Taming Wild Horses: The Need for Virtual Time-based Scheduling of VMs in Network Simulations

Srikanth B. Yoginath, Kalyan S. Perumalla
Computational Sciences and Engineering Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6085
yoginathsb@ornl.gov, perumallaks@ornl.gov

Brian J. Henz
ATTN: RDRL-CIH-C
U.S. Army Research Laboratory
APG, MD 21005
brian.j.henz.civ@mail.mil

***Abstract*— The next generation of scalable network simulators employ virtual machines (VMs) to act as high-fidelity models of traffic producer/consumer nodes in simulated networks. However, network simulations could be inaccurate if VMs are not scheduled according to virtual time, especially when many VMs are hosted per simulator core in a multi-core simulator environment. Since VMs are by default free-running, on the outset, it is not clear if, and to what extent, their untamed execution affects the results in simulated scenarios. Here, we provide the first quantitative basis for establishing the need for generalized virtual time scheduling of VMs in network simulators, based on an actual prototyped implementations. To exercise breadth, our system is tested with disparate applications: (a) a set of message passing parallel programs, (b) a computer worm propagation phenomenon, and (c) a mobile ad-hoc wireless network simulation. We define and use error metrics and benchmarks in scaled tests to empirically report the poor match of traditional, fairness-based VM scheduling to VM-based network simulation, and also clearly show the better performance of our simulation-specific scheduler, with up to 64 VMs hosted on a 12-core simulator node.**

Keywords-Virtual Machines, Network Simulation, Time Synchronization, High-fidelity, Multi-core, MPI, Ad-hoc Wireless

I. INTRODUCTION

Traditionally, development and testing of network protocols/applications have been performed using either (a) network simulators and/or (b) network emulators. Network simulators such as NS[1], OPNET[2], and SSFNet[3] are discrete-event simulation environments that support a variety of network protocol component models using which application scenarios are simulated. Although such network *simulators* incur modeling burden to re-implement real protocols/software as simulator-specific models, such simulators are attractive for their speed and scaling characteristics. On the other hand, network *emulators*, such as Emulab[4], ModelNet[5], PlanetLab[6] and GENI[7], can run unmodified, "as-is" network application/protocol test codes. However, emulators are significantly more limited by the physical resources than simulators, and increasing the scale of the emulators is an expensive endeavor. The scale and speed of simulators can be combined with the realism and modeling expediency of emulators by introducing virtual machines (VMs) as high fidelity surrogates for end host models. With the increase in the complexity of present day distributed network protocols and applications, such emulation-simulation environments built using VMs are gaining usage and attention. Figure 1 shows a schematic of a typical mapping of a subject test network onto VM-based simulation framework built using hypervisor technologies.

In order to efficiently make use of VMs in such network simulations/emulations, mechanisms are needed in hypervisors to maintain a common *simulation timeline* across VMs and also ensure time ordered execution of VMs along the timeline. The mechanisms must minimize or eliminate causality errors such that the messages of the future (in terms of virtual simulation time-line) are prevented from affecting the past on the simulation timeline. Unfortunately, almost none of the existing VM-based network simulators/emulators account for this requirement.

Time dilation, introduced in [8] and adopted in subsequent works, demonstrated time virtualization in network emulators, wherein a higher/lower bandwidth communication network behavior could be emulated using the same underlying network just by manipulating the perceived *rate of time elapse* 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[9], DieCast[10], VENICE[11], dONE[12], and Time-Jails[13], allowing some flexibility in configuring the emulation setup.

While the aforementioned approaches were adequate in uni-processor hosts that multiplex the VMs, they are inadequate in the context of multi-core hosts. While those systems may execute the scenarios on multi-core hosts, the required ordering by simulation time is not accounted for *within* a (multi-core) host node. Emulation environments hosting many VMs can no longer use wall-clock (unmodified real) time as the simulation time. Rate adjustment of time using Time Dilation also is inadequate to integrate with a discrete event simulator that is not constrained by real-time (e.g., executing in an as-fast-as-possible mode). Further, the user has limited or no control on the execution ordering of the VMs that is needed to avoid causality errors.

