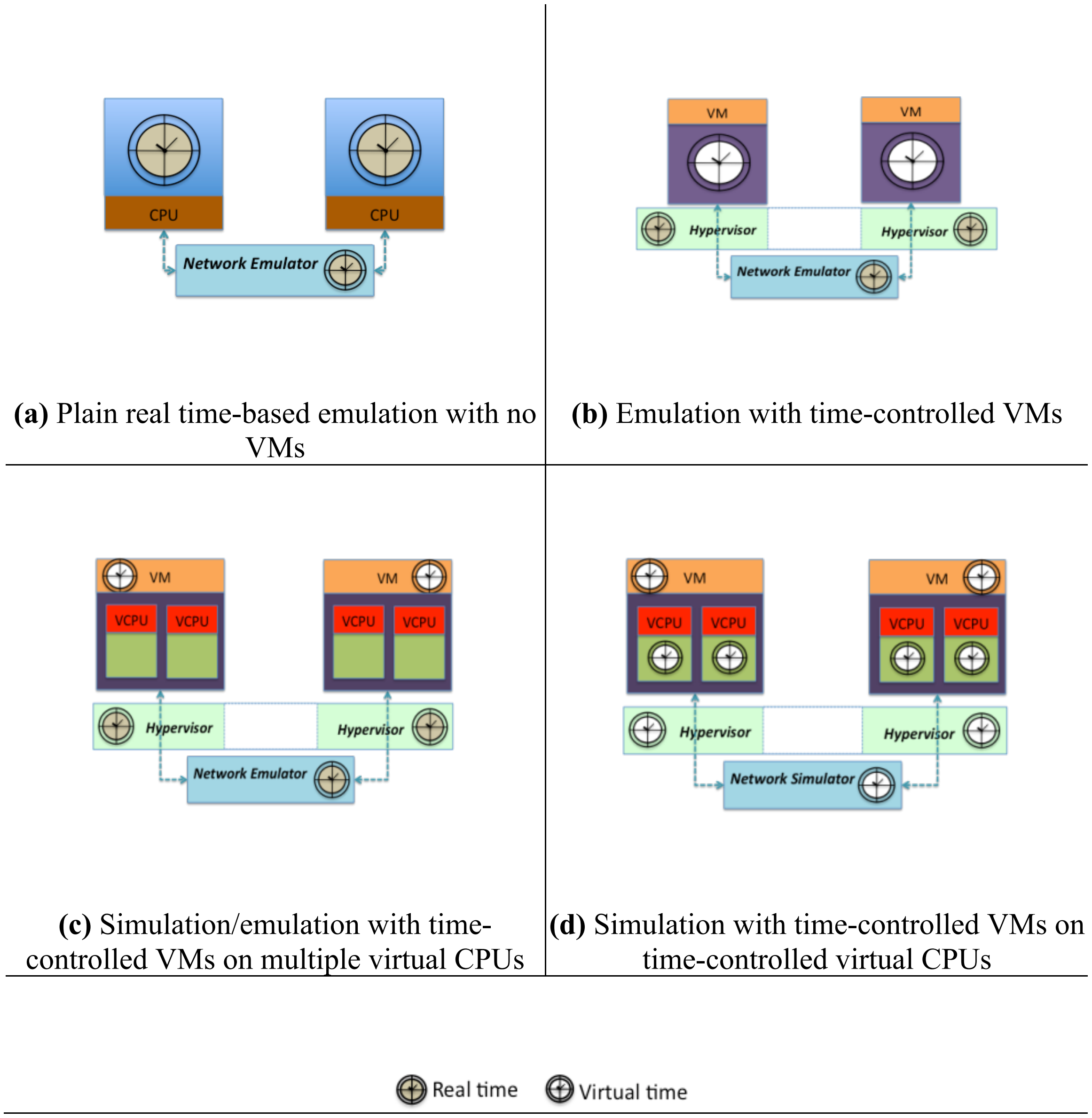
1.4.1.1 Dissociating Virtual Time from Real Time

By efficiently addressing this issue, we not only dissociate the simulator from real time for generic computer network systems but also open up new opportunities to realize high-fidelity simulations based on VM technologies. For example, in sensor-network simulations, the modeled sensor networks involve motes and these motes use processors that are clocked far slower than the generic processors. Though the VM technologies provide a means to host the OS used in sensors, the emulations pertaining to sensor networks are not reliable since the clock frequency of the processors used by VMs is very high compared to that of the motes. The dissociation of real-time from simulation time can ensure faster than real-time simulations of sensor networks.

1.4.1.2 Multi-core VMs for simulation

Figure 5 shows our approach in relation to past approaches: (a) represents free-running emulations with only real time-based clocks, (b) represents VMs integrated into emulations via virtual clocks controlled via approaches such as *time dilation* [36], (c) shows multi-core execution with one virtual clock per VM, and (d) shows a distinct virtual clock for every entity down to the level of each virtual core (VCPU). We support the capability to simultaneously schedule multiple single-core and/or multi-core VMs in virtual-time order on native multi-core hardware. Our design maintains a separate virtual clock for each virtual core (VCPU) of a multi-core VM to keep track of its virtual time.

This is analogous to a Logical Process (LP) in PDES that maintains its own local virtual time (LVT).



(a) Plain real time-based emulation with no VMs

(b) Emulation with time-controlled VMs

(c) Simulation/emulation with time-controlled VMs on multiple virtual CPUs

(d) Simulation with time-controlled VMs on time-controlled virtual CPUs

Figure 5 Real-time vs. virtual-time representations

1.4.1.3 Virtual Time-ordered Execution of VMs

Dissociating the simulation-time from real-time enables the ability to multiplex many VMs of varying capacities over a shared physical hardware and brings the traditional emulation system into the PDES conceptual framework. To ensure correctness of the simulation, the concurrently running VMs (with independent simulation times) need to adhere to the simulation-time-ordered execution of events of PDES. To this end, we identify, propose, implement and evaluate a novel technique of achieving time-ordered execution of the VMs by replacing the scheduling policy used by the hypervisor in scheduling the VCPUs.

1.4.1.4 Deriving Global Virtual Time

With each VCPU accounting for the virtual time, we need to synchronize and consolidate the timelines from each VCPU to a single global simulation time or a global virtual time. A timer-based approach that consolidates the VCPU virtual timelines to VM virtual timeline, and several VM virtual time lines into global virtual timeline, has been proposed, implemented and tested.

1.4.1.5 Virtual Time-ordered Network Control

Virtual time-ordered network control is another critical component in realizing correct models for high-fidelity cyber infrastructure/cyber-physical simulations. It enforces a user-specified virtual time latency and bandwidth to the packets in transit. Such virtual time-controlled inter-VM network communication is essential for virtual time-ordered execution of the entire system. In this thesis, we identify, propose and implement a novel, efficient design of a virtual time-controlled inter-VM network communication. Our virtual time-ordered network control is capable of capturing, buffering and delivering packets between VMs according to virtual time order of the overall simulation.

1.4.1.6 Synthetic Benchmarks to Measure Virtual Time-order Errors

To prove the existence of virtual time-order errors in traditional simulations when a large number of VMs are multiplexed on a limited resource physical machine, and also the absence of such errors in our virtual time-ordered systems, we designed and developed multiple synthetic benchmarks. These benchmarks provide a controlled way to detect if (and by how much) the simulation results deviate from theoretically correct results in terms of virtual time-order errors. In other words, the benchmarks serve to provide both qualitative as well as quantitative measures of simulation correctness in the use of VM within VTS.

1.4.1.7 Metric to Quantify Virtual Time-order Errors

To measure the virtual time-order errors in PDES application benchmarks, we invented an error-metric quantified in terms of a new concept we proposed, called *eunits*. A generic formula to calculate such virtual time-order errors was presented.

1.4.1.8 Application Benchmarks

Benchmarks with based on PDES applications were designed and developed to demonstrate the effects of virtual time-ordered execution of VMs. These benchmarks serve towards understanding the time-ordered execution correctness directly in terms of application-level phenomena, in contrast to the controlled variables of the synthetic benchmarks

1.4.1.9 Prototyping and Scaling Studies

All the proposed concepts were prototyped and tested for time-order errors. Their performance and scaling behaviors were also evaluated. We demonstrated a good runtime performance and excellent scaling behavior of our prototypes through the results from the synthetic as well as application benchmark executions.

1.4.2 VTS over VM

1.4.2.1 Guidelines for Better PDES Performance in VM Execution Environments

In comparison to the current execution environments of PDES applications, the Cloud computing services provide limited information about the actual hardware (host) platform that the Cloud uses for hosting the VMs. Similarly, the range of new configuration options provided by the Cloud to select and build a VM-based user cluster can confound the traditional PDES user. This is because such options have not been studied previously in the PDES literature, which had assumed a one-to-one mapping of simulation loops to processors. A set of guidelines on the performance of PDES on Cloud services was developed base on a detailed performance study, which is the first ever conducted for multi-core VMs. The study included multiple diverse PDES applications that exercised both optimistic and conservative synchronization techniques. Performance-critical and cost-critical choices of VM selections for PDES executions were highlighted, and the associated guidelines were provided. We were first to report such a detailed performance study involving synthetic and real-life PDES application benchmarks.

1.4.2.2 Identifying the Criticality of VM Scheduling for Performance

Although several Cloud-specific performance studies have been reported in the literature, the performance implication of the VM scheduling policy on VTS has never been studied. We were the first to highlight, demonstrate and associate the performance degradation of PDES applications executing on VM platforms to the scheduling policy of the VM platforms. We also proposed, designed, implemented and tested a new *Least-LVT-First* based scheduling policy for the VM hypervisor scheduler to address this problem and efficiently handle PDES application loads.

1.4.2.3 Algorithm to Overcome *Deadlock* and *Livelock* Conditions

The *Least-LVT-First Scheduling* (LLFS) policy in scheduling VCPUs on to PCPUs is *necessary* for efficiently hosting VTS applications on VM platform. However, it is not *sufficient* because a *purely* LLFS policy suffers from susceptibility to *deadlock* and *livelock* problems. We identified the presence of *deadlock* and *livelock* in a purely LLFS execution and traced the source of their appearance to GVT computations in the hosted PDES execution. We designed an efficient counter-based algorithm to overcome the *deadlock* and *livelock* conditions.

1.4.2.4 Efficient Implementation of PDES-specific Hypervisor Scheduler

Our new, livelock-free and deadlock-free LLFS algorithm was implemented and tested with multiple PDES application benchmarks. The implementation was demonstrated to yield very good performance gains across all the benchmark applications. Our implementation also demonstrated insignificant runtime variance in comparison to the variance observed from the default scheduler. Although the scaling behavior varies across application benchmark, our implementation demonstrated a good scaling behavior (for both strong scaling as well as weak scaling). A very competitive runtime performance in comparison to the native Linux-based (non-VM) PDES runs was also observed.

1.4.2.5 Order-of-magnitude Speedup from New VM Scheduling Policy

Based on the applications under consideration, PDES performance varies dynamically during runtime across individual LPs. The application-computing load on LPs depends on the event-processing rates that are highly dynamic in nature. We hypothesized that, by communicating the least virtual time information from the PDES application to the scheduler, the performance problem of the VM scheduling policy can be solved. By using the PDES-supplied virtual time information in allotting compute resource to the VMs that host LPs, the scheduler can efficiently address the dynamic load-balancing problem from time-varying event workloads. We demonstrated this critical aspect with our prototype implementation, which at its recorded best demonstrated around 20-fold speedup compared to the VM platform that used traditional fairness-based, general-purpose credit scheduler.

1.5 Thesis Organization

From Chapter 2 to Chapter 4, we discuss in detail the topics covering the use of VMs within VTS. In Chapter 5 and Chapter 6, we discuss the topics covering VTS execution over VMs.

Chapter 2 provides the concepts, issues, challenges, design, implementation and performance evaluation of NetWarp, the cyber-infrastructural and cyber-physical simulator that uses VM instances as end hosts. We also discuss the design, implementation and testing of a virtual-time ordered network control module prototype. In Chapter 3, we discuss the scaling, benchmarking, network control issues, and evaluate the NetWarp system for correctness and performance in greater detail. In Chapter 4, we perform a case study in which, a large scale MANET simulations are used to evaluate the NetWarp simulator.

In Chapter 5, we discuss the suitability of VM based execution platforms for PDES applications, and analyze important performance-critical observations discovered and provide recommendations for PDES users using VM-based execution platforms. In Chapter 6, we discuss the important role played by the VM scheduler during the execution of PDES applications. We design and develop a virtual time-aware hypervisor that allows the applications to pass virtual time to the hypervisor. We design and develop, a *deadlock* and *livelock* free algorithm Least-LVT-First (LLF) based hypervisor scheduler algorithm. We demonstrate the performance efficiency of the LLF based hypervisor scheduler in comparison with the default *fair-share* based hypervisor scheduler using several simulation scenarios from diverse set of PDES applications.

We conclude with discussions on future directions after summarizing our research work in Chapter 7.

CHAPTER 2

VM WITHIN VTS: CONCEPTS, DESIGN AND PROTOTYPE

2.1 Network Simulation/Emulation Overview

2.1.1 Background

Simulations and emulations have played an important role in the growth, sustenance and maintenance of networked systems of various kinds. Looking back at the evolution of tools used for realizing simulation models of computer networks, they can be broadly classified into two major groups as shown in Figure 6: (a) *Network-centric* and (b) *End-host-centric*.

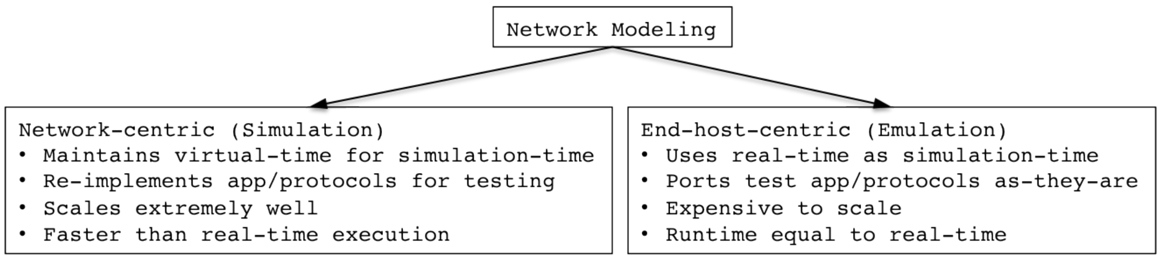

Network Modeling
Network-centric (Simulation)
• Maintains virtual-time for simulation-time
• Re-implements app/protocols for testing
• Scales extremely well
• Faster than real-time execution
End-host-centric (Emulation)
• Uses real-time as simulation-time
• Ports test app/protocols as-they-are
• Expensive to scale
• Runtime equal to real-time

Figure 6 Network modeling classification and terminology

The network-centric simulation group mostly contributes to the design of networks, understanding the dynamics of the interaction of end-node protocols, testing new protocols, determining the bottlenecks, and engineering the protocols for better resource utilization and flexibility, all of which are predominantly focused on the overall network operation. On the other hand, the end-host-centric group largely concentrates on the impact of network characteristics (such as, bandwidth and latency) associated with the dynamics of the communication protocols on the end-user application and porting actual implementations of the applications/protocols onto the end-hosts. More information on such a classification can be found in the literature [23].

Physical test-beds comprising networked computer systems spanning multiple continents, such as the PlanetLAB system [24], or spanning a country (United States), such as the GENI system [25], generally fall outside this classification since test-beds can be used for either network-centric and end-host-centric studies.

The analytical models and the network simulation tools such as NS2 or NS3 [26] and OPNET [27] fall under the network-centric category, where the abstraction of the end-nodes for modeling the network under study was acceptable and also desirable to achieve better scaling. The rapid growth in the size of internetworks and the Internet resulted in the development of parallel computing network simulators such as GTNets [28], SSFNet [29], GloMoSim [30], PDNS [31] and similar tools, to address the scaling and runtime performance needs. One characteristic to note in almost all the network-centric simulation tools is that the simulation time is completely different from real time or wall-clock time.

On the contrary, the application-centric tools, based on their design goals, rely on real-time. Tools such as dummy-net [32], ENTRAPID [33], ModelNet [34] and Emulab [35] fall in this category. To differentiate from the network-centric simulation models, they have been largely referred to as network emulation-tools or emulation-test beds in previous literature.

One apparent differentiating factor of these groups is their simulation time. The network-centric simulation tools keep track and maintain their simulation time, while the end-host-centric tools use the real-time as their simulation-time. We define the simulation-time maintained by the network-centric simulation tools as *virtual-time*, as the simulation time is common to both network-centric and end-host-centric models.

2.1.2 VM based Network Modeling

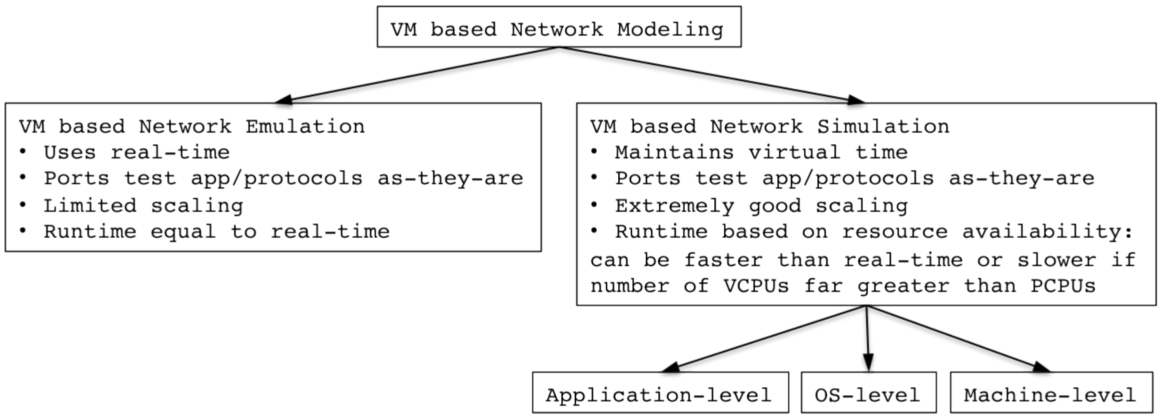

VM based Network Modeling
VM based Network Emulation
• Uses real-time
• Ports test app/protocols as-they-are
• Limited scaling
• Runtime equal to real-time
VM based Network Simulation
• Maintains virtual time
• Ports test app/protocols as-they-are
• Extremely good scaling
• Runtime based on resource availability: can be faster than real-time or slower if number of VCPUs far greater than PCPUs
Application-level
OS-level
Machine-level

Figure 7 VM-based network modeling classification

The end-host-centric simulators such as Emulab have started using VMs to address the scaling and capacity needs of tested applications. Concurrently, in the quest to achieve higher fidelity and ease of protocol/application portability without losing scalability, a few network-centric simulators such as NS2 started providing interfaces to feed live packets into their network simulators. Thus, a natural trend in interfacing network simulators with VMs to realize the goals of *both* network-centric and end-host-centric simulators has been pursued by various research groups around the world. In the context of this recent phenomenon of integrating VMs into cyber infrastructure simulations, several problems and challenges arise and remained to be addressed for efficient utilization of the VM technology in the simulations.

With the overlapping of two earlier distinct views, the tools that were clearly identified previously as either distinctly emulation-only or distinctly simulation-only tools have now become difficult to classify as only one and not the other. Essentially, the distinction begins to blur when the real-time aspects of VMs get mixed with the virtual time-driven simulations. Most literatures reporting the development of the combination of end-host-centric and network-centric simulation tools refer to them as network *simulation/emulation*. However, the terminology for differentiating these two groups of tools still continues based on the factors that previously qualified a system as either simulation or emulation. That is, if the tool uses real-time as simulation-time then it is referred to as emulator; if the tool maintains a virtual-time for simulation-time, it is referred to as a simulator. The terminology built on this notion is consistently used throughout the rest of this document.

2.1.3 VM-based Network Emulators

The concept of time dilation [36] was introduced in emulations, wherein a higher/lower bandwidth communication network behavior could be emulated using the same underlying network by simply manipulating the perceived time of the end-nodes/operating systems. This is often referred to as time virtualization in subsequent literature. Using *resource* virtualization from conventional hypervisors, augmented with *time* virtualization from time dilation, various network emulation systems have been proposed, such as V-eM [37], DieCast [38], VENICE [39], dONE [40] and Time-Jails [41], allowing flexibility in configuring the emulation setup. Note that although time dilation alters the perception of time for the end nodes the execution pacing is still real time based. This raises correctness issue when the total number of VCPUs hosted (across all the VMs) is greater than the number of physical cores (PCPUs) of the physical node. Even with total number of hosted VCPUs equaling the PCPUs, the problem of correctness persist (due to scheduling) and this can be overcome by pinning VCPUs to corresponding PCPUs.

2.1.4 VM-based Network Simulators

Unlike the network emulators that depend on real-time, the simulators using VMs must maintain their own virtual-time, synchronize across multiple VMs and consistently advance the simulation without committing any time-order errors. The tools that are published in the literature can be classified as (a) Application-level network simulators (b) OS-level network simulators, and (c) Machine-level network simulators.

ONE [42] can be considered as an application-level network simulator. It is realized using kernel-level protocol implementations and a few relevant applications such as FTP and Telnet software implementations, to generate a virtual application node. Multiple such virtual application nodes are run in parallel using PDES synchronization schemes. Specifically, they use null message-based conservative synchronization scheme [12]. This approach might yield better scalability when compared to the other two approaches but poses portability challenges as the code-base from the kernels and applications need to be compiled. Additionally, the source code base is often not available for all the OSs. Even when available, the instrumentation and maintenance of multiple operating systems along with their version variations poses a significant hurdle to using this approach.

The OS-level network simulator uses an OS-level VM for realizing the network simulator. The simulator presented in [43] using OpenVZ VMs is an example in this category. The advantages arise from the fact that the OS instances run as processes on the native OS, thus providing greater flexibility to exercise control, easy access to the state information of the guest-OS. Since all the OS instances essentially use the same kernel simulator, this approach is expected to scale well. However, since the VM in this case is an OS level machine level, the virtual hardware customization of the VMs cannot be handled. For example, it is not possible to realize simulations involving computing systems with varying number of processing cores, NICs, etc. Also, the simulator cannot host different types of OS kernels concurrently on the same physical machine using this VM technology. The inability to specify the VM machine configuration details is problematic as multicore architectures are widely prevalent, and single-core architectures have essentially become obsolete.

The machine-level network simulator uses either “Para-VM” or “Full-VM” virtualization techniques, as in the Xen or VMWare ESX server or MS-VServer implementations. Configurability-wise, this type of simulator offers a wide range of options to define the machine-specific details of a VM (number of cores, number of NICs, etc.). Further, the configurability is independent of the physical host of the VM; it also supports concurrent execution of different distributions of OS on the same physical platform. The scalability of Para-VM simulator is comparable to that of OS-level VMs. However, the guest OS instances are more isolated and the need to deal with the virtual devices of the VMs is unavoidable. VM-based simulation tools, such as the SliceTime [44] and our tool NetWarp, fall in this category.

SliceTime does not provide clarity in virtual time-keeping to clearly separate real time and virtual time for VMs (especially, multi-core VMs on multi-core hosts). Further, it uses a specific solution based on UDP broadcast for inter-VM virtual-time synchronization, while our approach provides a clear framework for virtualizing all aspects of the simulated inter-VM network. Virtual time-ordered execution on multi-core VMs needs a methodology to account for idle-time in simulations, which is lacking in the implementation of SliceTime. Our NetWarp design incorporates the mechanisms needed for accurate and efficient execution in the presence of idling VM cores.

2.2 NetWarp Simulator

2.2.1 NetWarp Goals

NetWarp is envisioned as a large-scale VM based high-fidelity network simulator that is capable of simulating the behavior of parallel/distributed applications and protocols. It envisions duplicating the operation of arbitrary configurations and combinations of wired and wireless networks. It is intended to reduce the time and effort involved in porting, configuring, instantiating, and executing network simulation scenarios.

2.2.2 NetWarp Architecture

In computer network simulations, two logically complete and functionally independent modules can be identified, namely, (a) the end-host module and (b) the network module. The functional completeness of the end-host module presumes the availability of actual implementations of the protocols/applications that need to be exercised in the simulation. Similarly, the functional completeness of the network module presumes the connectivity specification between the end-hosts with/without the specification of the bandwidth. With the functional completeness of the end-host and network modules, a modeler can evaluate distributed applications or protocols. However, this does not guarantee correctness because of time-order errors. The correctness of simulation-based evaluation only builds on completeness of software implementation, but additionally needs accurate time-ordered evolution of the system execution. The support of virtual time-ordered execution of all the end-hosts along with time-ordered network interaction that delivers packets to/from the end-hosts in the virtual time-order ensures correctness in the simulation execution.

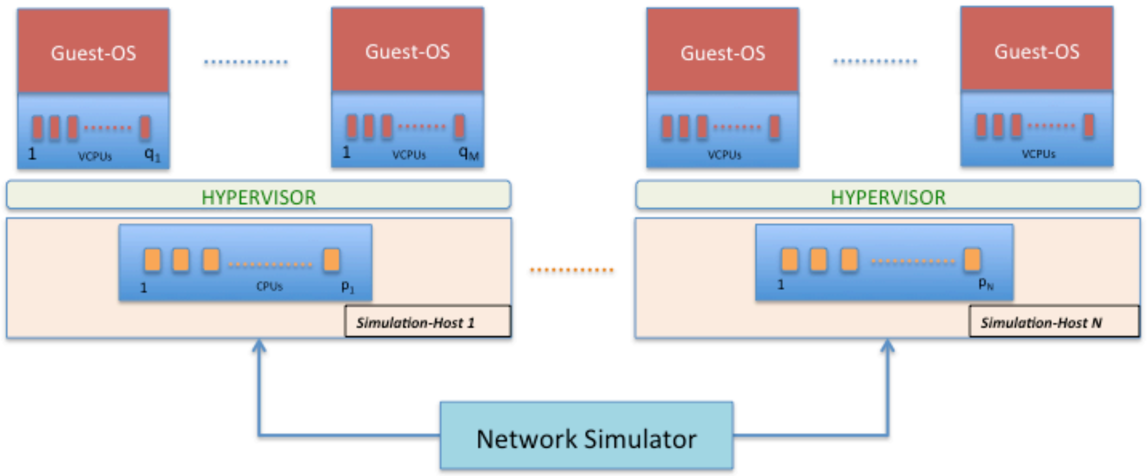


Figure 8 NetWarp simulator design

NetWarp is a network simulator geared toward cyber-infrastructural and cyber-physical simulations, that utilizes a large number of virtual end-hosts realized as multi-core VMs sustained on multi-core hosts in parallel. The execution architecture thus presents two distinct components of time-controlled execution, namely, the control of intra-node pacing among VMs within a multi-core node, and the control of inter-node pacing for virtual time coordination across multiple multi-core nodes.

This type of system model can be classified as a quasi discrete-dynamic-deterministic system model because the mapping of the VCPUs onto multi-core PCPUs might include minor randomness in virtual time-ordering making multiple runs deviate from each other; however, this deviation can be arbitrarily reduced by reducing the execution time slice, as will be described in greater detail in the rest of the document. Although the granularity of the time slices can be reduced to control the randomness to some extent, there is a trade-off with runtime performance because smaller virtual time slices to the VCPUs result in more frequent context switching. Hence, we qualify the NetWarp simulation system as a quasi-discrete-dynamic-deterministic system model.

2.2.3 NetWarp Core Conceptual Issues

In realizing a VM-based network simulator, the following conceptual issues need to be addressed:

- Accounting for simulation-time of virtual end-hosts
- Accounting for idle-time of VCPUs
- System/node-level accounting of virtual time
- Virtual time-ordered scheduling of VMs
- Adjustable granularity of the VM scheduling time unit
- Design of virtual time-ordered execution benchmark tests
- Virtual time-ordered network communication control
- Accuracy testing mechanisms in virtual time-ordered execution of applications.

In this section, we discuss the core issues, their relevance and the approach to realizing these concepts in VM-based simulators.

2.2.3.1 Accounting for Simulation Time

The network simulation platform is required to account for virtual time on each VM. Our design supports VMs containing multiple-VCPUs with arbitrary combinations of their composition (such as dual-core, quad-core, etc.). To support such heterogeneous mixture, virtual-time needs to be maintained for each VCPU. Further, we also need to consolidate the virtual times of VCPUs into VM-level virtual time periodically.

2.2.3.2 Accounting for Idle Time

During the course of a network simulation, within each multi-VCPU VM, not all the VCPUs are active all the time; similarly, not all VMs are expected to be active all the time in a large-scale network simulation. Hence, the idle time in the VCPU and VM processor times must also be accounted along with the utilization time of active processors participating in network simulations.

Within a VM, different VCPUs can be periodically synchronized to the largest value of VCPU virtual times across all VCPUs of that VM to account for the idle time of all VCPUs of the VM without actually burning the PCPU cycles for idle times. Similarly, by pulling the virtual time across all VMs to their maximum value periodically one can account for idle time of VMs across the simulation host node that houses all the VMs being used as end-hosts in the simulated network scenario.

2.2.3.3 System/Node Virtual Time

The progress of the entire network simulation comprising all the VMs running within a single physical host is tracked using the concept of a "System/Node Virtual Time," or SYS_LVT for short, which is derived using the virtual times of each VM. Virtual time within each VM can be accounted by taking the largest of virtual times across all the VCPUs of that VM, thus accounting for both utilized time and idle time of VCPUs. SYS_LVT at any point in the simulation is the minimum of local virtual times (LVTs) of all the VMs.

2.2.3.4 Virtual Time-ordered Execution

Since hypervisors were created for general-purpose usage (as opposed to being designed for network simulations alone), the host resources are shared fairly across all hosted VMs and are optimized to best utilize the host resources. In network simulations, the virtual time of a VM progresses with the actual utilization of the VM's VCPUs. In order to ensure virtual time-ordered generation and processing of events by VMs, the VCPUs of the VMs must follow virtual time-order. Given the distribution of the load across VMs can vary arbitrarily at runtime and that a large number of VCPUs will be multiplexed among limited number of PCPUs, a fairness-based VCPU scheduler is bound to generate virtual time order errors (as will be qualitatively and quantitatively demonstrated later in this dissertation). With this in mind, careful, virtual time-aware scheduling of VCPUs onto the PCPUs must be designed to control the LVT advancement. By ensuring that only the VCPU with the least LVT is scheduled at every scheduling opportunity, we can ensure virtual time-ordered execution of the simulation.

2.2.3.5 Adjustable granularity of VM Scheduling Time Unit

A hypervisor by default uses a fixed time slice size for each VCPU (e.g., 10ms in Xen). The larger the time slice provided to the VCPU of a VM, the greater is the potential for the system to commit virtual time-order errors. However, smaller time slices lead to a larger number of context switches of VMs (or VCPUs) and hence can affect the runtime performance. Additionally, the granularity of the slices also constrains the minimum delay that the inter-VM virtual network must enforce. This is because the delay to be enforced cannot be smaller than the VCPU time slice. Hence, the VCPU time slice needs to be adjustable at the hypervisor scheduler and be empirically studied to observe the effects of lower time slices on correctness and performance.

2.2.3.6 Testing Virtual Time-ordered Execution

While the virtual time order errors and their importance for the correctness is well understood in discrete-event simulations. Their occurrence, manifestation and effect on the simulation results are not apparent and clear on the outset, in the absence of actual implementation and experimentation. Conventional discrete event network simulations based on models are associated with discrete events (such as packet-sent or packet-received events). However, in VM based network simulations the virtual time is essentially continuous as opposed to being discrete because there are no specifically detectable discrete changes in time at the VM execution level. By making the virtual time progress evenly amongst all the VMs, we ensure that the concerned events (such as, packets sent and received) are correctly paced in virtual time. However, it becomes necessary to qualify and quantify the virtual time-order errors in VM based network simulators.

Consider three VMs, say, VM1, VM2 and VM3, being used to model end-hosts of a simulated scenario in which all point-to-point messages incur a fixed latency and all VMs are hosted on the same machine. If VM1 sends two messages to VM3, one directly and the other routed through VM2, two different ways of message arrivals at VM3 are possible: (a) the message VM1 sent directly to VM3 reaches first followed by the message routed through VM2, and (b) the message VM1 sent through VM2 reaches first followed by message directly sent to VM3. With homogeneous network conditions existing between VMs, it is possible to identify two modes of execution: (a) virtual time-ordered, and (b) free-run ordered. Running this experiment multiple times and by qualifying the observed correctness of the result in each case, we can statistically quantify the virtual time-order execution behavior of the VM based network simulator.

2.2.3.7 Virtual Time-ordered Network Control

To ensure virtual time-ordered delivery of the communication packets across VMs, we need to trap the packets in transit and ensure virtual time-order dispatch of the intercepted packets to their respective destination. This is necessary to ensure correctness and also to have an ability to control the inter-VM network characteristics. In a VM environment on a single physical host, all the packets exchanged between the guest VMs can be intercepted, delayed and forwarded or dropped at the privileged VM (e.g., DOM0 in Xen). This capability allows us to introduce user-specified latency or packet-loss characteristic to any stream of packets in transit.

2.2.3.8 Virtual Time-based Execution of VM Applications

In a configuration where multiple VMs are hosted on a physical machine such that the aggregate VCPU count of all hosted VMs is greater than the number of PCPUs, the simulation time progresses slower than the wall-clock time. Then, the applications that use wall-clock timers for their operation (such as for TCP time-outs), are adversely affected, resulting in an incorrect operation. To overcome this, we need to ensure that the *virtual time* is used by the applications hosted by VM in place of the wall-clock time.

2.2.4 NetWarp Prototyping Approach

NetWarp can be broadly classified into two distinct functional modules namely, (a) Virtual time-ordered execution of VMs (b) Virtual time-ordered network control of inter VM network traffic. The former addresses all the virtual time maintenance and evolution within and across VMs. Thus, it requires modifications to the hypervisor. The latter is concerned with runtime-efficient realization of the desired (virtual) network behavior by controlling the transfer of packets between VMs. This is addressed at the application-level in DOM0 utilizing low-level operating system mechanisms.

2.3 Prototyping Virtual Time-ordered VM Execution

2.3.1 Xen Scheduler

The essential functionality of the Credit Scheduler of Xen (CSX) is to efficiently multiplex many VCPUs onto a limited number of PCPUs for execution. Scheduling in Xen shares some concepts with an operating system that provides an *N:M* threading library. In such a system, the operating system kernel schedules *N* OS threads (typically one per physical context). Within each OS thread, a user-space library multiplexes *M* user-space threads. In Xen, the VCPUs are analogous to the OS threads, and the *domains* are analogous to the user-space threads [6]. Essentially, there exist three scheduler tiers in Xen, (1) user-space threading library maps the user-space threads to kernel threads (within a DOM), (2) user-DOM OS maps its kernel threads to VCPUs, and (3) hypervisor maps each VCPU to a physical CPU (PCPU) dynamically at run time.

Scheduling involving dynamic mapping of the VCPUs on to the PCPUs is discussed in this thesis. The strategy CSX uses for scheduling is based on the principle of fair share and uses *credits* for every DOM, these credits are expended as the DOM's VCPUs are scheduled for execution. Based on the amount of credits expended the VCPUs either become *over_scheduled* or remain *under_scheduled*, and these VCPU states are used during scheduling to ensure fairness. The scheduler that we developed for addressing the simulation concerns is called the NetWarp Scheduler for Xen (NSX), whose underlying principle for scheduling is to ensure simulation time-order. Hence, both CSX and NSX dynamically multiplex of VCPUs on to PCPUs using their respective scheduling strategies.

2.3.2 NetWarp Scheduler for Xen (NSX) Data-structures

Using the modular design of Xen new schedulers such as NSX can be built to replace the default CSX. The internal design of Xen prescribes a set of functions that needs to be implemented and interfaced by the new scheduler.

By replacing the relevant function pointers to the scheduler's interface variables, our NSX scheduler functional implementations is integrated into Xen. For example, in CSX the *init_domain* variable is a function pointer to *csched_dom_init*, which is used for the initialization a DOM. In NSX the *init_domain* variable is instead set to *nwsched_dom_init*, to initialize NSX. After integrating and booting with our scheduler, Xen invokes our newly added routines to make scheduling decisions.

Figure 9 shows the static object created for the NSX to interface with Xen. Similar to CSX, we maintain four different data-structures in NSX, namely,

a. *nw_pcpu* – control data corresponding to each PCPU

b. *nw_vcpu* – control data corresponding to each VCPU

c. *nw_dom* – control data corresponding to each DOM

d. *nw_private* – a global data area shared by all DOMs.

The *nw_pcpu* data-structure as shown in Figure 10, comprises a VCPU queue called *runq*, a *timer*, and a counter named *tick* that account for number of used ticks. The number of instances of the *nw_pcpu* data structure corresponds to the number of processor cores in the physical hardware. The next VCPU to schedule on a particular physical core is chosen by the Xen scheduler from the *runq* list of the *nw_pcpu* object corresponding to that physical core.

Figure 11 compares data-structures *csched_vcpu* and *nw_vcpu* (some variables concerned with keeping a log of VCPU status in *csched_vcpu* data-structure are not included in Figure 11 for clarity purposes). As seen, most of the variables in *nw_vcpu* are the same as in *csched_vcpu*, except that the *credit* and *pri* (priority) variables of the *csched_vcpu* are replaced by the *nticks* variable of *nw_vcpu* and the *ref_time*. The *nticks variable* keeps track of a number of ticks that the VCPU has used up, and the *ref_time* gives a reference-time from which *nticks* are tracked. The *ref_time* along with the *nticks* gives the Local Virtual Time (*LVT*) as $LVT = ref\_time + (nticks \times tick\_size)$ for a VCPU.

```
struct scheduler sched_credit_def = {
     .name           = "SMP Credit Scheduler",
     .opt_name       = "credit",
     .sched_id       = XEN_SCHEDULER_CREDIT,
     .init_domain    = csched_dom_init,
     .destroy_domain = csched_dom_destroy,
     .init_vcpu      = csched_vcpu_init,
     .destroy_vcpu   = csched_vcpu_destroy,
     .sleep          = csched_vcpu_sleep,
     .wake           = csched_vcpu_wake,
     .adjust         = csched_dom_cntl,
     .pick_cpu       = csched_cpu_pick,
     .do_schedule    = csched_schedule,
     .dump_cpu_state = csched_dump_pcpu,
     .dump_settings  = csched_dump,
     .init           = csched_init,
     .tick_suspend   = csched_tick_suspend,
     .tick_resume    = csched_tick_resume,
};

struct scheduler sched_nw_def = {
     .name             = "SMP Netwarp Scheduler",
     .opt_name         = "nw",
     .sched_id         = XEN_SCHEDULER_NW,
     .init_vcpu        = nw_vcpu_init,
     .destroy_vcpu     = nw_vcpu_destroy,
     .init_domain      = nw_dom_init,
     .destroy_domain   = nw_dom_destroy,
     .sleep            = nw_vcpu_sleep,
     .wake             = nw_vcpu_wake,
     .adjust           = nw_dom_cntl,
     .pick_cpu         = nw_cpu_pick,
     .do_schedule      = nw_schedule,
     .init             = nw_init,
     .tick_suspend     = nw_tick_suspend,
     .tick_resume      = nw_tick_resume,
};
```

Figure 9 CSX and NSX scheduler interface

Each element of *runq* of an *nw_pcpu* is of type *runq_elem*. As the *runq_elem* is related to the *runq* of the *nw_pcpu* object, in a similar way as the *active_vcpu_elem* object is related to the *nw_dom*'s (discussed next) *active_vcpu* queue. Both these data-structures aid the inserting and removing, to and from their corresponding queues by manipulating the *prev* and *next* pointer values.

Figure 12 shows the CSX's data-structure *csched_dom* and NSX's *nw_dom* for domains. The *nw_dom* comprises a list named *active_vcpu*, which is a list of *nw_vcpu* belonging to this DOM. It contains a member-variable named *active_sdom_elem*, which is initialized to point to self, and aids in the functioning and operation of the *active_sdom* queue in the *nw_priv* global object of type *nw_private*. The *nw_priv* holds all the active DOMs in this *active_sdom* queue.

```
struct csched_pcpu {
   struct list_head runq;
   uint32_t runq_sort_last;
   struct timer ticker;
   unsigned int tick;
};

struct nw_pcpu{
   struct list_head runq;
   struct timer ticker;
   unsigned int tick;
};
```

Figure 10 CSX's PCPU vs. NSX's PCPU structures

Xen maintains an object of type *domain* for each of the DOMs. The *nw_dom*'s *dom* pointer points to this data-structure object. The *nw_dom* comprises an integer variable that holds a record of active number of VCPUs in the DOM. It also has a member variable *dom_lvt* to keep track of the DOM's simulation time (max LVT value of all the VCPU's of this DOM) and a *spin-lock* used by the DOM's VCPUs to update *dom_lvt*. The *dom_lvt* variable is used periodically to coercively increase the LVT of lagging VCPUs to keep all the VCPUs of the DOM in sync. Note that the *nw_dom* data-structure does not need the *weight* and *cap* variables of the CSX.

```
struct csched_vcpu {
   struct list_head runq_elem;
   struct list_head active_vcpu_elem;
   struct csched_dom *sdom;
   struct vcpu *vcpu;
   atomic_t credit;
   uint16_t flags;
   int16_t pri;
     …
};

struct nw_vcpu{
   struct list_head runq_elem;
   struct list_head active_vcpu_elem;
   struct nw_dom *sdom;
   struct vcpu *vcpu;
   uint16_t flags;
   atomic_t nticks;
   s_time_t ref_time;
};
```

Figure 11 CSX's VCPU vs. NSX's VCPU structures

```
struct csched_dom {
   struct list_head active_vcpu;
   struct list_head active_sdom_elem;
   struct domain *dom;
   uint16_t active_vcpu_count;
   uint16_t weight;
   uint16_t cap;
};

struct nw_dom{
   struct list_head active_vcpu;
   struct list_head active_sdom_elem;
   struct domain *dom;
   uint16_t active_vcpu_count;
   spinlock_t lock;
   s_time_t dom_lvt;
};
```

Figure 12 DOM variables in CSX vs. NSX

As mentioned previously, the scheduler contains an *nw_priv* object of type *nw_private*. It contains a queue named *active_sdom*. The *ncpus* variable gives the number of cores supported by the underlying hardware. Importantly the scheduler object contains a time variable named *sys_lvt* that keeps track of the global *LVT* (the minima of all *dom_lvts* across all DOMs except for DOM0 and DOM 32767). The *sys_lvt* is equivalent to the traditional concept in simulators of *now* in simulation time. The *sys_lvt_next* holds the maxima of all *dom_lvts*.

```
struct csched_private {
   spinlock_t lock;
   struct list_head active_sdom;
   uint32_t ncpus;
   struct timer  master_ticker;
   unsigned int master;
   cpumask_t idlers;
   uint32_t weight;
   uint32_t credit;
   int credit_balance;
   uint32_t runq_sort;
};

struct nw_private {
   spinlock_t lock;
   struct list_head active_sdom;
   uint32_t ncpus;
   struct timer  master_all_lvt_ticker;
   struct timer  master_lvt_ticker;
   struct timer  master_ticker;
   unsigned int master;
   cpumask_t idlers;
   s_time_t sys_lvt;
};
```

Figure 13 CSX and NSX private data

The *nw_priv* maintains three timers. The first is the *master_ticker* used to update the *dom_lvt* across all DOMs. The second is the *master_lvt_ticker* used to synchronize the VCPUs corresponding to a DOM to their *dom_lvt*, thus artificially advancing the LVTs of the lagging VCPUs in the DOM to *dom_lvt*. The third timer named *master_all_lvt_ticker* synchronizes all LVTs of all VCPUs across all the DOMs to *sys_lvt_next*, thus advancing the LVTs among all the VCPUs to the higher value.

The variable named *idlers* keeps track of the idling *VCPUs* and the variable *lock* is used for updating this global object. Figure 13 compares the global data-structure that is *csched_private* of CSX and *nw_private* of NSX. Most of the CSX variables namely, *credit*, *weight*, *balance* and *runq_sort* of the *csched_private* data-structure are removed, while keeping the remaining others unchanged.

2.3.3 Virtual Times of NetWarp

As previously mentioned three conceptual timelines are maintained in the NetWarp system and these are extensively used in the following discussions.

(1) Local Virtual Time (*LVT*): Timeline corresponding to each VCPU that advances in discrete time as VCPU consumes PCPU cycles (measured in the units of *ticks*).

(2) DOM LVT (*DOM_LVT*): Timeline corresponding to each DOM or VM that is calculated using the *LVT*s corresponding to its VCPUs and it advances in discrete time as its corresponding VCPU *LVT*s advance.

(3) System or Node LVT (*SYS_LVT*): Timeline corresponding to the entire simulation setup involving a hypervisor as a simulation host, with the DOMs as the end nodes connected through the controlled virtual network of the hypervisor environment. This advances in discrete time as the *DOM_LVT*s advances.

Both the critical issues, namely, (a) maintenance of a single simulation timeline, and (b) time-ordered execution of VMs, can be addressed by replacing the scheduler of the Xen hypervisor.

2.3.4 Virtual Time Accounting

A periodic timer named *ticker* corresponding to each core (in a multi-core processor) is maintained in the *nw_pcpu* data-structure. This periodic timer goes off at every *tick*, and when it goes off, the accounting service routine charges a *tick* to the *current* runnable VCPU. Additionally, a soft-interrupt for scheduling this VCPU is raised on the physical core, which is selected either based on the provided VCPU-PCPU affinity-map or on the consideration that the VCPU has most idling neighbors in it's grouping. Thus at every tick generated by the *ticker* timer from any of the nw*_pcpu* objects corresponding to a PCPU, a VCPU is scheduled to run on that PCPU.

Another global timer named *master_ticker* is made to go off for every 10 ticks. This calculates the *DOM_LVT*, which is the maximum of the LVT values among all its VCPUs. This is carried out for every DOM and in each case the *DOM_LVT* is recorded in its corresponding *nw_dom* object.

2.3.5 Idle Time Accounting

If left unchecked, the VCPUs run asynchronously. Even within a single DOM, there could be differences in the LVT values of its VCPUs as this is dependent of the task assignment of the guest OS on to its VCPUs. To limit these differences and their subsequent propagation between the LVT values of the VCPUs within a single DOM, we adjust the LVT values of all the VCPUs to the maximum of the LVT values among them (i.e. *DOM_LVT*). This adjustment is carried out at regular intervals, the interval being a simulation parameter, set to a default value of one minute. A periodic timer *master_lvt_ticker* is used for this purpose. When this timer goes off, the *Adjust_lvt* routine is called which adjusts the LVTs of all the VCPUs to the *DOM_LVT*.