In contrast, a correct VM-based simulator must undertake the task of defining and maintaining the concept of an *intra-*

node simulation timeline and also ensure simulation time-ordered VM execution *within* each multi-core host node. None of the aforementioned simulation systems provides necessary mechanisms to address these issues. The need for virtual time-based inter-VM scheduling has only cropped up only recently, due to the wider availability of nodes with many cores.

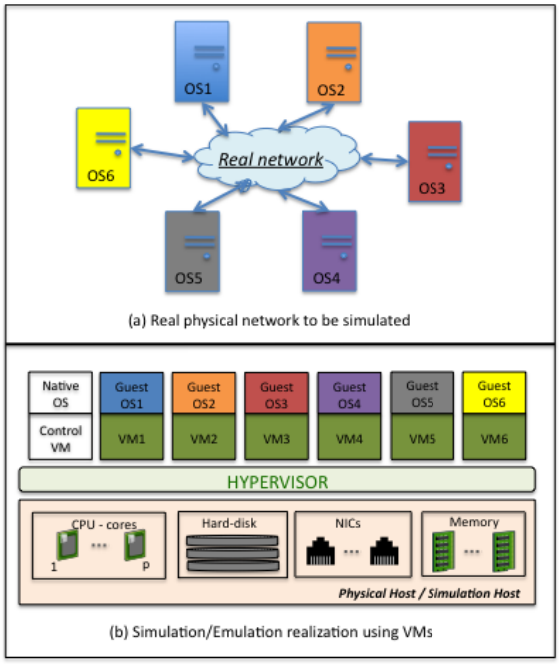

Figure 1 VM-based Network Simulations

Recently, in [16], we examined the issues of simulation timelines and time-ordered execution; however, the scaling of our proposed solution to larger problem sizes and its application for real-life network simulation scenarios was 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 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 paper we extend our previous work in [16] by (a) porting our new scheduler (NSX) based hypervisor environment onto a more powerful hardware platform, (b) designing new Message Passing Interface (MPI)[18] based benchmarks, (c) implementing a cyber-security application with complex timing behaviors and messaging functionality (d) porting an existing ad-hoc wireless application, and (e) performing a thorough analysis of scaling and accuracy.

Here, we establish the need and efficacy of our solution on a much larger number of cores, larger number of VMs, and a larger and more complex set of applications. Relative to [16], significantly larger configurations are reported here:

- execution on up to 12 cores or 24 hyper-threaded cores (compared to 2-core execution)
- multiplexing up to 64 VMs on a single node (increased from 4 VMs of previous results), all ordered via virtual time scheduling
- quantitative experimental results from three, widely different, applications: an MPI-based set of benchmarks, a cyber-security application, and a mobile ad-hoc wireless network simulation (compared to results from a single, highly simplistic synthetic benchmark in [16]).

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

In the following section, an overview of the concepts and terminology for virtual time-ordered scheduling is presented. In Section III, we describe the experimentation methodology and the details of the benchmarks. This is followed in Section IV by the experiment results and analyses, clearly showing the need for (and effectiveness of) virtual time-based intra-host VM scheduling for accurate simulation even with a large number of VMs per node. We summarize the findings and conclude the paper with a brief discussion on the future work in Section V.

II. Background: Concepts and Terminology

The underlying problem of interest in VM-based simulations on multi-core hosts can be traced to hypervisor scheduling. Conventional considerations such as fairness are inappropriate and mismatched with the relative orderings needed for modeling accuracy in cyber infrastructure simulations. Since VMs are by default free-running, they do not possess any conceptual ground work to conform to the specific time-ordered discipline needed for network simulation. New conceptual framework must be defined with VMs for use in network simulators. For self-containment, we briefly reproduce the ideas of simulation timelines and time ordered execution from our recent work [16].