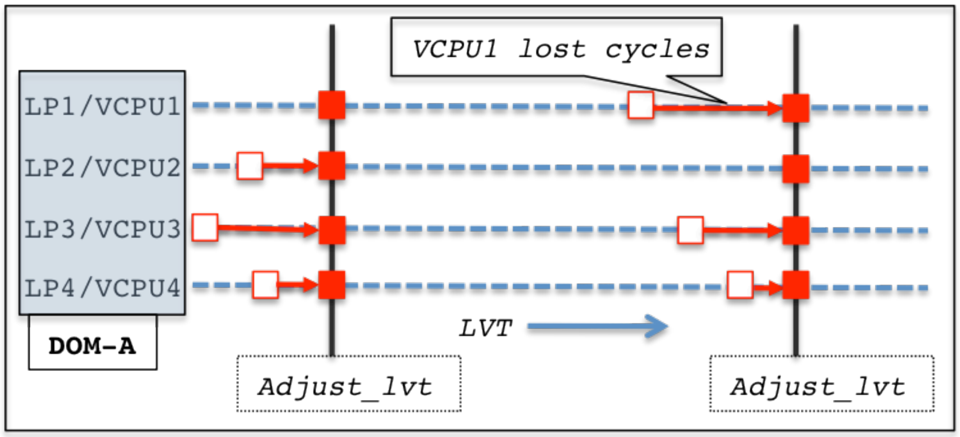


Figure 14 Accounting for VCPUs with lagging LVTs

The NSX scheduler enables us to address the issue of lost cycles that arises due to idle cycling of the VCPUs in a multi-core DOM. Note that the idle VCPUs are not charged and hence their LVT value remains stationary, while the LVTs of the active VCPUs increase. This results in a staggering of LVT values for different VCPUs within the same DOM as shown in the Figure 14. Since the behavior of VCPUs being either idle or active is dependent on the guest OS's scheduling and activity of the user applications (inside the DOMs), the lagging VPCUs can only be detected dynamically at runtime, and can be tracked with periodic updates. To illustrate, Figure 14 shows a DOM with 4 cores and the LVT values of VCPUs staggered and being pulled to the *DOM_LVT* value during periodic updates serviced by the *Adjust_lvt* routine.

Similarly, a *master_all_lvt_ticker* is used to synchronize the LVTs of the DOMs periodically, as illustrated in Figure 15. The only difference is that the *Adjust_all_lvt* routine that services this ticker checks if any of the DOMs are lagging beyond a period (*NW_MAX_DIFF*) behind the maximum of *DOM_LVT*s. In our experiments, the *NW_MAX_DIFF* was set to 10 ticks. This updating procedure of LVTs of all VCPUs across all DOMs accounts for the time of idle-DOMs during the network simulation. The *nw_priv* data-structure object maintains all the necessary global timers.

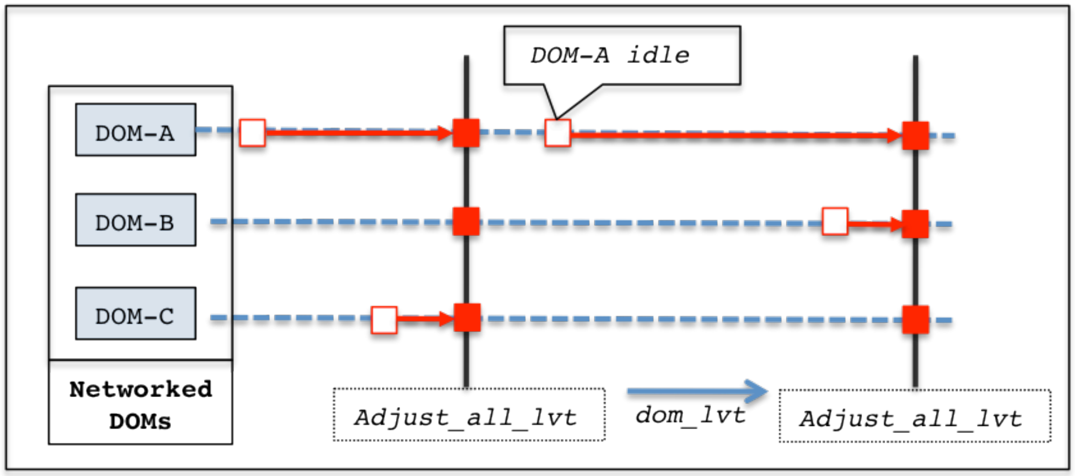


Figure 15 Accounting for idle DOM time

2.3.6 Global Virtual Time

To fulfill simulation time ordering requirement, each VCPU in its corresponding *nw_vcpu* data structure, keeps track of its utilized ticks of its execution using variables *nticks* and *ref_time* (the reference time from which point *nticks* has elapsed). These two values are used to calculate the LVT of the VCPU. The utilization of the VCPUs increases as the interconnected DOMs execute any application and as they do, the corresponding VCPU tick values increments. Hence the simulation time increases, as it is dependent on the LVT values of VCPUs. The least value among the LVT values of the VCPUs can be safely considered as the elapsed simulation time in a single core machine. In a multi-core system, we use the maximum LVT among the VCPUs of a DOM as the *DOM_LVT*, and the minimum of the *DOM_LVT*s is used as the *SYS_LVT*.

2.3.7 Modifying VM Time Slice Granularity

The CSX uses a default time-slice of 30ms, which is much larger than the network link latencies required in our simulation scenarios. Hence, for NSX we made modifications so that the time-slice is specifiable in terms of values as low as a few microseconds. Each time Xen needs to schedule a new VCPU, it invokes the *nw_schedule* routine. This routine picks up the least LVT valued VCPU to execute and provides a *time-slice* (number of *ticks*) for the VCPU to execute. The number of ticks in a *time-slice* was kept at unity. Thus, by changing the tick size in the pre-processor statements of the scheduler code, we were able to alter the tick size as needed.

2.3.8 Virtual Time-ordered Execution

The VCPU with the least LVT must always be scheduled for next execution. This is ensured during the insertion of VCPU into the run-queue, which happens either when *nw_schedule* is called or when the VCPU awakes from sleep (*nw_wake*). However, this criterion is not sufficient to ensure *Least-LVT-First* scheduling property on host platforms with multi-core processors, and the scheduling needs to be extended further as described next. The scheduler maintains a run queue for every PCPU. When the *nw_load_balance* routine is used, it safely steals the least LVT-valued VCPU from its peer PCPU's *nw_pcpu's runq* and schedules it for execution. This functionality is very similar to that carried out by the CSX. The only difference is that CSX looks for a VCPU with higher-priority value, while the NSX looks for the lowest LVT value.

The overall scheduling behavior in NSX is summarized in this paragraph. The *master_ticker* at every *tick* (same as *time-slice* in our implementation) increments the LVT of the *current* VCPU by a *tick* size (in μs). It then raises a *schedule* software interrupt on the core that has the most idling VCPUs. This *schedule* interrupt is serviced by the schedule service routine. Xen's scheduling framework ensures that there is always at least one VCPU in the *run-queues* of each PCPU. When the scheduler is interrupted, the service routine in NSX inserts the *current* (currently being serviced VCPU) into the *run_queue* of the PCPU that was interrupted. The VCPU with lowest LVT values is then searched across the *run_queues* of all the PCPUs. The VCPU with lowest LVT is removed from the *run_queue* and is scheduled to run next. The time-slice (number of ticks) that this VCPU should run is also provided in this routine. The selected VCPU and the *time-slice* are returned back from this service routine of NSX. Xen, during context switching process, assigns the selected VCPU as *current* and the *current* VCPU is executed on the actual physical core.

2.4 Prototyping Virtual-time Ordered Network Control

In this section the mechanisms necessary for realizing virtual time-controlled communication between VMs is discussed.

2.4.1 Virtual Network in Xen

Xen provides various options to setup a virtual network among the DOMs using software based network-bridge or network-router. Out of many options, the one most suitable for *secure* simulation-specific scheduling is the private bridge configuration: a virtual network-bridge in which all DOMs except DOM0 is connected. This configuration provides the best levels of security for performing network experiments because the virtual network is isolated from other (physical) networks connected to by DOM0. A schematic of such a setup is shown in Figure 16. Whenever a DOMU sends a message, it is forwarded from the DOMU's local virtual network interface to the software bridge driver maintained by DOM0, which is then buffered by our modified functionality for simulation delay and then routed to its destination network interface back through the virtual network interface.

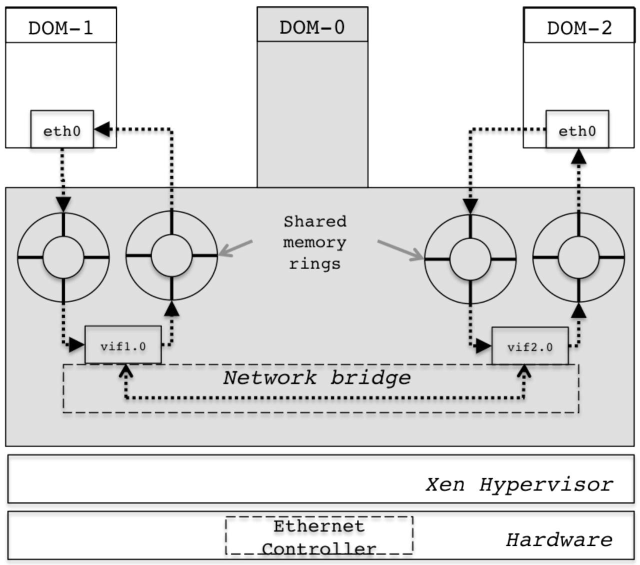


Figure 16 Virtual network using software bridge in DOM0

2.4.2 Trapping Transiting Packets

Every DOM specification file contains the details of the software bridge to which it should connect to after its creation. When a new DOM is created, a virtual network interface (referred to as *vif* in Figure 16) is also created in DOM0 and this interface is added to the specified bridge. To ensure the forwarding of Layer 2 packets from one virtual interface to the other Xen adds a new rule to the *iptables* [45] chain when a new DOM is created. For example: if DOM1 is created and if *vif1.0* were its corresponding virtual interface created in DOM0, then the *iptables* rule shown in the last line of Figure

17 is added to the *iptables* chain. The rule requests DOM0 to forward the Layer 2 packet originating from the virtual interface of the DOM1.

```
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT
-A FORWARD -m physdev  --physdev-in vif1.0 -j ACCEPT
```

Figure 17 *iptables* rule added on creation of a new DOM

Such a rule is added for every newly created DOM. By appropriately altering this rule we can trap, buffer and arbitrarily delay the packets transiting through the virtual network. This method of controlling the transiting packets is efficient because the transiting packets are not copied, are handled in kernel and results from a light-weight analytical or packet-simulation tools can be use determine the fate of the packet at this juncture in simulation execution. Interfacing with network simulators such as NS3 that handles live packets is another method of enforcing desired network control, but this makes the simulator unnecessarily heavy.

2.4.3 Enforcing Realistic Network Characteristics

Trapping the transiting packets only solves one part of the problem; enforcing realistic network characteristics requires additional work. In the overall operation, we envision a system such as ROSENET [46], a remote server based network emulation system, for this purpose. For prototyping the network control, we only introduce a tunable latency on the transiting packets.

As mentioned previously, we need to trap the transiting communication packets in the DOM0 to introduce a desired network delay. Figure 18 shows how this is achieved by trapping packets at the virtual network interface. The existing *iptables* rules are deleted that forward all the Layer 2 packets originating from virtual network interface (physdev) *vif1.0* (first-line in Figure 18). A new rule is then added to the table named *mangle* to forward the packets coming from *vif1.0* to a QUEUE (second line in Figure 18). The third command line in Figure 18 prints the *iptables* rules listed for different actions in the table *mangle*. In the output we can see *QUEUE* as a target for packets arriving from *vif1.0*.

```
#iptables -D FORWARD -m physdev --physdev-in vif1.0 -j ACCEPT
#iptables -t mangle -A FORWARD -m physdev --physdev-in vif1.0 -j
QUEUE
#iptables -t mangle -L
Chain PREROUTING (policy ACCEPT)
target  prot opt source  destination
Chain INPUT (policy ACCEPT)
target  prot opt source  destination
Chain FORWARD (policy ACCEPT)
target prot opt source   destination
QUEUE  all  --  anywhere anywhere  PHYSDEV match --physdev-in vif1.0
Chain OUTPUT (policy ACCEPT)
target prot opt source   destination
Chain POSTROUTING (policy ACCEPT)
target prot opt source   destination
```

Figure 18 *iptables* rule used to enforce communication control

The network utility library *Libipq* [47] provides a mechanism for passing IP packets out of the stack for queuing to user-space. It also allows the privileged user to set the verdict such as ACCEPT or DROP, on the trapped packet that are conveyed to the OS kernel. The ACCEPT verdict tells the kernel to continue the default processing of the packet (in our case the packet will be forwarded), while the DROP verdict would result in discarding the trapped packet. *Libipq* uses the *ip_queue* kernel module for this purpose. We developed a daemon service using the *Libipq* library functions for communication control for trapping and subsequently forwarding the packets transiting through the software-bridge maintained by DOM0.

To introduce network delay over the transiting packets the daemon in DOM0 uses the system-provided suspension procedure (*usleep*) over a high-resolution timer (in microseconds). The daemon traps the packet and sleeps for the *delay_time,* which is calculated using the arrival time of the packet and the *current* wall-clock time obtained from *gettimeofday* at DOM0. Coming out of *usleep*, the daemon sets the verdict as *ACCEPT* and sends it to the kernel, which forwards the trapped packet to its destination. The daemon then picks the next queued trapped packet and continues the same process. Note that the *delay_time* calculated using the arrival time of the packet and the current wall-clock time takes into consideration the time spent by the packet in the queue.

2.5 Prototype Evaluation

The aforementioned functionalities was implemented as the NetWarp system, and debugged and tested on Linux software platforms executed on Intel processor-based hardware. We now describe the benchmark test scenarios and the runtime results obtained in a performance study.

2.5.1 Benchmark Scenario

We implemented a simple parallel program using the Message Passing Interface (MPI) [48] library to test and also to compare NSX against CSX. The parallel execution is distributed across the user-DOMs such that exactly one process of the parallel execution is run on each user-DOM. Three domains, DOM-A, DOM-B and DOM-C, host three MPI processes of rank-0, rank-1 and rank-2, respectively, which participate in multiple *messaging rounds*. In a *messaging round*, the rank-0 process sends a message to rank-1 process and a then message to rank-2. The rank-1 process that was waiting for message from the rank-0 process receives it and then sends out a message to rank-2 process immediately. The message sent out using *MPI_Send* by rank-0 contains the integer 0 as its data, and the message sent out by rank-1 contains the integer 1. We use *MPI_ANY_TAG* and *MPI_ANY_SOURCE* while receiving MPI messages to ensure that *MPI_Recv* does not interfere with the observed ordering in reception.

Note that the interacting parallel processes are running on different DOMs. Hence, for an individual messaging round, if we were to model the *simulated* network delay experienced by the communication message as the same between any pair of VMs, and, if the DOMs are being scheduled in simulation time-order, then the causal order of arrival of messages received by rank-2 (in DOM-C) must contain data 0 followed by one (0-1). In other words, rank-2 must first receive the message from rank-0, before it receives the message from rank-1. Note, the absence of any computation between the send and receive MPI calls used in realizing the test-program largely reduces the time gap between the direct message from rank-0 and the relayed message from rank 1, at rank-2. This makes the test fine-grained and tight.

If we were to run multiple *messaging rounds* one after another, then the correct time-ordered messaging sequence received by rank-2 will be a sequence of [0-1-0-1-0-1…]. An occurrence of a break in this ordering is counted as a single breach in time order or a *unit error*. We count all such errors committed in a large number of *messaging rounds* to determine the *error percentage* from a single run. The mean or average of several of these runs is used to obtain a *mean-error-percentage* value. In all of our experiments, the number of *messaging rounds* in each run was 1000, and the *mean-error-percentage* value was obtained from averaging the *error-percentage* over 30 runs.

To record the *run time* of the experimental runs, the *real-time* of the parallel execution on the DOM on which it was initiated (DOM-A) was recorded. The *run time* was recorded for each of the 30 runs and their mean value is plotted.

2.5.2 Hardware and Software

The test setup comprised a Xen hypervisor v3.4.2, with Linux running as its DOM0 and all the other three user-DOMs (DOM-A, DOM-B and DOM-C). Each of the user-DOMs was assigned static IP address. In DOM0, a bridge named *privatebr* was created using the *brctl* tool, for establishing network connections between the user-DOMs. The DOM0 itself does not connect to *privatebr* and hence remains disconnected from the network of user-DOMs. For the CSX readings, the *weights* for all the DOMs were kept the same at 256, and none of the DOMs were capped to ensure maximum fairness in functioning of CSX.

The setup and the test runs were carried out on a MacBook-Pro with Intel® Core™ 2 CPU T7600 @ 2.33 GHz, with 3 GB memory, running OpenSUSE 11.1 Linux over Xen 3.3.1. The source code of Xen 3.4.2 distribution was used for making scheduler modifications. The resulting xen-3.4.2.gz was used to boot the hypervisor, and the boot-loader grub configuration was changed to enable the selection of modified Xen during the boot up. All the user-DOMs were configured to run OpenSUSE 11.1 Linux and they were installed as para-virtualized DOMs for best performance. The test program was written in the C programming language using MPI (OpenMPI v1.4.1) library.

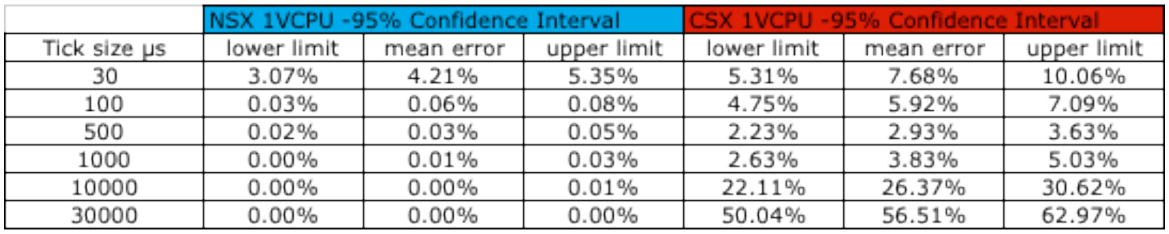
2.5.3 Virtual Time-ordered VM Execution Test Results

By default CSX maintains a tick size of 10ms and during scheduling every VCPU is allotted a time-slice equal to thrice the tick size. Hence, in our experiments, the maximum tick-size was set to 30ms, while a minimum to 30μs was exercised. Below 30μs, the interactivity suffers heavily (for both CSX and NSX), and the performance of the user-DOMs deteriorates because of high context switching rate. The performance was evaluated using two test-case scenarios. In the first scenario each DOM was configured with single VCPU, whereas, in the second each DOM was configured with two VCPUs.

2.5.3.1 Case 1: Single VCPU per DOM

Thirty runs with each run performing 1000 *messaging rounds* were carried out. In Figure 19 the mean-error is plotted against the tick-size for the NSX, the CSX, the NSX with 2x and 4x DOM0 tick sizes. Mean error was obtained from 30 runs with each run of 1000 messaging rounds in a 1-VCPU per DOM scenario. The corresponding runtime plots in seconds vs. tick-size in micro-seconds are presented in Figure 20.

Table 1 Mean error with confidence intervals for 1 VCPU/DOM scenarios



	NSX 1VCPU -95% Confidence Interval			CSX 1VCPU -95% Confidence Interval		
Tick size μs	lower limit	mean error	upper limit	lower limit	mean error	upper limit
30	3.07%	4.21%	5.35%	5.31%	7.68%	10.06%
100	0.03%	0.06%	0.08%	4.75%	5.92%	7.09%
500	0.02%	0.03%	0.05%	2.23%	2.93%	3.63%
1000	0.00%	0.01%	0.03%	2.63%	3.83%	5.03%
10000	0.00%	0.00%	0.01%	22.11%	26.37%	30.62%
30000	0.00%	0.00%	0.00%	50.04%	56.51%	62.97%

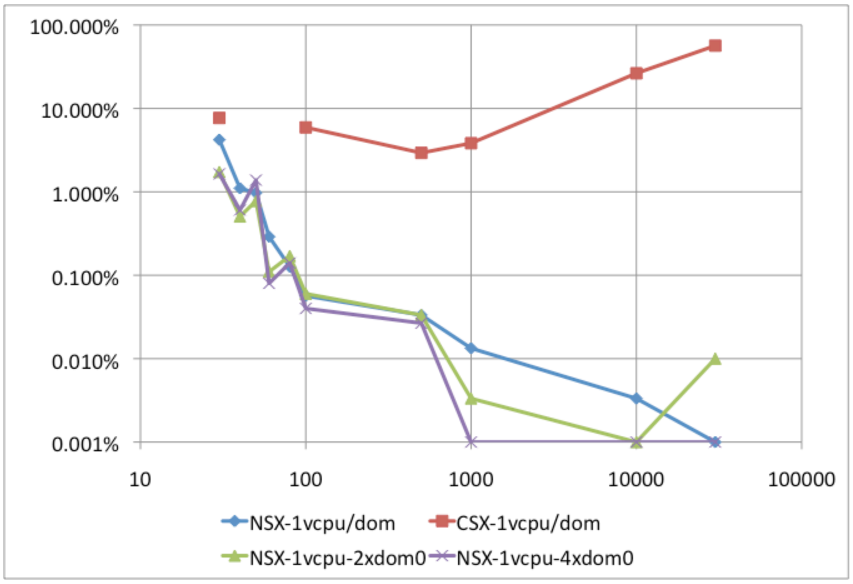


Figure 19 Mean-error (y-axis in %) vs. tick-size (x-axis in **µ**s), for single VCPU/DOM

For NSX, we observe a dramatic reduction in the *mean-error* with the increase in the tick-size. With CSX, the *mean-error* drops by a small amount initially and steeply increase later with the increase in the tick size. From Figure 19, we see that as the tick size increases from 30µs to 30ms, the mean error percentage decreases from 4% to 0%, whereas CSX increases from 3% to 56%. Figure 20 shows almost the same runtime for both NSX and CSX. However, the runtime of NSX suffers at lower tick sizes. The increase in the runtime with increase in the tick size is as expected because the tick-size allotted for every VCPU becomes larger than necessary, which essentially results in wasted compute cycles.

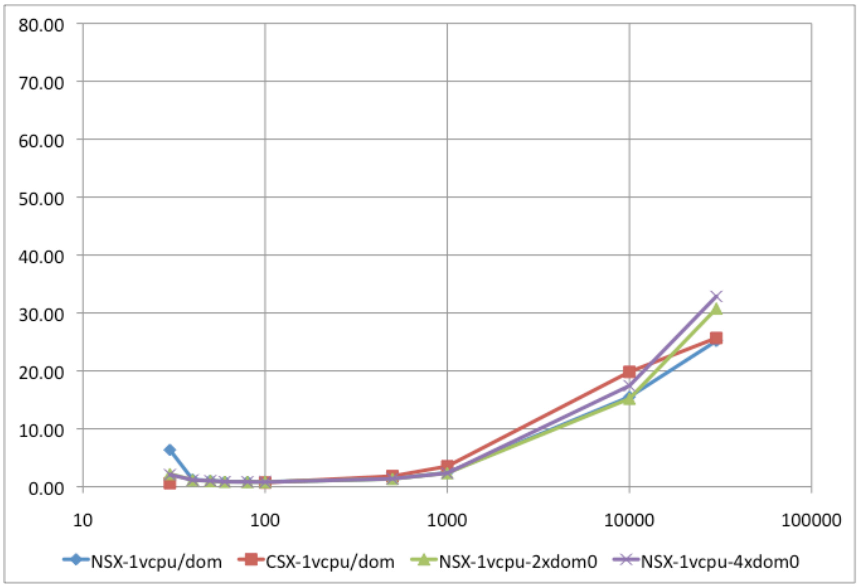


Figure 20 Runtime in seconds for the mean-error runs in single VCPU/DOM

2.5.3.2 <u>Case 2: Multiple VCPUs per DOM</u>

A similar reduction in the *mean-error* is observed in Figure 21. However, it does not go down to absolute zero just as observed in the 1-VCPU plots, but stays around 0.04%. Similar to the 1-VCPU scenario, an increase in CSX's mean-error with increase in the tick size is observed. At a tick-size of 30ms, the error was observed to be 49%. The runtime plot in Figure 22 shows a steep increase in the NSX runtime with increasing tick size, which is decreased by increasing the DOM0 tick size.

Table 2 Mean error with confidence intervals for 2 VCPU/DOM scenarios

	NSX 2VCPU -95% Confidence Interval			CSX 2VCPU -95% Confidence Interval		
Tick size µs	lower limit	mean error	upper limit	lower limit	mean error	upper limit
30	2.26%	3.52%	4.78%	6.31%	9.00%	11.68%
100	0.19%	0.25%	0.31%	4.40%	5.84%	7.27%
500	-0.02%	0.10%	0.22%	2.04%	2.92%	3.80%
1000	0.01%	0.03%	0.04%	2.52%	3.67%	4.82%
10000	0.02%	0.04%	0.06%	22.81%	26.63%	30.46%
30000	0.02%	0.03%	0.05%	43.60%	49.43%	55.25%

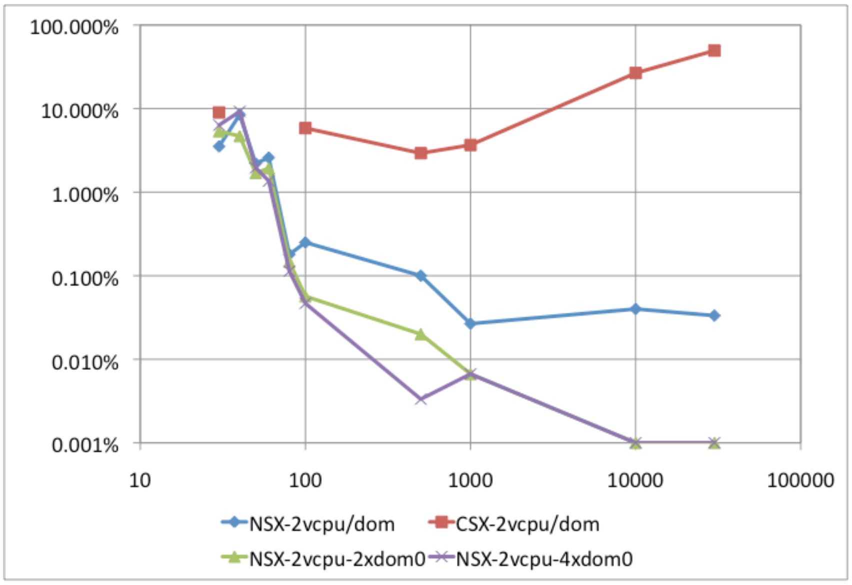


Figure 21 Percent mean-error against tick-size for 2 VCPUs/DOM

2.5.3.3 Time-order Error in CSX

The CSX caps the *over-scheduled* VCPUs and chooses *under-scheduled* VCPUs for scheduling; in doing so, it constantly re-orders the *run_queue* based on priority, and, when all VCPUs become over-scheduled, the credits are re-assigned and this process continues. In doing so, the CSX maintains fairness, but time-order suffers. This is because the program execution and communications are asynchronous, and the loads on the processes in the parallel program are usually not equal to each other. Capping one VCPU while scheduling others increases the probability of committing time-order errors. For example, capping the VCPUs of DOM-A holds back DOM-A from sending message(s) to DOM-C in time as expected and the under-scheduled DOM-B VCPU given more-cycles will be able to send message to DOM-C before DOM-A does, hence creating a time-order error.

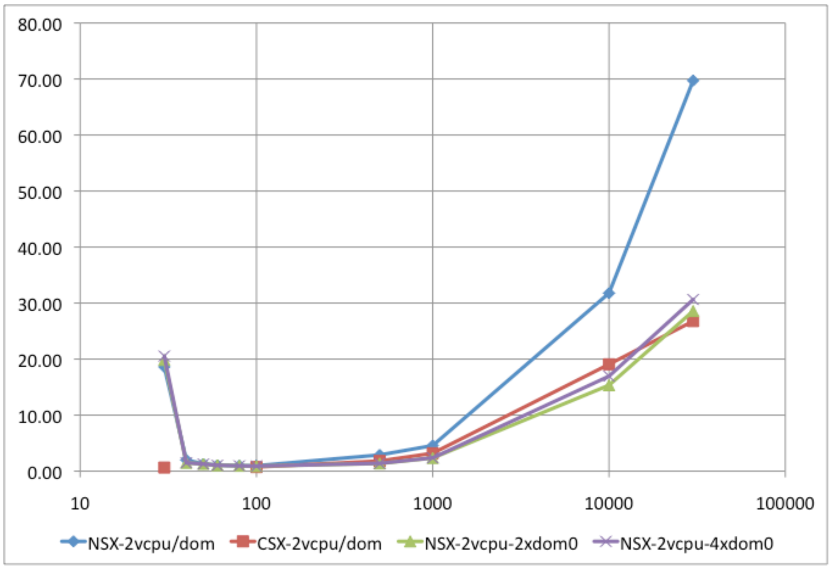


Figure 22 Runtime in seconds for mean-error runs in 2 VCPUs/DOM

2.5.3.4 Time-order error in NSX

In NSX, VCPU's cycles are always provided and are not blocked anytime. However, the VCPU with the least LVT value in the run-queue will be executed first. This results a staggered time line for each VCPU's LVT, but they are adjusted periodically as discussed earlier.

As seen in 1-VCPU and 2-VCPU case studies, the incidences of time-order errors are mostly at lower tick-sizes. The error occurrences tend to increase as the tick size decreases. The high-frequency context switching between the VCPUs is responsible for the errors. This reasoning is supported by the runtime plots showing higher runtime at lower tick sizes for executing the same set of experimental scenarios.

Since DOM0 manages the back-end drivers, network-bridge and also serves as interaction interface, its starvation due to a higher context switching frequency aggravates the error-rate. The error-rate can be significantly alleviated as seen in Figure 19 and Figure 21, by providing larger tick sizes to the DOM0. In the experiments, we provided twice (2x) and four times (4x) tick sizes to DOM0. This also reduces the runtime as seen in Figure 20 and in Figure 22, and, the impact of this change is clearly evident. As the DOM0's LVT does not affect the simulation time this change does not adversely impact the error in the test-scenario.

2.5.3.5 Mean-error and Runtime at Lower Tick-sizes

The smallest supported tick-size is very essential to determine the lower limits in the simulation of low-latency and high-bandwidth network simulation scenarios. Hence, the lowest tick-size at which the simulation time-order errors and runtime performance are acceptable is of importance. Empirically, at 20μs tick size, both CSX and NSX posed interactivity problem; the lowest tick-size with which we could run the experiments was 30μs. For the 1VCPU/DOM scenario, Figure 19 and Figure 20, with 2x DOM0 indicate the best error reduction; at 60μs, the error reduces to 0.1% and the runtime 0.86 seconds. From Figure 21 and Figure 22, we see that in the 2 VCPU/DOM scenario the 2x DOM0 gives an overall best error reduction and better runtime. At 80μs, the error reduces to 0.18% and the runtime by 0.95 seconds.

2.5.3.6 Result Summary

Table 3 Virtual time-order VM execution evaluation summary

Design requirements	Implementation
Simulation Time	
▪ **Global virtual time**	✓ Achieved by *sys_lvt*
▪ **Idle VCPU accounting**	✓ Done periodically
▪ **Idle DOM accounting**	✓ Done periodically
Causality	
▪ **Virtual time-order**	✓ Ensured using LLF scheduling
▪ **Time-order errors**	✓ Reduced to < 1%
New DOM needs	
▪ **DOM add/remove**	✓ Supported
▪ **Virtual time during VM boot**	✓ VCPU of new DOM is initialized to *sys_lvt*
DOM0 needs	
▪ **Sufficient CPU cycles**	✓ DOM0 VCPUs maintained at *sys_lvt*
▪ **Increased CPU cycle requirements of DOM0**	✓ Addressed by altering tick-size for DOM0
▪ **Starvation of DOMUs by DOM0**	✓ Absence verified by experimentation
Tick-size	
1VCPU/DOM	
▪ **Smallest tick-size**	✓ 60µs
▪ **Time-order error < 1%**	✓ 0.1%
▪ **Runtime for specified error**	✓ 0.86s
2VCPU/DOM	
▪ **Smallest tick-size**	✓ 80µs
▪ **Time-order error < 1%**	✓ 0.18%
▪ **Runtime for specified error**	✓ 0.95s

2.5.4 Virtual Time-ordered Network Control Results

In the previous section, the tests were primarily concerned with the virtual time-order execution without exercising network control. In this section, we introduce the network control on the same test scenarios and analyze the corresponding runtime behavior. We configured the network control program such that a constant finite delay is experienced by each of the packets transiting through the bridge. A single run comprising 1000 messaging rounds was run with single VCPU per DOM for varying constant-delays (10ms to 100ms in real-time) was introduced, and the NSX scheduler in all the test cases maintained a tick size of 1ms.

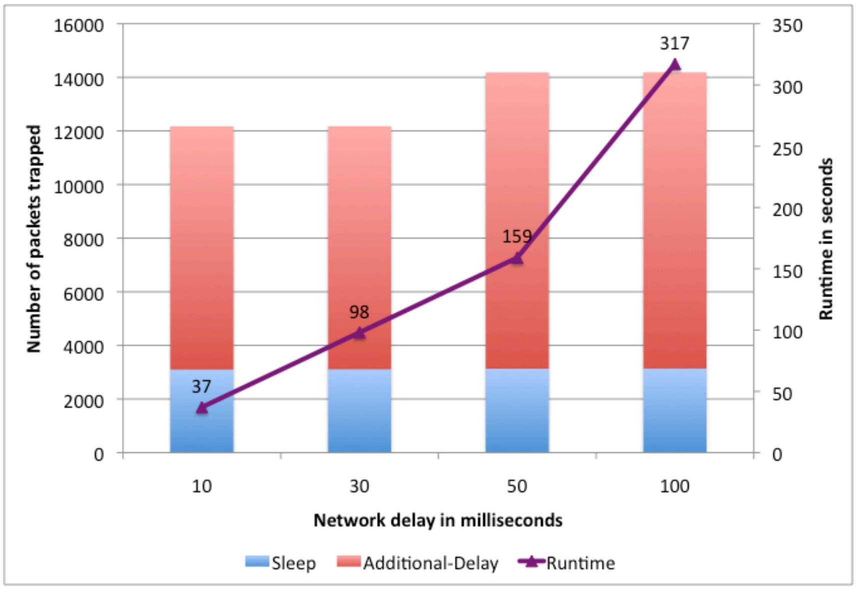


Figure 23 Number of packets trapped and runtime plots for a given network delay

Figure 23 shows the number of packets trapped during 1000 messaging rounds, with CND values ranging from 10ms to 100ms. The color-coding of the bar differentiates between number of trapped packets experiencing CND and packets experiencing delay greater than the specified CND. The curve represents the runtime.

As seen from Figure 23, most of the incoming messages incur additional delay in terms of virtual network processing overhead. This is because the network control daemon sleeps for *delay time* (calculated using arrival wall-clock time and current wall-clock time); due to this inaction in packet forwarding during this delay period more packets tend to queue up. Hence, the chances of experiencing additional delay are higher if many packets arrive almost simultaneously. Interestingly, for 50 and 100 millisecond CND scenarios, the number of messages involved for the same set of runs has considerably increased, which essentially suggests of packets being dropped. However, the runtime curve shown in Figure 23 is in line with expectation, which increases the run time with increase in the constant-delay value.

Over 75% of the transiting packets experienced additional packet delay in almost all the test cases. Further, since this method of trapping and forwarding the packets serializes the communication control, we observed negligible time-order errors in case of CSX and no errors using NSX. Nevertheless, such a behavior cannot be guaranteed on different test setup and larger parallel platform.

Note that the delay enforced on the transiting packets in this network control prototype is not based on the virtual-time. However, the constant-delay based on real-time is enforced in DOM0 is important because it reveals that this method of infusing artificial delay on to the transiting packets are useless, unless queuing delays in FIFO are circumvented.

2.6 Summary

We started by introducing the large area of network simulations and emulations. The known simulators and emulators were broadly classified into two categories based on their simulation goal as, end-host-centric and network-centric. Current state-of-the-art VM based network simulators and emulators were introduced and a nomenclature to identify and categorize them was provided. Within VM based network simulators a classification based on the types of VMs used namely, application-level, OS-level and machine-level VM simulators was introduced. Further, the advantages and disadvantages of each VM based simulator was discussed. Then the conceptual issues that one need to address to realize VM based network simulators were discussed. This was followed by the introduction to NetWarp Simulator, its goals, architecture and the implementation approach. The implementation of the NetWarp prototype was discussed under two sections namely, virtual time-ordered execution and virtual time-ordered network control. To evaluate the NetWarp prototype, we designed a benchmark to qualitatively capture time-order error and quantitatively measure the time-order errors. The results from the benchmark runs evaluating for virtual time-ordered VM execution and virtual time-ordered network control were presented and discussed. Major parts of this portion of research work were published in [49].

CHAPTER 3

VM WITHIN VTS: SCALING STUDY

In the last Chapter we dealt with the design concepts, prototyping and evaluation of virtual time-ordered VM execution and network control of NetWarp. In this Chapter, we focus on the scaling issues of the NetWarp system. We start with the discussions on the instrumentations performed to the VM scheduler, and the design, development and implementation of NetWarp Network Control (NNC). We then discuss the methodology used for the scaling study along with the benchmarks that were specifically developed to study correctness, runtime performance and scaling behavior of NetWarp. Finally, we present the results from the benchmark runs.

3.1 Staggering of Virtual Time

3.1.1 Virtual Time Evolution

In the NetWarp design, each VCPU maintains a virtual clock that advances based on the number of PCPU cycles it utilizes. We refer to this local virtual time maintained by the VCPU as VCPU-LVT. Each DOM maintains a DOM-LVT variable, a maximum of all VCPU-LVTs of the DOM's VCPUs that is computed periodically. The scheduler maintains a queue of VCPUs for each PCPU. After the exhaustion of the allotted time-slice the VCPU is enqueued into a PCPU queue, which is dynamically chosen during execution. To ensure virtual time-ordered execution, the NSX scheduler employs a *Least-LVT-first* (LLF) scheduling policy, using which the scheduler picks the VCPU with lowest VCPU-LVT (among all the VCPUs across all DOMs) for execution on a free

PCPU. Due to the presence of multiple PCPU queues, the scheduling involves searching of all the PCPU queues to pick the VCPU with lowest LVT value. LLF scheduling of VCPUs ensures regular progress of DOM-LVTs within the host node. However, the staggering of VCPU-LVT timelines during simulation results in the staggering of corresponding DOM-LVT timelines.

3.1.2 Virtual Time Staggering Test

```
struct MSG{ int ID, counter; } msg;
void Staggering_LVTs( )
{
    If(myrank == 0) {
        for(r = 1 to size-1) {
            recvfrom(ANY_RANK, msg);
            print msg.senderrank;
        }
        for(r = 1 to size-1) {
            sendto(r, msg);
        }
    } else {
        done = false;
        while(not done){
            sendto(rank-1, msg);
            recvfrom(ANY_RANK, msg);
            if (msg.sender==0) done=true;
        }
    }
}
```

Figure 24 Simulation time divergence test algorithm

To study the effects of staggering of unsynchronized DOM-LVT timelines, we designed a test algorithm that introduces imbalanced load on the parallel processes. Figure 24 gives the algorithm of our test program. The experimentation involves the hosting and execution of a parallel process on a DOM, which essentially results in *p* processes of a parallel task using *p* DOMUs. As per the algorithm, the parallel process with “rank” *r* sends a message to process with rank *r*-1 and waits to receive a new message (originating from any source). The *Lowest Ranked Process* (LRP), *i.e.*, *r*=0, does not send any messages until it receives *p*-1 messages, where, *p* is the number of processes in the parallel computing task. Thus, the *Highest-Ranked Process* (HRP) i.e., *r*=*p*-1 sends a single message before blocking on a receive call from LRP, while the process with *r*=1 receives (*p*-2) messages, and sends (*p*-1) messages to LRP. Consequently, the load (computation and communication) on the process ranks decreases with increasing rank. We periodically (2 seconds) collected the minimum and maximum DOM-LVT values in the hypervisor from DOM0 using Xen’s *libxcutil* library routines.

3.1.3 Virtual Time Staggering Test Results

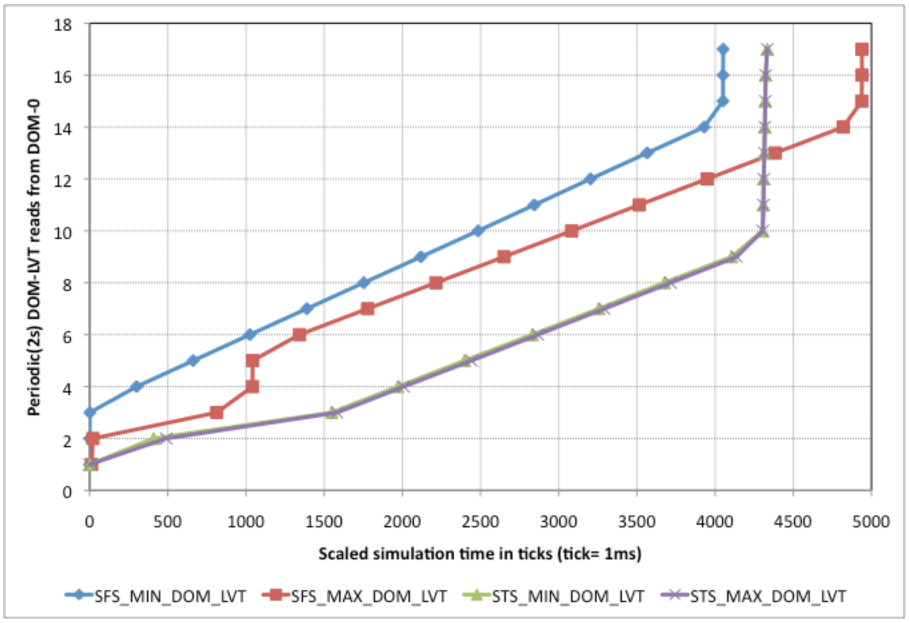


Figure 25 Experimental results demonstrating simulation time divergence

Two experimental cases, namely, *Scheduling Free of Synchronization* (SFS**)** and *Scheduling with Time Synchronization* (STS) were evaluated to demonstrate the staggering of the DOM-LVTs in SFS and its absence in STS. The STS resets the LVTs of all the DOMs and their corresponding VCPUs to the maximum VCPU-LVT (*max_lvt*), periodically. The periodicity of the synchronization was set to (*NVCPUS* × *tick_size*), where *NVCPUS* is the total number VCPUs in the test environment (including DOM0 VCPUs), and *tick_size* was the time-slice duration. For example: for a *tick_size* of 100μs, the periodic synchronization time for a test scenario comprising 64 single-VCPU DOMs and DOM0 with 24 VCPUs will be ((64+24) × 100μs)= 8.8ms. The experiment was performed over 64 single VCPU DOMUs and a DOM0 with 24 PCPUs (equivalent to number of CPU-cores supported by the machine). The *minimum* (MIN) and *maximum* (MAX) of sampled DOM-LVTs were collected periodically for both SFS and STS scenarios.

Except for the startup time, which appears to be an artifact of parallel job launching, the DOM-LVT values are observed to increase at uniform rate. At the end of the experimental run the simulation time remained constant, and can seen as almost vertical lines toward the end of test run, as shown in Figure 25. While two distinct curves representing MIN and MAX DOM-LVT values can be observed in SFS case, the same curves overlap in the STS, showing the need and the solution, respectively, for virtual timeline synchronization. At the end of the experimental run the MIN and MAX simulation time values in SFS were 4052*ms* and 4939*ms*, respectively. Note that the 4334*ms*, which is both MIN and MAX DOM-LVT values in STS falls in between the MIN and MAX of SFS DOM-LVT values. The sampling for the DOM-LVTs periodically in wall-clock time also reveals the runtime behavior of the STS and SFS setups. The plot from the Figure 25 suggests that the runtime of the STS method is better than that of the SFS method. We also ran experiments to compare the virtual time-order execution errors in the STS and SFS setups. We found that the errors from STS setup were far lesser than the SFS. Hence, we incorporated STS mode of operation in NetWarp.

3.2 NetWarp Network Control

In discrete-event network simulations, the arrival and departure of communication packets from and to the network respectively, at the end-hosts are generally modeled as events, since they determine the state-changes in the interacting nodes. Further, in contrast to the network emulation, discrete-event simulations takes leaps in simulation time as it processes the events, thereby potentially achieving faster-than-real-time execution. With VMs, however, to realize the simulation method of capturing communication activity as events, inter-VM communication must be virtualized. This can be done by capturing the packet in transit, time-stamping the packet with the virtual time, and delivering to the destination VM at a correct virtual time. In this section, we discuss our virtual network control (i.e., controlling inter-VM network traffic) methodology, design and implementation in conjunction with the virtual timelines established in prior section.

The hypervisors provide a variety of means to setup a virtual network to support the interaction across the hosted VMs. We use a private bridge that isolates the VMs involved in the simulation from the privileged VM (DOM0). By controlling the network, we would have the capability to introduce a specified virtual time-delay on any in-transit packet. This virtual time-delay can be specified based on the packet's source and destination addresses.

Since, we are aware of the virtual arrival-time of all the in-transit packets, by processing them in their emit-time order, we can also leap in virtual time. To achieve this, we provide the synchronization ability (as previously achieved by STS) to the network control subsystem. In this section, we discuss the NetWarp Network Control (NNC) subsystem design issues and the synchronization mechanism that it provides to the NetWarp, allowing it to leap in simulation time.

The virtual network control is intended to provide the following:

- Introduce a virtual delay without explicitly making a (byte-)copy of the packet buffer, or moving the transiting packets from kernel to user space
- Support the ability to introduce virtual time-delays that may be varied on a per packet basis; the delay specification may be static (e.g., for wireline networks) or dynamic (e.g., for mobie ad-hoc networks)
- Minimal overhead while processing the trapped in-transit packets

3.2.1 Network Control Approach

To establish a control on an in-transit packet, we should first be able to trap the packets in transit. This can be achieved in the control domain as all communication packets traverse through it as discussed in the previous Chapter.

The *netfilter* [50] a packet filtering framework inside Linux® kernel services is used for network control. The *iptables* rules redirect the in-transit packets to a specific *netfilter queue* (NFQ) maintained by the kernel packet filter, while the *libnetfilter_queue* function-APIs are used to control the queued packets from the userspace. The *libnetfilter_queue* function APIs allows copying the packet from the *NFQ*s in the kernel space to a servicing process in the user space. The process in user space ultimately decides the fate of the trapped packet by setting a verdict. The relevant verdicts that could be used for our purpose are: *NF_ACCEPT* – releases the packet to continue its journey toward its destination, *NF_DROP* – drops the packet, *NF_QUEUE* – inserts the packet back into the same or other similar NFQs.

While using *netfilter* for the purpose of network control we need to be aware of (a) the NFQs are FIFOs and are unaware of any *virtual time* (b) the service thread processing packets from a NFQ *must* set a verdict on current packet before servicing the next in NFQ.

3.2.2 Network Control Design Alternatives

In this section, we explore a range of possible strategies for network control using *netfilter*. In the process, the infeasible and/or inefficient approaches are identified and are discounted before arriving at a specific control mechanism that we finally adopt in NNC.

Let ST, SQ, MT and MQ stand for Single-Thread, Single-Queue, Multiple-Threads and Multiple-Queues, respectively. The combination of single- vs. multi-threaded processing and single vs. multiple queues provides four options.

In single-threaded operation, only one thread of control in DOM0 handles all tasks in ordering tasks of all emitted packets from all VMs. In the multi-threaded scheme, multiple threads share the task of introducing virtual time-delay on the incoming packets, and emitting them to the destination when the destination reaches virtual time equal to reception time of the packet.