A. *VM Timelines and Execution Ordering*

Simulation Timelines: In network emulations, where the physical resources are perfectly matched with the logical processes (LP) of simulation, the wall-clock time is the simulation time. One of the main reasons the VMs have become attractive in network simulations is because of their inherent support for resource conservation. This property can be efficiently exploited if we were able to multiplex a large number of VMs of varying capacity (each virtually configured as dual-core, hex-core, etc.) onto the limited resources of a physical host. Clearly, by such multiplexing, a mismatch between the hosted virtual resources and the hosting physical resources is created. The physical resources are time-shared by the virtual resources of VMs. Thus, two dual-core VMs running on a dual-core physical host utilizing 100% of their (simulated/virtual) CPU cycles would clearly take twice the (real) wall-clock time to finish certain task. Hence, the wall-clock time cannot be used as the simulation time. There is also the important issue of wasted host-CPU cycles if any of the

guest cores idle. To address this issue, in [16] we proposed maintaining a virtual clock for each virtual CPU core that advances based on the virtual CPU core utilization. This essentially results in multiple *virtual* timelines corresponding to the number of virtual CPU cores supported by the VM. For the purpose of network simulation, these multiple virtual CPU core timelines must be ideally reduced to a single VM timeline corresponding to a single *guest node*. Similarly, a single global simulation timeline can be derived from the multiple VM timelines on the *host node*.

Time-ordered Execution: The strategy in hypervisor scheduling used for processor sharing among the VMs is fair share-based and is performed independently of any time principle present in the VMs of network simulation. However, the simulations need to progress in the order of their simulation time, violating which leads to time-order errors resulting in scenarios where in the events of the future in one VM affect the events from the past in other VMs. A major challenge in network simulations using VMs on multi-core hosts is to enforce this time-ordered processing of events. To simulate a network behavior without time-order errors, it is essential that the events (such as packet arrival and departure events) be processed in simulation-time-order, along the previously defined timelines. Simulation time-ordered scheduling of the virtual CPU-cores onto the physical CPU-cores of the physical host can achieve time-ordered execution of all the VMs. The multiplexing time granularity of the VMs can be made as fine as necessary to achieve any desired degree of accuracy (e.g., in practice, even a scheduling time unit in the μs range can dramatically reduce or eliminate errors).

B. Virtualization Platform

All our concepts are implemented and experimented with in our prototype network simulation system named NetWarp. For virtualization support in NetWarp, we chose the Xen hypervisor [17] (in para-virtualization mode). Xen refers to VMs as Guest Domains or DOMs. Each DOM is identified by its DOM-ID. The first DOM, "DOM-0," affords special hardware privileges. Each DOM has its own set of virtual devices, including virtual multi-processors called virtual CPUs (VCPUs). The hypervisor scheduling mainly deals with efficient mapping (multiplexing) of all the VCPUs of multiple VMs onto the available physical processor cores (PCPUs).

Credit Scheduler of Xen (CSX): The "untamed" execution of VMs corresponds to the default (simulation-unaware) execution of Xen using this scheduler. The credit-based scheduler [17] is the default Xen scheduler, which schedules VCPUs on to PCPUs based on the principle of fair-share. CSX uses *credits* for every DOM, these credits are expended as the DOM's VCPUs are scheduled for execution. It provides control to the user to alter the configuration of scheduling through parameters called *weight* and *cap*. By default the *weight* value for all DOMs are 256 and *cap* is 0, ensuring fair CPU allocation to all of the DOMs. This scheduler is very widely used, and works excellently for a very large variety of virtualization uses.

Netwarp Scheduler for Xen (NSX): In our new (simulation-aware) virtual time-ordered scheduler (NSX) the Xen scheduling framework was customized for network simulations. Each VCPU accounts for its execution time in terms of *ticks* and this time is referred to as local virtual time (LVT) corresponding to that VCPU. Each PCPU maintains a queue of VCPUs. To schedule a VCPU for execution on a selected PCPU, the VCPU is inserted into the PCPU's queue. In each DOM, in addition to the LVTs of the DOM's VCPUs, a DOM-LVT is also maintained, which is the maximum among all its VCPU LVTs. The DOM-LVTs among all DOMs on the host node are periodically computed and updated. Thus, the LVT specific to a VCPU and DOM are referred to as VCPU-LVT and DOM-LVT, respectively.

To ensure time-ordered execution of events the NSX scheduler employs a least-LVT-first (LLF) policy, according to which the VCPU with the least VCPU-LVT value is scheduled for execution whenever a PCPU becomes available. Note that, due to the presence of multiple VCPU queues corresponding to each PCPU, the scheduling operation involves searching all processor queues for identifying the least-LVT VCPU and migration of the selected VCPU from its original queue to the currently active PCPU queue. By controlling the advancement of VCPU-LVT, the DOM-LVT is controlled; similarly employing LLF scheduling ensures virtual time-ordered progress of DOM-LVTs within the host node. Additional implementation details of NSX are documented in [16].

III. Experimentation: Methodology and Benchmarks

A. 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 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 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 time-ordered modes. In the third (Adhoc wireless network test), a real-time MANET emulation is used as a baseline to determine the correctness of the results from untamed (simulation unaware) scheduler and our new scheduler.

To increase the scope with respect to scale, we experiment with varying number of VMs, from 1 to 64 (for a single simulator host node).

B. MPI Benchmarks

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 2: 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 2.

2) *Varying Network Delay (VND) Test*

By this experiment we test how well 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 3. 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 3: Algorithm for VND test benchmark

Again, the objective of this benchmark is to bring out the anomalies introduced by any simulation 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) *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-ordered 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 j^{th} message in i^{th} run, and O_{ij} is the observed identifier of the j^{th} message in the i^{th} 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.

C. *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 4. 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 uniformly 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 each of 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 5. 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.

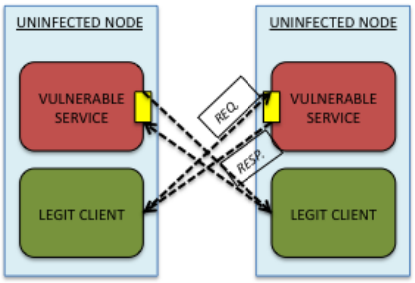

Figure 4: Interacting multi-service and multi-client scenario

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.

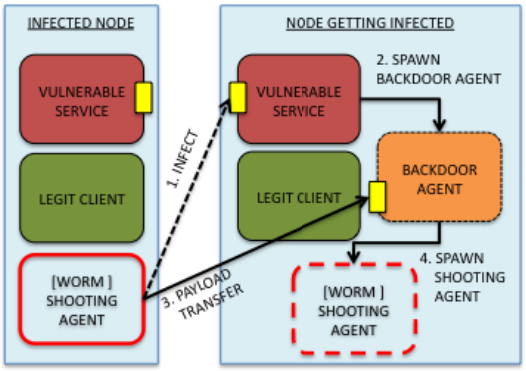

Figure 5: Worm infection and propagation

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.

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 DOM-0 and, using the *libxcutil* functions, we are able to get the simulation time into the *xenstore*[17][19], 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.

D. Adhoc Wireless Network Benchmark

In this benchmark we compare the results of emulating a Mobile Ad-hoc NETwork (MANET) [20] scenario run on a dedicated hardware cluster with the results from a similar scenario executed on our system configuration. We run this scenario with the untamed CSX and our time-ordered NSX hypervisor schedulers, and compare the results against the original hardware-based execution. Apart from highlighting the significance of time-ordered based scheduling this benchmark also demonstrates the versatility of the single node hypervisor based simulation host for realizing complicated network simulations.

1) EMANE based MANET Emulation Scenario

EMANE is the Extensible Mobile Ad-hoc Network Emulator from DRS Cengen [21]. EMANE contains a number of wireless *network emulation modules* (NEM) that model the *physical* (PHY) and layer2 or MAC layers of the network stack. This emulation was run on a dedicated cluster comprising 10 dual quad-core nodes. Of these ten nodes, 8 servers were used and each node was dedicated to host 8 VMs each and, each VM was pinned to a core. Each EMANE instance is hosted on a VM.

The network connectivity in this study is based on a star topology centered on coordinates of 39.70°N, 111.22°W, located in Utah. The RF propagation path loss is computed using the Longley-Rice algorithm with a cut-off value of -94 dB. The connectivity and routing table is computed at runtime using Optimized Link State Routing (OLSR)[22]. The OLSR daemon (olsrd)[23] is executed on each VM in order to dynamically compute the routing tables. The resulting one hop neighbor list is shown graphically in Figure 6.

A VoIP application is used in our experiments and is based on Signal Initiation Protocol (SIP)[24] for initiation and termination of calls. RTP[25] is used as the means to transmit the actual audio packets. Nodes 49 and 57 are noted in red at opposite ends of the star in the Figure 6 are the VoIP receiver and caller, respectively. The number of simultaneous calls between nodes 57 and 49 is increased from 1 up to 64, to study effects on the application performance due to varying communication loads over MANET. The software used for VoIP interactions is PJSIP[26].