The number of queues, similarly, can be varied to delay the packets, while the packet's destination has not reached the appropriate virtual times. We do not discuss Multiple Thread-processing using Single Queue (MT-SQ) approach, as it would not be a feasible given the limitations in processing an NFQ in consideration with our requirements, as discussed previously. As many as 64K queues can be put to work at once using the *iptables* and *libnetfilter_queue* library.

3.2.2.1 Single Thread-processing using Single Queue (ST-SQ)

In this mode, all the VM-generated traffic is passed through a single NFQ and a single service thread processes the packets in the order they are enqueued. By processing, we mean that the service thread would determine the virtual *emit time* of the packet based its source and destination, then sets an NF_ACCEPT verdict when the simulation time catches up with the virtual *emit time*.

Considering that the arrival sequence of the packets would be virtual time-ordered, if the delay to be enforced varies on every packet, the approach becomes infeasible due to the limitation that the currently processed packet must be emitted before processing the next. Further, even if we were to assume that the user just wants to enforce a constant delay, this method suffers from *queuing delay* issues, wherein the packets that arrive almost during same period of time due to the queuing nature of processing incur significantly large additional delays based on its position in the queue and the required magnitude of the delay. These delays increase with the increase in the number of DOMs involved in the simulation.

An alternative to avoid *queuing delay* is to postpone the processing of the first packet. Since, we must avoid packet copying to minimize runtime and memory overheads, we can use an NF_QUEUE verdict, which reinserts it back into the NFQ. However, by this approach we lose the arrival order of the packet and hence this scheme would not be correct for virtual network control purposes.

3.2.2.2 Single Thread-processing using Multiple Queues (ST-MQ)

Since *netfilter* allows usage of multiple NFQs, several different strategies of enqueuing and processing can be designed. For example: packets can be enqueued, based on their source or destination, into a specified NFQ, and processed as they arrive in their respective NFQs. The *libnetfilter_queue* API based service thread servicing multiple NFQs tries to handle all of them equally regardless of their queue size.

This method of processing is attractive because the order in which the packets are processed can be adequately controlled. We can realize *arrival time-ordered* processing, if NFQs are *packet source* based, i.e. enqueuing the trapped packet into a NFQ based to its source. Similarly, the *departure time-ordered* processing can be realized with the NFQs based on *packet-destination*. But, due to the presence of single service thread, we need to deal with the *queuing delay* problem similar to ST-SQ, if our processing involves waiting to release the packet till the *simulation time* catches up with the *emit time* of the packet. Additionally, due to the presence of multiple queues, there will also be lapses in the time-ordered processing of events as the service thread processing multiple NFQs does not follow any time-order in processing. Hence, this also is not a feasible approach.

Alternatively, one can realize service thread processing the packets from the multiple NFQs in an almost time-ordered fashion and also accommodate variable delays on to the individual packets, if the strategy of enqueuing in multiple NFQs is altered. In this approach, there is not a fixed NFQ on which a packet arrives as in the former approach; instead, each arriving packet sequentially moves from one NFQ to the other in increasing order of the NFQ identifier (could also be decreasing order). The service thread looks into the packet arriving at the NFQ for its *emit time* and if it is greater than or equal to the simulation time, the packet is released by setting an NF_ACCEPT verdict; otherwise NF_QUEUE verdict is used to enqueue the packet into a NFQ with next higher identifier and by doing so the *queuing delay* is minimized. However, although the processing ensures that a packet is released when the *emit time* catches up with *simulation time*, we cannot be sure of maintenance of the *virtual time-order* as a single thread services multiple queues. Additionally, in this approach, we do not know the number of NFQs to create for a specific simulation and the means of handling the packet in the last NFQ whose *emit time* is still ahead of *simulation time*.

3.2.2.3 <u>Multiple Thread-processing using Multiple Queues (MT-MQ)</u>

The MT-MQ approach utilizes multiple threads for processing packets from equal number of queues. One relatively straight forward approach is allotting a queue for packets based either on the source or destination of the packet. A service thread corresponding to each queue processes the incoming packets in parallel. However, with this approach, we will not be able to overcome the *queuing delay* problem similar to ST-SQ and ST-MQ, but it would definitely be better in comparison to the latter because of dedicated service threads per queue. Even if we were to ignore the *queuing delay* issues, the introduction of variable delays can also be problematic. For example, the virtual *emit time* of the processed packet can be greater than the virtual *emit time* of the next packet, and hence, a service thread cannot *wait* for the *simulation time* to catch up with the virtual *emit time* for the release of the packet being serviced. Such scenarios can occur regardless of whether the queues are source-specific or destination-specific. Additionally, MT-MQ also introduces a large performance overhead because every thread will be making hypercalls to obtain simulation time in regular intervals. With a scenario involving 128 DOMs, 128 threads will be continuously burdening the hypervisor in regular intervals for virtual time progress information.

3.2.3 NetWarp Network Control Design

Consider a method in which multiple queues are serviced by multiple threads regardless of incoming packet's source and destination (as discussed in ST-MQ). In such an operation, the *queuing delay* can be minimized, and the varying delays on transiting packets can also be realized. However, to ensure *emit time-order* and minimize the performance overhead, the operation of multiple *service-threads* needs to be well orchestrated. This strategy is used in the design of NNC.

Multiple NFQs along with their corresponding *service-threads*, equaling the *number of DOMs* (specific to the simulation scenario) are used in NNC. The *iptables* rules in the control domain (DOM0) are set such that all the in-transit packets are routed to a single NFQ, with 0 *queue identifier* (*qid*). In Figure 26, this functionality is schematically presented using directional pointers suggesting the path of packet movement from a DOMU application to the front-end network device, and then to its back-end counterpart before being enqueued in the NFQ with qid=0. The *service-thread* (*service-thread0*) corresponding to this NFQ determines and marks the packet with the *emit time* before setting a *NF_QUEUE* verdict on the packet that results the enqueuing of the packet into a NFQ, with *qid*=1 (the next higher queue identifier). The first *service-*

thread performs only this operation, and hence, it continuously processes the arriving packets without introducing any additional delay other than the processing itself.

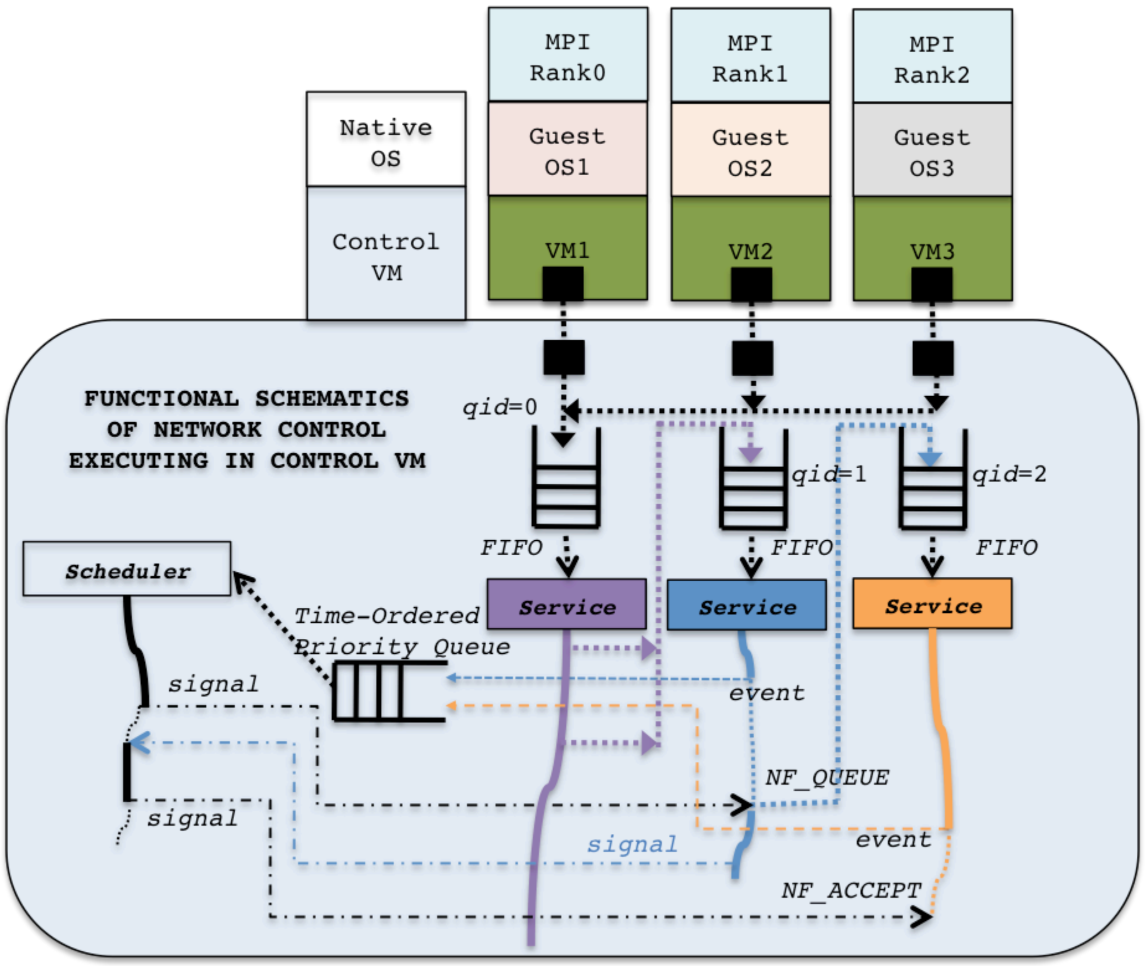


Figure 26 Functional schematics of NNC operation

Apart from *service-thread0*, all the other *service-threads* (corresponding to the other NFQs) on receiving the packet query for the *simulation time* and checks if it is greater than (*emit time* – INT_DELAY) value. If it is equal or greater then the packet is released from the NNC subsystem by issuing *NF_ACCEPT* verdict, as shown by the third service thread. If the desitnation DOM's virtual time has not advanced to the *emit time* yet, then the corresponding *service-thread* generates an *event* with an *event time* of (*simulation time* + INT_DELAY), inserts the *event* to the *eventlist*, and then blocks itself waiting on a signal from the *scheduler-thread*. This interaction is schematically presented in Figure 26, in which the solid-line represents the *service-thread* receiving and subsequently processing of the transiting packet, while, the thin dotted-line represents *service-thread* waiting for a signal from its peer.

The INT_DELAY mentioned previously refers to *intermediate-delay* in virtual time and is computed as

$$INT\_DELAY = MIN\_DELAY + ceil\left(\frac{MAX\_DELAY}{NUM\_DOMs - 1}\right)$$

Where, MIN_DELAY is a constant (usually 1) and MAX_DELAY is the maximum of the range of delays to be enforced by NNC on an in-transit packet. With segmented intermediate delays we ensure that an in-transit packet is released from NNC before it reaches the last NFQ, and it also ensures that the specified delay is introduced in its transit. Also, note that all delays are enforced in terms of *virtual time*. As mentioned earlier, the *simulation time* is kept track in terms of the *ticks*, and in our implementation, each *tick* corresponds to 100μs. For example, if we wish to enforce a MAX_DELAY of 10*ms* that correspond to100 *ticks* in *virtual time* on every transiting packet, and if our simulation scenario were to use 128 DOMs, then (1+ceil(100/128))=2 *ticks* (200μs) will be the INT_DELAY.

The *scheduler-thread* continuously processes the events in *event time-orde*r, and signals the respective thread when the *simulation time* advances to the *event time* and waits for a signal from the signaled *service-thread* before processing with next event. On receiving the signal from the scheduler thread the *service-thread* enqueues the packet into a NFQ with next higher *qid*, using the *NF_QUEUE* verdict. Thus, at every NFQ, the *service thread* either releases the packet or introduces a virtual time delay of INT_DELAY. With this method, we can greatly minimize *queuing delays*, efficiently process packets with varying delays, and release the packet from NNC in a perfect *emit time-order*.

3.2.4 NetWarp Network Control Implementation

The *virtual time* on every DOM advances when its corresponding VCPUs use of CPU cycles of the physical core and further the *simualtion time* advances as the DOM timelines advance. However, when a DOM is waiting (for a packet) or waiting for a signal from the scheduler (as in our NNC), the *virtual time* advance is very minimal as there is no physical CPU usage. Since, the *scheduler-thread* does not signal the relevant *service-thread* until the *simulation time* has advanced to the *event time*, during the enforcement of large delays *virtual time* advancement almost creeps, resulting in communication time-outs. This issue is resolved by leaping in *simulation time* in steps of *event time*. This technique is desirable because it not only addresses the virtual time-advancement issue in simulation experiments but also yields better performance.

Iptables rules in the DOM0 ensure that the packets on the virtual network are routed to the NFQ with identifier 0, which is serviced by *service-thread0*. Each *service-thread* is a posix thread developed using *libpthread*, *lib_netfilterqueue*, and *libxcutil* library functions and they spawn off from the main process, which itself becomes the *scheduler-thread*. A globally accessible *singleton* object comprising an *eventlist* (priority-queue) and thread synchronization related data-structures such as, mutex-locks, and conditional variables, is maintained. Each *service-thread* (except for thread-0) maintains a *state-variable*, which could be one of the following: *processing*, *wait_on_scheduler* and *wait_on_packet*. The *processing* state suggests the thread is busy processing a newly arrived packet, the *wait_on_scheduler* state means that the service thread is busy waiting for a signal from scheduler and the *wait_on_packet* means that the service thread is busy waiting for an arrival of new packet. The scheduler thread pulls out a new event for processing only when the service threads (except thread-0) are not in *processing* state. This ensures that all the to be handled *events* are in the *eventlist* and are thus time-ordered, and the event that is removed from the *eventlist* is the one with minimum *event time*. We use *libxcutil* library function interfaces to retrieve the *simulation time* from the Xen hypervisor from DOM0.

With a few modifications to the NSX source code and the usage of *libxcutil* library functions, we developed a feature using which a user program resets the *simulation time* to a higher value. This is achieved by pulling forward the virtual time of all the VCPUs (hence, their DOMs) whose current *virtual time* value is lesser than the *specified virtual time* value. The scheduler thread in NNC uses this feature to advance the simulation time during event processing. With this capability in place, periodic time synchronization is unnecessary for maintaining a single time line, and hence, the scheduler's periodic time synchronization is unused with NNC.

3.3 Scaling Study: Methodology and Benchmarks

In the previous Chapter we examined the issues of virtual timelines and virtual time-ordered execution; however, the scaling of our proposed solution to larger problem sizes and its application for real-life network simulation scenarios were not discussed. We gave preliminary evidence of the detrimental effects on the correctness of simulation results arising from the absence of explicit simulation-specific support in conventional hypervisor schedulers, and proposed a solution based on maintaining a separate virtual (simulation) time clock at the level of each virtual CPU core (VCPU). While it served as proof-of-concept that was performed on a small two-core host machine, the issues of scalability, correctness and efficiency on large numbers of guest VMs and host cores remained unexplored.

In this section we extend the work presented in Chapter 2 by (a) porting our new scheduler (NSX) based hypervisor environment onto a more powerful hardware platform, (b) designing new Message Passing Interface (MPI) based benchmarks, (c) implementing a cyber-security application with complex timing behaviors and messaging functionality (d) performing a thorough analyses for scaling and accuracy.

An important objective here is to ascertain whether, and to what extent, virtual time-based scheduling and networking, affects the correctness of VM-based simulations, under heavy multiplexing conditions, and in the presence of complex messaging dependencies.

3.3.1 Methodology

To increase the scope with respect to applications, we exercise our system with three qualitatively very different benchmarks. The benchmarks are intended to reflect sufficient complexity to overcome concerns of bias and generality, and sufficient in terms of simplicity to make them manageable for verification and duplication. The applications vary in terms of inter-entity dependencies and timing characteristics. The benchmarks were logically designed to infer how well our new hypervisor scheduler would support time-ordered execution, when compared to the default (fairness-oriented) hypervisor scheduler on the simulation host. In the first (MPI-based) benchmarks, the outcome from a correct, time-ordered execution is known, which is quantified and used to observe the extent to which untamed VM execution gives incorrect results. In the second (worm propagation), the expected qualitative nature of the outcome is used as a determinant of correctness; the degree of repeatability of the simulation is also compared in untamed and virtual time-ordered modes. To increase the scope with respect to scale, we experiment with varying number of VMs, from 1 to 128 (for a single simulator host node).

3.3.2 Hardware and Software

The experiments were performed on a Mac-Pro with two hex-core Intel® Xeon processors at 2.66 GHz, 6.4 GT/s processor interconnect speed with 32G of memory. With hyper-threading enabled, Xen sees 24 cores. With Xen creating 24 PCPUs to handle this, all our experiments view this system as a 24-core machine. OpenSUSE 11.1 with Xen-3.3.1 and Xen-3.4.2 source code was used on this hardware. The test machine is capable of hosting a maximum of 128 instances of OpenSUSE 11.1-based Linux VMs, limited only by the memory with which we configured each VM.

We used the OpenMPI v1.4.3 distribution of MPI to implement our test programs, in order to easily realize controlled point-to-point communications, for ease of experiment initiation, termination, statistics gathering, and to easily facilitate reusability and/or peer verification by the research community.

3.3.3 MPI Benchmarks

3.3.3.1 Constant Network Delay (CND) Test

By this experiment we test how well the NSX and CSX schedulers support time-ordered event execution when the communication structure and dynamics across the VMs (DOMs) is deterministic and only the observed message generation order differs.

```
struct MSG{ int ID, counter; } msg;
void CND( )
{
    If(myrank == size-1) {
        for( r = 1 to size-2) { //Initially populate
            msg.ID = r;
            sendto(r, msg);
        }
    } else if(myrank == 0) {
        for( r = 1 to size-2) {
            recvfrom(ANY_RANK, msg);
            print msg.ID; //Record observed ordering
        }
    } else {
        recvfrom(size-1, msg);
        sendto(0, msg);
    }
}
```

Figure 27 Algorithm for CND test benchmark

If *r* is the process rank and *p* is number of processes involved in the MPI test application, the "high rank" process (HRP) with rank *r*=*p*-1, sends out messages to other processes whose rank *r*>0 iteratively in ascending rank-order (from 1 to *p*-2). Upon reception, every receiving process "forwards" (i.e., sends another message) to the "least rank" process (LRP) with rank *r*=0. A single run of the test ends when the LRP receives all (*p-2*) sent messages.

With an assumption of constant delay incurred in the virtual network, the receive-order at the LRP must follow the send-order of the HRP. For example, if 1-2-3-4-5 is the message send-order, then the expected order in which the messages are received in a system following time-ordered execution of events must also be 1-2-3-4-5, since, all the sent messages experience the same network delay. We will refer to this test as the CND test. The pseudo code for the test algorithm is shown in Figure 27.

3.3.3.2 Varying Network Delay (VND) Test

By this experiment we test how well the time-ordered execution is supported/affected by NSX and CSX, when the generated messages vary both in their generation order and the communication load experienced. The pseudo code of the algorithm is shown in Figure 28. In this algorithm, the HRP generates *p*-2 messages; each message is populated with a variable counter (*msg_counter*) and a constant identifier (*msg_id*). In the first round, the HRP sends out the messages to processes whose rank corresponds to their *msg_id* iteratively in ascending rank-order. When the processes receive this message they decrement the counter and send it back to the HRP if the *msg_counter*>0 in the received message; otherwise the message is forwarded to the LRP. When the HRP receives the message back, it picks a random process rank from the set ranging from (1 to *p*-2) and forwards the message to the random ranked process. This continues until all (*p*-2) generated messages reach the lowest ranked process, at which point the LRP sends an end signal to all the processes marking the completion of a single run.

```
struct MSG{ int ID, counter; } msg;
void VND( )
{
    If(myrank == size-1){
        for(r = 1 to size-2){ //Initially populate
            msg.ID = r; msg.counter = r;
            sendto(r, msg);
        }
        done = false;
        while(not done){
            recvfrom(ANY_RANK, msg);
            if( msg type == ENDMSG ) {
                done = true; //Terminate
            } else {
                //Forward to random destination
                sendto(RANDOM(1:size-2), msg);
            }
        }
    }
    else if(myrank == 0){
        for(r = 1 to size-2){
            recvfrom(ANY_RANK, msg);
            print msg.ID; //Record observed ordering
        }
        for(r = 1 to size-1){
            sendto(r, ENDMSG); //Signal termination
        }
    }
    else{
        done = false;
        while(not done){
            recvfrom(ANY_RANK, msg);
            if( msg type == ENDMSG ) {
                done = true;
            } else if(msg.counter == 0) {
                sendto(0, msg);
            } else {
                msg.counter--; //Decrement
                sendto(size-1, msg);
            }
        }
    }
}
```

Figure 28 Algorithm for VND test benchmark

Again, the objective of this benchmark is to bring out the anomalies introduced by any virtual time-unaware execution. In a correct, time-ordered execution, the receive order in the LRP must be equal to the send order. This follows from the original order of message generation as well as from the number of hops that each message incurs. For example, the message with *msg_id*=1 takes only 2 (i.e., 2 × *msg_id*) hops to reach the LRP, whereas message with *msg_id*=*n*, takes (2×*n*) hops to reach the LRP. Under fixed network latency in the interconnecting virtual network, the expected receive-order must follow the message generation order.

3.3.3.3 Error Metric

The observed results from the parallel test programs can give a *qualitative* evaluation of the presence or lack of time-ordered event execution. However, to *quantify* the effect, we need an error metric. Hence, we define an error metric to characterize the ordering behaviors, designed such that the smaller the number, the closer it is to an ideal time-ordered execution.

Note that the benchmarks yield the expected (perfect) output ordering, if they followed time-ordered execution. However, in the absence of time-ordered execution, a different order would result. Thus, the *observed order* (*O*) in the output could be different from the *expected order* (*X*). To be able to measure the disparity in the *expected* and *observed* orderings, we introduce a notion of "*e*unit" as a unit measure of error, and use the following metric, *E*, in *e*units for measuring the time-order errors:

$$E = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m}\left|X_{ij} - O_{ij}\right|$$

where, n is the number of replicated runs, m is the number of parallel processes (ranks), X_{ij} is the expected identifier of the jth message in ith run, and O_{ij} is the observed identifier of the jth message in the ith run.

The error calculation metric stresses on the positioning of the output sequence elements such that the larger the gap between the *expected* and observed element values, the greater will be the error incurred. Hence, the error calculation metric penalizes the *observed* value that is too distant from the *expected* value.

3.3.4 Cyber Security Benchmark

To compare the performance of the NSX and CSX based simulation platforms while simulating a more complex network application, we developed a test program mimicking simple computer worm propagation; we will refer to this as the Worm Propagation (WP) test. This experiment emulates the behavior of worm-infection and its subsequent propagation across the service hosts in an interacting multi-server and multi-client scenario, as shown in Figure 29. The propagation in the system proceeds as a simple instance of the well-known "SI" epidemic model.

The experiment involves Vulnerable Services (VS) listening for requests from legitimate or non-malicious clients, referred to as Legit-Clients (LC). Upon receiving a request from any LC, a VS responds by spawning a service thread that would subsequently transfer data of uniform-randomly distributed size ranging from 1 to 10KB. Every LC generates requests to randomly selected service hosts, with inter-request interval ranging uniformly randomly from 10ms to 100ms.

One instance of each VS and LC are spawned on each VM node in the experiment. One among all the VSs is set to be initially in an *infected* state. The *infected* VS spawns an independently running Shooting Agent (SA) embedded in the worm script. The SA process, which is exactly similar to LC in its operation, picks a random VS to infect and makes a request similar to an LC; in addition to the normal data-transfer, this malicious request also initiates the process of opening a backdoor-port for worm payload transfer in the VS host, as shown in Figure 30. The payload file (4 KB) makes the VS host *infected* and the spawned SA of this host subsequently starts infecting its peers similar to the infected (seed) VS. Eventually, due to continuous interaction between the hosts the worm infects the VSs on all the hosts.

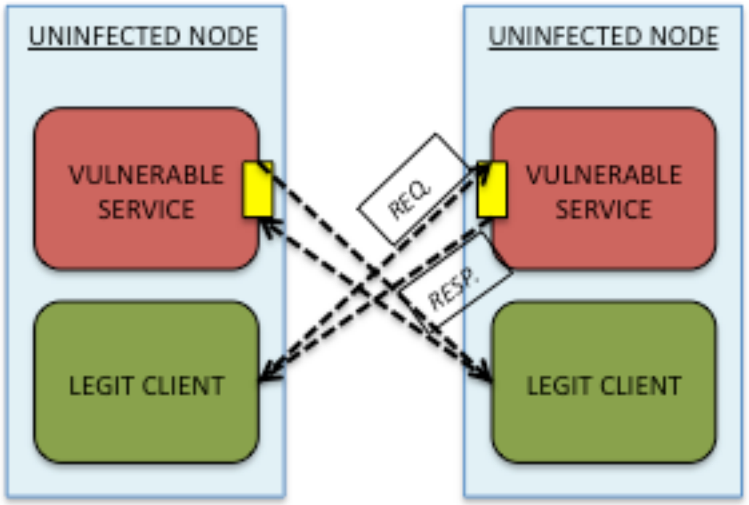


Figure 29 Interacting multi-service and multi-client scenario

TCP/IP (Berkeley) sockets were used to realize the VS, LC and SA communication operations. We conducted the experiments on 64 VMs and each VM starts up an instance of VS and LC. All LCs and the very first SA spawned by the seed VS are delayed by a 5-second sleep at their startup, to ensure all VSs are ready to accept requests.

After being infected (i.e. after payload transfer), every VS sends out a message to a pre-assigned VS, which, on reception, marks the sender as *infected.* When the pre-assigned VS determines that the specified fraction (90%, in our experiments) of VSs are infected, it sends out messages to all VSs to terminate. Each VS in turn communicates this message to the LC and SA hosted on the same VM by creating a *end_process_file* for each process before self-termination. The LC and SA terminate themselves on the detection of existence of their respective *end_process_files* after clean up. This ensures smooth termination of the WP test.

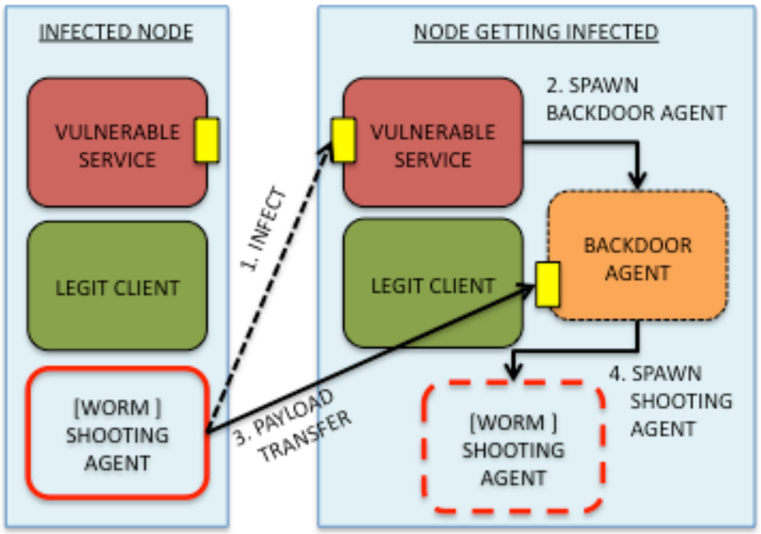


Figure 30 Worm infection and propagation

In the NSX based simulation host the simulation time is maintained within the scheduler data-structure, which is in the core of Xen hypervisor kernel. This simulation time has to be communicated to the VMs so that they can record their *infection* time. To accomplish this we developed a daemon named *update_lvts,* whose functionality is to get the simulation time value from the Xen scheduler data-structure and update it in a common data-structure that is accessible to all the VMs. We start up this daemon in DOM0, and using the *libxcutil* functions, we are able to get the simulation time into the

Xenstore [6], whose location is known and can be read by all the VMs. Hence, the VSs that record the simulation time in the NSX setup make use of *libxenstore* library functions to read simulation time from the *Xenstore*. For the CSX-based runs, the real time value returned by *gettimeofday* function is used as the simulation time.

3.4 Scaling Study: Results

3.4.1 MPI Simulation Results

3.4.1.1 CND Test Results with 64 VMs

From Figure 31 the curves obtained for the CND tests for virtual time-ordered scheduling (NSX) show its effectiveness in keeping the time-order errors at negligible values of below 2 *e*units for single VCPU/DOM test and below 2.5 *e*units for 2 VCPUs/DOM test. In contrast, although untamed execution (CSX) starts out very well with almost no errors for scenarios with 8 and 16 DOMs, it starts to perform very poorly at 24 DOMs and higher. This is expected because of its poor behavior with multiplexing more than one VM per core.

As long as the virtual resources match the physical host resources the fairness also ensures time-ordered execution of events as well, but as the number of DOMs increase the mismatch between virtual and physical resources plays significantly against time-ordered event execution. This is clearly evident from the observed *e*unit plots. Note also that it is indeed possible to get eunits down to zero, by reducing the time slice. However, reducing the time slice to very small values only increases the run time significantly; in practice, the errors become negligible even at time slice of 100 μs.

The CSX, which does not use any notion of time in scheduling VCPUs, is bound to yield increased time-order event errors with increased mismatch between virtual and physical resources. Hence, the errors increase with the increase in the number of DOMs in the test. For the 1-VCPU/DOM and 2-VCPU/DOM scenarios the error is 12-13 *e*units on average.

Figure 32, shows the wall clock time taken by NSX and CSX to complete the same experiment. The run time is seen to be similar across the schedulers, showing that the virtual time scheduling can be efficient with little runtime overhead.

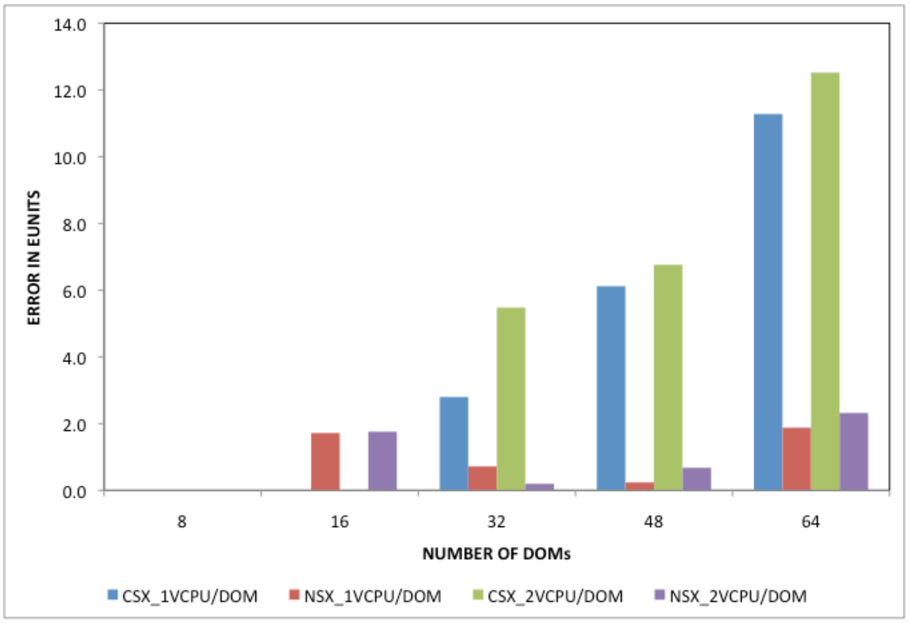


Figure 31 CSX and NSX time-order errors for CND benchmark

In essence, for tests with constant network delay that differentiates the messages from each other only based on the sent order the NSX scheduler maintains the expected time-ordered execution better than the fair-share CSX, without significantly compromising on the runtime performance. This characteristic of consistent maintenance of lower error-rate even with increasing number of DOMs in the tests demonstrates a good scaling behavior of the NSX scheduler.

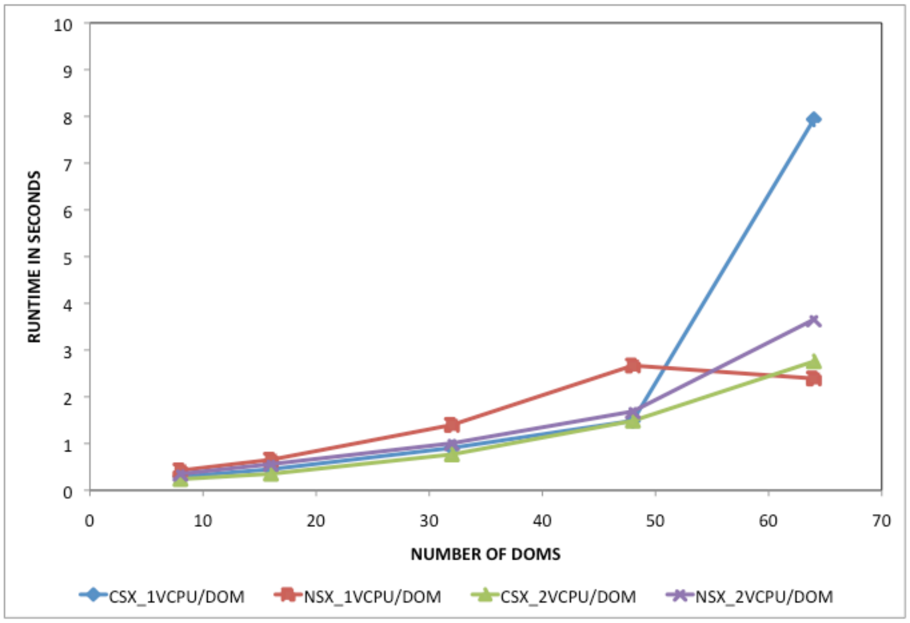


Figure 32 CSX and NSX runtimes from CND test runs

3.4.1.2 CND Test Results with 128 VMs

As observed from the error graph in Figure 33, the NSX scheduler for both 1VCPU/DOM and 2VCPU/DOM scenarios shows very low errors until the number of DOMs in the test scenario increases from 64 to 128, at which point the 1VCPU/DOM time-order errors are almost same as its peer CSX run. However, the runtime curves in Figure 34 show that NSX provides significantly better runtime performance and hence better scaling with increase in the number of DOMs in the experiments. CSX with lower tick-size does not perform any better in controlling the time-order error or in terms of runtime performance than its performance using the default setting in CND tests in

Figure 32. The 2VCPU/DOM case with CSX performs poorly both in terms of errors and runtime.

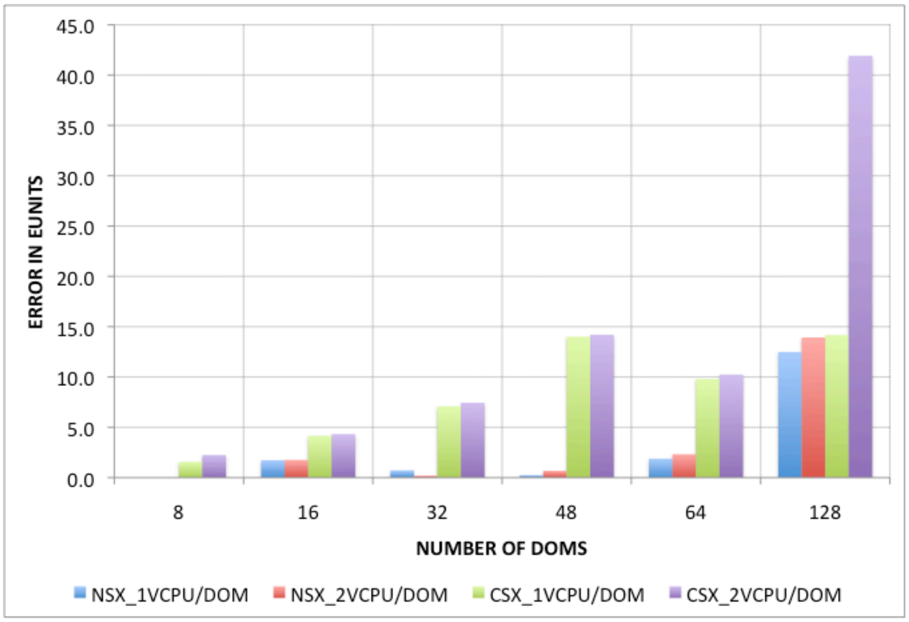


Figure 33 CSX (with lower-tick size) and NSX time-order errors for CND benchmark

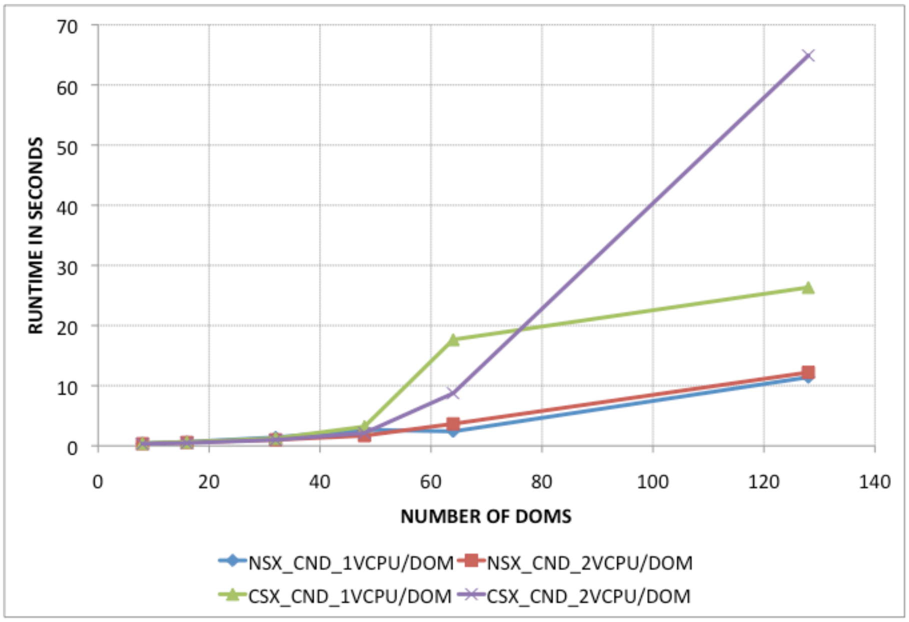


Figure 34 CSX (lower-tick size) and NSX runtime plots for CND benchmark

The DOM0 VCPUs are maintained at the minimum of all VCPU-LVTs in between synchronizations and this contributes significantly in the reduction of time-order errors in CND, where time-ordering only depends on the sent-order. From the CND error plot it is clear that except for the scenario with 128 VMs sufficient PCPU time for all the VCPUs is provided (despite being continuously subjugated by the 24 VCPUs of DOM0) before the VCPUs time-order priorities are flattened due to periodic synchronization.

3.4.1.3 VND Test Results with 64 VMs

In this test, the message generation-order, in addition to the network transit latency, plays an important role in the receive-order of the messages at the LRP. The test results for scenarios with number of DOMs involved in the tests ranging from 8 to 64 are shown in Figure 35. Similar to the CND results, the CSX scheduler in the VND tests perform better when the DOM resources fall closer to the physical resource limit but shoots up abruptly once it has been exceeded. Further, the runs carried out with 64 DOMs using CSX were very unstable, and only a few of them ran to successful completion. The error readings plotted for VND in 1-VCPU/DOM scenario are from averaging 20 runs, while all other runs are from averaging 50 runs. The VND with 2-VCPUs/DOM using CSX failed to execute to completion.

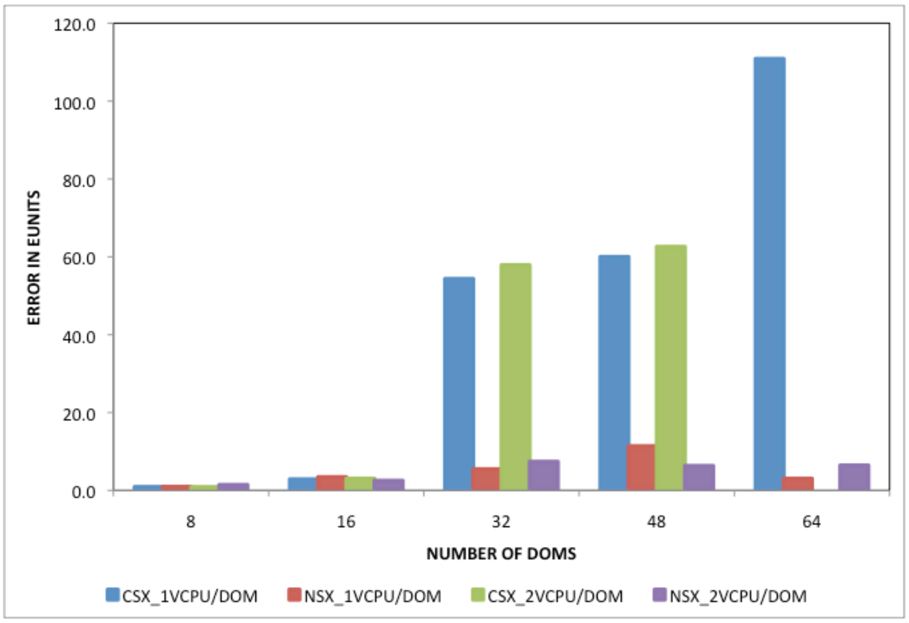


Figure 35 CSX and NSX time-order errors for VND benchmark

The NSX runs deliver almost negligible time-order error across all the test scenarios with 8 to 64 DOMs. The error value is just 3 *e*units with NSX as opposed to 110 *e*units with CSX in 1VCPU/DOM runs. Further, NSX incurs just 6 *e*units, while CSX simply fails to complete a single run in 2VCPU/DOM scenario, because of the gross mismatch between fair scheduling and time-ordered execution in this benchmark. For both 1-VCPU/DOM and 2-VCPU/DOM test scenarios, the time-order errors remain consistently smaller even as the number of DOMs in the test increases.

The runtime chart in Figure 36 shows NSX performing better than CSX especially in the tests with higher number of VCPUs. One of the reasons why NSX runs faster than CSX could be that CSX using credit based scheduling needs to update or re-sort the PCPU queues often, which is not required by the NSX. Hence, as the number of VCPUs increases, the efficiency of the CSX scheduler is reduced.

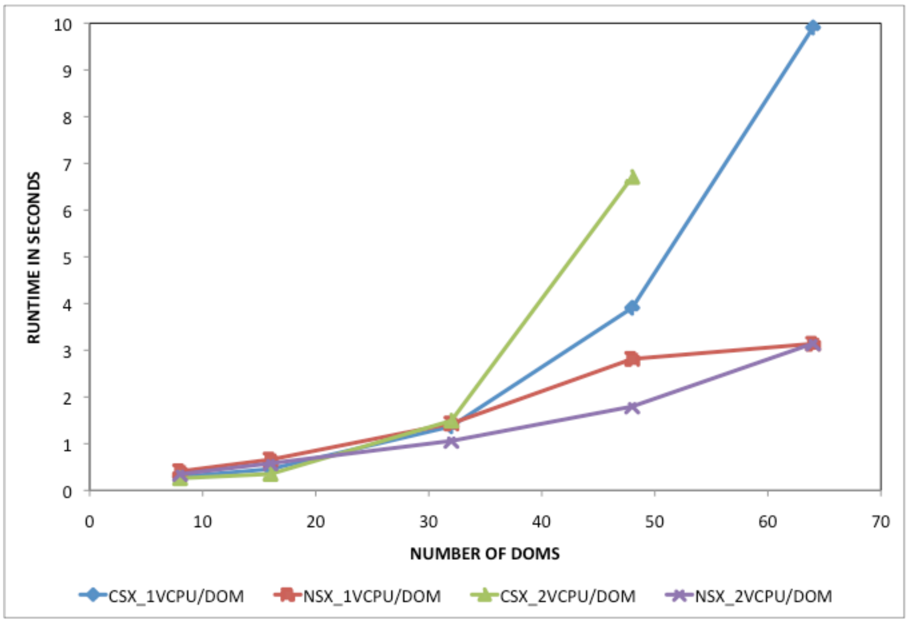


Figure 36 CSX and NSX runtimes for VND benchmark

3.4.1.4 VND Test Results with 128 VMs

In the VND tests, the NSX scheduler for 2VCPU/DOM scenario perfectly keeps both the time-order error rates and the run times low, even as the number of DOMs in the test scenario increases as shown in Figure 37. On the other hand, NSX for 1 VCPU/DOM varies quite a bit in controlling time-order errors as the number of DOMs in test scenario increases, worsening when the number of DOMs increases to 128. However, it has the best run time performance among all the other runs. The CSX with smaller tick-size performs better than its default setting performance in Figure 35 in terms of controlling errors, however the runtime suffers in both 1 VCPU/DOM and 2 VCPU/DOM scenarios as seen in Figure 38.

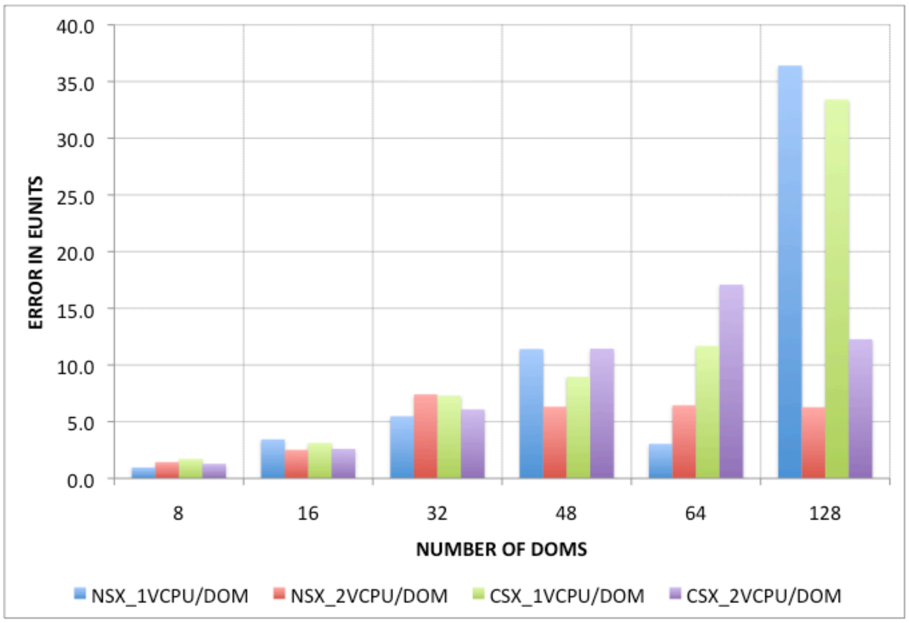


Figure 37 CSX (with lower tick size) and NSX time-order errors for VND benchmark

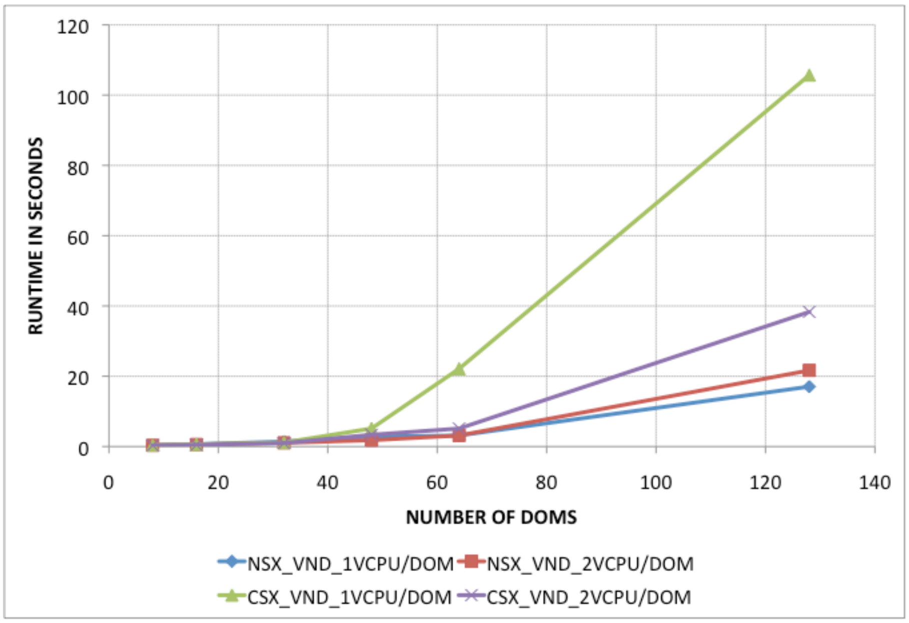


Figure 38 CSX (lower-tick size) and NSX VND runtime plots

Here too, the availability of PCPU time for all VCPUs in between synchronization can be reasoned for the relatively widely varying NSX time-order errors across the scenarios. The relatively consistent lower time-order errors in NSX with 2VCPU/DOM wherein the interval between two synchronizations is almost twice compared to 1VCPU/DOM for the same benchmark serves as strong evidence.