2) Replicated MANET Emulation Scenario

We do not use EMANE-based emulation software in our setup, since the EMANE traffic is a simulation artifact. We directly use the virtual Ethernet devices of the VMs instead and the ad-hoc network is setup across these virtual devices.

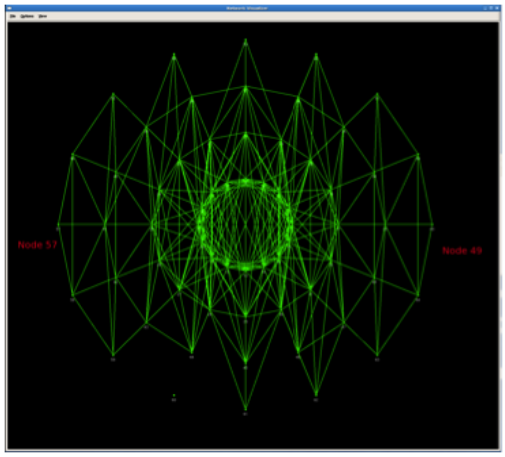

Figure 6: Plot of one hop neighbors for the MANET topology. The location of the VoIP caller and receiver nodes 57 and 49, respectively, are highlighted.

In this replicated scenario we use an input file that provides a distance-matrix specifying the signal strength between all entity pairs. The cut-off value of -94dB is used to determine the existence of a communication path between any two nodes/VMs. This data is communicated to the VMs through the *xenstore* of the Xen hypervisor. Based on this data each VM generates corresponding *iptables* rules to either drop or allow the packet based on its source MAC address. Of the packets that are allowed by the local host the bandwidth of this connection is controlled by dropping the packets that arrive after a specified threshold in arrival rate for that particular source MAC address has been reached. This threshold is enforced as *iptables* rules using *limit* module. An empirical value of 350 packets per second was used in our setup. The OLSR protocol dynamically establishes routes between the virtual layer2 interfaces of the communicating VMs and, this should yield same network connectivity as in Figure 6.

IV. Experiment Results

A. 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 currently capable of hosting a maximum of 64 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.

B. MPI Simulation Results

1) CND Test Results

From Figure 7 the curves obtained for the CND tests for virtual time-ordered scheduling (NSX) show its effectiveness in keeping the time-ordered 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 8, 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.

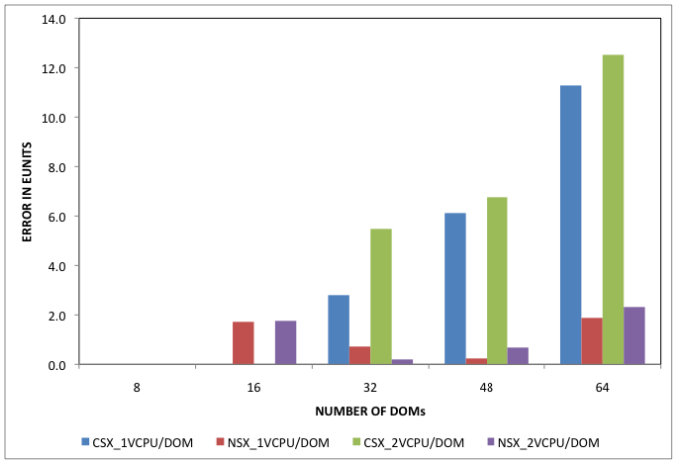

Figure 7: Comparing CSX and NSX using CND test results for time-ordered execution *errors*

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.