3.4.2 Cyber Security Simulation Results without NNC

Figure 39 and Figure 40 show the results from the runs of WP test with CSX and NSX, respectively.

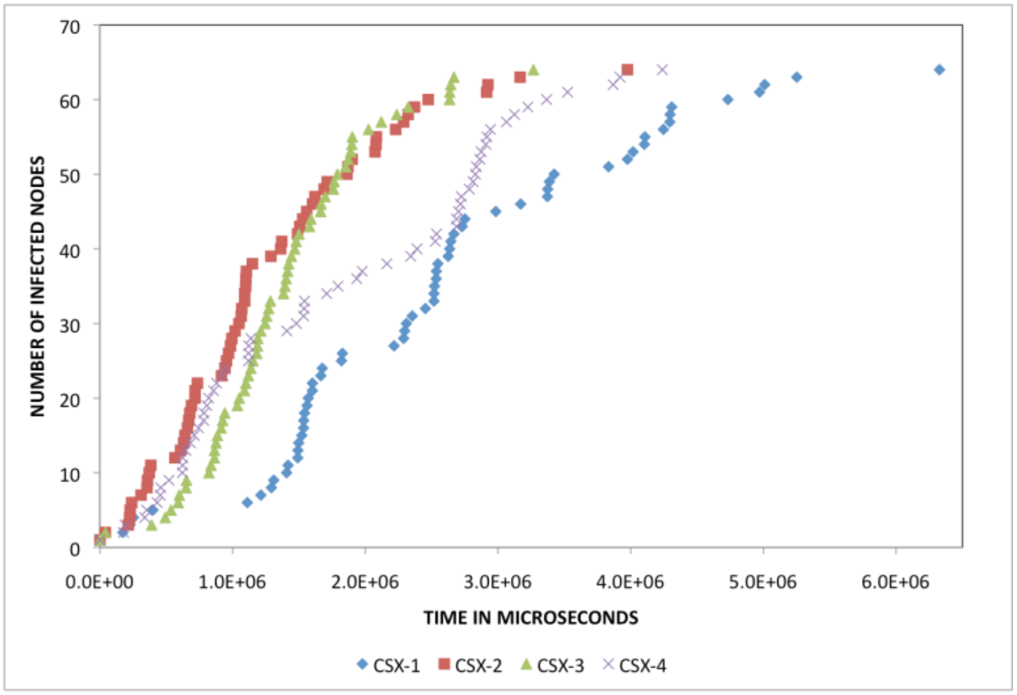


Figure 39 Worm propagation curves with CSX

These test results clearly highlight the importance of using time-ordered scheduling. Out of ten consecutive runs with CSX, only four succeeded in completing the WP test and of the successful ones, none achieved clean termination. The reason for failures is the mismatch between fairness-based (or utilization-based) scheduling and the actual need for correct, time-based advances across VMs. Since resources are heavily shared (64 VCPUs mapped on 12 CPUs), every incorrect choice in the scheduling decision incurs a stiff runtime penalty. Since the worm propagation phenomenon takes total activity that is quadratic in the number of nodes, the penalty of poor scheduling decisions increases sharply with the number of nodes. On the other hand all NSX runs were successful and achieved clean termination, without exception.

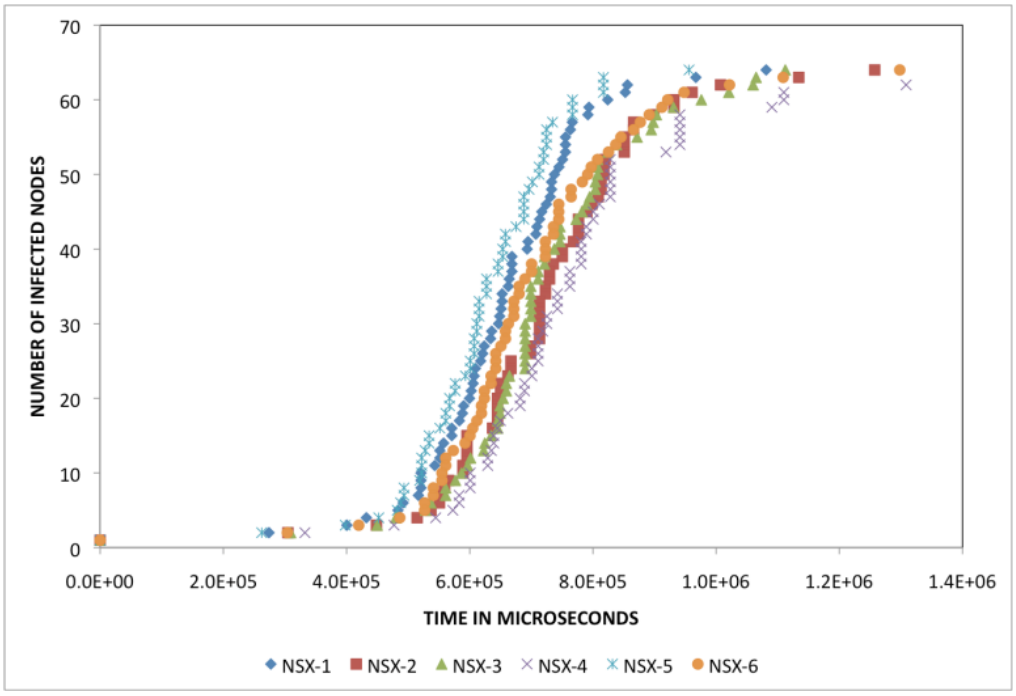


Figure 40 Worm propagation curves with NSX

It is well known from the classical simple epidemic model that the self-replicating and propagation behavior of worms without recovery or the death of infected entity, the spread of infection is a sigmoid curve very similar to the one observed in Figure 40. The consistent expression of this behavior in a controlled WP test across several runs of NSX (Figure 40) and its absence in the CSX (Figure 39) based simulation host environments strongly supports the necessity of time-ordered execution in the VM based network simulations.

Figure 41 shows the curves from CSX and NSX of Figure 39and Figure 40 in a single chart, allowing us to compare the behaviors directly. It is clear from Figure 41 that the CSX curves are widely varying, and a poor representation of the phenomenon, while the NSX curves show excellent simulation support with highly repeatable curves.

Note that the slight variability among multiple runs of NSX-based simulations is within the margin of time slice of 100 μs. Similar to the previous MPI benchmarks, this variability can be reduced as desired, by reducing the time slice further.

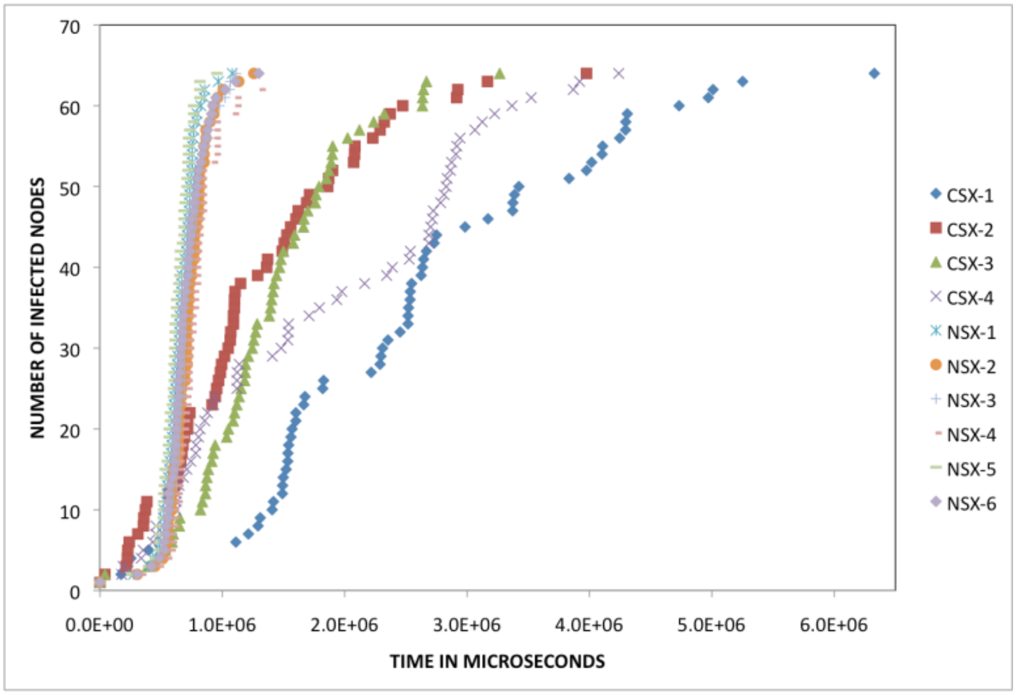


Figure 41 Worm Propagation behavior from multiple runs of CSX and NSX

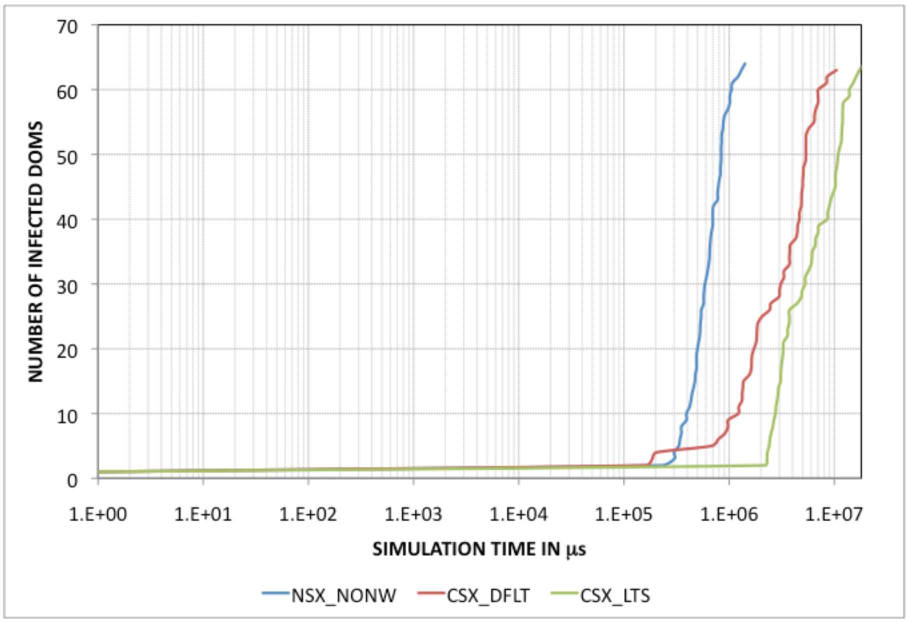


Figure 42 Worm propagation plots for different CSX configurations, and NSX without network control

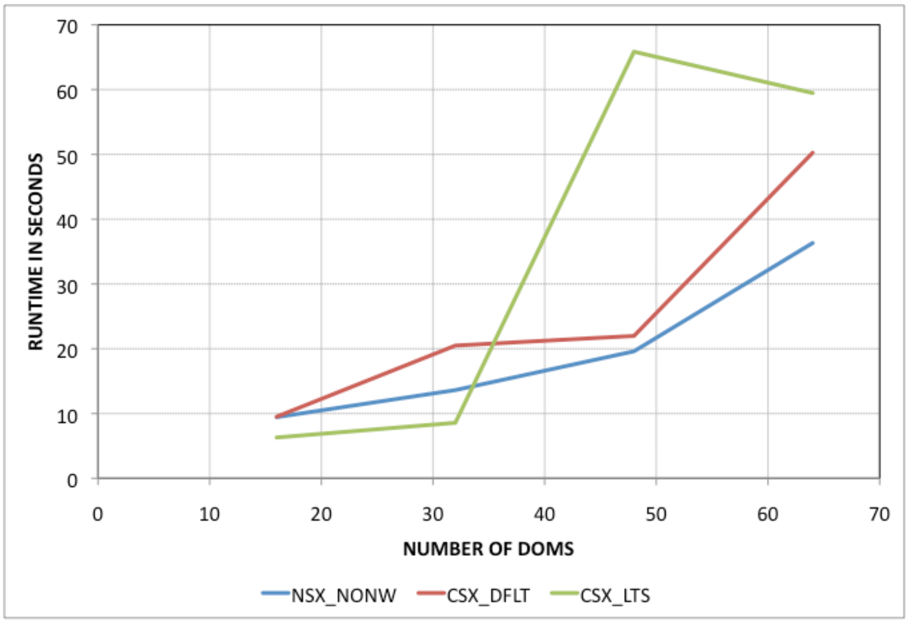


Figure 43 Runtime curves from cyber security benchmarks for both CSX configurations and NSX without network control

The plots in Figure 42 show the infection propagation curve using the cyber security benchmark in a 64-DOM scenario. The CSX_DFLT and CSX_LTS refers to CSX with default setup, and CSX with lower tick size, respectively. The NSX_NONW refers to NSX using STS synchronization without network control. Note that CSX with different tick-sizes provides different infection profiles, because CSX uses the wall-clock time as the simulation time, and hence varies based on the run-time of the simulation. In the runtime plots shown in Figure 43, for 64-DOM scenario, the run time of CSX_LTS is greater than that of CSX_DFLT and hence the infection curve is laterally shifted in the log plot shown in Figure 42.

This comparison reinforces the finding that the wall-clock time is not reliable, especially when large numbers of DOMs are multiplexed on limited physical resource, and hence, emulation methods fail at scale. Even if virtual time is tracked using an alternative method as opposed to the NetWarp method of maintaining a virtual clock per VCPU, the emulation/simulation methodologies using CSX_DFLT or CSX_LTS suffer in run time as seen in Figure 43, especially in with large number of VMs.

3.4.3 Cyber Security Simulation Results with NNC

The curves in Figure 44 show the spread of infection across the connected nodes. In the *NNC_1ms_DELAY/PKT, NNC_10ms_DELAY/PKT* and *NNC_100ms_DELAY/Pkt* scenarios, the NNC subsystem introduces a virtual-time delay of *1ms*, *10ms* and *100ms*, respectively, on every in-transit packet in the virtual network. The lateral shift of the curves to the right with the increase in enforced delay demonstrates that the NNC subsystem is appropriately enforcing the specified delays.

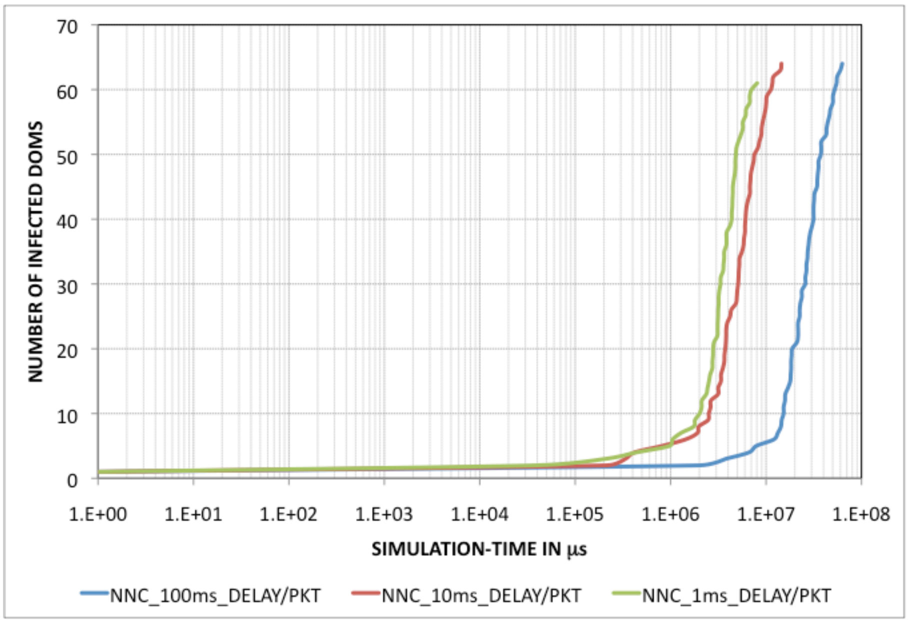


Figure 44 NNC verification results with cyber-security application benchmark

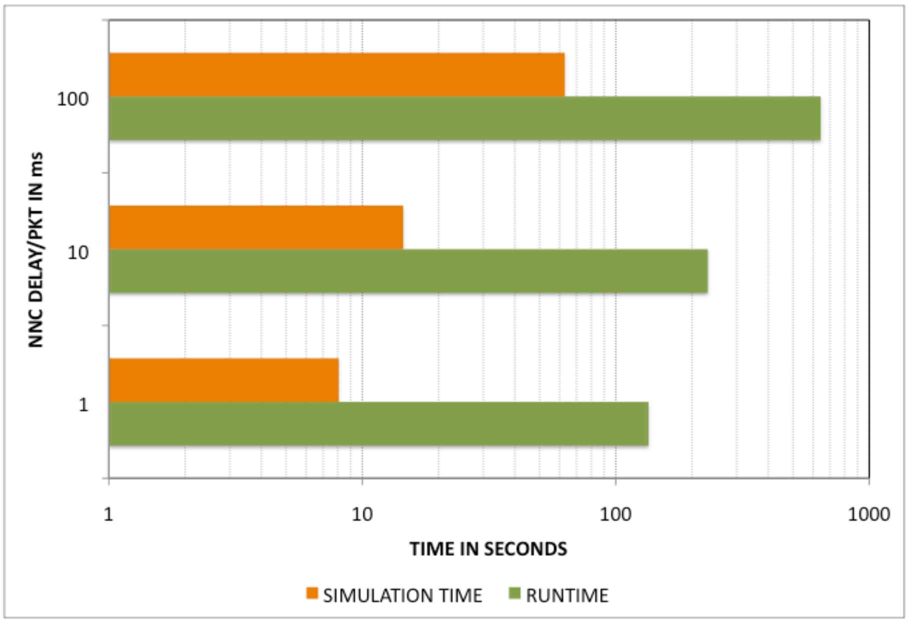


Figure 45 NNC runtime performance results with cyber-security application benchmark

In Figure 45, we plot the *simulation time* and the *runtime*, and these plots compare the wall-clock time required to simulate a distributed computing cyber security application scenario involving 64 DOMs on a 12-core machine, whose *runtime* on 64 independent nodes (ignoring the simulation overhead) would at least be equal to *simulation time*.

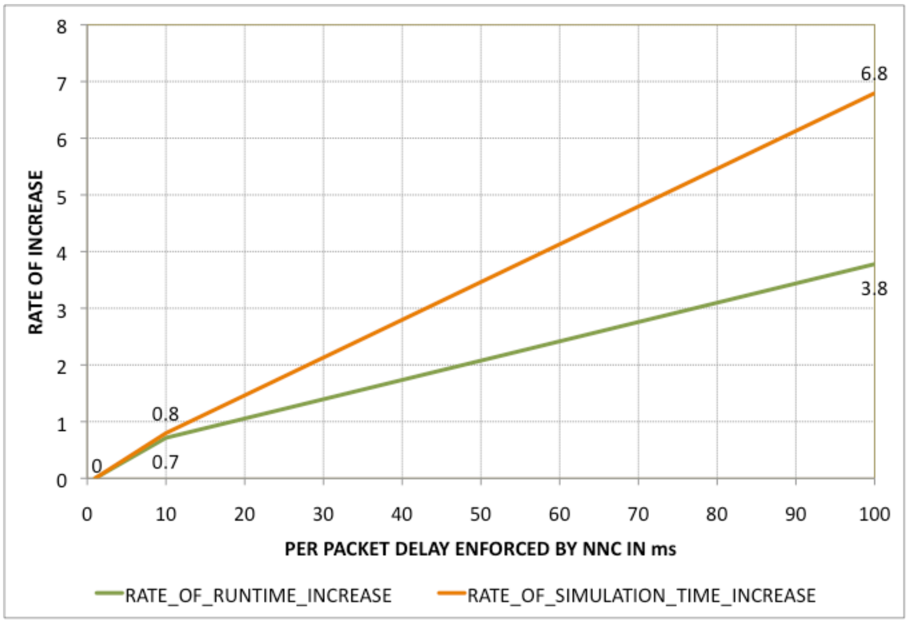


Figure 46 Rate of increase in *runtime* and *simulation time* with increase in virtual packet delay

In Figure 46, we compare the rate at which the *run time* and the *simulation time* increase with respect to their minimum values. In this case, their minimum values correspond to 1ms of virtual delay/packet enforced by NNC. The proportion by which the *runtime* increases is seen to be lesser than that by which the *simulation time* increases. This is not possible in time-stepped simulation or emulation approaches, in which the *simulation time* advances (due to VCPU accounting) in regular time steps. The reduction in the *runtime* increase rate can thus be attributed to the discrete-event nature of operation of NNC.

3.5 Summary

Starting with a virtual time execution prototype of Chapter 2, we identified and addressed the problem of staggering virtual time. We redesigned, discussed the design alternatives and implementation specifics of the parallel computing virtual time-ordered network control NNC. We designed the synthetic benchmarks and defined error metric to quantify the virtual time order errors. We also designed real-life cyber security application modeling the worm propagation in the computer networks. A detailed performance and scalability studies involving synthetic benchmarks was carried out with simulation scenarios using up to 128 VMs. The cyber security benchmark using 64 VMs to demonstrate effect of virtual time ordered execution on real life applications and the ability of NetWarp to scale was demonstrated using this real life application. The cyber security application was also used to demonstrate the correct working of NNC and NNC's capability to run faster than time-stepped simulations by making intermediate leaps in virtual time. Portions of this Chapter were published in [51] and [52].

CHAPTER 4

VM WITHIN VTS: APPLICATION CASE STUDIES

The versatility of a NetWarp simulator lies in its ability to adapt for new simulation applications to be modeled. In this section we present a case study, where in NetWarp simulation system is adapted to simulate the behavior of a complicated and extremely dynamic system of Mobile Ad-hoc NETworks (MANET).

4.1 Background and Related Work

In MANET, the mobile nodes form and break communication networks with their peers as they move, and they do not need or utilize any infrastructural support to perform such actions. The challenges involved in the simulation of a complex system such as MANET can be found in [53]. In [54], the strengths and weaknesses of several MANET simulators differentiated based on the simulation granularity are discussed. The application of MANET simulations in defense sector can be found, in [55]. In this case study our focus is on realizing a high-fidelity MANET simulation on NetWarp, and on demonstrating the correctness of the MANET simulation results.

4.1.1 MANET Emulation

Ability to simulate/emulate large-scale MANET networks is of greater interest to the military. Extensible Mobile Ad-hoc Network Emulator (EMANE) from DRS Cengen [56] is used to realize emulations. EMANE contains a number of wireless *network emulation modules* (NEM) that model the *physical* (PHY) and layer2 or MAC layers of the network stack. For high-fidelity emulation purposes EMANE applications are hosted on VMs, and to maintain the real time as simulation time, the number of VMs (each with single VCPU) hosted on the physical host is matched to the number of PCPUs of the host. The connectivity and routing tables are computed at runtime using Optimized Link State Routing (OLSR) [57]. The OLSR daemon (olsrd) [58], is hosted on each VM to dynamically compute the routing in real time, as the routing paths need to vary dynamically with the movement of MANET nodes and the geographical landscape. The EMANE application highjacks the communication packets from VM network interfaces and passes them through the physical and MAC layers to enforce the mobility characteristics. However, to enforce emulation network conditions the path loss information is periodically communicated to the relevant EMANE modules.

With this setup a wide range of distributed applications can be tested as EMANE emulators are hosted on VMs that hosts a fully-blown operating system. Since, the EMANE uses real-time as the simulation time, the issue of correctness of the emulation runs arises. This is because, in addition to EMANE objects there are many independent components that are communicating in real-time, which are artificial instrumentations that do not exist in realistic scenario and their asynchronous runtime is inadvertently accounted in the emulation. Such artificial instrumentations will definitely affect the correctness of the emulation predictions. Note that these concerns are in addition to the other correctness issues of VM-based cyber simulations that have been elaborately discussed in the previous Chapters.

4.1.2 NetWarp Simulation of MANET

By porting MANET application scenario on to the NetWarp simulator, the simulation time reliance on the real time as in case of EMANE based emulations is removed. This enables NetWarp to host far more VMs on a single machine without compromising on the correctness of the simulation.

An initial experiment was conducted to compare a MANET scenario involving voice-over-IP (VOIP) application on EMANE and NetWarp setups. The VoIP application used Signal Initiation Protocol (SIP) [59] for initiation and termination of calls. The RTP [60] protocol was used to transmit the actual audio packets. A recorded voice message was transmitted between the nodes over VOIP. The out-of-box PJSIP [61] application (that implemented SIP and RTP protocols) was used for interactions. The changes in the network characteristics and the quality of the VOIP messages with the increase in number of VOIP sessions were observed. This EMANE based emulation, which did not involve mobile nodes were executed on a dedicated cluster to obtain results. This static MANET experimental scenario with the EMANE hosted VMs representing MANET nodes that were interconnected through ad-hoc networking form a network graph, as shown in Figure 47.

This exact scenario was simplified to execute on our generic VM test platform without EMANE. In our simplified model we directly used the virtual Ethernet devices of the VMs instead of EMANE NEMs. The RF propagation path loss for the static MANET scenario was read from file. This data along with the cutoff value of -94 dB was used to generate the static network connectivity graph of MANET nodes. Using this connectivity graph information, the *olsrd* executing on each VM created the same multi-hop network as shown in Figure 47 on our generic VM test platform. The exact details of utilizing the connectivity graph to setup a multi-hop MANET network of VMs is discussed in the later sections. In our experiments, we compared the EMANE results with the results from our generic VM test platform that also exercised different hypervisor schedulers (CSX and NSX).

For experimenting, the VoIP calls were initiated between two of the ad-hoc wireless nodes (the ones numbered 57 and 49, in this case). Node 57 sent a recorded message (*wmv* file) and 49 responded similarly. Once the files were transferred the test ended. The number of simultaneous calls between the sender and receiver was varied (1 to 64) across runs.

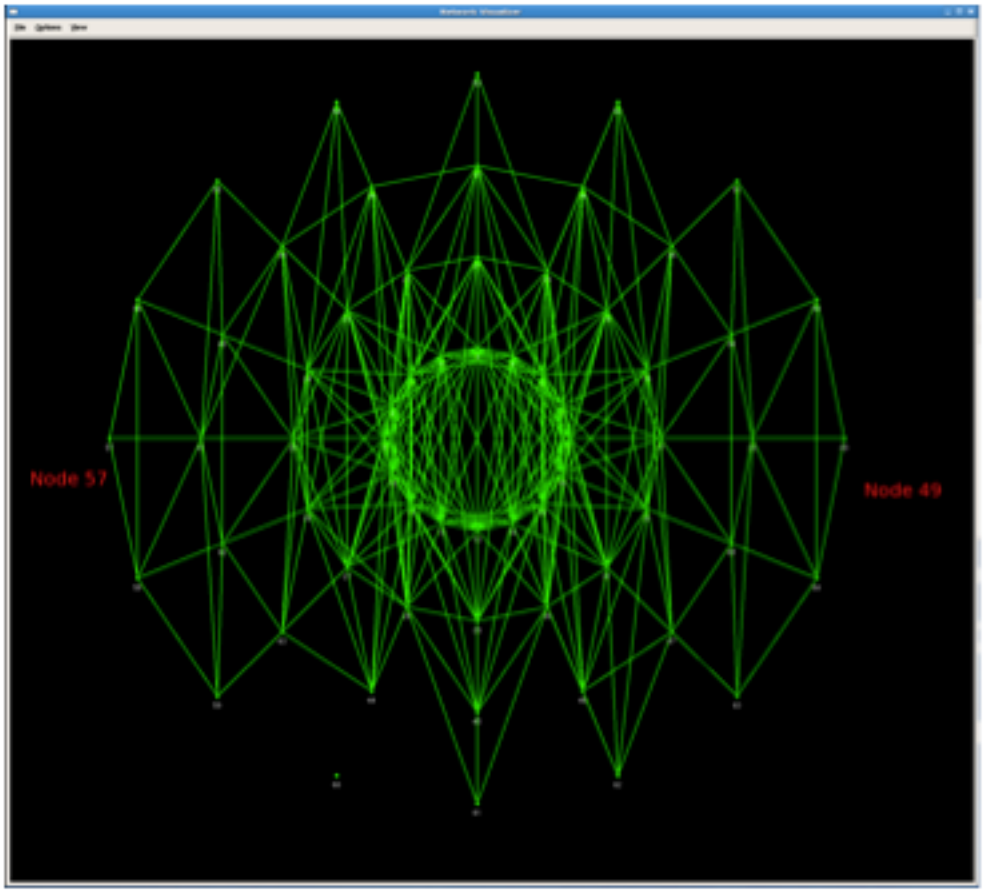
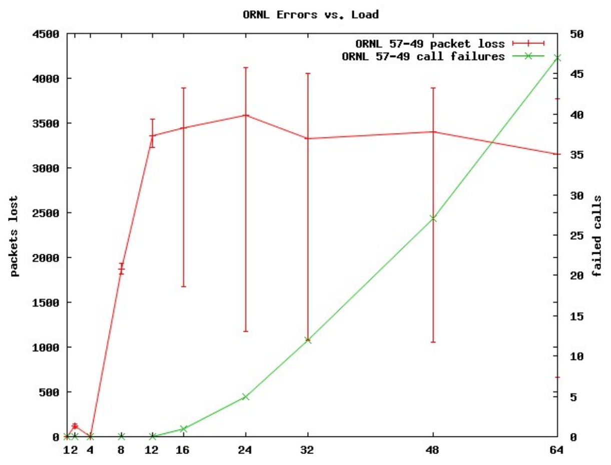
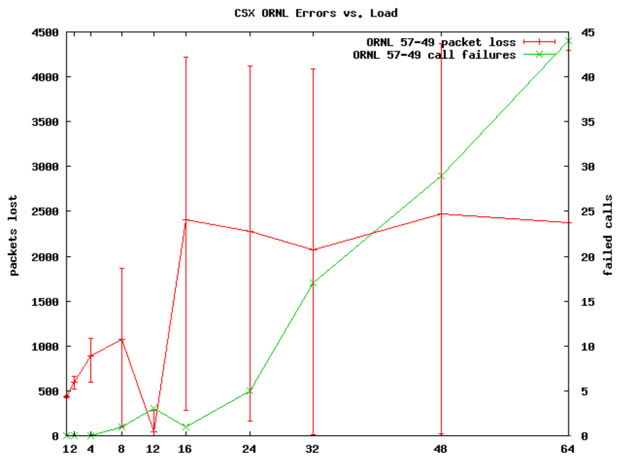


Figure 47 Plot of one-hop neighbors for the MANET topology

(a) 

(b) 

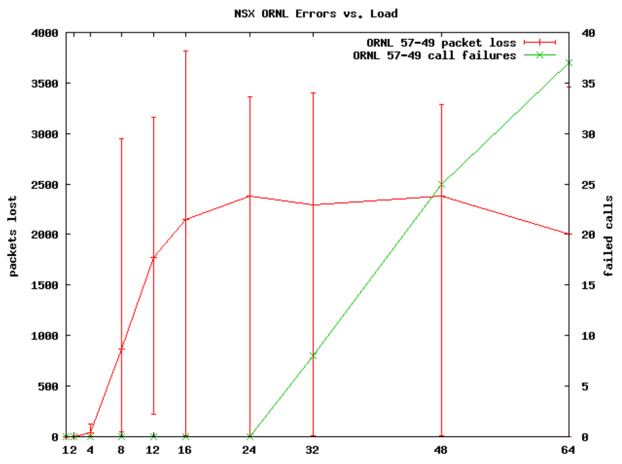
(c) 

Figure 48 Call failures and packet-loss vs. the number of calls in (a) Dedicated cluster (b) CSX (c) NSX

Figure 48 shows the packet loss and call failures on same of tests executed on the (a) dedicated cluster using EMANE (b) CSX setup on a single 12-core MacPro running Xen capable of hosting 64 VMs (c) NSX setup on the same hardware as in b. On comparison of the curves from the three plots in Figure 48, we found that plots using NSX setup was more similar to the results from the runs from EMANE emulation with dedicated hardware. Comparatively, the runs from CSX setup show a greatly different packet-loss curve with increase in the number of simultaneous calls. However, the call failures curve more or less look similar in all the three plots. These results were published in [51] and this methodology served as a preamble for a detailed case-study.

4.2 Instrumentation Specifics

To simulate MANET experimental scenarios on generic VM platforms without using emulation software like EMANE, the following requirements should be met.

1. Virtual time-controlled execution of VMs
2. Virtual time-controlled communication of messages among VMs (latency)
3. Support for ad-hoc network setup and operation
4. Virtual network bandwidth control
5. Virtual mobility support
6. Benchmark applications

The first two requirements are generic and their significance in determining the correctness of the simulation results has been elaborately discussed in the last two Chapters. The remaining four requirements are specific to the MANET simulations on NetWarp.

Unlike wired computer networks, the MANET deals with mobile nodes that interact with each other using ad-hoc networking protocols and this results in constant change in their connectivity. OLSR was used for this purpose.

In MANET, the virtual network bandwidth varies based on the distance between the mobile nodes and the surrounding environment. To enforce bandwidth specification in the MANET scenario, we require the VMs utilized in MANET simulations to communicate the bandwidth or throughput information to their virtual network interfaces, and to ensure that they are enforced during simulation.

Simulating the mobility feature is one of the basic requirements of MANET because by definition the interconnected nodes are mobile. This requires the VMs utilized in MANET simulations to be able to dynamically tear down existing network connections and form new network connections during the course of simulation, based on a specified mobility pattern.

Benchmark applications are important because it is through these applications, the correctness of the complex MANET simulations are evaluated. In the previous Chapters we devised several benchmark applications to measure correctness and runtime. The CND and the cyber security application benchmarks are used for evaluating MANET simulations.

4.2.1 Ad-hoc Network Setup and Operation

Installation and execution of *Olsrd* on all VMs representing a mobile node in the MANET simulation, results in single hop connectivity among all the VMs. In order to enforce realize the characteristic connectivity pattern of the MANET simulation scenario, an input file providing the RF bandwidth in between every pair of MANET nodes was used. A cutoff bandwidth was used to determine the existence and absence of connectivity between any pair of MANET nodes.

After obtaining this connectivity graph as shown in Figure 49, the setup of a multi-hop network in NetWarp involved two tasks, (a) communicating the connectivity information to VMs and (b) enforcing connectivity.

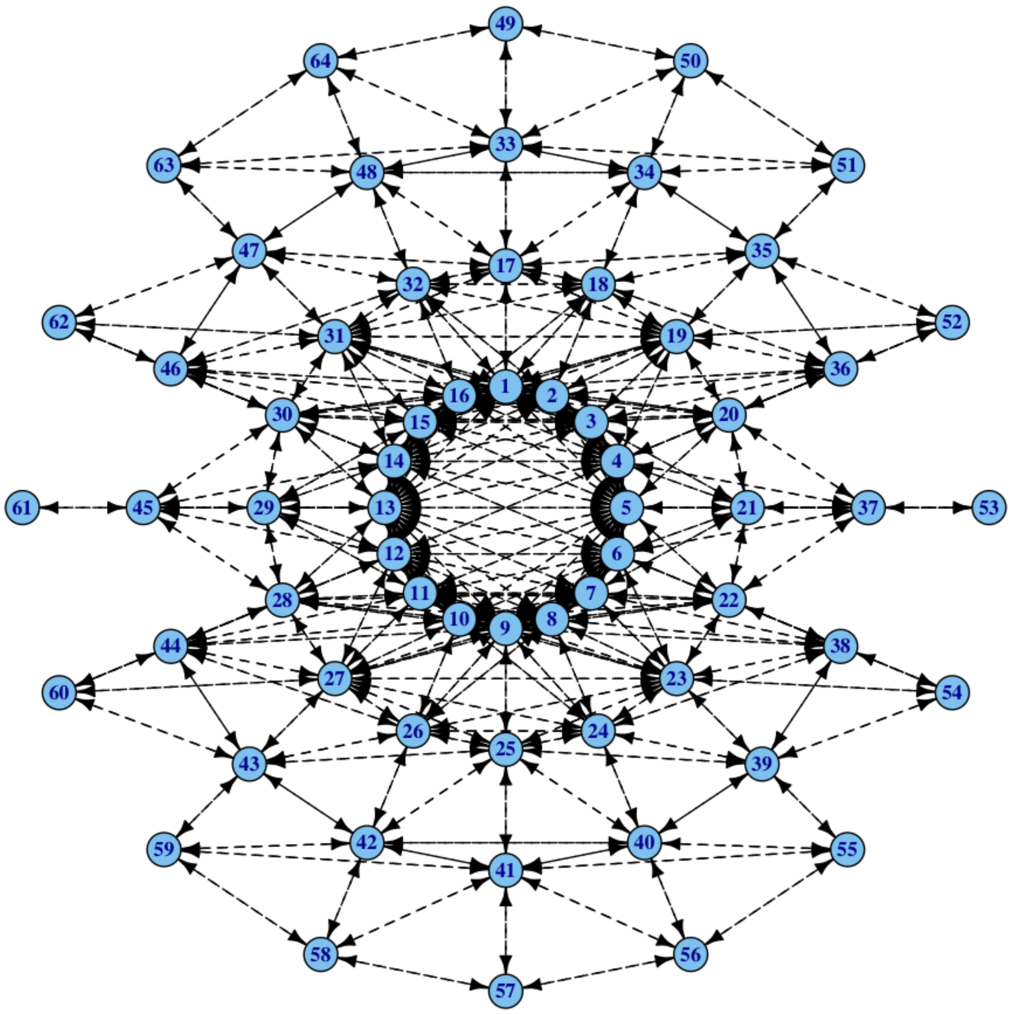


Figure 49 Network connectivity graph of MANET simulation scenario

To communicate the connectivity information to the VMs, a bit (0 or 1) representing the existence of connectivity between all pairs of MANET nodes was written in *Xenstore*. This data in the *Xenstore* is visible to all hosted VMs and, hence can be read by all the VMs that represent the MANET nodes.

To enforce specified connectivity among the virtual network interfaces, every VM on boot read the *Xenstore*, and executed necessary *iptables* rules specifying either to *accept* or *drop* the packets originating from the peer VM network interfaces. The connectivity graph shown in Figure 49 is actually constructed from the next-hop route information collected from all 64 VMs involved in the MANET simulation scenario. Except for very few modifications that were made to suit our study, this connectivity graph is same as the MANET graph shown in Figure 47.

4.2.2 Network Control

4.2.2.1 Network Bandwidth Control

If connectivity existed between two VMs, then dropping packets from a peer that exceeded specified arrival-rate threshold ensured bandwidth control. This threshold was enforced by the VMs using *iptables* rules with *limit* module. If no connectivity existed between a peer-node then the packets from that peer were just dropped. Thus, the VMs running as MANET nodes enforced these *iptables* rules based on the connectivity information read from the *Xenstore*. Figure 50, lists the necessary *iptables* rules that were used for this purpose. Each VM enforces a definite rule for every other peer-node involved in the simulation.

```
#iptables rule to ensure no connectivity with mac address XXX
iptables -A INPUT -m mac --mac-source XXX  -j DROP

#iptables rule for bandwidth controlled connectivity with mac
address XXX
iptables -I INPUT 1 -m limit --limit PKT_PER_SEC/sec -m mac --mac-
source XXX -j ACCEPT
iptables -I INPUT 2 -m mac --mac-source  XXX -j DROP
```

Figure 50 *iptables* rules enforcing non-connectivity and bandwidth

4.2.2.2 Network Latency Control

The Netwarp Network Control (NNC) discussed in the last Chapter is used to enforce delay on packets transiting from one MANET node to another. The NNC executing in DOM0 utilizes *iptables* rules to trap the inter-VM packets and redirect them to NNC, as done previously. Additionally, in case of ad-hoc networks the dynamic connectivity of the mobile nodes are made possible due to periodic exchange of broadcast packets (hello packets) between the nodes as per OLSR protocol. For performance reasons we did not redirect the broadcast packets to NNC and hence, only the application specific packets passed through NNC. Figure 51, lists the set of *iptables* rules enforced in DOM0 for this purpose.

```
#iptables rule deleting the default rule of Xen for DOM0
iptables -D FORWARD -m physdev --physdev-in vifXXX –j ACCEPT

#iptables adding rule to accept the broadcast packets
iptables -A FORWARD -m physdev --physdev-in vifXXX -j –d
192.168.X.255 -j ACCEPT

#iptables rule to redirect the other non-broadcast packets to NNC
iptables –t mangle –A FORWARD -m physdev --physdev-in vifXXX -j -d
!192.168.X.255 -j NFQUEUE --queue-num 0
```

Figure 51 *iptables* rules to aid NNC functioning

4.2.3 Virtual Mobility Support

With the *iptables* rules for connectivity and latency control of the inter-VM packets as shown in Figure 50 and Figure 51, the setup requirements necessary for static MANET scenarios was fulfilled.

In the mobility-feature supported MANET scenario considered for performance analysis only node 61 was made mobile. As shown in Figure 52, the node 61 virtually revolved around other static MANET nodes in anti-clockwise direction forming and tearing the connection with the peripheral nodes in regular virtual time intervals. Note that at any instance of simulation time the node 61 was only connected to only one peer-node.

To support mobility feature, we developed an additional mobility module executing in DOM0 that refreshed connectivity (based on physical positions) information of MANET nodes at periodic virtual time intervals. To simulate the revolving behavior of node 61 in NetWarp, the *Xenstore* was periodically updated with connectivity information (at every t_1 sec), and the VMs representing the MANET nodes read the *Xenstore* periodically (at every t_2 sec) and updated their *iptables* rules to reflect the MANET connectivity at any instance of the virtual time. For this to work correctly, we ensured $t_1 >> t_2$.

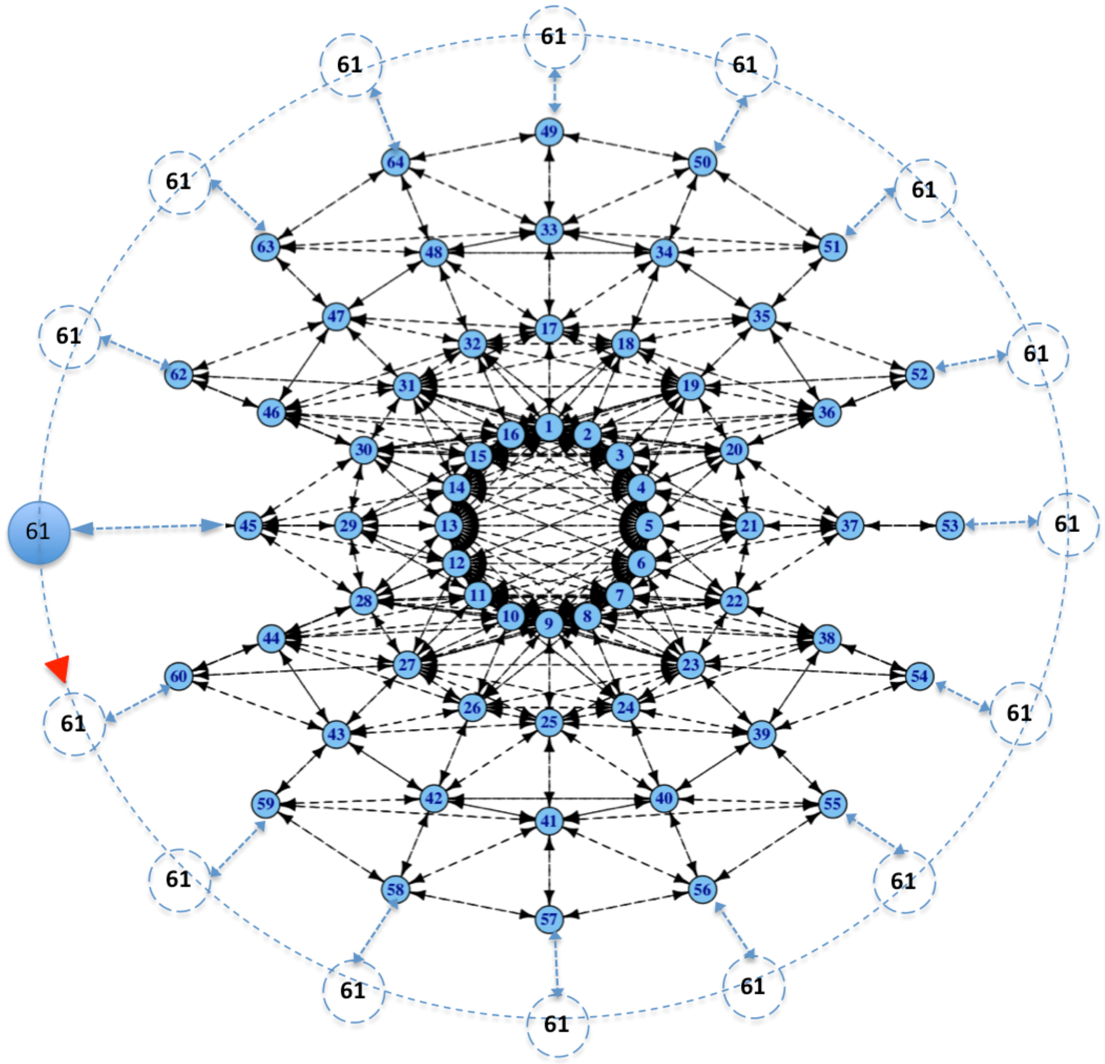


Figure 52 MANET scenario with mobility support

4.3 Benchmark Applications

4.3.1 CND Benchmark

We used the CND benchmark discussed in Section 3.3.3.1 for evaluating the correctness of MANET simulator functioning on NetWarp. It can be recalled that in CND the highest rank sends sequence of messages to the lowest rank, each of these messages before reaching their destination made an intermediate hop. If the lowest rank were 0, the first message would hop on rank1 process, the second message would hop on rank2 process and so on. This benchmark verified the correctness and measured the errors in *eunits* using the message receive-order and the message sent-order. For perfect correctness both sent and receive orders were same.

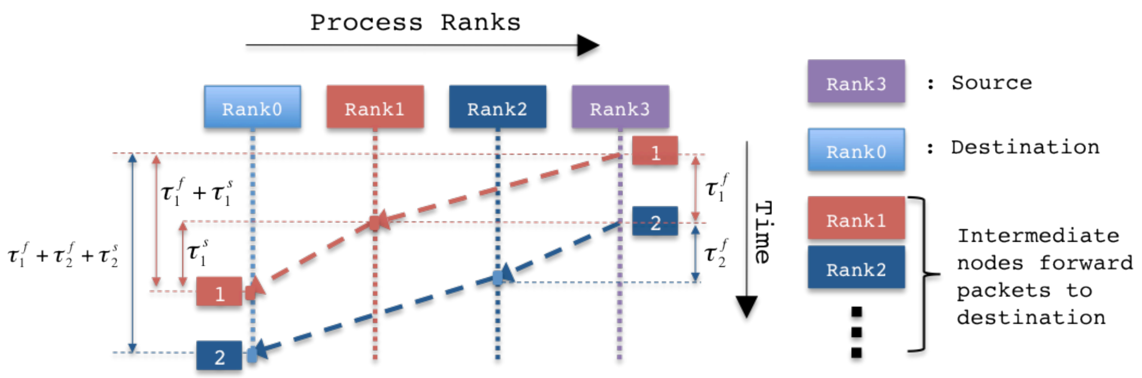


Figure 53 CND functional diagram

Note that CND was designed under the notion that every message made exactly one hop (physical and logical) before reaching the destination. However, this notion was invalid in the MANET scenario because, a single logical hop of a message at the application level is not always equal to number of physical hops taken by the message in a MANET network. The number of physical hops is actually dependent on the shortest distance between the source and destination nodes. For example, even though the first message passing from node64 (rank 63) to node1 (rank 0) logically makes a hop at node2 (rank 1), the message actually makes 3 physical hops to reach node2 from node64 and makes a single hop from node2 to node1. In this particular example the message actually takes 4 physical hops to fulfill one logical hop. Hence to evaluate the correctness, we first determined the correct receive-order of messages at the lowest rank for the given MANET scenario.

Note that the CND application benchmark uses blocking MPI calls in implementation. Hence, the MPI_Send would not return until the message sent were buffered at the destination host as shown in Figure 53. As previously alluded, in MANET setups a single hop can correspond to multiple physical hops. However, with the shortest-path between the nodes as a distance measure, the correct receive-order for the highly deterministic CND benchmark can be estimated. Using, the shortest-path, the relative-time of packet arrivals at the lowest-rank can also be determined.

If τ_i^f corresponds to the first logical hop taken by the *i*th message to reach the *i*th ranked process and τ_i^s corresponds to the second logical hop taken from the *i*th ranked process to the lowest rank (rank0). Then the reception time of the packet γ_i for any *i*th packet can be determined as,

$$\gamma_i = \tau_i^f + \tau_i^s$$

$$\tau_i^f = \sum_{k=1}^{i} \tau_k^f$$

$$\gamma_i = \sum_{k=1}^{i} \tau_k^f + \tau_i^s$$

$$\tau_i^f \ \alpha \ shortestPath(highestRank, i)$$

$$\tau_i^s \ \alpha \ shortestPath(i, lowestRank)$$

Note here that τ_i^f is the summation of all previous first logical hop times. This is because we used blocking MPI routine to send out the message and hence every generation of consecutive message suffers from this delay. In calculating τ_i^s, which corresponds to every i^{th} ranked process receiving and sending only one message does not incur additional overheads. The number of physical hops in the shortest-path distance required to perform one logical hop is used to determine the receive-order and to calculate the relative receive time of the arriving packets.

In addition to the receive-order, approximate receive-time and hence the receive pattern was obtained from this benchmark. The receive-order and receive-pattern of this benchmark were utilized for verification of correctness in the static MANET scenario.

4.3.2 Cyber Security Benchmark Application

The cyber security benchmark simulates the spread of a Worm from an initially infected vulnerable service to all of its connecting nodes. One major difference between this benchmark and the benchmark discussed in Section 3.3.4 is, in this benchmark the infection is spread only to the next-hop nodes.

The cyber security benchmark is a socket-based application using TCP/IP protocol and is agnostic of ad-hoc (OLSR based) network underneath. The infection-spread time, infecting nodes and the pattern of spread of worm over time are the study aspects of this benchmark. Note that the entire software-stack starting from the cyber security application to the lowest-level OLSRD were ported on to a VM without any modifications to the source to suit the simulation purpose. Only exceptions were little instrumentation that was performed for recording the virtual time and to support virtual mobility.

The hypervisor scheduler ensured the virtual time-controlled execution and the virtual time-ordered delivery of the packets were performed by the NNC daemon. Hence, and rightly so, the cyber security application was completely unaware of both virtual time control and virtual network control. To utilize this benchmark in the MANET scenario we slightly modified this benchmark.

The mobility had to be triggered at the start of simulation. To ensure this, the benchmark application needs to communicate its readiness to the mobility module. This was accomplished by the initial-infected service node, which informed the mobility module about the setup readiness through the *Xenstore.* This communication to the mobility module happens after all services on the VMs wait on a barrier before starting the simulation.

In our benchmark scenario node61 was considered as the infected node for both mobile and non-mobile MANET based cyber security benchmark scenarios. Note that node61 was also the mobile node in the MANET scenario with mobility.

The infection spread in this benchmark happens in two phases, in the initial phase a worm infects the vulnerable service hosted on its *next-hop* nodes and this results in opening of the backdoor socket on the infected node. In the final phase the worm transfers payload to the already infected service. Note that it is not necessary that the same infecting node perform both initial and final phase infections. After the initial infection the worm tries thrice to connect to the backdoor and transfer the payload, in event of failure it moves on to infect the next node. If the backdoor is already open, some other worm from a different node might complete the final phase of infection with payload transfer, after which this node (a new resident for the worm) actively tries and infects its neighbors.

This benchmark was utilized to verify the mobility operation. With node61 being the infection start-point in both static and mobile MANET scenarios, one can expect node61 to infect only one node45 in static MANET scenario, while the same node61 can be expected to perform more infections in the mobile MANET scenario.

4.4 Performance Results

4.4.1 Experimental Setup

The experiments were performed on the Mac-Pro hardware with two hex-core Intel® Xeon processors at 2.66 GHz, 6.4 GT/s processor interconnect speed with 32G of memory. With hyper-threading enabled, Xen sees 24 cores. With Xen creating 24 PCPUs to handle this, all our experiments view this system as a 24-core machine. OpenSUSE 11.1 with Xen-3.3.1 and Xen-3.4.2 source code was used on this hardware.

The CSX setup was configured to use 4-VCPUs in DOM0 and each the weight of DOM0 processors were maintained 10 times more than the other VCPUs. During scheduling each VCPU was assigned a time-slice of 100µs. While, the bandwidth restriction of 300 packets/sec per peer network device was enforced to all CSX runs, no latency was introduced. The scaled wall clock time is used as the simulation time for the CSX runs. The scaling factor was determined by ratio, $\frac{number\ of\ PCPUs}{number\ of\ VCPUs} = \frac{24}{64}$. For the mobile scenarios one-second wall clock delay was used for the mobile node (node-61) to hop from one peripheral node to another as shown in Figure 52.

The NSX setup was also configured to use 4-VCPUs in DOM0 and the time-slices for DOM0 VCPUs were increased by 10 fold in comparison with other VCPUs, which used 100µs time-slice. The bandwidth restriction same as CSX was enforced in NSX setup and the NNC was used to ensure latency control. A 10ms delay was enforced on every (non-broadcast) transiting packet. The virtual time was collected through the

Xenstore interface as presented previously in Chapter 3. A virtual time equal to one second was enforced for the mobile node (node61) to hop from one peripheral node to another.

4.4.2 CND Benchmark Results

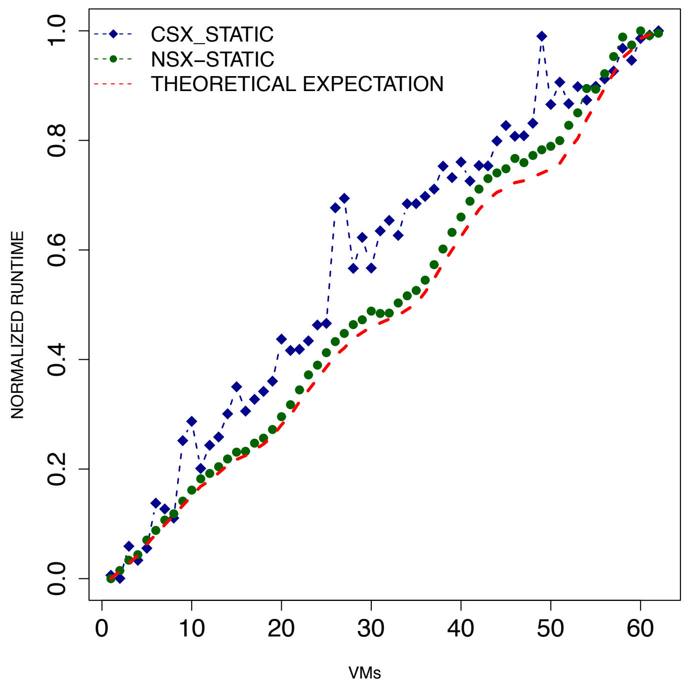


Figure 54 CSX and NSX receive-pattern comparison with theoretical expectations

Figure 54, shows the normalized arrival pattern of the packets at the VM hosting the lowest-rank MPI process. As observed, the NSX-STATIC, i.e., the NSX runs for the static MANET scenario have a very close correspondence to the theoretically derived result, and in contrast the CSX runs show irregular pattern highly differing from expectations. The errors measured from NSX and CSX runs in terms of *eunits* are presented in Figure 55, which shows highly increasing errors with the increase in VMs, while almost no errors were observed NSX runs. In Figure 56, the variability of the CSX and NSX packet-receive simulation times with 95% confidence intervals are presented. A high-variability in CSX and almost no variability in NSX runs can be observed.

While, we derived a means to theoretically determine the correct receive-order and receive-pattern of the packets at the lowest rank and in the static network setup of MANET. Similar, recognition of receive-order and receive-pattern in the mobile MANET scenario is extremely difficult as one of the nodes is mobile. Hence, we present only the runtime of CSX and NSX packet-receive simulation time with 95% confidence intervals. Figure 57 shows a very low variability in the packet-receive simulation time of NSX for internal MANET nodes, while a bit higher variability at the peripheral nodes, which actually are affected by the mobile node-61. However, the trends of simulation-time variability shows similar irregular, high variability trend as seen in the static MANET scenario in Figure 56. Though the NSX readings show less variability, it plots provide no insight on the verification of mobility feature or the correctness of the readings.

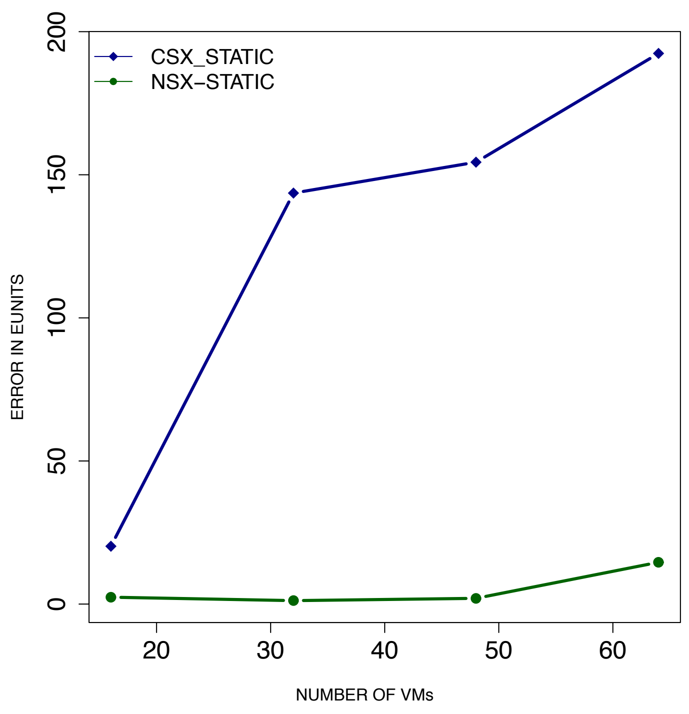


Figure 55 CSX and NSX errors in eunits

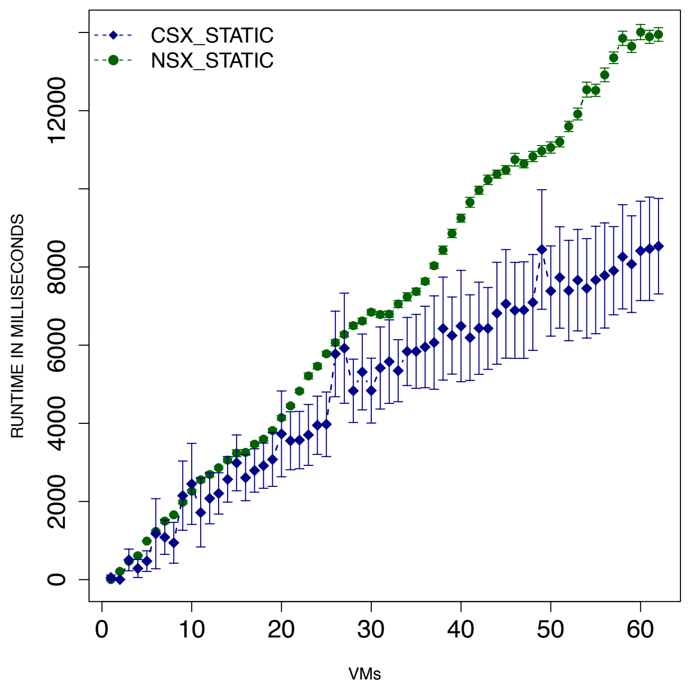


Figure 56 CSX and NSX simulation time in static MANET with 95% CI

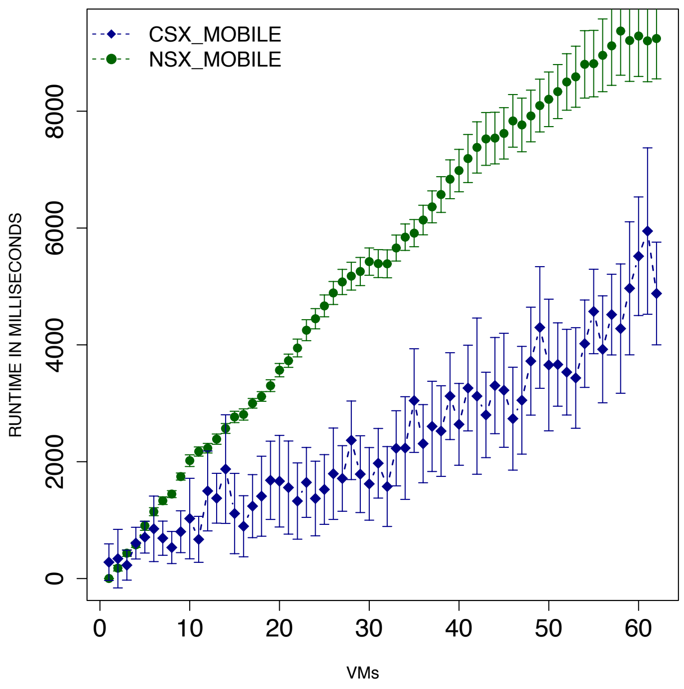


Figure 57 CSX and NSX simulation time in mobile MANET with 95% CI

4.4.3 Cyber Security Benchmark Results

With the cyber security benchmark results from NSX runs we verify the correctness of mobility feature of the MANET scenario. As mentioned in Section 4.3.2, the worm in the cyber security benchmark infects only the next-hop nodes. Hence, in the static MANET scenario one can expect the worm from the vulnerable-service at node61 infecting its only neighbor node45 and then spreading of the worm from there on, as observed in Figure 58.

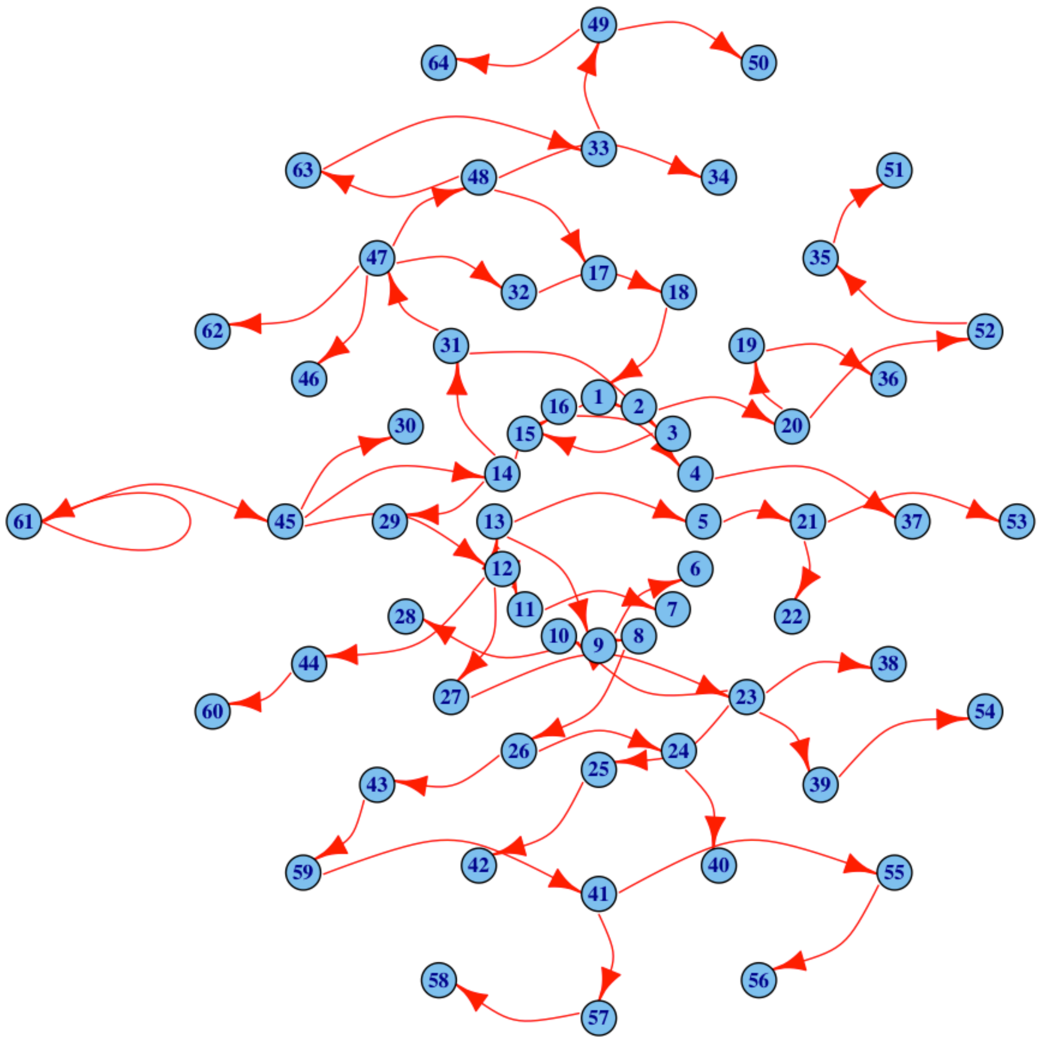


Figure 58 Worm spreading from node-61 in static MANET scenario

Similarly, in the mobile MANET scenario, where the mobile node61 revolves around the periphery of the MANET network as shown in Figure 52, one expects the node61 to infect other peripheral nodes other than node45. This is observed in Figure 59, the node61 not only infects node45 but also infects node59, node56, node53, node52, node50 and node62. This verifies the correctness of the mobility feature in mobile MANET scenario.

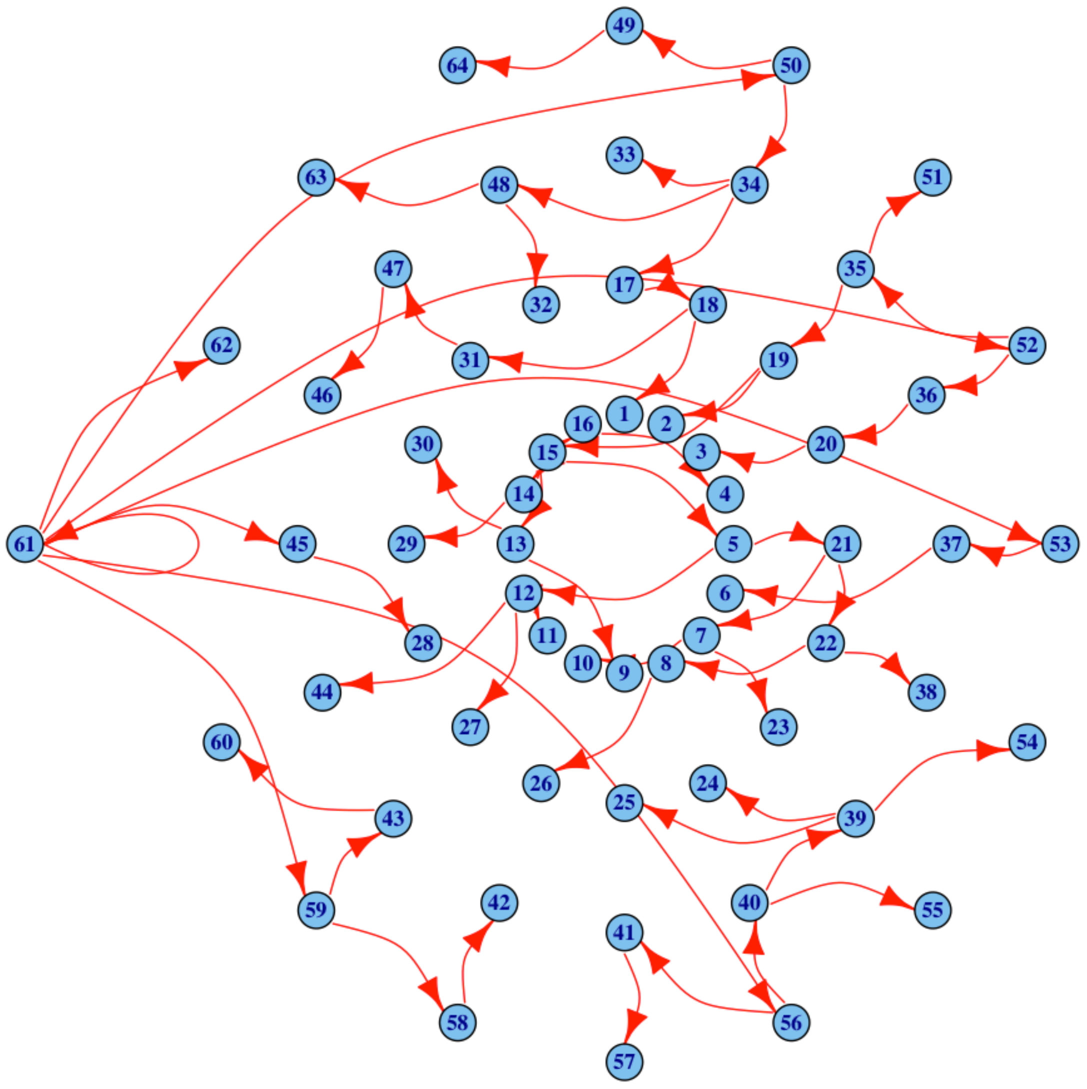


Figure 59 Worm spreading from node-61 in mobile MANET scenario

In Figure 60, we plot the virtual time of infection across number of VMs for both static and mobile MANET scenarios, with 95% confidence intervals. The curves show the trend infection spreads in static and mobile scenarios. In analyzing the curves one needs to keep in mind that that the infection spreads only through one-hop neighbors and the infection spreads in two phases, and only after the second phase (after payload transfer) the infected node actively infects others. Hence, if a MANET node has less number of neighbors the parent node aggressively try and quickly converts its neighbor into a worm propagating node.

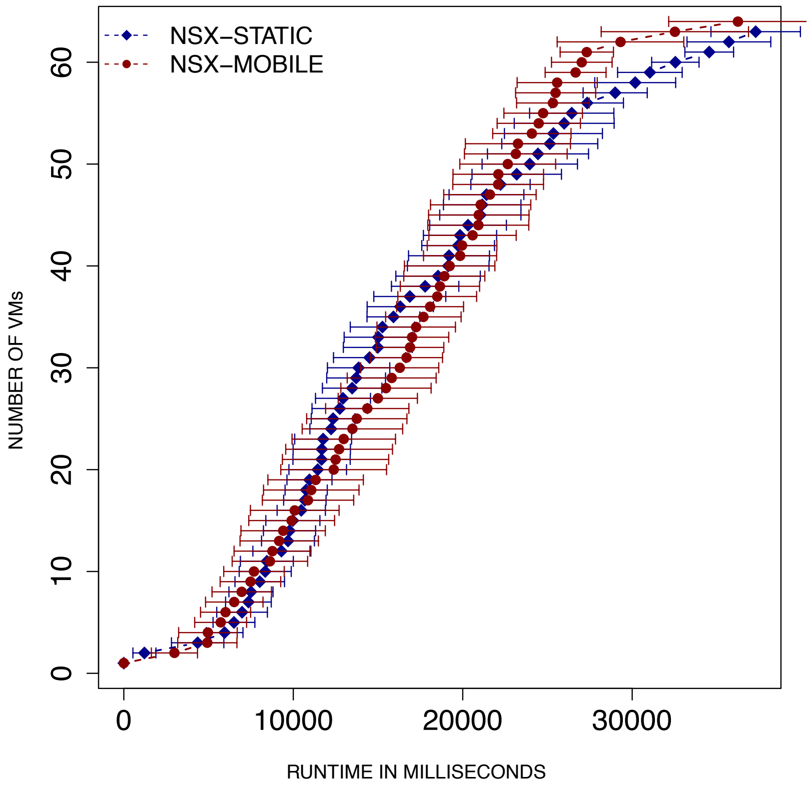


Figure 60 Worm spreading in static and mobile MANET setups

To understand the behavior of the static and mobile curves, we need to refer to all the three Figure 58, Figure 59 and Figure 60. In Figure 58, we see that the infection starts from node61 and infect all. In contrary, the Figure 59 although shows the movement of node61 it also shows that it is not able to infect all its visited neighbors successfully. Hence the static MANET seems to have a higher infection rate initially. However, even with lower number of infected nodes (seven out of 16 visited neighbors) the infection rate in the mobile scenario picks up at later stages of the simulation as shown in Figure 60. Hence, the spread of infection rate is dependent on the number of neighbors the mobile node coverts into a worm propagator.

4.5 Summary and Conclusion

In this Chapter, we discussed the application of NetWarp in MANET simulations. The details in realizing a highly complex MANET simulation application scenario in NetWarp were discussed. A VM along with its virtual network interface together formed a MANET node. On such a MANET network test-bed, desired complex applications can be put to test without additional effort of porting the test-applications to a simulation environment. We verified the correctness of MANET functioning using CND benchmark. For this purpose, we first derived the theoretical expectation of the CND behavior and compared our experimental results from CSX and NSX setups. From the results we found that NSX results to be highly accurate. Further, we verified the mobility feature using cyber security benchmark for mobile MANET scenario. We also compared and analyzed the infection spread in static and mobile MANET scenarios.

CHAPTER 5

VTS OVER VM: PERFORMANCE EVALUATION

5.1 Problem Space

5.1.1 Virtual Machines and Cloud Computing

Virtual Machine (VM) technology moves the traditional operating system (OS) away from the actual hardware interface and transplants it to work over a software interface. The decoupling enables entirely new modes of execution from a user's point of view, and provides many benefits such as flexibility, multiplexed use, fault tolerance, dynamic migration, automated load balancing, and cost sharing. Anyone can exploit the benefits of VM technology by deploying the VM implementations on their own hardware. Cloud computing is a term generally used to refer to such installations that provide the advantages of virtualized computing (and storage) interfaces. Due to economies of scale, only large commercial, dedicated installations provide the most cost-effective provisioning of VM technologies and make them accessible over the Internet via Web-based interfaces for very attractive prices. They provide on-demand access to compute resources without the burden of housing, installation, maintenance, and upgrading needs.

5.1.2 Problem Statement

Historically, PDES has largely assumed the luxury of picking the highest-end among the set of computer configuration choices one speeds on the chosen high-end system. However, with the introduction of the Cloud platforms, the new dimension of *price* is introduced into consideration. Since there is a price one must pay for all compute cycles used by the application, a “dollar value” is now attached to each PDES run. The most interesting aspect about this new dimension is that the price variation is non-linear. The user might have to pay more than double the price for double the performance.

Alternatively, doubling the cost does not guarantee double the performance. Since commercial offerings of Cloud computing are profitable mostly due to the ability to multiplex many smaller units of virtual hardware on larger units of physical hardware, one can expect a price structure that permits the most flexibility for multiplexing. In particular, the price structure favors smaller virtual units, and, more importantly for PDES, charges a non-linearly larger price for the highest-end virtual units. Thus, virtual machines whose resources (e.g., speeds and numbers of virtual processors) are close to the capacity of the underlying physical hardware can be expected to be the most expensive.

Figure 61 illustrates the cost model for the Amazon AWS-based EC2 Cloud service [62] that is based on the allotted hardware resource sizes. Suppose the user requires the Cloud for executing a parallel job, and further suppose the user’s application enjoys an ideal parallel execution by which the runtime decreases in proportion to the number of processors. The runtime for a parallel job is plotted against different computational units offered by the Cloud platform. Along the abscissa, the size of each indivisible virtual computational unit increases moving from left to right. On the left ordinate, the ideal parallel runtime is plotted. On the right ordinate, the cost for the computational unit is plotted. The non-linear aspect of the price is notable. Also, additional non-linear effects can be expected due to shared-memory effects, shared network effects, scheduling effects, and inter-VM communication effects. Thus, unless the parallel job is embarrassingly parallel in nature, it is rather difficult to predict the trends of runtime and overall cost of parallel jobs, and an empirical study is inevitable for properly understanding the overall tradeoffs.

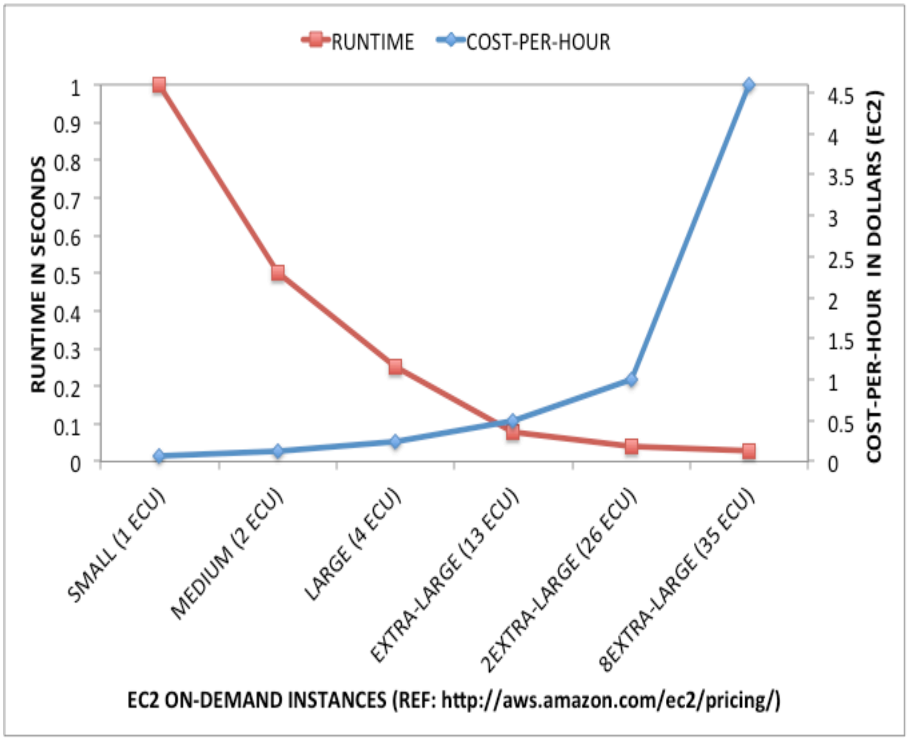


Figure 61 EC2 cost-value model

In this new milieu, little is known about the price-performance features of PDES execution on Cloud platforms, and about the configuration choices of PDES over VM platforms in general. There are several questions that arise. What, if any, is the level of performance penalty taken by a PDES application when moving from a traditional native execution to a VM? Is there any performance gain obtained by insisting that the VM be a privileged one versus the default, unprivileged mode of VMs? Does the highest-end

VM/Cloud hardware configuration always deliver the least total execution time, and at what overall cost? If the highest-end VM configuration is too expensive for the user, what is the next best configuration to choose, considering overall cost? How well does a Cloud platform designed primarily for embarrassingly parallel jobs execute tightly coupled PDES applications? Are runtime and dollar value largely opposed to each other as one might expect? In general, how does the total execution time and total cost vary with different VM/Cloud instance configuration options?

5.1.3 Study Approach

Here, an empirical approach is undertaken to help answer such questions. Results and findings are reported to understand the new configuration space for PDES enabled by the introduction of new, Cloud-specific concepts such as abstracted speeds of virtual processors, normalized units of processors and memories, price per packaged compute-unit, and overall “bottom-line” cost. Using actual PDES application runs executed on a Cloud platform and on high-end VM hosts, we study the configuration space to uncover new insights, trends, and guidelines on the problem of economically executing PDES applications in Cloud environments. For experimentation purposes, we chose the popular Amazon AWS (EC2) Cloud computing platform, and gathered data from a variety of configurations with varying price-performance characteristics. The empirical study makes use of two benchmarks: one is the popular synthetic PHOLD benchmark, and the other is a complex disease spread model at the individual level in a large population. Both optimistic and conservative synchronization schemes are exercised, with varying levels of locality of events.

5.1.4 Related Work

Evaluating the HPC applications performance on Cloud infrastructure has been reported in [63], but these applications are largely high-performance scientific applications such as Community Atmospheric Model (CAM), and are not PDES applications. Network performance on Amazon EC2 data-centers has been studied and an evaluation of the impact of virtualization on network parameters such as latency, throughput, and packet-loss was discussed in [64], again in non-PDES context. There is a good overview and discussion of generic utilization of Cloud infrastructures for PDES applications focusing on the advantages and challenges it poses [65], which also serves as a good motivation and background for PDES on Cloud platforms. The Master-Worker approach to distributed (and fault tolerant) PDES has been explored [66] [67] [68], and is somewhat related, although it is different from the traditional PDES execution view in which all processors are equal. Recently, an evaluation of a set of conservative synchronization protocols on EC2 was reported [69]. Overall, the area is nascent, and much additional research in PDES execution is needed to explore the space opened by the new metrics of VM/Cloud computing beyond the raw speed execution.

5.2 Empirical Study Setup

5.2.1 Performance Benchmarks

Using the PDES applications listed in the previous section, four benchmark applications, namely, conservative and optimistic executions for each of *PHOLD Simulation Benchmark* (PSB) and *Disease Spread Benchmark* (DSB) were designed. In all the benchmarks using PHOLD a *lookahead* of 1 was used.

5.2.1.1 <u>PHOLD Scenarios (PSB)</u>

With PHOLD, we used scenarios with 100 LPs/Federate, 100 and 1000 messages/LP, with 32 Federates for 50% and 90% LOC values. With performance data gathered for both optimistic and conservative synchronization scenarios, a total of 8 sets of readings are gathered for this benchmark.

For 100 LPs/Federate and 100 messages/LP on 32 Federates, 3200 LPs are hosted on 32 DOMs to simulate exchanges of 320,000 PHOLD messages over 100 units of simulation time. Similarly, for 100 LPs/Federate and 1000 messages/LP on 32 Federates, 3200 LPs are hosted on 32 DOMs to simulate exchanges of 3,200,000 PHOLD messages over 100 seconds of simulation time. With a locality value LOC of 50% half of the messages generated are destined to LPs on remote Federates (outside the VCPU). With a locality value LOC of 90%, only 10% are destined to LPs on remote Federates. Thus, LOC 50 is much more taxing on the network than LOC 90.

5.2.1.2 <u>Disease Spread Benchmark (DSB)</u>

DSB simulates the disease spread across regions. Each Federate is mapped to a region, which are formed of number of locations that are mapped to LPs. In the experiments 32 Federates and 10 locations per Federate are instantiated (representative of a small city sized scenario for disease propagation) and each such location has a population of 1000 people. Hence the benchmark involves simulation of spread of disease across 320 locations across a population of 320,000 for a simulation time of 7 days. The mobility of the population can be set to a certain percentage, similar to PHOLD. In the DSB we experiment with 50% and 90%. The LOC 50 mobility suggests that 50% of the trips tend to travel across regions, while LOC 90 suggests only 10% of the trips travel across regions. DSB is slightly I/O intensive generating around 32M of output data compared to less than 200 KB of output data of PSB.

5.2.2 Test Platforms

We utilize two platforms for the VM-based experiments. One is a local high-end machine in our laboratory, and the other is a commercial Cloud offering. The details of these two platforms are provided next.

5.2.2.1 Local Test Platform (LTP)

LTP is our custom-built machine with a Supermicro® H8DG6-F motherboard supporting two 16-core (32 cores in total) AMD® Opteron 6276 processors at 2.3 GHz, sharing 256GB of memory, Intel Solid State Drive 240GB and a 6TB Seagate constellation comprising 2 SAS drives configured as RAID-0. Ubuntu-12.10 runs with Linux® 3.7.1 kernel runs as DOM0 and DOMUs, over Xen 4.2.0 hypervisor.

All DOMUs were para-virtual and networked using software bridge in DOM0. DOM0 was configured to use 10GB of memory and the user-DOMs were configured to use at least 1GB memories each, which were increased as necessitated by the application benchmarks. Each user-DOM used 2GB of LVM-based hard disk created over SAS drives, while the DOM0 uses an entire Solid State Drive (SSD). OpenMPI-1.6.3 (built using gcc-4.7.2) was used to build the simulation engine and its applications. A machine-file listing the IP addresses of the VMs was used along with *mpirun* utility of OpenMPI to launch the MPI-based PDES applications onto VMs.

5.2.2.2 EC2 Cloud Platform

We also ran our benchmarks on Amazon's EC2 Cloud platform. We built a cluster of para-virtual VM instances of Ubuntu 12.04 LTS. The following are the VM clusters used to run the benchmarks (typical offerings available to Amazon EC2 users).

- *m1.small* is a single-core VM with compute power of 1-ECU and has a memory of 1.7 GB.
- *m1.medium* is a single-core VM with compute power of 2-ECUs and has a memory of 3.7 GB.
- *m1.large* is a 2-core VM with compute power of 4 ECUs and has a memory of 7.5 GB.
- *m1.xlarge* is a 4-core VM with compute power of 8 ECUs and has a memory of 1.5 GB.
- *m3.2xlarge* is an 8-core VM with compute power of 26 ECUs and has a memory of 30 GB.
- *hs.8xlarge* is a 16-core VM with compute power equivalent to 35 ECUs and has a memory of 117 GB.

The term ECU here refers to a "EC2 Compute Unit," which is an abstraction defined and supported by Amazon as a normalization mechanism to provide a variety of virtual computation units independent of the actual physical hardware support that they use/maintain/upgrade without user intervention. OpenMPI-1.6.3 was built on the virtual instance, which was used to build the simulation engine and all the PDES applications. A machine-file listing the DNS names of the allotted instances was used to launch the MPI-based PDES applications using *mpirun*.

5.3 Performance Study

5.3.1 Local Test Platform (LTP) Results

PDES being a parallel computing application, two important factors, namely, computation and communication, determine the overall application performance. The hypervisor essentially virtualizes the hardware resources and hence a VM running over hypervisor uses a VCPU and a virtual network interface for computation and communication, respectively. The hypervisor essentially maps the VCPUs onto the available CPUs, while networking is performed using front-end and back-end virtual interfaces. Hence, the hypervisor essentially introduces some overhead due to its presence.

5.3.1.1 Virtual Computational Performance

We know that the hypervisor is a necessity to realize Cloud computing. However, this implies that native execution of PDES is not possible in the presence of a hypervisor, which may introduce overheads such as context switching costs and system call (and hypercall) trap costs. Hence, a performance comparison between the native and VM runs is essential to determine the amount of degradation, if any, that PDES suffers simply for the fact that the execution is moved from native to VM platforms.

In Figure 62, the performance results of both PSB and DSB benchmarks runs are presented from *Native*, *DOM0* and a single *DOMU*. The *Native* readings correspond to a setup where Linux® runs directly over the hardware as usual, without the hypervisor. The *DOM0* readings correspond to a setup where the control- DOM with Linux® runs over the Xen hypervisor as the only running instance and is configured to use all 32 CPUs. A single DOMU readings correspond to a setup where a user-DOM with Linux®

runs in the presence of control-DOM (dOM0); however DOM0 is not loaded with any load during performance runs. These results demonstrate how the presence of the hypervisor affects the compute-performance of a PDES application. As seen from Figure 62, somewhat surprisingly, the results from all the three setups are almost identical across all the runs, suggesting that the overhead of the Xen® hypervisor in delegating the CPU resources is almost negligible.

This data is helpful in addressing the issue of native *vs*. VM-based performance of PDES execution, and may encourage the community to move towards a Cloud environment by allaying uninformed fears of incurring a significant performance penalty.

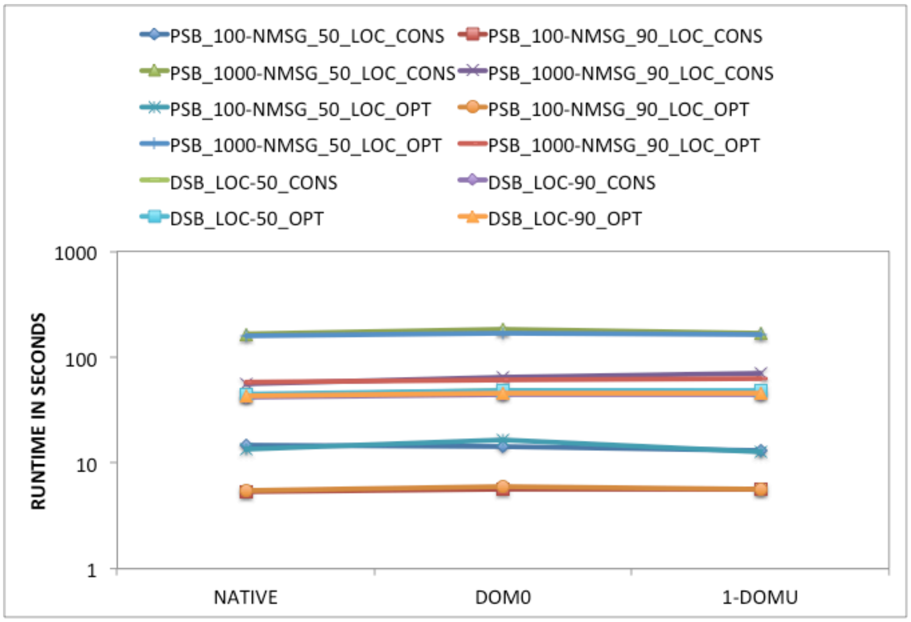


Figure 62 Native, DOM0 and single DOMU performance comparison on LTP

5.3.1.2 Effects of Virtual Communication and I/O via DOM0

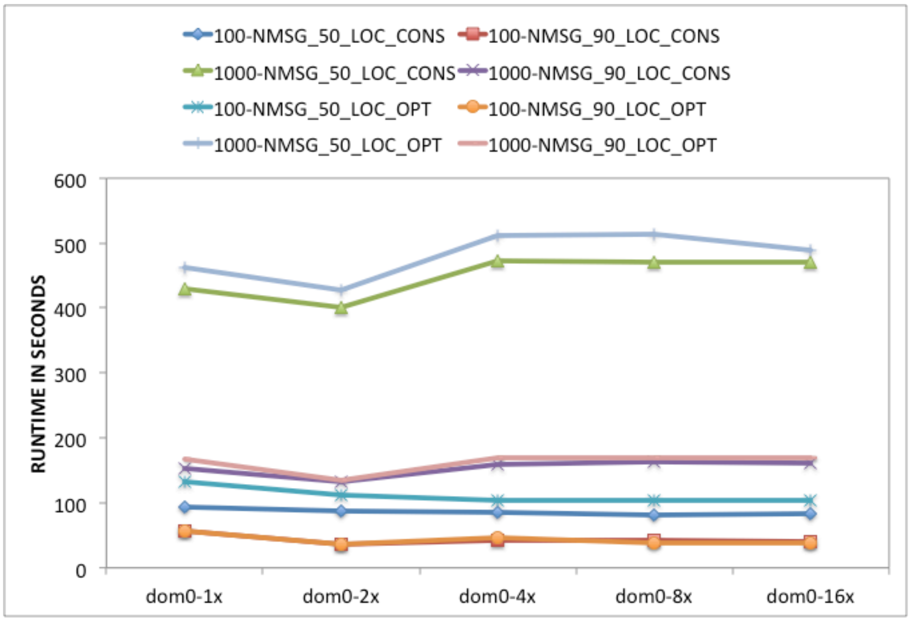


Figure 63 PSB runtime performance with 32 VMs for varying DOM0 weights

Since DOM0 is partially involved in servicing the network communication and input/output (I/O) for all DOMU, DOM0 may need to receive sufficient number of CPU cycles (or a higher priority weight). The need for higher DOM0 weights for better performance was demonstrated in [70] using an older version (3.4.2) of the Xen distribution. To understand such requirements in the current version, the weight assigned to DOM0 relative to all DOMU is varied from 1 to 16. For example, a factor of 4 implies that DOM0 has four times more credits than any DOMU. This provides an overburdened DOM0 more CPU cycles compared to DOMUs.

The runtimes with varying weights are plotted in Figure 63 for the PSB, and in Figure 64 for the DSB. The seemingly flat curves of Figure 63 suggest very slight or no impact of higher weights for DOM0 on the runtime performance with the newer version of Xen. This change can be attributed to the incorporation of Netchannel2 [71] in Xen networking, which transfers the burden of copying network data from DOM0 to the DOMUs. However, for DSB, a good improvement in the performance as the weight of DOM0 is doubled is seen in Figure 64. DSB being I/O intensive and also due to the fact that DOM0 services the I/O, the additional weight provided for DOM0 helps significantly in speeding up the I/O functionality during DSB runs.

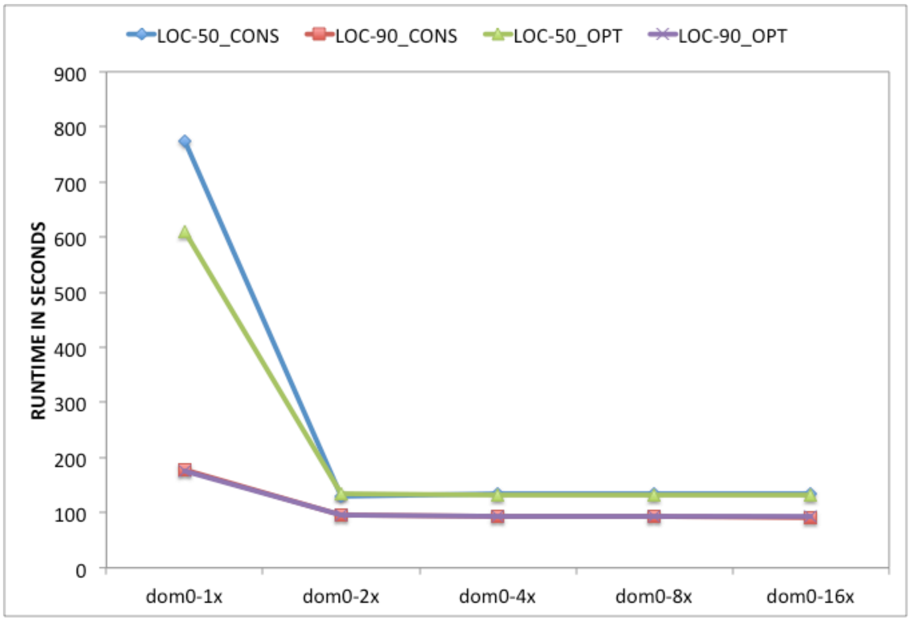


Figure 64 DSB runtime performance with 32 VMs for varying weights of DOM0

5.3.1.3 Virtual Computation and Communication Performance

To study the impact of combined virtual computation and communication effects on PDES applications, we ran the benchmarks on our LTP, varying the number of VMs in the experiments from a single DOMU with 32 VCPUs to 32 DOMUs with 1 VCPU, keeping the total number of VCPUs constant.