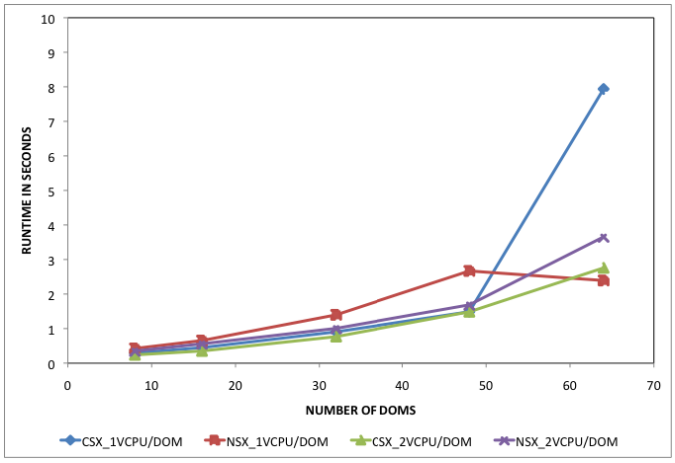

Figure 8: Comparing CSX and NSX *runtimes* from CND test runs

2) VND Test Results

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 9. 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.

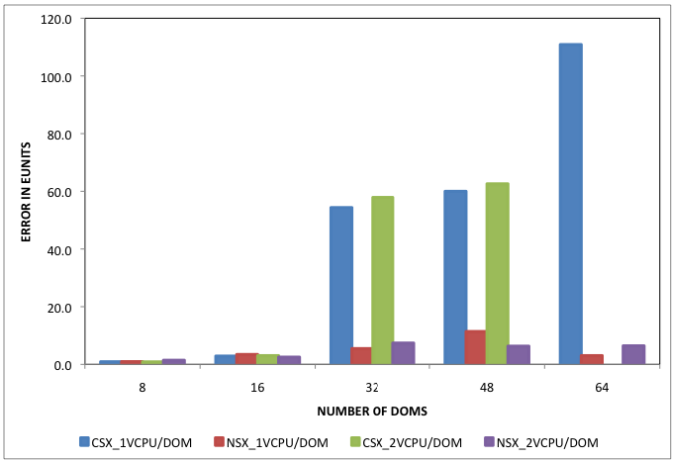

Figure 9: Comparing CSX and NSX using VND test results for time-order execution errors

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 10 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.

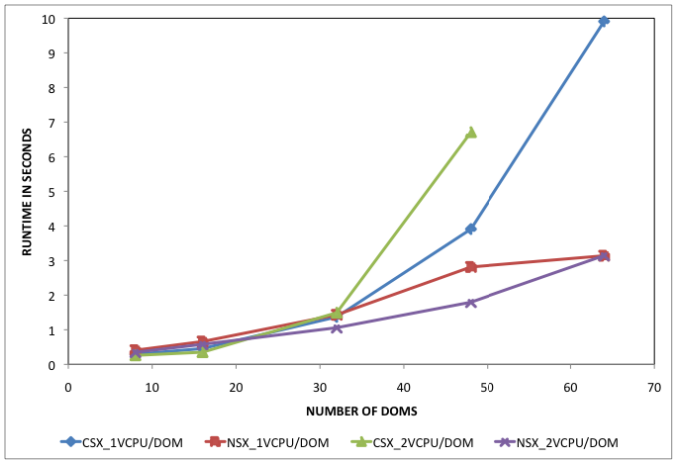

Figure 10: Comparing CSX and NSX runtimes from VND test runs

C. *Cyber Security Simulation Results*

Figure 11 and Figure 12 show the results from the runs of WP test with CSX and NSX, respectively. 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.

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 12. The consistent expression of this behavior in a controlled WP test across several runs of NSX (Figure 12) and its absence in the CSX (Figure 11) based simulation host environments strongly supports the necessity of time-ordered execution in the VM based network simulations.

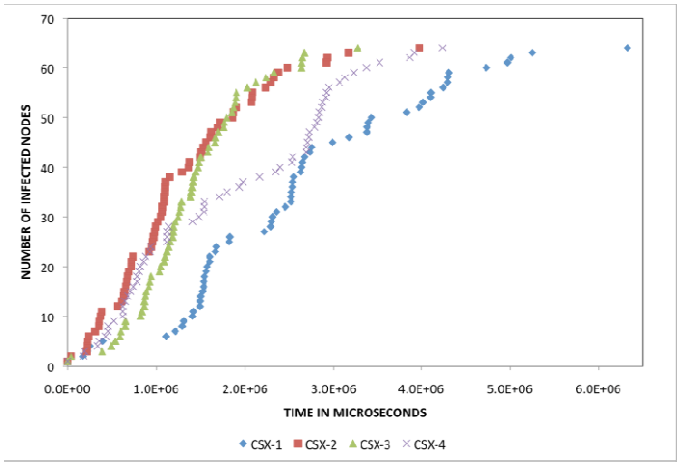

Figure 11: Worm propagation with CSX setup, showing high variability and abnormal simulation of propagation