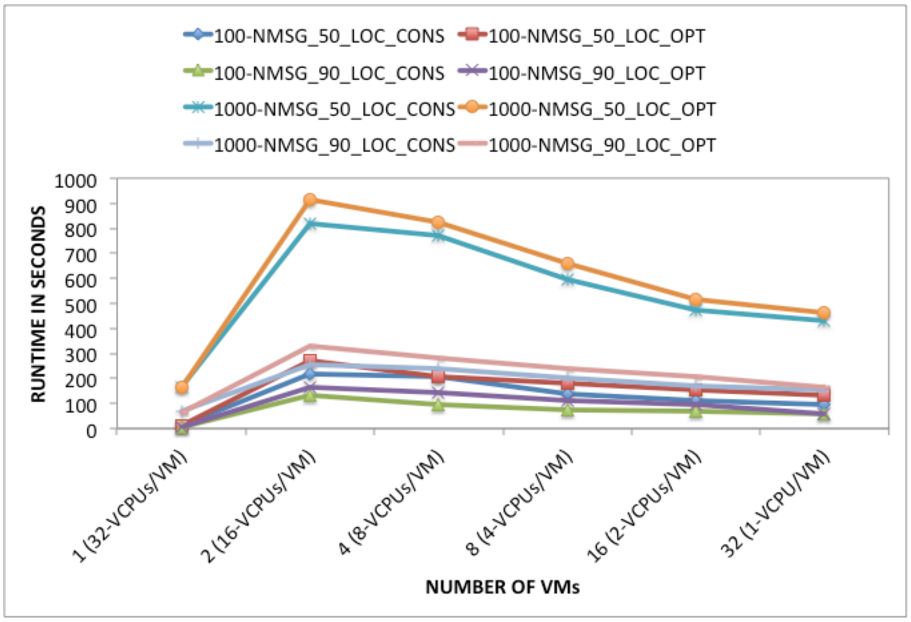


Figure 65 Performance comparison with increase in number of DOMs using PSB on LTP

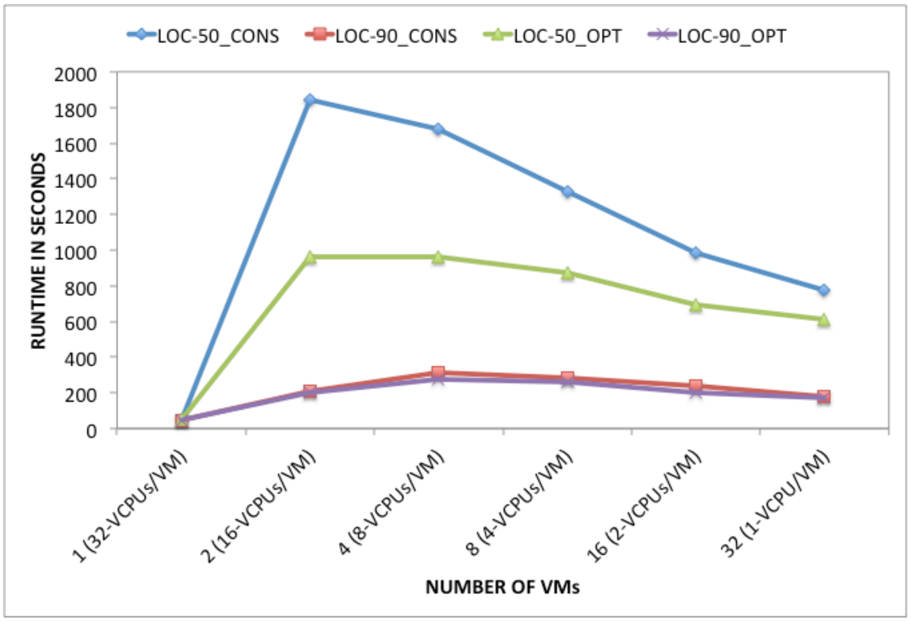


Figure 66 Performance comparison with increase in DOMs using DSB on LTP

Figure 65 and Figure 66 show the benchmark results obtained from varying the number of DOMs hosted on LTP using PSB and DSB, respectively. For the same benchmark the figures show how the performance varies with the increase in the number of DOMs. Note that as the number of DOMs running the benchmark increases, the number of VCPUs within each DOM also decreases. Also note that in each of these benchmark runs, the number of Federates is equal to the number VCPUs, i.e., Federates have a 1:1 mapping to the VCPUs. Hence, in a fast network environment, we expect the runtime across all types of DOM configurations to be largely identical, since the same number of VCPUs are involved in the computation.

A very interesting and common trend across all the benchmark runs is the degradation of performance with fewer numbers of VMs beyond 1, and its betterment with the increase in number of VMs hosted. In other words, there is a steep rise in runtime when moving from 1 VM to 2 VMs but a gradual drop from 2 to 32 VMs. The effect is predominant in the cases where the network traffic is high (LOC=50), as seen in both Figure 65 and Figure 66, for PSB and DSB, respectively.

This trend is counterintuitive to a general parallel-computing user because a better performance is expected in scenarios involving VMs with more VCPUs. The intuition is that the parallel processing libraries such as MPI generally use shared-memory to communicate across processes within the same VM. Hence, one expects to observe better performance by increasing the communication across Federates within a DOM (shared memory) and reducing the inter-DOM messages by reducing number of DOMs.

The underlying reason for the counterintuitive trend is as follows. When a VM contains many VCPUs, a bottleneck is created at the virtual network interface card (NIC) because of serialization. Further, DOMUs doing most of the networking work as observed in the Figure 63 in previous section adds on to this performance degradation.

In real hardware, the communication is often highly optimized via direct memory accesses, cache coherence mechanisms between NIC and CPU, and so on. However, in the case of VMs, the NIC is a software implementation, and all synchronization is performed in software, which significantly reduces the network speed. This degradation increases in a quadratic nature with the number of VCPUs sharing the virtual NIC. Unfortunately, there is little that can be done regarding this issue other than reduce the network traffic generated per VCPU or reduce the number of VCPUs per VM.

As previously observed in Figure 61, the Cloud operators charge a lower price for low-end machines and higher cost for high-end machines. The benchmark performance results suggest that PDES applications can in fact take an advantage of the lower cost of smaller sized VMs *and* gain lower execution time simply by moving to the other extreme of 1 VCPU/VM, and greatly benefit from the existing cost model offered by the Cloud infrastructures, such as EC2.

In Figure 67, we show the time that the simulator takes to compute a lower bound on incoming timestamps (LBTS) for the most-affected PSB runs namely, 1000-NMSG_50_LOC_CONS and 1000-NMSG_50_LOC_OPT runs along with LOC-50_CONS and LOC_50_OPT DSB runs. However, the number of LBTS computations remained the same across the same benchmark runs even while the number of VMs is changed. Hence, within PDES, it is the prolonged LBTS computation that affects the overall runtime of the simulation application.

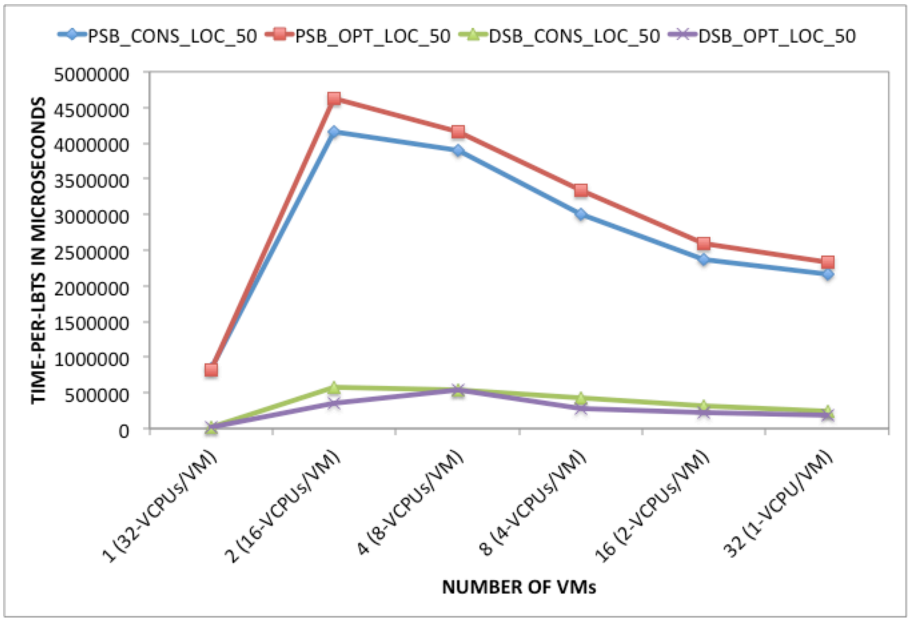


Figure 67 Time per LBTS computation with increase in the number of VMs on LTP

5.3.2 EC2 Cloud Computing Platform Results

To deal with possible variance of performance in the Cloud due to periodicity of loads and other uncontrollable phenomena, each data point in the results is derived as an average from three runs executed on three different days and times. Note that each request for VMs from Cloud assigns a different set of machines and hence, the virtual cluster built for every run is different from the other. This averaging for variance applies to the Cloud performance results in Figure 68 through Figure 72.

Note also that all the machines that the EC2 provides are VMs. This provides the Cloud operator an ability to multiplex multiple VMs more numerous than the available hardware resources. However, by overloading the host machine, the compute cycles of the physical machine are shared among the VM instances. By defining the Elastic Compute Unit (ECU) that is always lesser or equal to the compute cycles offered by physical CPU-core, the Amazon EC2 is able to overload the host machine and still guarantee the provision of the assured ECU worth of computational service.

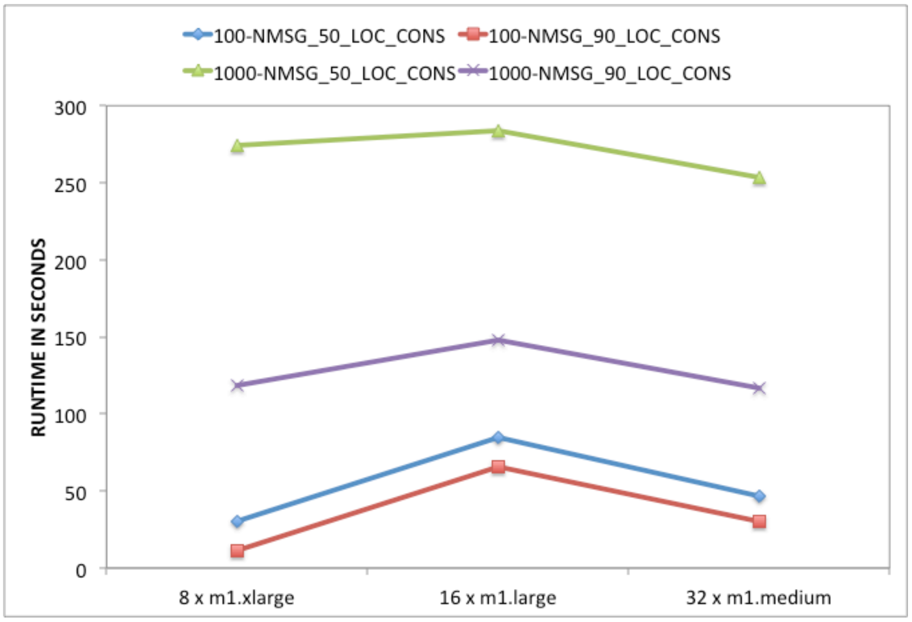


Figure 68 Runtime performance of PSB with conservative synchronization on EC2

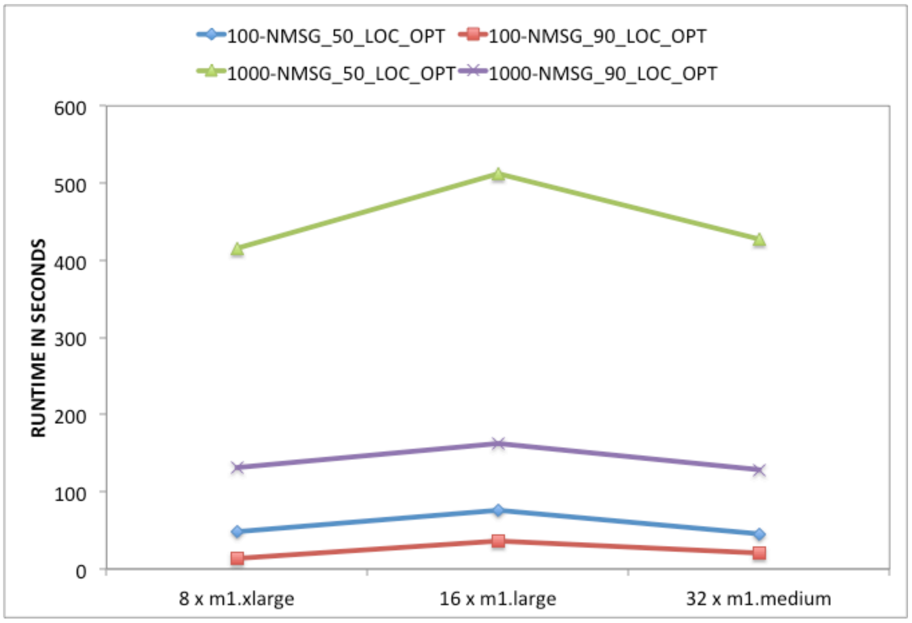


Figure 69 Runtime performance of PSB with optimistic synchronization on EC2

With a Cloud infrastructure, the user is not guaranteed in advance specific details of the physical hardware. The user is only assured of the ECU, number of cores and amount of memory for an instance created. The performance unit of the CPU-core is provided in terms of ECUs. For example: *m1.small* and *m1.average* are both single-core VMs but with compute units of 1 ECU and 2 ECUs, respectively. The *m1.small* instance's assured compute cycles (in ECUs) can be compared to a low priority task that can be migrated across physical nodes or multiplexed flexibly at the Cloud runtime's discretion. Hence, a good performance from *m1.small* instances is not guaranteed. Since, we would not be able to characteristically determine *a priori* the hardware-specific details of a physical CPU-core from the Cloud, the next best option is to choose the VM configuration with a single core with high number of ECUs and use it as a baseline for selecting multi-core machines, if needed.

To observe the performance trend observed with PSB and DSB benchmarks on our LTP we use *m1* set of machines comprising *m1.small*, *m1.medium*, *m1.large* and *m1.xlarge*. Of this set the *m1.medium*, *m1.large* and *m1.xlarge* VMs are *single*, *dual* and *quad* support VCPUs, respectively. Further, the compute cycles of these VMs increase by a factor of 2; i.e., 2 ECUs, 4 ECUs and 8 ECUs in the same specified order. The *m1.small* that provides 1ECU worth of compute cycles is not considered for this set of runs because it is difficult to use it for a fair comparison with other configurations. We built three virtual clusters using these VM instances. *32×m1.medium*, *16×m1.large* and *8×m1.xlarge* are the 3 virtual clusters built using 32, 16 and 8 instances of *m1.medium*, *m1.large* and *m1.xlarge* VMs, respectively.

The conservative and optimistic synchronization-based PSB runtimes from EC2 runs are plotted in Figure 68 and Figure 69, respectively. Figure 70 presents the results for DSB runs on EC2. Interestingly, similar to LTP results, we observe a consistent trend across all the plots. The runtime in most of the cases is at its best with *8×m1.xlarge* virtual cluster, which worsens with *16×m1.large* virtual cluster and gets better with *32×m1.medium* virtual cluster runs. Note that in each of these benchmark runs the number of Federates is equal to number VCPUs.

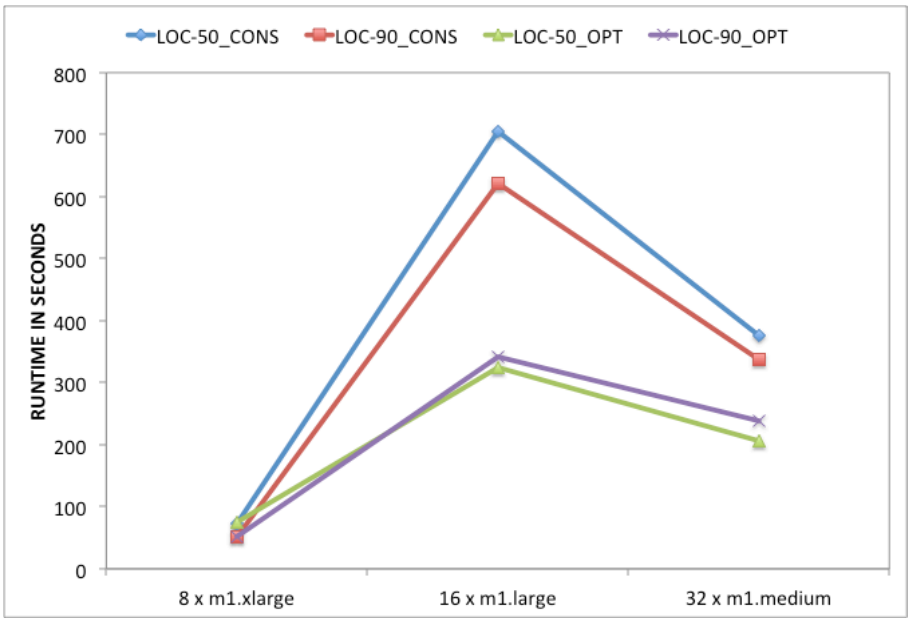


Figure 70 Runtime performance of DSB on EC2

VM *m1.xlarge* is the most powerful among all the other offered VMs in the *m1* set. The observed counterintuitive behavior on the Cloud can be reasoned using our prior understanding of the benchmark behavior on LTP, with *m1.xlarge* considered as the physical node capacity on which the all VMs belonging to *m1* can run. Given, the lack of *a priori* guarantees about the physical hardware properties of machinery that hosts EC2

VMs, this is a fair assumption for all *m1* set of VMs. In this case, the runtime reduction in the *8×m1.xlarge* virtual cluster can be attributed to the very low involvement of the virtual network as the quad-core VMs occupying entire physical node mainly use the high-speed physical inter-connect during parallel computation. The increase in runtime in *16×m1.large* setup (instead of decrease as observed in LTP runs) can be attributed to the presence and active utilization virtual-networking as more than one *m1.large* could have been hosted on a physical node. The reduction in the runtime with increase in number of VMs in the *32×m1.medium* setup is consistent with our observations on our LTP.

5.3.3 LTP vs. EC2

In comparing the results from LTP and EC2, note that all VMs running on LTP use the virtual network, whereas an indeterminate combination of real and virtual-network is typical of EC2 environment. While the LTP uses 32 CPU-cores of AMD Opteron at 2.3 GHz, the *m1* set of EC2 is perceived to use Intel Xeon CPU-cores at 2.6 GHz (*cpuinfo* of Cloud instance).

The PSB's LTP and EC2 runtime comparisons using 32 VM scenarios are shown in Figure 71; similar runtime comparisons for DSB are shown in Figure 72. An important aspect of PSB plots in Figure 71 is the close similarities for the LTP and EC2 trends.

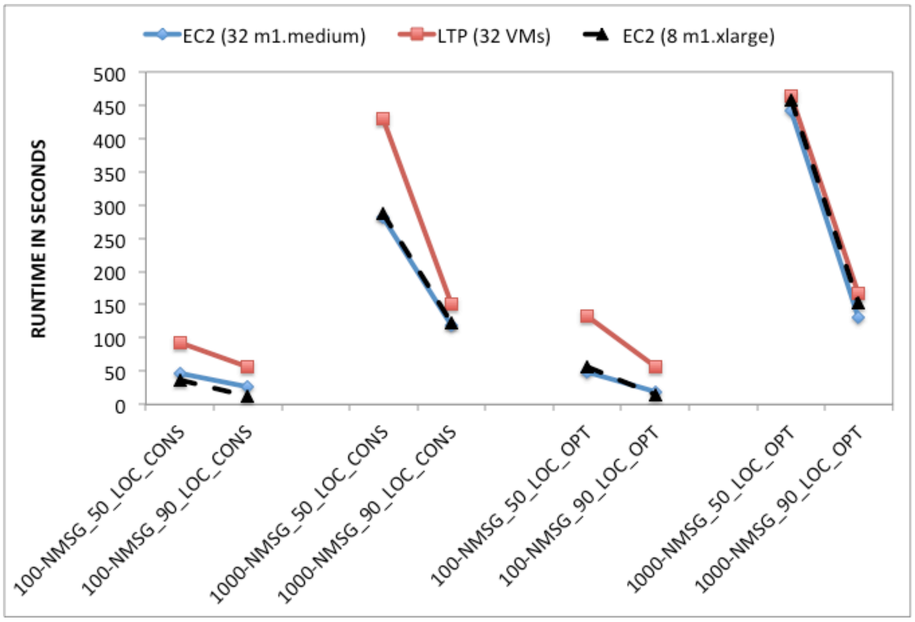


Figure 71 LTP and EC2 runtime comparison for PSB

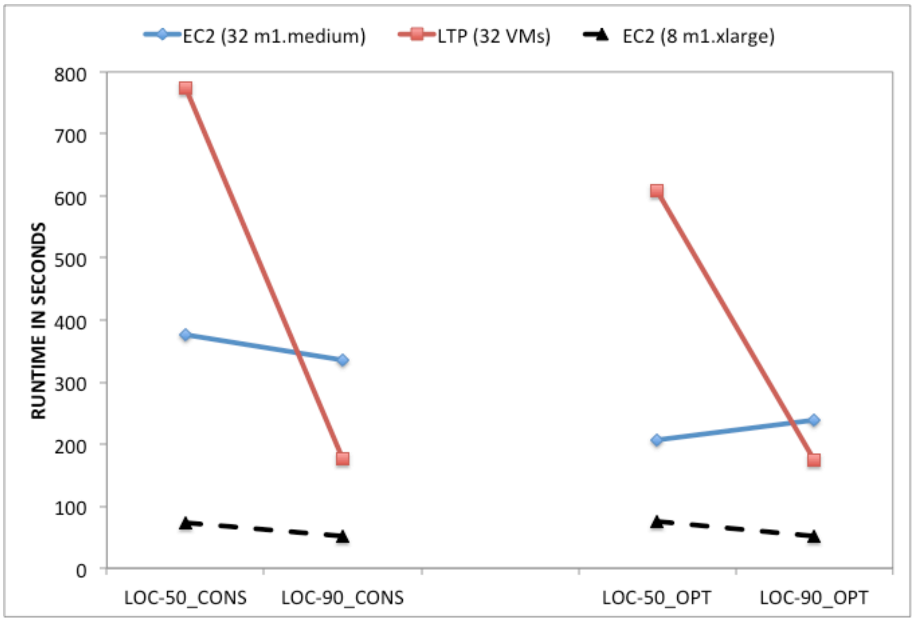


Figure 72 LTP and EC2 runtime comparison for DSB

However, the DSB runtime plot comparisons in Figure 72 differ from this view, especially in 32 VM runs. Note that LTP runs are highly affected by network load as suggested by huge drop in runtimes as LOC value changes from 50 to 90. However, the corresponding 32 *m1.medium* EC2 runs seem unaffected, essentially suggesting the absence or minimal utility of virtual network. Further, 32 *m1.medium* EC2 runtime during LOC 90 is greater than its LTP peer, suggesting EC2 performance being effected by distribute I/O. The 8 *m1.xlarge* EC2 runtimes provide the best runtimes on EC2, suggesting that distribute I/O affects VM dispersed across many nodes more than on fewer nodes. This observation seems to be consistent with LTP, where runtimes are more affected by network performance than I/O.

Finally, we note that these observations can be helpful in determining an upper bound runtime on the Cloud environments utilizing processors of similar clock speeds.

5.3.4 Cost-Value Evaluation on EC2

Table 4 Details of EC2 on-demand instances

EC2 Instances	Cost/hour in Dollars	Number of VCPUs	Assured Performance in ECUs
m1.small	0.06	1	1
m1.medium	0.12	1	2
m1.large	0.24	2	4
m1.xlarge	0.48	4	8
m3.2xlarge	1.00	8	26
hs.8xlarge	4.60	16	35

In Table 4, the relevant details of on-demand VMs provided by the Amazon EC2 service are tabulated. For cost-value evaluation we selected set of VMs based on the specified ECU value. To run the PSB and DSB, we built 4 clusters of VM instances. The cheapest VM instance for the least compute unit of 1-ECU is *m1.small* and a VM cluster formed using 32 such instances is called *32×m1.small*. Similarly, VM clusters of *16×m1.medium, 8×m1.large and 2×m3.2xlarge,* are formed using 16, 8 and 2 instances of *m1.medium, m1.large, m3.2xlarge* VMs, each of these VMs have an ECU of 2, 4 and 26, respectively. The *hs1.8xlarge* is the most powerful and most expensive VM that EC2 offers , and it assures an ECU of 35.

5.3.4.1 PSB and DSB Runtime Performance

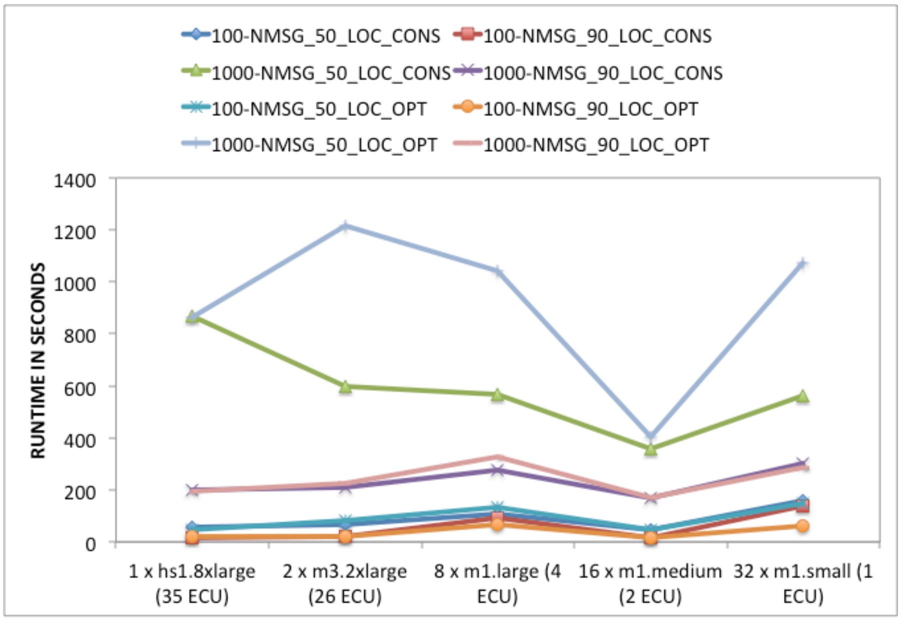


Figure 73 PSB runtime performance on EC2

Figure 73 plots the runtimes of various PSB scenarios. While the runs with lower network traffic are almost flat, the optimistic and conservative curves vary significantly across different VM clusters. Three significant observations can be made from the

1000_NMSG-50_LOC_CONS and 1000_NMSG-50_LOC_OPT runtime plots. They are (a) contrary to the trend observed in the LTP runs the runtime of both OPT and CONS curves worsen on *32×m1.small*, (b) the runtime on high-end *hs1.8xlarge* VM, where the Federates are hosted on a single node and in the absence of network utilization during parallel computing, the runtime is the worst among all (c) the best performance across almost all runs is obtained with *16×m1.medium* cluster setup, where 32 Federates are hosted on 16 instances of single-core VMs.

A VM with ECU 1 on a hypervisor running on CPU-cores whose compute capacity is often multiple of ECUs, can be realized either in scenarios where *m1.small* VMs are overloaded on the hypervisor or on nodes where it's often run as VM with lower weight and are generally capped so that they do not exceed their provision. Either of these cases is detrimental for highly asynchronous parallel computing PDES applications. Hence, the poor performance with *32×m1.small* is expected.

Regarding the poor performance on *hs1.8xlarge* runs, note that the *hs1.8xlarge* is a 16-core VM and is loaded with 32 Federates. In overloaded scenarios such as these the hypervisor VCPU scheduler in quest of ensuring fairness in physical CPU utilization among all VCPUs affects the performance. This is a known problem [70].

Further, good runtime performance can be expected from *16×m1.medium* virtual cluster runs based on our previous observations both on LTP and EC2.

Figure 74 plots the runtimes of various DSB scenarios. Here, the runtimes are almost same on *hs1.8xlarge*, *2×m3.2xlarge* and *16×m1.medium* virtual clusters. Same reason as stated for PSB explains the bad performance of DSB on *32×m1.small.* The readings for *8×m1.large* are consistent with the observations seen with *16×m1.large* virtual cluster runs shown in Figure 70 and the performance degradation can be attributed to virtual-networking.

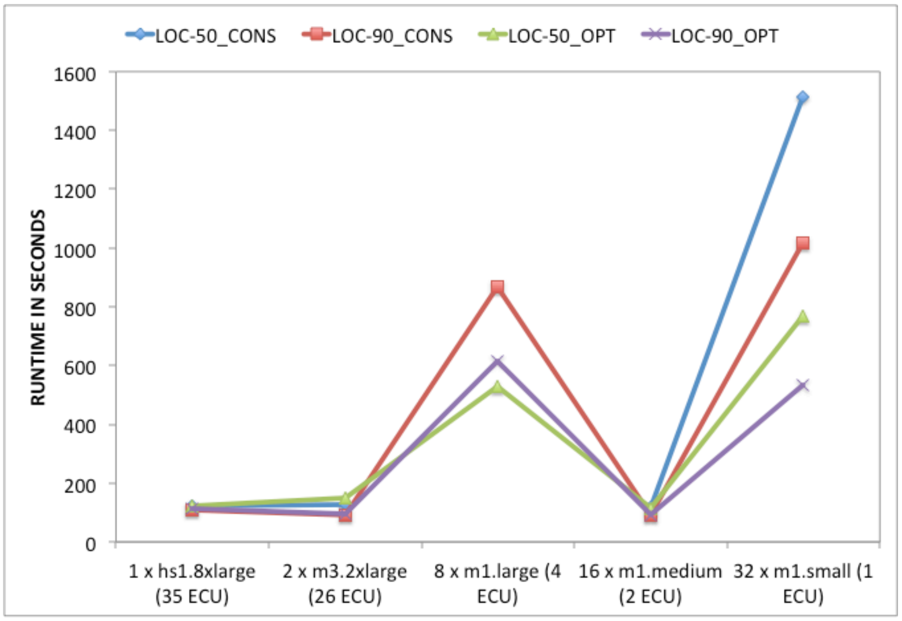


Figure 74 DSB runtime performance on EC2

5.3.4.2 Cost factoring to PSB and DSB Scenarios

After obtaining the runtime from the PSB and DSB runs and the cost-per-hour from the EC2 specifications, we computed the overall cost for the PSB and DSB scenarios. Figure 75 and Figure 76 plot the cost of execution in terms of dollars on various virtual clusters. For most of the runs it was found that the *16×m1.medium* virtual cluster provided the best cost-value across almost all runs for both PSB and DSB scenarios.

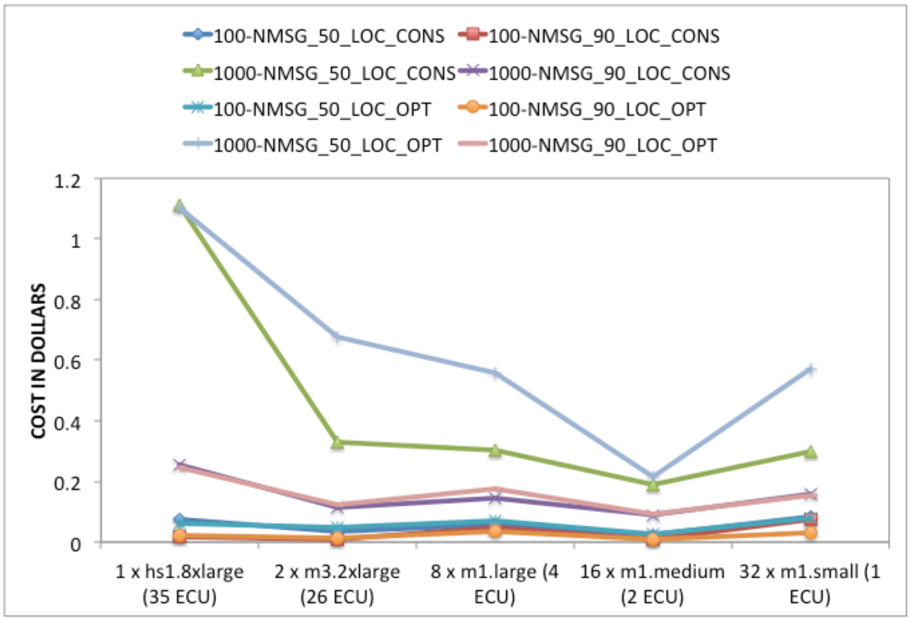


Figure 75 Overall cost of PSB on EC2

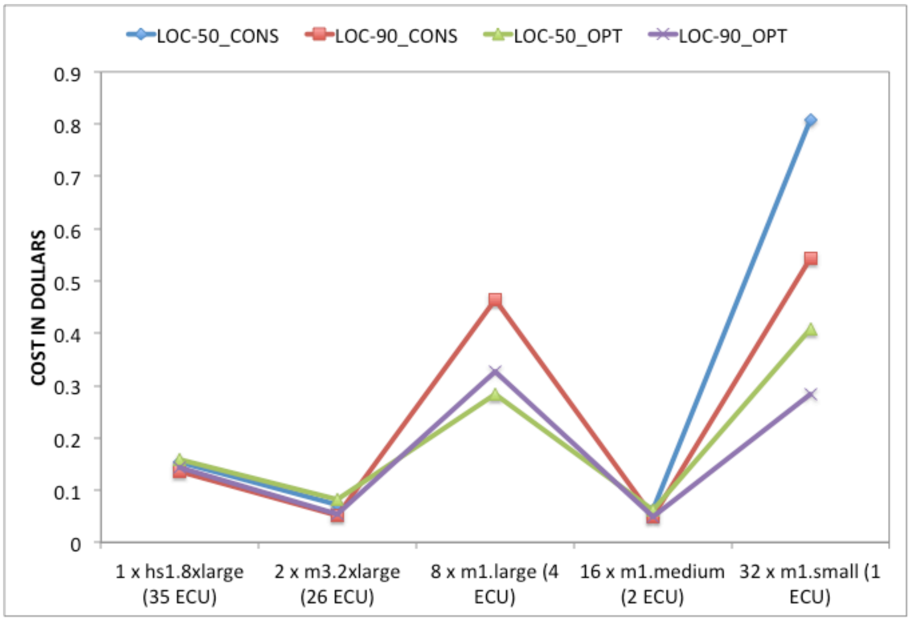


Figure 76 Overall cost for DSB on EC2

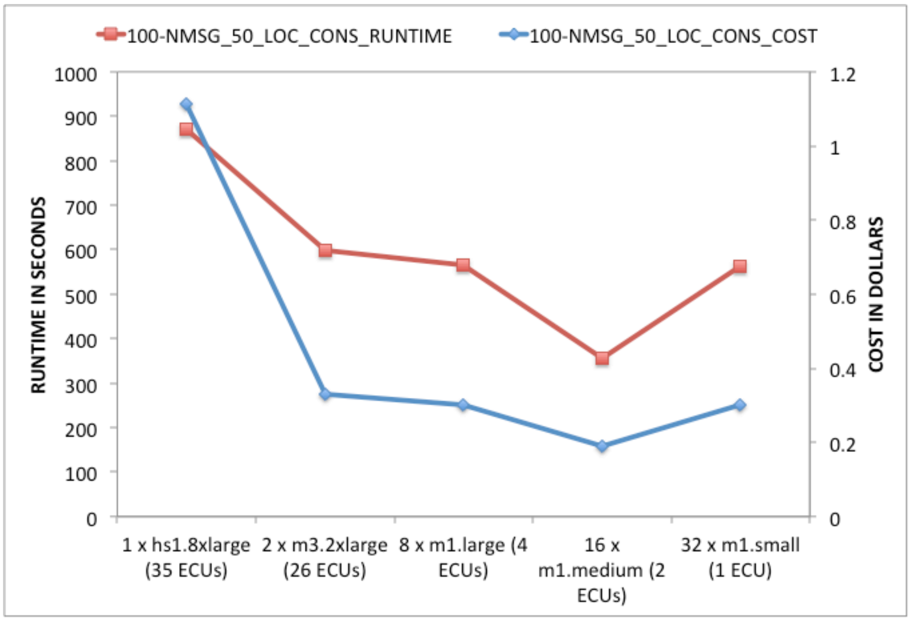


Figure 77 Cost and runtime of PSB 1000-NMSG, 50-LOC, CONS run on EC2

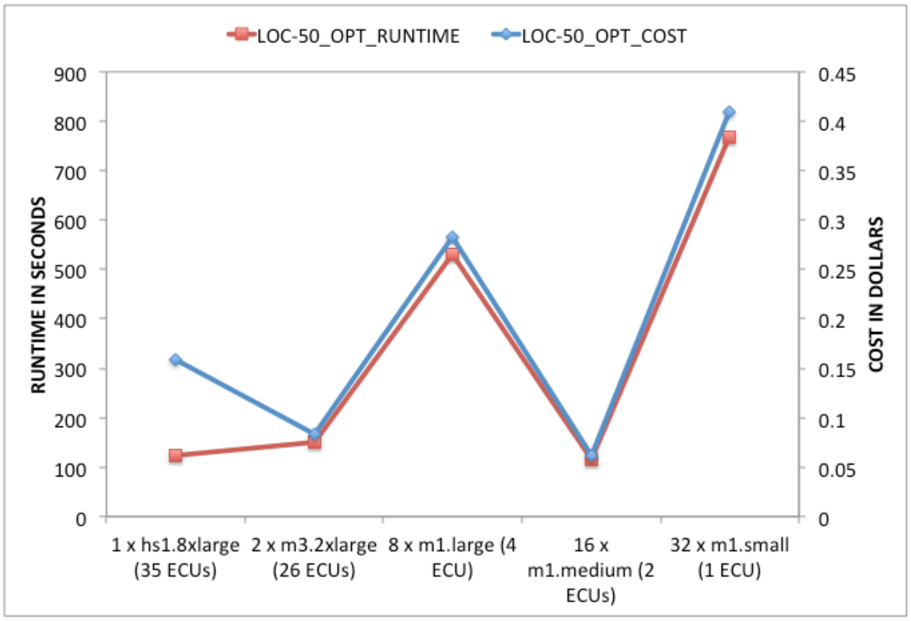


Figure 78 Cost and runtime plots of DSB with LOC-50, OPT run on EC2

To compare the runtime and costs, we pick the better-performing large-scale scenarios with high-network traffic from PSB (100-NLP_1000-NMSG_50-LOC_CONS) scenario and DSB (LOC-50_OPT) scenario, as shown in Figure 77 and Figure 78, respectively. The PSB plot in Figure 77 shows best runtime and best cost associated with virtual cluster of decently compute-intensive instances of VM, i.e. *16×m1.medium*, which is against popular belief. The cost and runtime on the expensive high-end resource is far higher than that on the *16×m1.medium* cluster.

Similar to PSB, the DSB plot shown in Figure 78 also provides better runtime performance and cost at *16×m1.medium*. Although the runtime provided by the expensive high-end compute resource compares well with runtime, the cost of computation is higher than that of *16×m1.medium*.

5.4 Summary

5.4.1 Performance Summary

From the benchmarks and scenarios, we find that VM-based execution can be as fast as native execution, with little perceivable performance degradation. Also, privileged and unprivileged VMs deliver the same runtime, indicating that it is not worthwhile to elevate privileges with the goal of increasing performance for PDES runs.

On dedicated machines in which the number of virtual cores is exactly the same as the number of physical cores, the fastest execution is obtained by using only a single VM that contains all the virtual cores. However, such a dedicated allocation of virtual to real cores is almost impossible to ensure in a typical Cloud environment because the underlying physical machine is opaque and also subject to change. Thus, the fastest execution that is competitive with native execution cannot be obtained on the Cloud. In fact, due to complex scheduler artifacts that arise due to a fundamental mismatch between virtual time order and fair scheduling order, the PDES execution on the highest end VM configuration in the Cloud suffers from degraded performance. To make matters worse, since the computational cycles on the highest-end configuration also cost significantly more than other lower end configurations, the overall cost can be much higher, hence less competitive, than execution on lower end configurations. Thus, on the Cloud, it seems to be more economical to choose some of the least expensive configurations (which have only one or two virtual cores per VM), which deliver a dramatic reduction in cost coupled with good runtime relative to the high-end configurations.

On dedicated VM hosts outside the Cloud, there is also an interesting tendency towards the extremes: while the best runtime is obtained on one VM with all the virtual cores, the next best is obtained on the other extreme of the spectrum in which each VM has only one virtual core. In other words, to obtain the best performance, either 1xN or Nx1 should be chosen (N is the number of physical cores), but all other configurations in between should be avoided as they suffer from worse performance. This empirical performance study presented in this Chapter was published in [72]

5.4.2 Recommendations

While PDES has largely focused so far on speed, a need to address the associated dollar value can no longer be ignored when the PDES applications are executed in a Cloud environment. From the performance study it is clear that low cost and small runtime are not always opposed to each other, and that trade-offs exist.

On a node with *N* physical processor cores, the overheads of the virtual network interface should be avoided either by using the entire physical node with a single-VM using *N* VCPUs or or by using *N* VMs each with only one VCPU.

When a PDES application is executed over a single VM (that uses the entire physical node), the host node must avoid being overloaded with more VCPUs than physical cores; i.e., the number of federates per VM should equal the number of VCPUs (not the number of ECUs). This avoids the undesired effects of VM scheduling on PDES performance.

CHAPTER 6

VTS OVER VM: VIRTUAL TIME-AWARE SCHEDULING

6.1 Problem Space

6.1.1 VM Execution Platform

Newer parallel computing platforms, such as cloud computing, based on virtualization technologies are maturing of late, and are seen as a good alternative to native execution directly on specific parallel computing hardware. There are several benefits to using the virtualization layer, making such platforms very appealing as an alternative approach to execute parallel computing tasks. In the context of parallel discrete event simulation (PDES), the benefits include the following:

- The ability of the virtualization system to simultaneously host and execute multiple distinct operating systems (OS) enables PDES applications to utilize a mixture of simulation components written for disparate OS platforms
- The ability to over-subscribe physical resources (i.e., multiplex larger number of VMs than available physical compute resources) allows the PDES applications to *dynamically* grow shrink the number of physical resources as the resources become available or unavailable, respectively
- The dynamic imbalances in event loads inherent in most PDES applications can be efficiently addressed using the process migration feature of the virtual systems

- The fault tolerance features supported at the level of VMs in concert with the VM migration feature also automatically helps in achieving fault-tolerance for PDES applications.

6.1.2 Problem Statement

A critical component of the virtualized system is the *hypervisor*, which provides the ability to host and execute multiple VMs on the same physical machine. To support the largest class of applications, a *fair-sharing* scheme is employed by the hypervisor for sharing the physical processors among the VMs. The concept of fair sharing works best either when the VMs execute relatively independently of each other, or when the concurrency across VMs is fully realized via uniform sharing of computational cycles. This property holds in the vast majority of applications in general. However, in PDES, fair-share scheduling does not match the required scheduling order, and, in fact, may run counter to the required order of scheduling. This mismatch arises from the fundamental aspect of inter-processor dependency in PDES, namely, the basis on the global simulation time line.

In PDES the simulation time advances with the processing of time-stamped simulation events. In general, the number of events processed in a PDES application varies dynamically during the simulation execution (i.e., across simulation time), and also varies across processors. This implies that the amount of computation cycles consumed by a processor for event computation does not have any specific, direct correlation with its simulation time. A processor that has few events to process within a simulation time window ends up consuming few computational cycles. It is not ready to process events belonging to the simulation-time future until other processors have executed their events and advanced their local simulation time. However, a fair-share scheduler would bias the scheduling towards this lightly loaded processor (since it has consumed fewer cycles) and penalize the processors that do in fact need more cycles to process their remaining events within that time window. This type of operation works against the actual time-based dependencies across processors, and can dramatically deteriorate the overall performance of the PDES application. This type of deterioration occurs when conservative synchronization is used. Similar arguments hold for optimistic synchronization, but, in this case, the deterioration can also arise in the form of an increase in the number of rollbacks. The only way to solve this problem is to design a new scheduler that is aware of, and accounts for, the simulation time of each VM, and schedule them in a *Least-LVT-First* (LLF) order.

6.1.3 Related Work

The Master-Worker approach to distributed (and fault tolerant) PDES [68] is also a related but complementary approach, different from our support for the traditional PDES execution view in which all processors are equal. We adopt a different approach by focusing at the lowest level, i.e., at the level of the hypervisor itself. Incidentally, the Time-Warp Operating System [73] of the 1980's is one of the earliest works that addressed PDES performance issues by realizing the simulation scheduler (and related functionality) at the bottom-most hardware levels; however, this was limited to a single operating system, as opposed to a hypervisor system.

There is also a superficial semblance with our own prior related work in VM-based network simulations discussed in Chapter 2 to Chapter 4. However, VM-based network simulations are fundamentally different from PDES execution over VM platforms. In VM-based network simulations, the simulation time of each VM is determined by the hypervisor itself (in terms of computation time consumed by each VM, tracked and accounted by the hypervisor), whereas in PDES over VMs, the virtual time for scheduling is entirely determined by the user's simulation model. The hypervisor does not (in fact, cannot) have any way of influencing the virtual time at which the simulator executes inside each VM. The virtual time can only be communicated *from* the PDES engine *to* the hypervisor via the VM's OS, and the hypervisor is obligated to respect the value of the virtual time supplied by each VM (albeit, with the guarantee that the global minimum of the times across all VMs will never decrease).

6.2 Issues and Challenges

6.2.1 PDES Characteristics

Parallel discrete event simulation (PDES) has traditionally assumed execution at the highest-end of the computing platform available to the user. However, the choice is not so straightforward in Cloud computing due to the non-linear relation between actual parallel runtime and the total cost (charged to the user) for the host hardware.

For example, suppose a multi-core computing node has 32 cores on which a PDES with 32 logical processors (i.e., 32 concurrent simulation loops) is to be executed. Generally speaking, traditional PDES maps one logical processor (i.e., one simulation loop) to one native processor. However, with Cloud computing, the monetary charge for such a direct mapping (i.e., a virtual machine with 32 virtual cores) is typically much larger than the total monetary charge for aggregates of smaller units (i.e., 32 virtual machines each with only 1 virtual core).