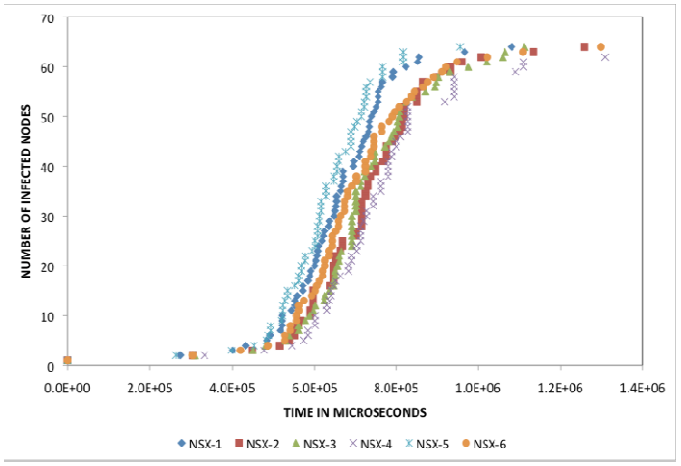

Figure 12: Worm Propagation plots with NSX, showing expected behavior of sigmoid curves, and very close matches across multiple runs, indicating repeatability within margin of a few microseconds (for *x*-axis, note the difference of scales with CSX above)

Figure 13 shows the curves from CSX and NSX of Figure 11 and Figure 12 in a single chart, allowing us to compare the behaviors directly. It is clear from Figure 13 that the CSX curves are widely varying, and a poor representation of the phenomenon, while the NSX curves show excellent simulation support. Note that the slight variability among multiple runs of NSX-based simulations is within the margin of time sliceof 100 µs. Similar to the previous MPI benchmarks, this variability can be reduced as desired, by reducing the time slice further.

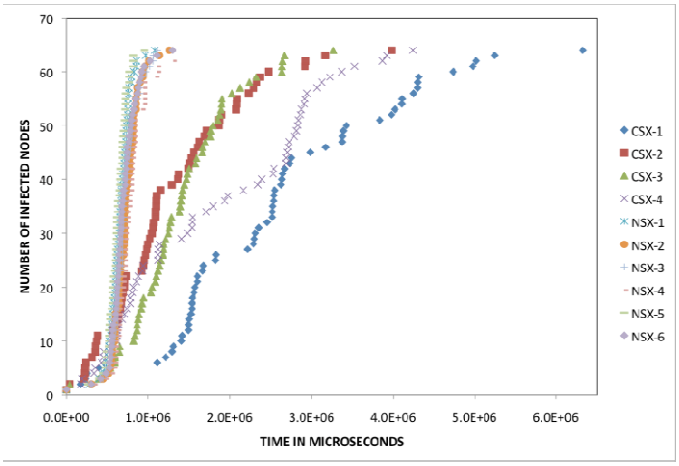

Figure 13: Worm Propagation behavior from multiple runs of CSX and NSX, showing the highly repeatable and clean curves from virtual time ordered scheduling (NSX) and non-repeatable and abnormal behavior from untamed (simulation-unaware) native scheduling

D. Adhoc Wireless Network Simulation Results

In this test, the experiment involved VoIP calls between two of the ad-hoc wireless nodes (the ones numbered 57 and 49, in this case). Node 57 sends a recorded message (*wmv* file) and 49 responds similarly. Once the files are transferred the test ends. The number of simultaneous calls between the sender and receiver is varied (1 to 64) across runs. The increase in number of simultaneous calls will obviously increase the load on the ad-hoc network setup. OLSR responsible for setting up ad-hoc network might change its routing behavior to accommodate increased load. Such changes by OLSR will affect the network dynamics. These set of experiments were designed to study such dynamics in the ad-hoc network with varying load.

Figure 14 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. Note the wavering packet loss curve with increase in the load on ad-hoc network; in comparison self-configuring behavior lacking general network would have resulted in increase in packet loss with increase in the application load.