6.2.2 Non-linear Cost Structure

The non-linear cost structure is fundamentally rooted in the principles of economies of scale -- the Cloud hosting company gains flexibility of movement and multiplexed mapping of smaller logical units over larger hosting units, ultimately translating to monetary margins. Moreover, a high-end multi-core configuration on native hardware is not the same as high-end multi-core configuration on virtual hardware because the inter-processor (inter-VM) network appears in software for VMs, but in ``silicon-and-copper'' for native hardware. The aggregate inter-processor bandwidth is significantly different between the virtualized (software) network and in-silico (hardware) network.

6.2.3 Multiplexing Ratio

Given that multiple VMs must be used to avoid the high price of a single many-core VM, the performance of PDES execution now becomes dependent on the scheduling order of the VMs (virtual cores) on the host (real hardware cores). This makes PDES performance to be at the mercy of the hypervisor scheduler's decisions. When the multiplexing ratio (ratio of sum of virtual cores across all VMs to the sum of actual physical cores) even fractionally exceeds unity, the PDES execution becomes vastly sub-optimal. In all Cloud offerings, this multiplexing ratio can (and will very often) exceed unity dynamically at runtime. Thus, we have a conflict: one-to-one mapping (multiplexing ratio of unity or smaller) incurs a higher monetary cost, but increasing the multiplexing ratio incurs a scheduling problem, and increases the runtime, thereby stealing any monetary gains.

6.2.4 Scheduling Problem

The conflict arises due to the hypervisor scheduler: the default schedulers designed for general Cloud workloads are a gross mismatch to PDES workloads. The hypervisor is a critical component of the virtualized system, enabling the execution of multiple VMs on the same physical machine. To support the largest class of applications on the Cloud, a fair-sharing scheme is employed by the hypervisor for sharing the physical processors among the VMs. The concept of fair sharing works best either when the VMs execute relatively independently of each other, or when the concurrency across VMs is fully realized via uniform sharing of computational cycles. This property holds good for vast majority of applications in general. However, in PDES, fair-share scheduling does not match the required scheduling order, and, in fact, it may run counter to the required order of scheduling. This mismatch arises from the fundamental aspect of inter-processor dependency in PDES, namely, the basis on the global simulation time line.

6.2.5 Virtual Time-based Scheduling

In PDES the simulation time advances with the processing of time-stamped simulation events. In general, the number of events processed in a PDES application varies dynamically during the simulation execution (i.e., across simulation time), and also varies across processors. This implies that the amount of computation cycles consumed by a processor for event computation does not have any specific, direct correlation with its simulation time. A processor that has few events to process within a simulation time window ends up consuming few computational cycles. It is not ready to process events belonging to the simulation-time future until other processors have executed their events and advanced their local simulation time. However, a fair-share scheduler would bias the scheduling towards this lightly loaded processor (since it has consumed fewer cycles) and penalize the processors that do in fact need more cycles to process their remaining events within that simulation time window. This type of operation works against the actual simulation time-based dependencies across processors, and can dramatically deteriorate the overall performance of the PDES application. This type of deterioration occurs when conservative synchronization is used. Similar arguments hold for optimistic synchronization, but, in this case, the deterioration can also arise in the form of an increase in the number of rollbacks. The only way to solve this problem is to design a new scheduler that is aware of, and accounts for, the simulation time of each VM, and schedule them in a LLF order.

A final twist in the tale is that a scheduling algorithm based solely on LLF-order is susceptible to *deadlock*, and simple schemes to resolve the *deadlock* may suffer from *livelock* (these issues are elaborated later). Thus, a new *deadlock* and *livelock* free hypervisor-scheduling algorithm is needed for efficient execution of PDES on Cloud/VM platforms. Also, its implementation must allow scalability with respect to the number of VMs multiplexed by the hypervisor.

6.3 PDES Scheduler Design

In PDES, since LPs (and consequently, VMs) can have widely differing event loads, they exhibit different ratios of simulation time to wall clock time. Event load imbalance can arise across VMs, which is not only inherent but also hard to predict due to its dynamic nature. Fundamentally, this dynamic, scenario-specific variation of the ratio of simulation time to wall clock time is the critical factor that must be accounted for in the design of the PDES-specific VM scheduler. However, in PDES we do know that the LP with the lowest value of local virtual time (LVT) affects the progress of its peers and hence the entire simulation application. Hence, if the LVT of the LP were used as the criterion in allocating processor time to VMs by the hypervisor (i.e., LPs with lower LVT values are prioritized over those with higher LVT values), then the runtime performance can be optimized. This can be achieved if the LPs running on different VMs are able to communicate their LVT values to the hypervisor, and the hypervisor in turn uses this information during the scheduling of VCPUs on to PCPUs. An additional aspect in relation to global virtual time computation also becomes an important design consideration.

6.3.1 PDES Hypervisor Scheduler Architecture

Figure 79 shows the system architecture of a hypervisor-based parallel computing environment with a scheduler optimized for PDES execution. For simplicity of explanation, let us assume that a single LP is hosted on each PDES federate and each VM has a single VCPU (note that this is *not* a requirement or a limitation of our system, but it simplifies understanding). As illustrated in Figure 79, the LVT of an LP is passed to the VCPU of its DOM. The scheduler that performs the task of multiplexing VCPUs onto PCPUs uses the VCPU-LVT and employs LLF scheduling. With a LLF order, the scheduler gives the highest priority to the VCPU with least VCPU-LVT value, as opposed to the default fair distribution of compute cycles across all the DOMs.

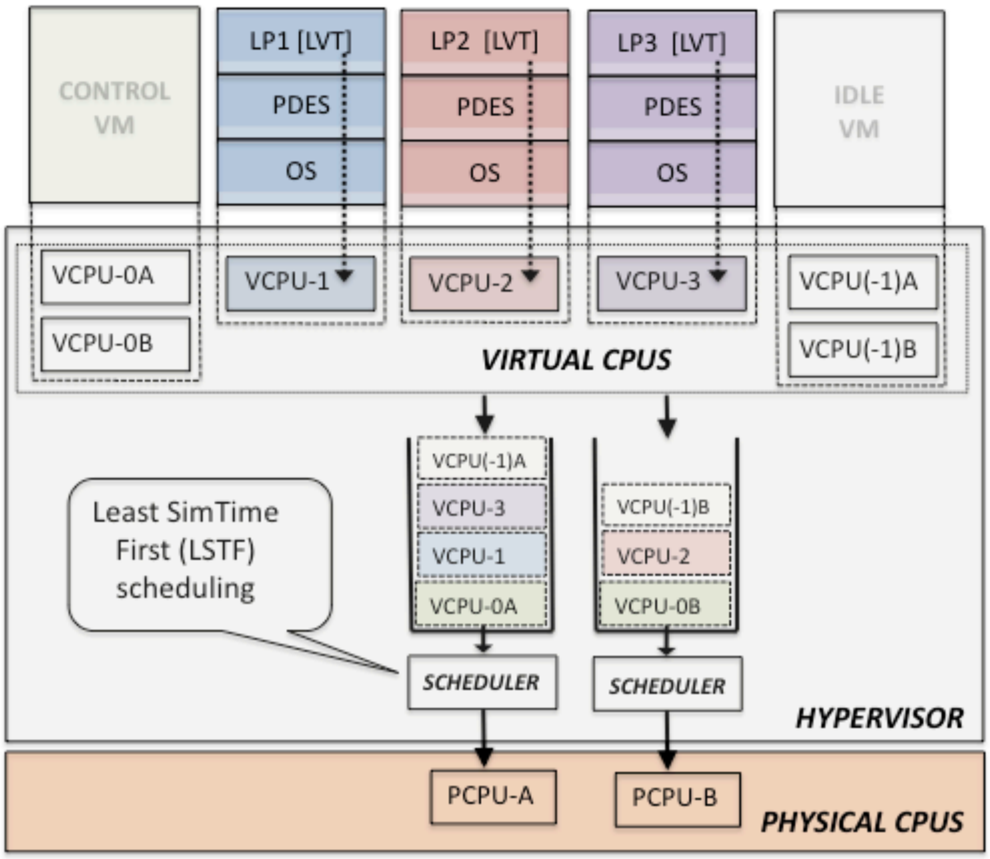


Figure 79 The design of the PDES-customized scheduler

However, the passing of LVT from the application to the hypervisor and LLF policy based scheduling, are not sufficient to ensure the scheduler execution. This is because the VCPUs with lower VCPU-LVT values (i.e., having a higher scheduling priority) would not allow the VCPUs with a higher VCPU-LVT to be chosen for scheduling. This results in blocking the GVT computation at the application level, as some of the LPs (with a higher LVT value) would never get a chance to respond during GVT computation.

Hence, a *deadlock* and *livelock* free algorithm needs to be designed to ensure proper working of the PDES scheduler. A preliminary version of this hypervisor scheduler design was discussed in [70] using a simplistic scheme of LVT toggling to overcome *deadlock* issues in brute force fashion.

Note that the special VMs (DOM0 and Idle-DOM) in Xen do not participate in the PDES simulation. The DOM0 is the privileged DOM, and the Idle-DOM is a Xen mechanism to ensure that the PCPU run-queues are never empty.

6.3.2 Deadlock-Free and Livelock-Free PDES Hypervisor Scheduler

```
Input: Just executed VCPU (VCPU$_\alpha$)
Output: Next VCPU to be scheduled for execution (VCPU$_\beta$)

Read shared_info to update VCPU$_\alpha$ LVT
Insert VCPU$_\alpha$ on to local PCPU runq
VCPU$_\beta$ = next VCPU from local PCPU runq
For all peer PCPU runqs do
  If (LVT of VCPU$_\beta$ > LVT of any VCPU$_\gamma$, in peer PCPU runq) then
    VCPU$_\beta$ = VCPU$_\gamma$
  end
end
```

Figure 80 LVT based hypervisor scheduler algorithm

In addition to event processing, the LPs also need to participate in periodic GVT computation. This periodic computation is necessary to consolidate the independent LVTs of each LP into a global GVT. With LLF scheduling as shown in algorithm in Figure 80, the federate with higher LVTs never get past the federate with lower LVTs and hence do not get any PCPU time to participate in GVT computation. Without successful GVT computation, LPs cannot determine safely processable events. Without a special consideration for GVT computations, a strict LLF based PSX does not allow completion of GVT computation, hence the PDES execution *deadlocks*.

6.3.2.1 Counter-based Algorithm to Resolve Deadlock

Deadlocks can be efficiently overcome using the *counter-based GVT_Threshold* algorithm. According to this algorithm, anytime a VCPU is inserted into the PCPU *runq* using *least-LVT first* principle, the *gvt_counter* variable of the peer VCPUs with higher-lvt and that are already in the *runq*, is incremented. If a VCPU's *gvt_counter* reaches *gt_threshold* during this process, it is picked up for scheduling regardless of its LVT value, by the scheduler. Doing this locally within each *runq*, we can get rid of the *deadlock* problem. Further, the *gvt_counter* of the selected VCPU is reset to 0, when it is picked up to schedule. The pseudo code of the algorithm in Figure 81 adds necessary logic to resolve *deadlock* in the LVT-based SMP scheduler algorithm in Figure 80.

```
Input: Just executed VCPU (VCPU$_\alpha$)
Output: Next VCPU to be scheduled for execution (VCPU$_\beta$)

Read shared_info to update VCPU$_\alpha$ LVT
Insert VCPU$_\alpha$ on to local PCPU runq
Increment gvt_counter for all VCPUs with greater LVT than (VCPU$_\alpha$)
VCPU$_\beta$ = NULL

For all VCPU$_\delta$ in local runq do
  If (VCPU$_\delta$ gvt_counter > gvt_threshold)
    VCPU$_\beta$ = VCPU$_\delta$
  End
End

If (VCPU$_\beta$ == NULL) then
  VCPU$_\beta$ = next VCPU from local PCPU runq
  For all peer PCPU runqs do
    If (LVT of VCPU$_\beta$ > LVT of any VCPU$_\gamma$, in peer PCPU runq) then
      VCPU$_\beta$ = VCPU$_\gamma$
      Break loop
    End
  End
End

VCPU$_\beta$.gvt_counter = 0
```

Figure 81 LVT based hypervisor algorithm to resolve deadlock

6.3.2.2 Counter-based Algorithm that Resolves Deadlock and Livelock

```
Input: Just executed VCPU (VCPU$_\alpha$)
Output: Next VCPU to be scheduled for execution (VCPU$_\beta$)

Read shared_info to update VCPU$_\alpha$ LVT
Insert VCPU$_\alpha$ on to local PCPU runq

/* start code to resolve deadlock */
Increment gvt_counter for all VCPUs with greater LVT than (VCPU$_\alpha$)
VCPU$_\beta$ = NULL

For all VCPU$_\delta$ in local runq do
  If (VCPU$_\delta$ gvt_counter > gvt_threshold)
    VCPU$_\beta$ = VCPU$_\delta$
  End
End
/* end code to resolve deadlock */

If (VCPU$_\beta$ == NULL) then
  VCPU$_\beta$ = next VCPU from local PCPU runq
  For all peer PCPU runqs do
    If (LVT of VCPU$_\beta$ > LVT of any VCPU$_\gamma$, in peer PCPU runq) then
      VCPU$_\beta$ = VCPU$_\gamma$
      Break loop
    End
    /* start code to resolve livelock */
    If (VCPU$_\beta$ !=  next VCPU from local PCPU runq) then
      Increment gvt_counter for all VCPUs in the local PCPU runq
    End
    /* end code to resolve livelock */
  End
End

/* reset gvt_counter of the selected VCPU -required to resolve both
deadlock and livelock*/
VCPU$_\beta$.gvt_counter = 0
```

Figure 82 LVT based hypervisor scheduler algorithm to resolve *deadlock* and *livelock*

However, algorithm in Figure 81 does not address the *livelock* problem. In an SMP scheduling using LLF policy, the lowest LVT VCPU is searched not only in local *runq* but also on the peer *runqs* of different PCPUs. The *livelock* problem persists because the *gvt_counter* of the VCPUs in local *runq* are left unaltered when the least-LVT VCPU is picked from the peer PCPU *runqs*. This essentially leads to the runtime scenarios, where in the lower LVT VCPUs are continuously exchanged among PCPU *runqs* without incrementing the *gvt_counter* and thus not allowing the higher-LVT VCPUs any cycles for GVT computation.

This problem can be resolved by considering the act of picking of lower-LVT VCPU from peer PCPU *runq* to be equivalent to an insertion in the local *runq*. In which case, the scheduler needs to increment the *gvt_counter* of all the existing VCPUs in its *runq*. The pseudocode for the least LVT first based SMP scheduling algorithm that resolves both *deadlock* and *livelock* problems is given by the algorithm shown in Figure 82.

6.3.3 Implementation Approach VM Environment for PDES Execution

To realize the PDES scheduler for Xen (PSX) we need to address two issues namely, (a) efficiently communicate the LVT of each LP (which is at the application layer) to the hypervisor, and (b) efficiently utilize this LVT information from within the hypervisor scheduler during scheduling, with minimal overheads (such as, avoiding locking-based synchronization for LVT value transfer from the VM to the hypervisor data structures).

6.3.3.1 Communicate LVT to Hypervisor

To communicate the LVT of an LP from the application layer to the hypervisor, the LVT must first pass from the user-space of the PDES process to the kernel-space of the guest-OS and then to the hypervisor data regions. One way to accomplish this is by adding a system call to the guest-OS to enable the transit of user-space data to kernel-space. However, to make this data accessible to the Xen hypervisor the guest kernel uses a shared memory page named *shared_info*, which is used by the Xen hypervisor through out its runtime to retrieve information about the global state [6]. The *shared_info* contains information that is dynamically updated as the system runs. In fact the Xen hypervisor uses the *shared_info* for time-keeping functionality of its para-virtual guest-OS. The LVT value from the guest-OS kernel-space is written into the *shared_info*, thus making it available to the hypervisor.

6.3.3.2 Hypervisor Scheduler Modifications to use LVT

Next, we need to implement the hypervisor scheduler that employs a LLF policy instead of the default credit-based fair scheduling strategy. Implementing the Xen hypervisor scheduler for the application-specific requirements has been previously accomplished and has been discussed elaborately in Chapter 2. Each PCPU maintains a *runq* (priority-queue) in which the VCPUs requiring clock-cycles are en-queued. The scheduler inserts the VCPUs into the PCPU *runqs* based on the LVT value. Hence, every VCPU of a DOM that hosts the LPs is required to maintain a variable (VCPU-LVT) representing LVT value of the LPs. Based on the VCPU-LVT value, the VCPUs are inserted in the *runq* of the PCPU. With a LLF policy, the VCPU that the scheduler picks for allotting PCPU cycles will have the lowest LVT among all its peers.

6.3.3.3 Specific Instrumentation to Accommodate GVT Computations

In addition to these two major requirements, it is also necessary to ensure that the LPs receive sufficient number of computational cycles to participate in GVT computation regardless of its LVT priority in relation to other LPs. The counter-based algorithm resolving *deadlock* and *livelock* are to be incorporated in the VCPU scheduling algorithm to accommodate GVT computations.

6.4 Implementation

To realize the PDES-optimized hypervisor scheduler, we require (a) each *μsik* kernel instance running on a VM to independently communicate its LVT value to the Xen scheduler, and (b) a new Xen hypervisor scheduler implementation that utilizes the communicated LVTs to optimize compute-resource sharing. These implementation details are described next.

6.4.1 Communicating LVT to Xen Scheduler

The mechanisms for communicating the simulation time from the simulation LPs at the user-level down to the scheduler data structures at the hypervisor level is conceptually trivial but implementation-wise non-trivial, especially to keep the runtime overheads low. The scheme involves modifications to the guest-OS kernel (Linux, in our test implementation), and corresponding modifications to the simulation engine (*μsik*, in our test implementation).

6.4.1.1 Linux Kernel Modifications

To send the LVT information from the application level, which is a *μsik* federate, we define and implement a new system call for the Linux® OS. This system call is invoked from within the simulation loop of the *μsik* library. This system call allows the LVT information to transit from user-space to kernel-space; once reaching the kernel-space, the LVT value is written into the *shared-info* data structure of the host VM. Thus the information is made accessible to the hypervisor at runtime.

```
struct shared_info {
    struct vcpu_info vcpu_info[MAX_VIRT_CPUS];
    unsigned long evtchn_pending[sizeof(unsigned long) * 8];
    unsigned long evtchn_mask[sizeof(unsigned long) * 8];
    uint32_t wc_version; /*Version counter: see vcpu_time_info_t.*/
    uint32_t wc_sec;          /*Secs  00:00:00 UTC, Jan 1, 1970.*/
    uint32_t wc_nsec;        /*Nsecs 00:00:00 UTC, Jan 1, 1970.*/
    uint32_t switch_scheduler;
    uint64_t simtime;
    struct arch_shared_info arch;
};
```

Figure 83 Modified *shared_info* data-structure

However, the para-virtual guest-OS kernel has to be re-built, after the addition of a new system call and incorporation of the changes to the *shared_info* data-structure (Figure 83). Two fields namely, *simtime* and *switch_scheduler* are added to the *shared_info* data-structure. Each guest-OS maintains a *shared_info* page, which is mapped on to memory by the hosting VM, during its creation. While *simtime* is used for holding the LVT value of the federate mapped on to the VM, the *switch_scheduler* is a flag, which indicates the switch between two different modes of the scheduler operation namely, *normal-mode* and *simulation-mode*.

Using the system call, the *μsik* federate writes the LVT to *simtime* of the *shared_info* along with a variable that either sets or unsets the *switch_scheduler* variable. The *switch_scheduler* in *shared_info* is set to suggest that PDES scheduler is in *simulation-mode*, and is maintained in this mode until the simulation ends. This flag is also used as an indication for the scheduler to read the LVT values from the *shared_info* of the DOMs into their VCPU and to use these values during scheduling. Note that the PDES federate running on the guest-OS simply updates the *shared_info* and is operationally independent of the *shared_info* variables usage by the hypervisor.

6.4.1.2 μsik Library Modifications

In order to communicate the LVT value from the μsik federate to the hypervisor scheduler, the μsik library was modified. It is required for the μsik library to indicate the *start* and the *end* of the PDES run to the hypervisor scheduler so that the scheduler can switch its mode of operation in accordance, from *normal-mode* to *simulation-mode* and back. During μsik's initialization, the *switch_scheduler* in *shared_info* of its host DOM is set to *true* using the custom *system call*. The scheduler reads this variable to change its mode of operation from *normal-mode* to *simulation-mode*. Similarly, during the termination of simulation the *switch_scheduler* is set *false* to revert back to its *normal-mode* of operation.

In μsik, the LPs hosted by the PDES federate are event-oriented, and during the simulation run, the LP with the least LVT is chosen by the federate for event processing. The *simtime* variable of the *shared_info* can always be kept updated to the LVT value of the recently processed event by the federate. However, we limit the number of writes to *shared_info* by updating it only when the subsequent changes in the federate LVT value are greater than the *lookahead* value.

Every μsik federate maintains a variety of simulation times based on its event processing state at any given moment. They are distinctly classified into four classes, namely, *committed*, *committable*, *processable* and *emittable* [16]. In practice, we observed that the use of the "*earliest-committable-time-stamp*" resulted in better performance than the others, and hence, this is the simulation time value used in all our experiments.

6.4.2 Xen Scheduler Implementation

The PDES Scheduler for Xen (PSX) scheduler replaces the default Credit Scheduler of Xen (CSX) in scheduling the virtual CPU (VCPUs) onto the physical cores of CPU (PCPU). The strategy that we take to replace the scheduler is similar to the one presented in [49].

6.4.2.1 PSX Data-structures

The *switch_sched* (corresponding to *switch_scheduler* in *shared_info*) is a field of global *ps_priv* global variable, which is an instance of *ps_private* data-structure (shown in Figure 84), and by default the value of *switch_sched* is *false* (*normal-mode*). The scheduler regularly checks the *shared_info* associated with the user-DOM of the VCPU it services. Hence, when the *switch_scheduler* value the *shared_info* of any user-DOM is updated, the scheduler reads it from the *shared_info*, and writes it to *switch_sched* field of *ps_priv* variable. The scheduler uses spin-locks in this process to avoid any un-desirable race conditions during its SMP execution. Each VCPU reads the LVT value from the *shared_info* into its *sim_time* variable. Figure 84, shows the *sim_time* and *switch_sched* variables in the PSX's VCPU and *ps_private* data-structures, respectively.

```
struct ps_vcpu
{
    struct list_head runq_elem;
    struct list_head active_vcpu_elem;
    struct nw_dom *sdom;
    struct vcpu *vcpu;
    uint64_t sim_time;
    s_time_t sim_time;
    atomic_t gvt_counter;
    int switch_sched;
    ...
};

struct ps_private
{
    spinlock_t lock;
    struct list_head active_sdom;
    int switch_sched;
    uint32_t gvt_threshold;
    ...
};
```

Figure 84 VCPU and ps_private global data-structures in PSX

6.4.2.2 Scheduling in Normal-mode

The scheduler is referred to be in *normal-mode* if the *switch_sched* (*ps_private* data-structure Figure 84) is false. This corresponds to the mode in which the VMs are booted and operational, but no PDES run has been started (and hence LVT-based scheduling is undefined). PSX by default maintains the *sim_time* (VCPU data-structure Figure 84) of all DOM0 VCPUs lower than all the DOMUs. In the *normal-mode* all the DOMU VCPUs will have their *sim_time* initialized to 1, while DOM0 VCPUs have their *sim_time*s initialized to 0. Only after the *switch_sched* is set true by PDES federate the *sim_time* value of the relevant VCPU is updated after reading the *shared_info*. However, the *sim_time* of VCPUs of DOM0 continues to be 0 even after switching to *simulation-mode*.

Note that the *sim_time* corresponding to the VCPUs of the DOM0 is always maintained to be lower than that of other VCPUs regardless of the PSX's mode of operation. This guarantees that DOM0 VCPUs are always preferred over the other VCPUs, which in turn ensures better performance during inter-DOM communications (as all the virtual network traffic passes through DOM0) and a responsive user-interactivity with DOM0 during simulation execution.

6.4.2.3 Scheduling in Simulation-mode

The hypervisor switches to the simulation mode after the PDES execution is started on all the VMs. Each PCPU maintains a *runq* (priority-queue) as shown in Figure 85, and in the *simulation-mode* PSX en-queues the VCPUs to be scheduled in a prescribed priority.

```
struct ps_pcpu
{
    struct list_head runq;
    struct timer ticker;
     ...
};
```

Figure 85 PSX physical CPU-core specific data-structure maintained by PSX

We use the LVT value as the VCPU priority – the lower the *sim_time* (VCPU data-structure Figure 84), the higher is its priority in the *runq*, and hence the earlier it is picked by PSX to allocate compute resource. Every PCPU schedules itself for every *tick* using the timer named *ticker*. The PCPU performs accounting for the VCPU currently being serviced by incrementing the *vcpu_ticks*, and updating the *sim_time* by reading the *shared_info*. The PCPU also generates a *schedule* interrupt for the VCPU being serviced on a less loaded PCPU. During scheduling the SMP scheduler enqueues the VCPU being serviced and picks the VCPU with least *sim_time* across all PCPU *runq*s to service. Our implementation of the scheduler allots a *tick* size (1ms) of PCPU time for the VCPU picked to service.

6.5 Performance Evaluation

6.5.1 Hardware and Software

We use the same custom-built Local Test Platform (LTP) that was used for PDES performance evaluation in the Chapter 5. A 1ms tick size was used with both CSX and PSX schedulers. Three PDES applications using µsik, namely, PHOLD (a synthetic PDES application generally used for performance evaluation), Disease Spread Simulation and SCATTER (a reverse-computation-based vehicular traffic PDES application) are used in our performance studies.

6.5.2 Performance Expectations with CSX

6.5.2.1 <u>Performance Deterioration CSX on PDES for Multiplexing Ratio > 1</u>

To demonstrate the hypervisor scheduler effects on PDES performance, we executed the synthetic PDES PHOLD benchmark for a wide range of application parameter settings. To demonstrate the effects of VCPU scheduling on the PDES application performance, we launch single VCPU VMs equal to remnant PCPUs (30) and increase the number of VMs hosted until the number of VCPUs become 10% greater than number of PCPUs. The Figure 86 plots the runtimes for varying PDES loads for a mere 10% increase in the number of hosted VCPUs.

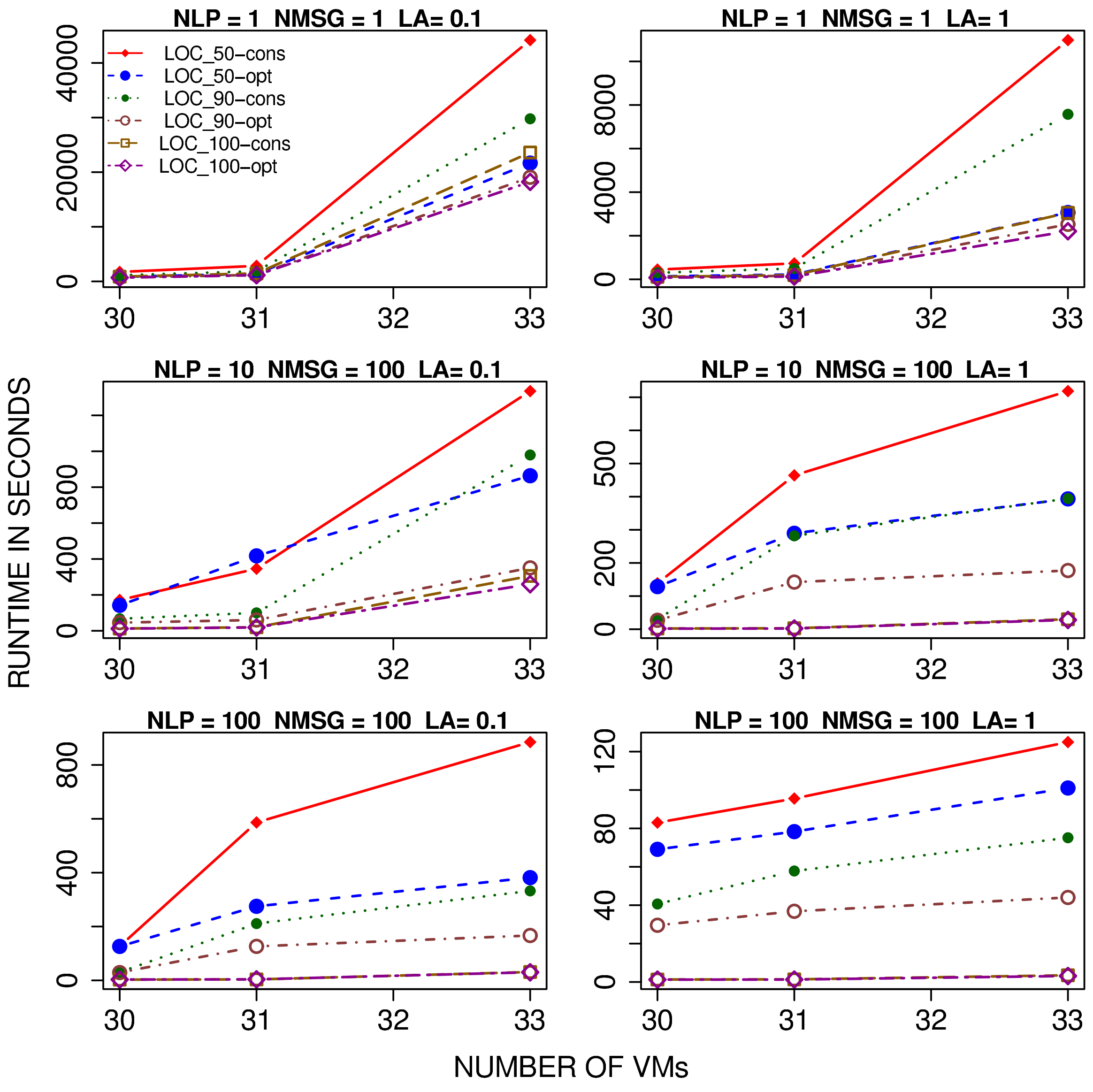


Figure 86 Drastic increase in runtime with CSX just beyond multiplexing ratio

The top two graphs in Figure 86 plot performance runs with lowest possible computational load for varying communication loads and for varying lookaheads 0.1 (left) and 1.0 (right), these show the effects of *fair-share* based VCPU scheduling in absence of significant computational load. The plots show a several orders-of-magnitude of degradation in performance with negligible increase in load. These readings constitute some of the worst possible performance that can be expected on a cloud platform.

The bottom two graphs in Figure 86 plot performance runs with highest possible computational load for varying communication loads and for varying lookaheads of 0.1 (left) and 1.0 (right). This set of readings represents one of the best performances that PDES applications can expect on a cloud platform. Yet, based on the communication load the performance varies significantly, at worst it is closer to an order-of-magnitude for LOC=50% and low lookahead of 0.1.

The center two graphs in Figure 86 plot average computational load for varying communication loads and for varying lookaheads 0.1 (left) and 1.0 (right). These set of readings can be considered representative of an average PDES application behavior. With the performance degrading by several folds with increase in VMs, especially with increase in the communication load highlights the impact of scheduling on PDES application performance.

6.5.2.2 PHOLD Performance with Reduced Tick-size

The time-slice provided for each VCPU during scheduling affects the performance of the PDES application. Altering, the time-slice values are made easier in the recent releases of Xen hypervisor using the *xl* tool. By default, CSX provides a time-slice of 30ms in quantum of 10ms tick-size for each scheduled VCPU. The time-slice can at most be reduced to 1ms using the *xl* tool. Figure 87 compares the runtimes of PHOLD benchmark scenario (NLP=100, NMSG=100, LOC=95, LA=0.1 and endtime=1e3) for CSX with default time-slice with CSX with 1*ms* time-slice. As seen in the Figure 87, the conservative synchronization performs extremely well with reduced time-slice as evident in 128 VM and 64 VM scenarios. Close to an order-of-magnitude performance gain is in 128 VM scenario. However, the same is not true while using the optimistic synchronization case. In the 128 VM scenario using optimistic synchronization 1ms time-slice makes no difference in runtime when compared to default time-slice, further the performance suffers very badly in 64 VM scenario. This is because high number of reversals (tens of millions) in case of 1ms time-slice runs compared to lower (few hundred thousands) number of reversals, while using default time-slice. In the absence of high reversals optimistic is expected to perform better than conservative. Hence, in all the following performance runs using CSX a 1ms time-slice is used.

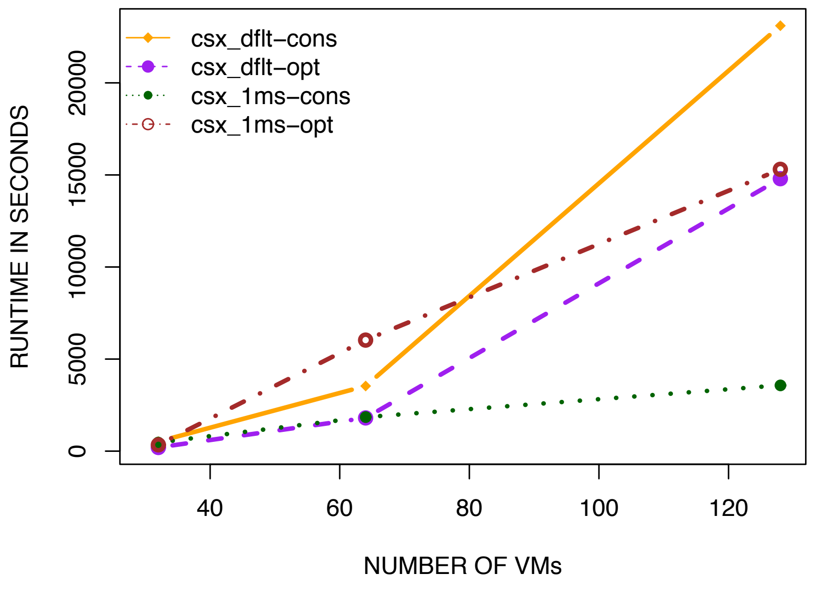


Figure 87 Runtime performance of PHOLD for varying time slices

6.5.3 Performance Comparison with PHOLD Benchmarks

For this set of performance results we used a PHOLD scenario with NLP=100, NMSG=100, LOC=95 and we hosted one μsik Federate on a VM. NLP=100, ensured that each VM/Federate hosted 100 LPs. The NMSG=100, ensured that each LP exchanged 100 messages amongst its peers. Thus at any instance in a simulation scenario with 128 VMs exchanged 1.28 million messages among 12800 LPs. The locality (LOC) was set to 95% suggesting 95% of the randomly generated messages were local, while the 5% were sent to a random peer LP hosted on other VMs. Locality of 95% ensured that the Federate had enough local events to process at any instant, hence the affect of scheduling was minimal as observed in Figure 86. For all the PSX runs, a value of 10 was assigned to GVT_Threshold (GT).

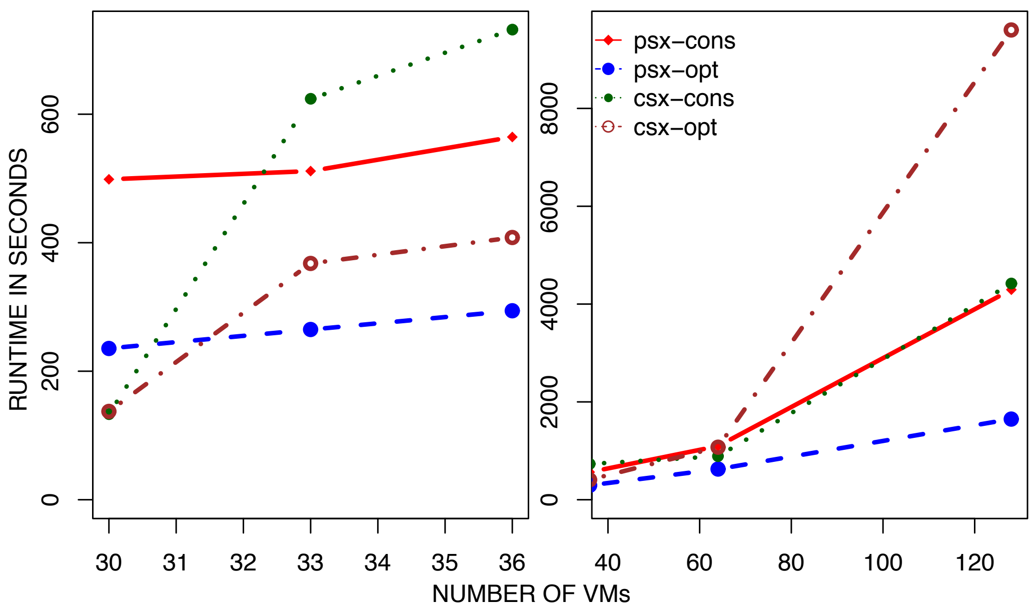


Figure 88 PSX and CSX comparison PHOLD with LA=1

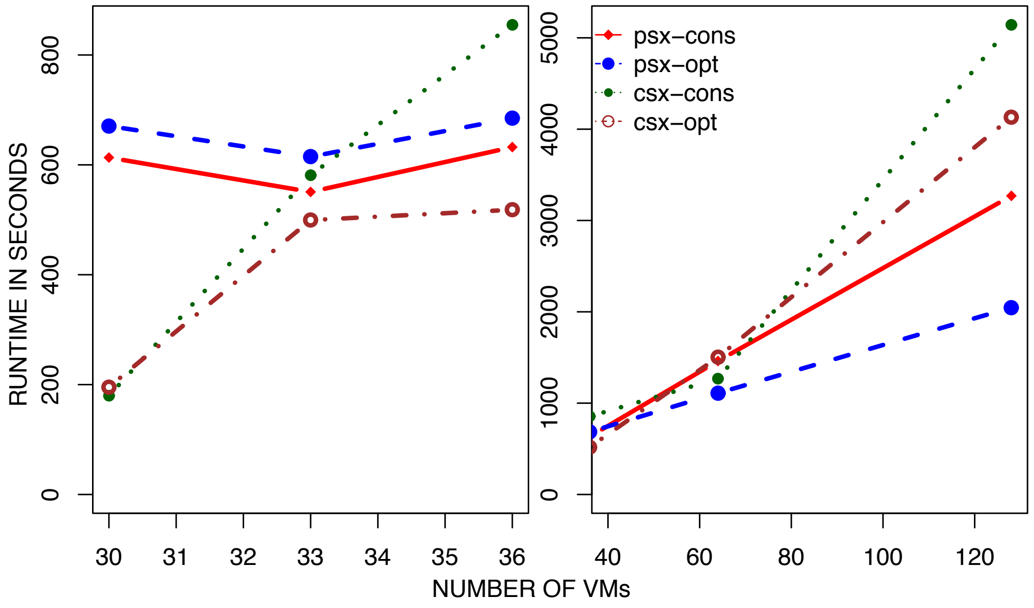


Figure 89 PSX and CSX comparison PHOLD with LA=0.1

Figure 88 plots the conservative and optimistic curves for lookahead 1. As the number of VMs hosted on the physical machine increases both optimistic and conservative runtimes of PSX perform very well in comparison with their CSX counterparts. The conservative runs of CSX perform extremely well in comparison with the CSX optimistic runs. This is expected because of 95% locality and a higher lookahead ensures sufficient local events in between GVT synchronization, while CSX optimistic suffers due lot of reversals. The total number of reversals is over 6 million and 8 million in 64 and 128 VM scenarios, respectively).

Figure 89 plots the conservative and optimistic curves for lookahead 0.1. While the PSX using conservative synchronization also suffers along with CSX runs (both conservative and optimistic synchronizations), the PSX using optimistic synchronization performs well with increase in number of VMs.

The plots on left in both Figure 88 and Figure 89 are magnifications of the initial set of data-points. They plot the behavior of PSX with respect to CSX when the multiplexing ratio of VCPUs on to PCPUs is low. As observed, CSX performs best when no mismatch between PCPUs and VCPUs exist and suffers significantly even due to a slight mismatch. On contrary, PSX suffers in the absence of mismatch due to unnecessary overhead of writing LVTs to hypervisor and performs better than CSX as the mismatch grows, as expected.

6.5.4 Performance Comparison with Disease Spread Benchmarks

The μsik Federates mapped to the regions and the LPs are mapped to the locations in the DSB scenarios. Each region (μsik Federate) hosted multiple locations (LPs). Each region or Federate was hosted on a VM. Here, the DSB scenario comprised 100 locations per region and a population of 100 per each location. Thus in a scenario with 128 VMs, we simulated the disease spread across a population of 1,280,000 people spread across 128 regions, each with 100 localities. The simulation scenario simulated the disease spread among the population over a week. Two sets of runs were performed based on the movement limitations of the population across regions. The first set limited the movement of population to 10% (LOC=90%), while the second set limited it to 50% (LOC=50%).

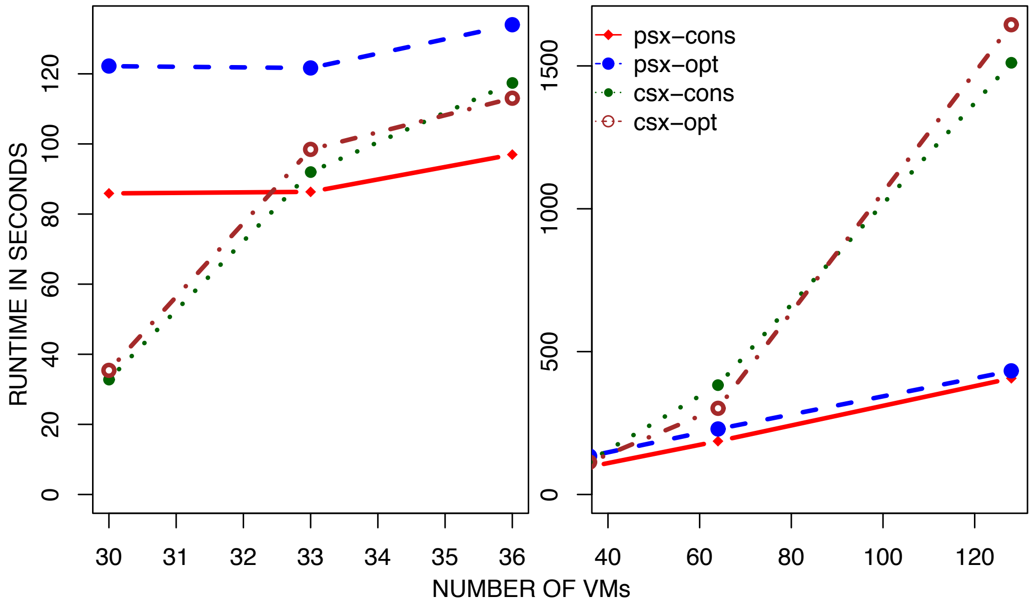


Figure 90 PSX and CSX performance comparison DSB with LOC=90

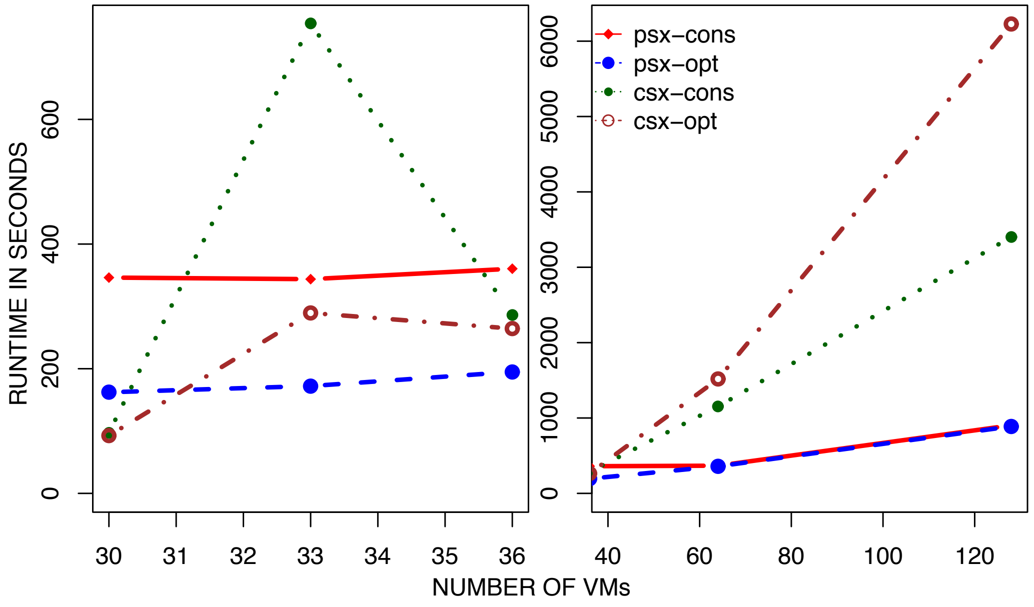


Figure 91 PSX and CSX performance comparison DSB with LOC=50

Figure 90 and Figure 91 plot the runtime curves for the first set and second set, respectively. Both optimistic and conservative runs with PSX scheduler perform extremely well as the number of VMs used in the experimental setup increases. As expected a better performance is observed in LOC=50% (Figure 91 scenario where interaction between across VMs is higher and consequently optimistic synchronization performs slightly better when compared to the LOC=90% (Figure 90) scenario.

Similar, to the previous PHOLD plots, the left hand plots of Figure 90 and Figure 91 show the performance of PSX when the multiplexing ratios of VCPUs on to PCPUs are low.

6.5.5 Performance Comparison with SCATTER Benchmarks

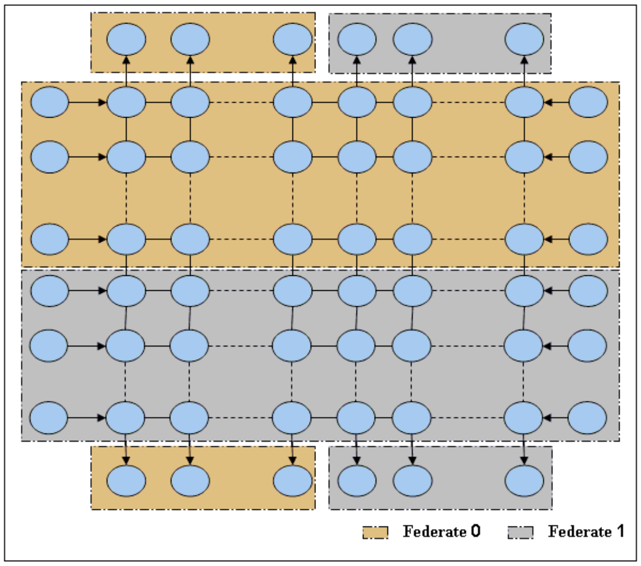


Figure 92 Road network layout

As opposed to the two prior benchmarks that exemplify weak-scaling (increase in computational load with increase in number of VMs), this benchmark evaluates strong-scaling (computational load remained same across all scenarios, which varied in terms of number of VMs used) behavior. The SCATTER benchmark simulated the vehicular traffic evacuation scenario of 3.2 million vehicles originating from 256 sources. Each vehicle made its way across 128×128 (16K) grid of intersections toward its destination (one of the 256 sinks), using the djkstra's shortest path algorithm. The vehicles were generated at the source at a rate of 50 vehicles/sink/hour for an hour. Vehicles were injected through source-nodes placed on either side (left and right) and they moved toward the sink-nodes (top and bottom) of the 128×128 grid. The same simulation scenario was executed on 32, 64 and 128 VMs. The intersections, sources and sinks were modeled as LPs of PDES. The spatial decomposition ensured that equal number of intersection LPs, source LPs and sink LPs were allotted to each μsik Federate hosted on a VM, as shown in Figure 92. The color-coding show the distribution scheme of LPs (i.e. intersections, sources and sinks) on to two μsik Federates.

The corresponding performance plots are presented in Figure 93. The optimistic curve of PSX remains almost same with the increase in number of VMs, the PSX conservative also shows similar trend except when number of VMs hosted is 128, where its runtime slightly increases. In comparison the CSX runtime curves suffer as the number of VMs hosted increases. However, the CSX using optimistic synchronization is able to curtail the performance degradation significantly in comparison with its conservative synchronization.

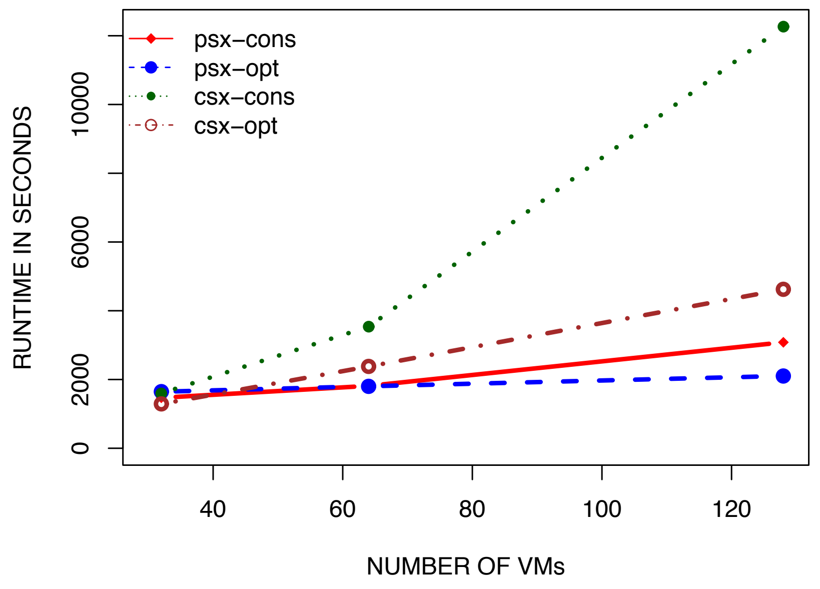


Figure 93 PSX and CSX performance for SCATTER vehicular traffic simulation

6.5.6 Performance Relative to Native Linux

In this section, we compare the performance benchmarks run on VM platform with native Linux platform, on the same hardware device. For fair comparison, the number of µsik Federates (processes) equivalent to number of VMs used is spawned on Linux. The executions involving 128 VMs using PHOLD, DSB and SCATTER were used for comparison. The best runtime regardless of the PDES synchronization scheme was used for comparison. In particular, from the PHOLD benchmark we used 0.1 lookahead readings and from DSB benchmark runs we used LOC=50 readings. This is just because runtimes of respective PHOLD and DSB are higher in the considered scenarios.

As seen in Figure 94 the PHOLD benchmark runtime on Linux using 128 processes is several order-of-magnitude faster than results from CSX and PSX. Though PSX is able to alleviate the performance degradation to a certain extent it still is inefficient because the PHOLD benchmark has very low computational load and very high communication load due to low lookahead (0.1) requiring frequent synchronization. This is despite of optimistic synchronization trying its best to keep the runtime lower. However, the native runs are almost over an order of magnitude faster than VM environment using PSX or CSX.

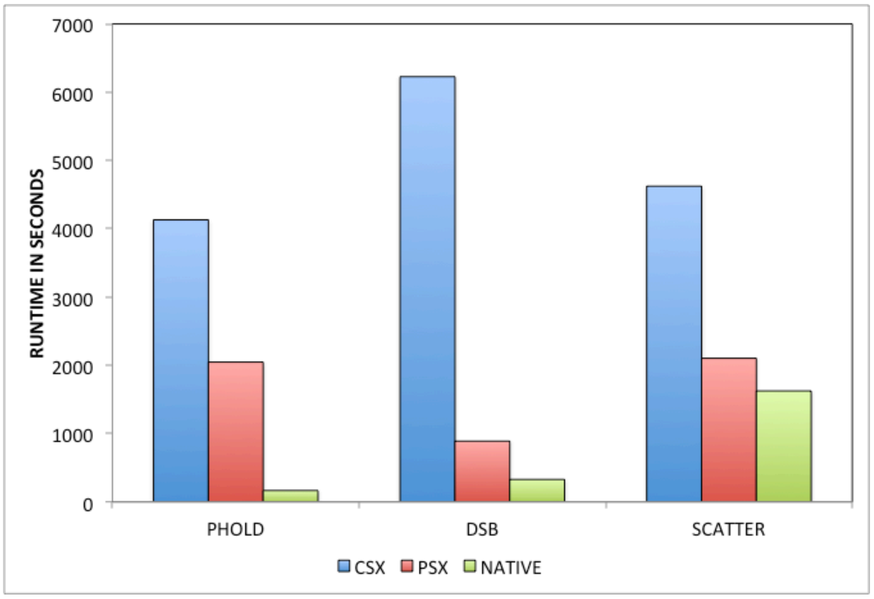


Figure 94 PSX, CSX and Native performance with PHOLD, DSB and SCATTER

For the DSB benchmark, the best runtime with CSX is extremely bad, however PSX has been able to significantly boost the performance of DSB benchmark bringing it closer to the native Linux performance. The DSB benchmark has higher computational load in comparison with PHOLD, although the communication load (LOC=50) is higher, with efficient scheduling both conservative and optimistic synchronizations perform very well.

For the Scatter benchmark, PSX performs extremely well. While CSX is only few times slower than native Linux runtime, PSX is very close to the native runtime performance. This is because Scatter scenario is computationally intensive, very well load-balanced and optimistic synchronization with zero-rollbacks yields very good performance and PSX with its LVT based scheduling further betters the performance.

6.5.7 Variance in Performance

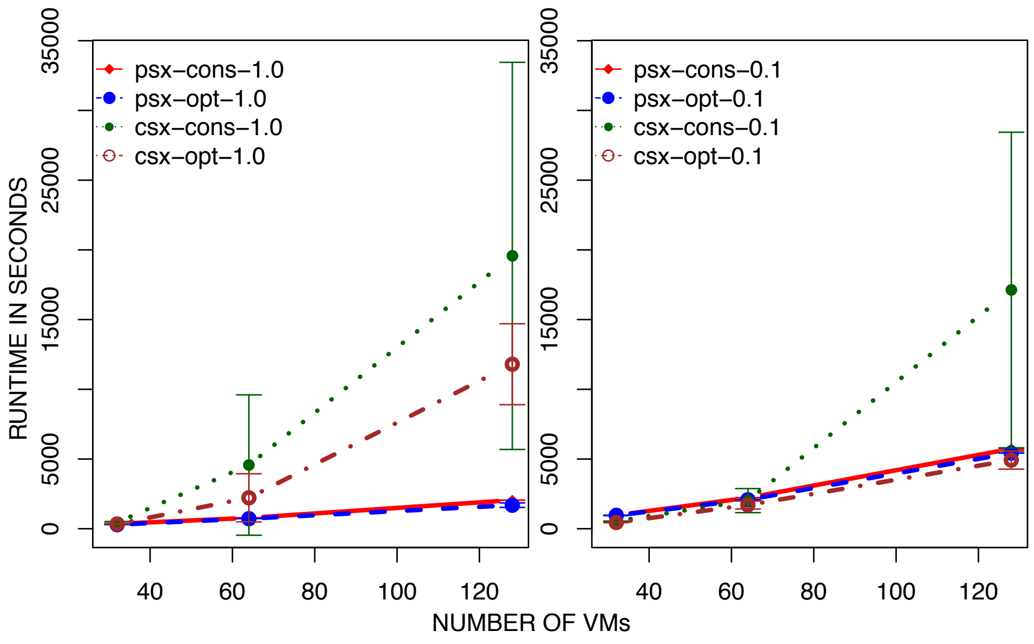


Figure 95 PSX and CSX runtime variance with PHOLD for lookaheads 1 and 0.1

The CSX runtime show high variance when the number of VCPUs multiplexed was greater than the number of PCPUs. Figure 95 plots the PSX and CSX runtimes with

95% confidence intervals for PHOLD benchmark with lookaheads 1(left) and 0.1(right). In our previous plots we have used the best runtimes obtained using CSX plots from multiple runs for comparison with the runtimes of PSX. At worst the CSX readings are several times (more than 5) slower than the readings plotted. This behavior can be expected from CSX as a result of the strategy it uses for scheduling VCPUs. In contrast to CSX the PSX readings very low variance regardless of the number of VMs multiplexed by the hypervisor.

6.6 Summary

With the proliferation of Cloud and VM-based platforms for parallel computing, it is now possible to execute parallel discrete event simulations (PDES) over multiple VMs, in contrast to executing in native mode directly over hardware as has been traditionally done over the past decades. However, while most VM-based platforms are optimized for general workloads, PDES execution exhibits unique dynamics significantly different from other workloads.

In this Chapter we presented the results that identify the gross deterioration of the runtime performance of VM-based PDES simulations when executed using traditional VM schedulers, quantitatively showing the bad scaling properties of the scheduler as the number of VMs is increased. The mismatch is fundamental in nature in the sense that any fairness-based VM scheduler implementation would exhibit this mismatch with PDES runs.

To overcome the mismatch, a new algorithm was presented for PDES-specific scheduling of VMs by a hypervisor. The algorithm schedules VMs primarily by their local virtual time (LVT) order, and incorporates mechanisms that prevent *deadlock* and *livelocks* that are otherwise possible in a purely LVT-based scheduling. The new scheduler has been implemented and exercised in an actual hypervisor system (Xen) that is popularly used in major Cloud platforms worldwide. Experimental results have been documented from detailed experiments with multiple discrete event models over a range of scenarios (with different lookahead values, inter-processor event exchange frequencies, and conservative and optimistic synchronization), all of which show (a) the high variability and sub-optimality of the default credit-based VM scheduler that is PDES-agnostic, and (b) the well-behaved scalability and significantly faster execution of our new algorithm.

CHAPTER 7

CONCLUSION AND FUTURE DIRECTIONS

7.1 Conclusion

Starting with the definitions of VM and VTS, we identified two broad fields, wherein their combined symbiotic functioning holds a greater promise for the simulation community. We referred to the two fields as (a) VM within VTS and (b) VTS over VM. While the former strongly holds a promising future in realizing high-fidelity cyber infrastructural and cyber physical simulations, the latter has an even wider applicability, suggesting a paradigm shift in the execution platform for discrete event simulations as a whole.

7.1.1 VM within VTS

We systematically addressed the challenges in utilizing the VMs for cyber simulations, from the ground up. We started the discussion with the classification of emulators and simulators as end-host centric and network-centric. We introduced the current state-of-the-art VM based network simulators and emulators and a nomenclature to identify and categorize them. The core conceptual issues falling under the two topics, namely, (a) virtual time-order execution of VMs and (b) virtual time-order network control, were discussed. A prototype for VM-based network simulation was designed and developed to address the conceptual issues in an actual implementation. The prototype implementation to realize virtual time-ordered execution of VM required accounting, maintenance and synchronization of virtual time across the VMs, and a replacement of conventional hypervisor scheduler with a VTS-aware scheduler. Similarly, the prototype implementation of the virtual network control involved new methodology to trap and control the inter-VM communication.

The requirement of virtual time-ordered execution for correct simulation results was uncovered, and the requirement of virtual time-ordered execution was demonstrated. Specific performance benchmarks were designed and a methodology to quantify the simulation errors was devised. With our prototype implementation of NetWarp and the benchmarks, we categorically demonstrated that the simulations using default VM platforms result in virtual time-order errors, and also demonstrated that such errors can be controlled and made arbitrarily small using our NetWarp system. Further, we achieved this without any appreciable loss in runtime performance.

Detailed scaling study and scaling-relevant instrumentation of NetWarp was performed. We also developed a highly efficient virtual network control called Netwarp Network Control (NNC) and, several new synthetic and real-life application benchmarks. We demonstrated a good scaling, efficient runtime performance and greater time-order error control of NetWarp with simulation scenarios involving 64 and 128 VMs on a 12-core physical machines. We also demonstrated the efficient capability of NNC using actual application codes.

This scaling work was followed by a detailed case study of simulating a complex physical phenomenon of MANET on NetWarp. Instrumentations at application level to support features, such as virtual mobility, virtual ad-hoc network creation, etc., were performed to make NetWarp support MANET simulations. With the benchmarks, we demonstrated the correctness of the simulation results obtained from NetWarp. We also tested the supported features using the benchmark applications.

The prototypes, benchmark applications, performance runs and the corresponding results enable us to convincingly state that virtual time awareness is necessary and sufficient for correct and efficient execution of VM based systems.

7.1.2 VTS over VM

Cloud computing services provide several desirable features such as ease of resource accessibility, runtime migration capability for dynamic load-balancing, support to fault tolerance, and maintenance-free resource usability. These features are useful to exploit in the execution of VTS systems. To this end, we conducted a thorough performance study of PDES execution on VM platforms. With this performance study, we addressed several critical questions regarding the performance of PDES on VM platforms. We demonstrated that VM-based executions could be as fast as native execution; a privileged VM execution yielded the same performance as that of an unprivileged VM, and the virtual network performance remained unaffected with increase in DOM0 VM weights. We also observed and highlighted the counterintuitive behavior of the VM platforms for PDES applications that had profound effect on the cost-value aspect of the Cloud service. Based on the performance study results, we provided recommendations and guidelines to the Cloud service user for PDES loads.

We found that the runtime performance of VM platform deteriorates as the multiplexing ratio of VCPU to PCPU exceeds unity. In reasoning the cause for poor runtime performance with increase in multiplexing ratio, we identified the hypervisor scheduling policy mismatch with the PDES runtime execution dynamics as the underlying cause. To address this performance issue, we designed a new, virtual time-aware hypervisor scheduler that allotted processor resources in coherence with the PDES runtime dynamics. We devised a robust scheduling algorithm for this purpose, which is built on a Least-LVT-First scheduling policy, with additional algorithmic components to handle deadlock and livelock conditions. We tested the runtime performance of VM execution platform with the new virtual time-aware simulation specific scheduler (PSX), with the default VM scheduler (CSX) using several PDES applications, such as PHOLD, disease spread simulations and vehicular traffic simulations. We demonstrated excellent runtime performance and scaling behavior of PDES applications with PSX in comparison to CSX using multiple simulation applications with varying scenario configurations. We also compared the runtime performance of the application benchmarks from PSX and CSX, with the native Linux platform and demonstrated that PSX runtime performance always performed significantly better than CSX and approached the performance of the native execution runtimes.

7.1.3 Concluding Remarks

In conclusion, we list highlights of this research findings.

- Virtual time-order VM execution and virtual time-order network control are the two necessary and sufficient conditions that ensure correctness of the simulation results, in the VM-based network simulators.
- Virtual time-order execution of VMs can be realized by changing the hypervisor scheduler’s VCPU scheduling policy.
- Virtual time-order supporting hypervisor scheduler can be realized without incurring any additional performance penalty.

- Fairness based hypervisor schedulers are inappropriate for VM-based network simulations.
- Virtual time-order errors can be measured and quantified on any VM-based network simulation/emulation platform.
- By varying the time-slice allotted to the virtual CPUs during hypervisor scheduling the virtual time-order errors can be controlled.
- It is possible to realize virtual time-order executions of multi-core VMs (with varying number of cores) for VM-based network simulations.
- It is possible to obtain correct results from a VM-based network simulation, even if the physical hardware platform is highly overloaded with VMs.
- The PDES simulations using VM execution platform with perfectly matched virtual and physical compute resources yield same runtime performance as a native platform. As even the virtual compute resources start to outnumber the physical resources the PDES runtime performance quickly deteriorates.
- PDES performance on a VM execution platform is agnostic to the privileges of the VM.
- A high-cost VM from a Cloud service might not necessarily yield better PDES performance and similarly, low-cost VMs can yield better performance.
- Reduction of time-slices in default VM scheduler most often yields in better PDES performance over VM execution platforms.
- The virtual-CPU scheduling policy (fairness-based) mismatch with the PDES runtime execution dynamics is the reason for poor runtime performance of PDES applications.

- Virtual time-aware scheduling policy is necessary and sufficient to ensure better runtime performance of PDES simulation workloads, over VM execution platforms.
- Least-LVT-First based scheduling policy in a virtual time-aware VM execution platform for PDES workloads, is prone to deadlock and livelock conditions, which has to be overcome.

7.2 Future Directions

7.2.1 Outstanding Issues of Least-LVT First (LLF) Scheduling

One common principle of the schedulers of both VM within VTS and VTS over VM paradigms is the LLF strategy used for scheduling. In the multi-core environments, based on the number of PCPUs supported by the hardware, equal number of VCPU run-queues is maintained. After the exhaustion of allotted time-slice of a PCPU and while assigning the next VCPU to the PCPU, the scheduler reads the least-LVT VCPU from the local PCPU run-queue and looks for a lower LVT VCPU than local least LVT VCPU in the peer PCPU run-queues. The very first lower LVT VCPU found while searching peer run-queues is picked next. If no lower LVT VCPU is found in other run-queues, prior selected VCPU from the local PCPU queue is picked.

The search for the lower LVT VCPU starts from the local-queue and moves along with its neighbor in a circular fashion. Although the incidence of situations wherein, large number of VCPUs with lower LVTs are left behind while the one with higher LVT is picked is very less, nevertheless, the LLF policy implementation is not pure. Hence, the hypervisor schedulers used in both paradigms are not pure LLF implementations, as there could always be one another VCPU with lower LVT in other run queues than the one picked for scheduling.

One other approach we tried was to insert lower LVT VCPU (obtained during global search) into the local run queue instead of picking it for scheduling. Hence, after searching all run queues the least-LVT VCPU from the local run queue is picked for scheduling. However, this approach does not eliminate the possibility of existence of lower LVT VCPU in the run queues. This is because while searching for other VCPUs in the peer run queues, the local run queue has to be locked, and the peer run queue where a lower LVT VCPU is found must also be locked. Further, when the scheduler is not able to lock a peer run queue it moves on to the next. This essentially leaves the problem of leaving lower LVT behind in a peer run queue while picking up higher LVT VCPU unaddressed. The better among these techniques needs to be studied and reasoned. However, pure LLF scheduling algorithm for hypervisor scheduler executing over a multi-core physical hardware is still open for research.

7.2.2 Outstanding Issues in NetWarp Network Control (NNC)

NNC is a multithreaded application, which along with *iptables* captures network packets in transit and allows simulation modeler to enforce desired network characteristics in the network simulations. The *iptables* rules capture the transiting packet and put them in a *netfilterqueue*.

The initial packet processing (IPP) thread determines the *emit time* from NNC and pushes them into next *netfilterqueue* for further processing. To determine the *emit time* the IPP thread requires the *arrival time*, which is considered as the virtual time when the packet is pulled out of the first *netfilterqueue*. Although the thread is continuously processing the packets arrived and even though the processing is light (just involves determination the *emit time*), the timing errors can occur. This is because the NNC synchronizes the VCPU LVTs of all the VMs as necessary and if the packet were being processed just after synchronization of VCPU LVTs, there would be a significant difference in the *actual-arrive-time* of a packet and the *read virtual time*. Hence, the *emit time* calculated based on such *arrival time* will also be erroneous.

Hence, to reduce the error one needs to ensure that the virtual time of the packet on its arrival is recorded. This can be done using an additional *netfilterqueue* and an additional thread that just continuously stamps the incoming packets with the virtual time of the packet's arrival. This packet is then sent to the next *netfilterqueue* that continuously determines the *emit time* using the *arrival time* stamped on the packet.

7.2.3 NetWarp Simulator Spanning Multiple Physical Hosts

In the VM within VTS, all the discussions and experiments pertaining to VM based network simulations was limited to a single physical machine that hosted multiple, multi-core VMs. However, additional work is necessary to realize large-scale network simulations spanning many physical machines. Outstanding problem is to maintain a single simulation time line across multiple hypervisors on physically separate hardware. Although synchronization algorithms from parallel discrete event simulations can be utilized for this purpose. The synchronized advancement of VMs dispersed across multiple physical systems is challenging.

7.2.4 Virtual Time Communication to Lower-Level Protocols

In NetWarp although each VM is provided cycles based on the virtual time and progress in virtual time, the lower-level protocols such as, TCP/IP used currently utilize the wall clock time. We need to make these lower-level protocols use virtual time instead of wall clock time. The process of making the application and lower-level protocols adhere to virtual time is not complex and has been previously published. However, the functioning of such model where, the protocols use the virtual time that does not evolve continuously, might give way to some interesting problems.

7.2.5 Real life Application Performance Evaluation

NetWarp simulation environment has been extensively tested using the designed synthetic benchmarks. Although, some benchmarks such as cyber security benchmarks were realistic, they were not out-of-box commercial or real-life distributed applications. We need evaluate the NetWarp with scenarios using real-life applications.

7.2.6 Outstanding Issues regarding CPU-Pools

Our implementation of PSX scheduler on the new version of Xen-4.2.0, built over CPU-Pool functionality. Though our scheduler uses CPU-Pools all the resources PCPU resources are maintained in a default Pool-0. We can currently boot with our scheduler, create additional CPU-Pools using different schedulers, remove PCPUs from one CPU-Pool and add CPU-Pools to other. However, additional work to resolve the failure when DOMUs are created on particular CPU-Pool, is required.

The CPU-Pools is an important feature that needs to be supported because it is the gateway for porting the currently developed, and future hypervisor schedulers on to Cloud resources for simulation purposes. With this feature, the scheduler of the hypervisor can be altered at the runtime for certain pools of PCPU cores. This can be easily achieved, if a service option to choose the hypervisor scheduler is provided by the Cloud service provider to the user.

7.2.7 Scheduler Classification based on Interrelation between PDES Processes

Table 5 shows the set of possible scheduling policies that a physical hardware can host concurrently using CPU-Pools. In the Table 5 four categories of scheduling policies are mentioned.

If the processes hosted on VMs are both virtual time constrained, as in the general case in PDES simulations that we have discussed, then the LLF scheduling scheme, as we know works best.

If suppose, the PDES LP is constraining but not exactly constrained by its peer process hosted on a different VM, as in the case of the LP which is not constrained by the output/display process but is constraining the display process. A different scheduling scheme needs to be adopted for optimal performance.

Another possibility is that a process hosted on VM is constraining the PDES LP but not constrained by PDES LP. For example, the real-time concurrent input feeding process, which is not constrained by PDES LP but actually is constraining PDES LP. An optimal scheduling policy needs to be designed to accommodate such requirement.

Table 5 Possible scheduling policies in PDES simulations

PEER / SELF	Virtual time constrained	Virtual time constraining but not constrained
Virtual time constrained	LP with respect to its peer LP	LP with respect to output/display process
Virtual time constraining but not constrained	Concurrent input process with respect to LP	Default fair-share scheduling (credit) behavior

Finally, if the peers are constraining but are not constrained by each other. One example for such behavior is fairness, each process is constraining its peer to achieve fairness but neither of the processes is constrained by their peer. The default credit-scheduler, where the quantum of PCPU time is fairly divided among different VMs works best for such a requirement.

These options lead to a design of schedulers tailored for particular task in simulations. Since, the CPU-Pools allow concurrent execution of hypervisor schedulers the area of research and development in this direction holds a promising future.

7.2.8 Outstanding Issues with Counter-Based Algorithm (Deadlock and Livelock)

We derived a counter-based algorithm to prevent *deadlock* and *livelock* conditions during a load-balanced VTS over VM execution. However, the threshold value of the counter that we used is an empirical in nature and hence, might yield varying performance with change of applications, hardware and synchronization schemes.

Intuitively, a large threshold value could better application performance that is not prone to *deadlock* and livelocks. Similarly, a small threshold value could better applications prone to *deadlock* and *livelock*. Further, these threshold values are also dependent on the dimension of the time-slices provided to the VCPUs during scheduling, i.e., larger time-slices execute more concurrent events than smaller time slices. Also, the *deadlock* and *livelock* are dependent on events computed, where as the threshold value counts the exhausted time-slices of the PCPU.

Hence, future work needs a comprehensive study performance dynamics for varying threshold values or a better PSX algorithm that does not rely on such empirical value is necessary.

7.2.9 Hardware-Supported Network with VM-Based Virtual Network Comparison

Efficient communication of event and time-stamps among the LPs in a PDES application is an important factor that determines the performance of the application. Hence, historically PDES applications have always relied on high throughput and low latency interconnects in their execution platforms. As the community move towards the Cloud Computing platforms, we need to deal with the software-based or hardware-supported virtual interconnect components to understand the performance dynamics of the PDES executions on Cloud. This especially becomes important, as cost is associated for the resource utilization time.

While, we have identified that the packet loss at the software bridge heavily determines runtime performance. The suitability of the software based networking support provided by the hypervisor or hardware-assisted virtual interconnect for executing wide range of PDES applications under varying multiplexing (VCPUs over PCPUs) ratios and varying synchronization algorithms have not been very well studied yet. We have found that overall bandwidth provided by hardware-assisted virtual interconnect was over an order of magnitude higher than that provided by its software counterpart on our LTP. However, even with this we have found that the runtime performance observed using software based virtual network to be better than the hardware-assisted interconnects for same scenarios, especially when using the default scheduler (CSX) of Xen.

7.2.10 Sensor Network Simulations

High-fidelity sensor network simulations are hard to realize. For example, although the OS used in the sensors or motes, such as the TinyOS, may be ported to work over Xen [74], the main challenge is the mismatch of the clock-cycles in the low-power sensors and the processors on which simulations are executed. Sensors usually have very low clock rates in comparison with the modern day processors. To realize high-fidelity sensor network simulations, one has to maintain the clock rate, evolve in time-ordered manner, and scale the utilized clock cycles appropriately in accordance to the sensors being simulated. With in our NetWarp network simulation platform, we are able to fulfill the aforementioned requirements of self-maintained virtual time line and virtual-time ordered execution. In addition, the virtual time is recorded in terms of elapsed time slices or tick sizes, which are based on the clock rates. For example: in most of our experiments each tick size measures 100μs. This virtual time recorded in terms of tick sizes can be easily scaled with appropriate factor to suit the sensors being simulated. Further, with CPU-cores clocking faster than the sensors, we can achieve faster than real-time sensor network simulations on hypervisor platforms, when the virtual to physical multiplexing ratio 1.

7.2.11 Reviving the Concept of a Time-warp OS

The VM platforms provide a great opportunity to revive Time-warp OS, a simulation based operating system for PDES applications. With the hypervisor we can realize the simulation processes or PDES federates as an operating system rather than an application running on an OS. The Xen hypervisor provides sufficient support to start in this direction, for example: the Mini-OS distributed along with the Xen hypervisor provides the capability to the user to over-ride the default initial process of the Mini-OS with a user-specified function, which as well could be a simulation process. Such a platform of PDES over a hypervisor platform would be highly advantageous, as it also leverages on other powerful features supported by the hypervisor. For example: runtime migration of the PDES-OS, which would play a significant role in dynamic load balancing and would enhance the fault-tolerance of the PDES application. Since, such an OS forms a container, which one could positively expect that Cloud service providers to host in future. If this happens the Cloud would prove to be a powerful execution platform for PDES applications in the coming future.

7.2.12 GVT Synchronization within Hypervisor

By pushing the GVT synchronization algorithms to the level of hypervisor a powerful simulation platform can be envisioned. Discrete event simulators developed to function on particular operating system can be used to work with other, and similar diverse simulators in parallel and synchrony. Further, this can be achieved by simple instrumentation such as writing and reading its simulation time values to and from the hypervisor. Thus creating a scalable platform for parallel execution of diverse discrete event simulators that were developed for execution on equally diverse OS platforms.

7.2.13 Realizing High-Fidelity Simulation Framework using Characteristically Distinct Simulators

The computing systems with relevant software are used to monitor, control and analyze the complex industrial systems. For example the supervisory controls and data acquisition (SCADA) units are used for such purposes in electric-grid. We envision a simulation framework is required to assess the unwanted, unintentional and unanticipated affects on the electric-grid due to the changes to the SCADA control software, as a future work. This requires integration of electric grid simulator and the high-fidelity computer network simulator as shown in Figure 96. Two distinct networks namely, electric grid and the networked SCADA units or telecommunication grid is seen in Figure 96. The runtime dynamics involved in the electric grid is very different from that of the telecommunication network and hence, their simulation models are different as well. However, with the combination of using VM as execution platform for electric grid with SCADA simulations and VMs as SCADA unit surrogates a framework for secure electric grid can be realized. Although we use electric-gird domain in this discussion, the set of design principles are equally valid for other industrial system.

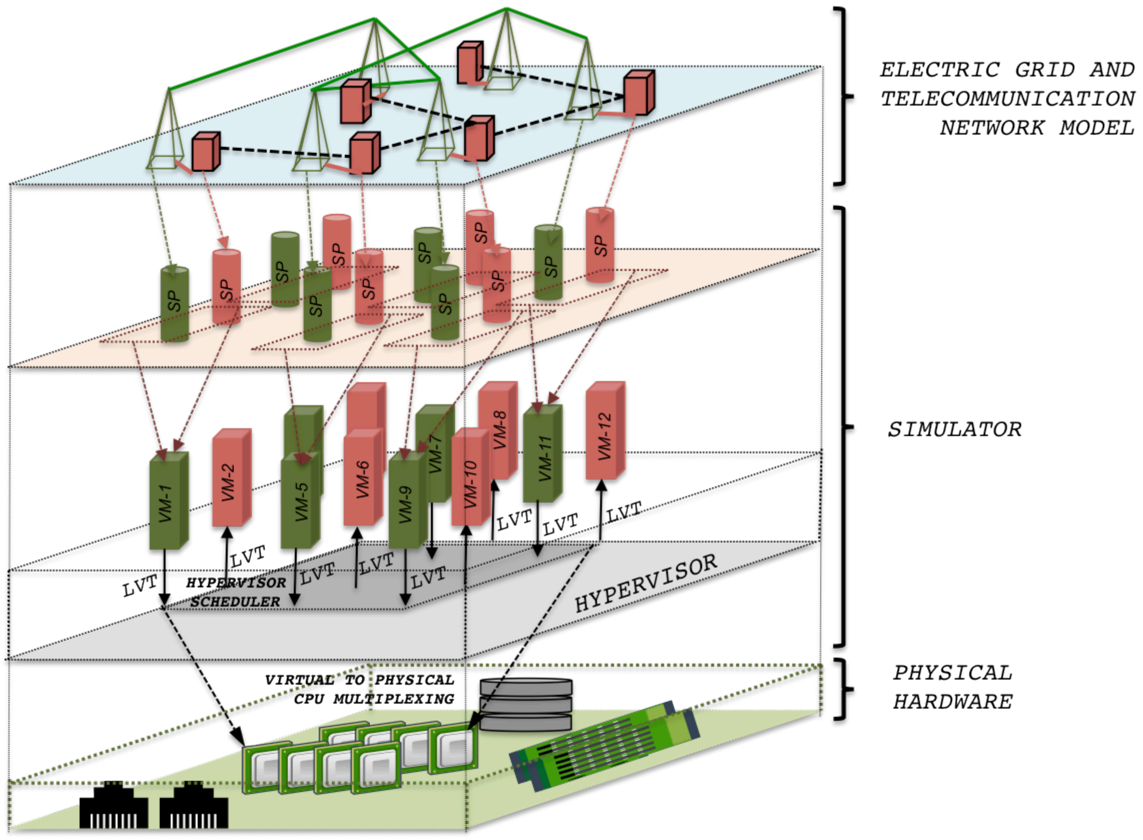


Figure 96 Functional diagram of the software test framework for secure electric grid

7.2.14 Runtime Load Balancing

The point of strength in PSX design discussed in Chapter 6 is, the application characteristic that hints runtime behavior of a highly-dynamic parallel application is rightly delegated to the compute resource allocator to reap performance benefits. This particular aspect is not limited to PDES alone but can be applied to other similar dynamic parallel applications. One offshoot of this work is in the direction applying this runtime load-balancing concept to similar parallel dynamic systems after deriving the runtime-characteristic determinant factor.

7.2.15 Dynamic Load Balancing

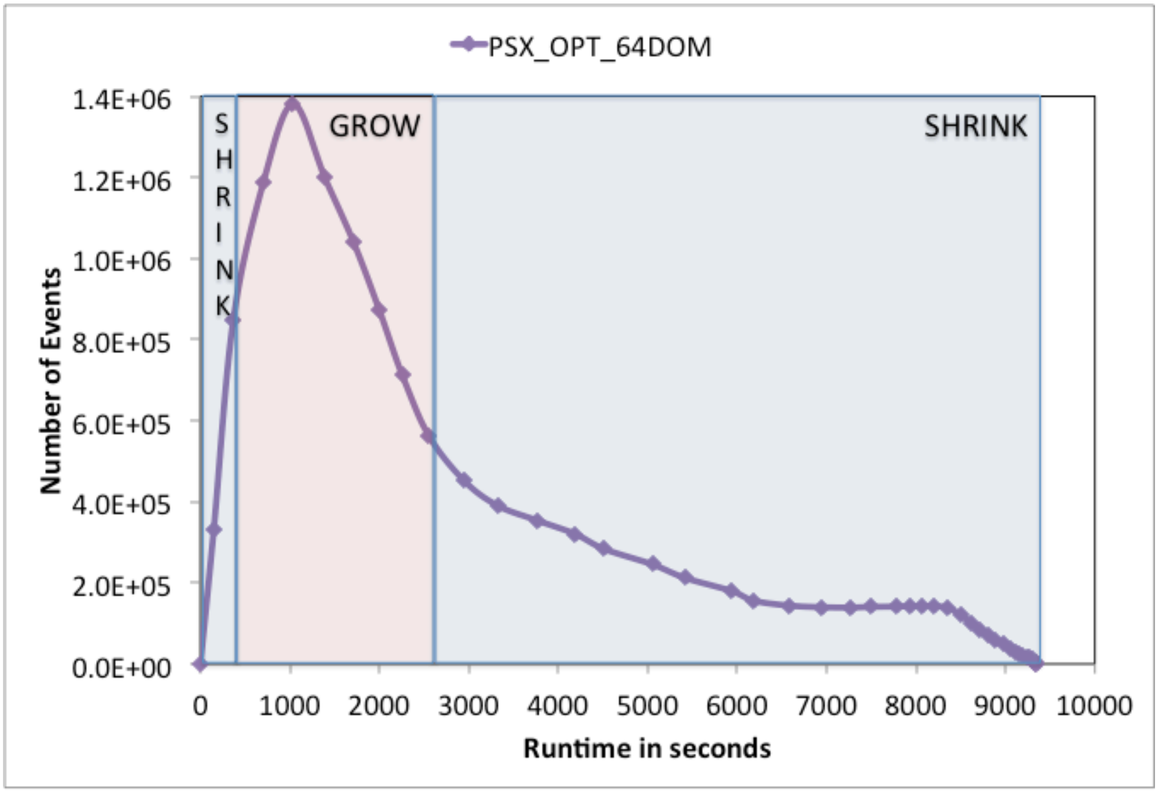


Figure 97 Vehicular traffic simulation plot of number of events against runtime

In the PDES applications, compute cycles consumed directly corresponds to the number of processed simulation events. In such scenarios the capability of a parallel simulation execution environment to grow or shrink its physical compute resources as necessitated by the varying workload is extremely beneficial for efficient utilization of available resources. Figure 97 plots the results from vehicular traffic simulation experimental setup comprising 64 VMs where, number of events processed by all federates are plotted against the runtime. This figure also pictorially shows the potential points (based on some number-of-processed-events threshold) at which simulation execution environment could grow and shrink. One means to achieve the capability to *grow* and *shrink* in physical resource utilization is via *oversubscribing* of virtual resources at the beginning of the simulation, *growing* using *process-migration* of virtual resources on to newly available physical resource, and *shrinking* by *oversubscribing* again. The efficiency achieved and/or the hindrances to overcome in exercising this advantageous technique for dynamic load balancing is another important direction of our future work.

REFERENCES

[1] Verners, B., Inside Java Virtual Machine. 1996, McGraw-Hill, Inc.

[2] Kirill K., Virtualization in Linux, 2006, http://download.openvz.org/doc/openvz-intro.pdf

[3] Kamp, P.H. and R.N. Watson. Jails: Confining the omnipotent root. In Proceedings of the 2nd International SANE Conference. 2000.

[4] Price, D. and A. Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In LISA, vol. 4, pp. 241-254. 2004.

[5] VMware ESX Server User's Manual, http://www.vmware.com/pdf/esx_15_manual.pdf

[6] Chisnall, D., The Definitive Guide to the Xen Hypervisor. 2007: Pearson Education.

[7] Matthews, J.N., et al., Running Xen: a hands-on guide to the art of virtualization. 2008: Prentice Hall PTR.

[8] Popek, G.J. and R.P. Goldberg, Formal requirements for virtualizable third generation architectures. Commun. ACM, 1974. 17(7): pp. 412-421.

[9] Bugnion, E., et al., Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. ACM Transactions on Computer Systems (TOCS), 2012.

[10] Mell, P. and T. Grance, The NIST Definition of Cloud Computing, US Department of Commerce Special Publication 800-145. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[11] Birta, L.G. and G. Arbez, Modeling and simulation: exploring dynamic system behaviour. 2007: Springer.

[12] Chandy, K.M. and J. Misra, Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. IEEE Transactions on Software Engineering, 1979. SE-5(5): pp. 440-452.

[13] Bryant, R.E., Simulation Of Packet Communication Architecture Computer Systems. 1977, Massachusetts Institute of Technology.

[14] Jefferson, D.R., Virtual time. ACM Trans. Program. Lang. Syst., 1985. **7**(3): pp. 404-425.

[15] Fujimoto, R.M., Parallel and Distributed Simulation Systems. 2000, 605 Third Avenue, N.Y.: Wiley-Interscience.

[16] Perumalla, K.S., µsik-a micro-kernel for parallel/distributed simulation systems. In the Workshop on Principles of Advanced and Distributed Simulation, PADS, 2005: pp. 59-68.

[17] Perumalla, K.S., A.J. Park, and V. Tipparaju, GVT algorithms and discrete event dynamics on 129K+ processor cores. In 18th International Conference on High Performance Computing, 2011: pp. 1-11.

[18] Mattern, F., Efficient algorithms for distributed snapshots and global virtual time approximation. Journal of Parallel and Distributed Computing, 1993. 18(4): pp. 423-434.

[19] Fujimoto, R.M., Performance of time warp under synthetic workloads. 1990.

[20] Perumalla, K.S. and S.K. Seal, Discrete event modeling and massively parallel execution of epidemic outbreak phenomena. Simulation, 2012. 88(7): pp. 768-783.

[21] Yoginath, S.B. and K.S. Perumalla, Parallel Vehicular Traffic Simulation using Reverse Computation-based Optimistic Execution. In Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation. 2008: pp. 33-42.

[22] Yoginath, S.B. and K.S. Perumalla, Reversible discrete event formulation and optimistic parallel execution of vehicular traffic models. International Journal of Simulation and Process Modelling, 2009. **5**(2): pp. 104-119.

[23] Liu, J., A primer for real-time simulation of large-scale networks. In 41st Annual Simulation Symposium, ANSS, 2008.

[24] Chun, B. et al., Planetlab: an overlay testbed for broad-coverage services. ACM SIGCOMM Computer Communication Review 33.3, 2003: pp. 3-12.

[25] Elliott, C., GENI-global environment for network innovations. LCN. 2008.

[26] Fall, K., Network emulation in the VINT/NS simulator. In Proceedings of IEEE International Symposium on Computers and Communications: pp. 244-250.

[27] OPNET: www.opnet.com

[28] Riley, G.F., Large-scale network simulations with GTNetS. In Proceedings of Winter Simulation Conference, 2003.

[29] Cowie, J., A. Ogielski, and D. Nicol, The SSFNet network simulator. Software on-line: http://www. ssfnet. org/homePage. html, 2002.

[30] Zeng, X., R. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In Proceedings of Twelfth Workshop on Parallel and Distributed Simulation, PADS. 1998.

[31] Fujimoto, R.M., et al. Large-scale network simulation: how big? how fast?. In 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, MASCOTS, 2003.

[32] Rizzo, L., Dummynet: a simple approach to the evaluation of network protocols. SIGCOMM Comput. Commun. Rev., 1997. 27(1): pp. 31-41.

[33] Huang, X.W., R. Sharma, and S. Keshav. The ENTRAPID protocol development environment. In the Proceedings of Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM, 1999.

[34] Vahdat, A., et al., Scalability and Accuracy in a Large-scale Network Emulator. SIGOPS Oper. Syst. Rev., 2002. 36(SI): pp. 271-284.

[35] White, B., et al., An Integrated Experimental Environment for Distributed Systems and Networks. SIGOPS Oper. Syst. Rev., 2002. 36(SI): pp. 255-270.

[36] Gupta, D., et al., To Infinity and Beyond: Time-warped Network Emulation. In Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3. USENIX Association: San Jose, CA, 2006.

[37] Apostolopoulos, G. and C. Hassapis, V-eM: A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation. In Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation. Washington DC, USA, 2006: pp. 117-126.

[38] Gupta, D., K.V. Vishwanath, and A. Vahdat, DieCast: Testing Distributed Systems with an Accurate Scale Model. In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation. USENIX Association: San Francisco, California, 2008: pp. 407-422.

[39] Liu, J., R. Rangaswami, and M. Zhao, Model-driven network emulation with virtual time machine. In Proceedings of the Winter Simulation Conference, WSC, 2010: pp. 688-696.

[40] Bergstrom, C., S. Varadarajan, and G. Back, The Distributed Open Network Emulator: Using Relativistic Time for Distributed Scalable Simulation. In Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation. Washington DC, USA, 2006: pp. 19-28.

[41] Grau, A., et al., Time Jails: A Hybrid Approach to Scalable Network Emulation. In Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation, Washington, DC, USA, 2008: pp. 7-14.

[42] Duggirala, V. and S. Varadarajan, Open Network Emulator: A Parallel Direct Code Execution Network Simulator. In Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation. 2012: pp. 101-110.

[43] Yuhao, Z. and D.M. Nicol, A Virtual Time System for OpenVZ-Based Network Emulations. In IEEE Workshop on Principles of Advanced and Distributed Simulation, PADS, 2011.

[44] Weingartner, E., et al. SliceTime: a platform for scalable and accurate network emulation. In Proceedings of the 8th USENIX conference on Networked systems design and implementation. 2011.

[45] Hoffman, D., D. Prabhakar, and P. Strooper. Testing iptables. In proceedings of the conference of the Centre for Advanced Studies on Collaborative research. IBM Press, 2003.

[46] Gu, Y., ROSENET: A remote server-based network emulation system. 2008, Georgia Institute of Technology.

[47] Team, Netfilter Core. Libipq-Iptables Userspace Packet Queuing Library. (2006). http://linux.die.net/man/3/libipq

[48] Gropp, W., E. Lusk, and A. Skjellum, Using MPI: portable parallel programming with the message-passing interface. Vol. 1. 1999: MIT press.

[49] Yoginath, S.B. and K.S. Perumalla, Efficiently Scheduling Multi-Core Guest Virtual Machines on Multi-Core Hosts in Network Simulation. In Proceedings of the IEEE Workshop on Principles of Advanced and Distributed Simulation. 2011.

[50] Netfilter: Netfilter - Firewalling, NAT and Packet Mangling for LINUX, http://www.netfilter.org, 2012.

[51] Yoginath, S.B., K.S. Perumalla, and B.J. Henz, Taming Wild Horses: The Need for Virtual Time-Based Scheduling of VMs in Network Simulations. In IEEE 20th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, MASCOTS, 2012.

[52] Yoginath, S.B., K.S. Perumalla, and B.J. Henz, Runtime performance and virtual network control alternatives in VM-based high-fidelity network simulations. In Winter Simulation Conference (WSC), 2012.

[53] Kurkowski, S., T. Camp, and M. Colagrosso, MANET simulation studies: the incredibles. ACM SIGMOBILE Mobile Computing and Communications Review 9.4, 50-61, 2005.

[54] Hogie, L., P. Bouvry, and F. Guinand, An overview of MANET simulation. Electronic notes in theoretical computer science, 150.1, 81-101, 2006.

[55] Henz, B.J., et al. Large scale MANET emulations using U.S. Army waveforms with application: VoIP. Military Communications Conference, MILCOM, 2011.

[56] EMANE User Manual 0.7.3, DRS Cengen, Bridgewater, NJ, 2012.

[57] Clausen T. and P. Jacquet, Optimized Link State Routing Protocol (OLSR)". RFC 3626.

[58] OLSRD: www.olsr.org

[59] Rosenberg J., Schulzrinne H., Camarillo G., Johnston A., Peterson J., Sparks R., Handley M. and Schooler, E., SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141.

[60] Schulzrinne H., Casner S., Frederick R. and Jacobson V., RTP: A Transport Protocol for Real-Time Applications, RFC 3550 (Standard), July 2003. Updated by RFCs 5506, 5761, 6051, 6222.

[61] PJSIP. Open source SIP stack and media stack for presence, instant messaging, and multimedia communication, http://www.pjsip.org

[62] Amazon Elastic Compute Cloud User Guide, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Welcome.html

[63] Jackson, K.R., et al., Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In IEEE Second International Conference on. Cloud Computing Technology and Science, CloudCom, 2010: pp. 159-168.

[64] Wang, G. and T.S.E. Ng, The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In Proceedings of the 29th Conference on Information Communications, San Diego, California, USA, 2010: pp. 1163-1171.

[65] D'Angelo, G., Parallel and distributed simulation from many cores to the public cloud. In International Conference on High Performance Computing and Simulation, HPCS, 2011: pp. 14-23.

[66] Fujimoto, R.M., A.W. Malik, and A. Park, Parallel and distributed simulation in the cloud. SCS M&S Magazine, 2010.

[67] Malik, A.W., A. Park, and R.M. Fujimoto, An Optimistic Parallel Simulation Protocol for Cloud Computing Environments. SCS M&S Magazine, 2010.

[68] Park, A.J., Master/worker parallel discrete event simulation. 2009, Georgia Institute of Technology.

[69] Vanmechelen, K., S. De Munck, and J. Broeckhove, Conservative Distributed Discrete Event Simulation on Amazon EC2. In IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing, Washington, DC, USA, 2012: pp. 853-860.

[70] Yoginath, S.B. and K.S. Perumalla, Optimized hypervisor scheduler for parallel discrete event simulations on virtual machine platforms, in Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques. 2013, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering): Cannes, France. pp. 1-9.

[71] Santos, J.R., et al., Netchannel 2: Optimizing network performance. Proceedings of the XenSource/Citrix Xen Summit, 2007.

[72] Yoginath, S.B. and K.S. Perumalla, Empirical evaluation of conservative and optimistic discrete event execution on cloud and VM platforms. In Proceedings of the ACM SIGSIM conference on Principles of advanced discrete simulation, Montreal, Qubec, Canada, 2013: pp. 201-210.

[73] Jefferson, D., et al., Time Warp Operating System. SIGOPS Oper. Syst. Rev., 1987. 21(5): pp. 77-93.

[74] Maclean, A. Xen Meets TinyOS, University of Glasgow, 2008.