By comparing the curves from the three plots in Figure 14, we see that plots using NSX setup are more similar to the results from the runs on the emulation-based results on the hardware with dedicated servers. 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.

The number of packets lost and the calls failed were higher in the dedicated server runs Figure 14(a) in comparison to other two plots. This is expected because the EMANE module used in the dedicated cluster runs introduces transmission delays on to the communication packets.

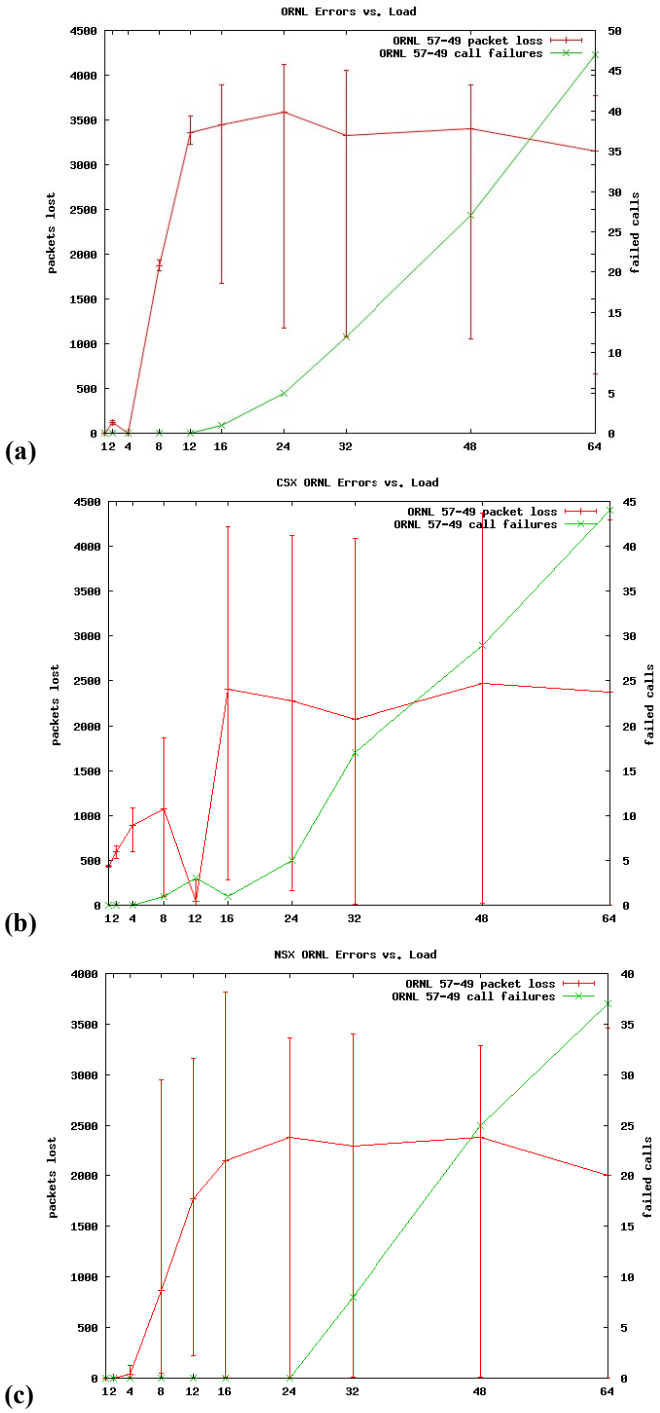

Figure 14: Plot of call failures and packet loss between nodes 57 and 49 versus the number of calls in (a) Dedicated cluster (b) CSX setup (c) NSX setup.

V. SUMMARY AND FUTURE WORK

Here, we have empirically shown how important simulation time-aware scheduling is for VM-based network simulations. This is the first quantitative evidence that highlights the pitfalls of using untamed native schedulers to host a large number of VMs on a multi-core host. Simulation runs with utilization-based or fairness-based schedulers give incorrect simulation

results at best, and are infeasible and non-scalable at worst. To strengthen the findings, we exercised our prototypes on a range of disparate applications, and tested with up to 64 VMs on a single node. The results strongly point to the weakness of existing schedulers and the clear strength of simulation-aware schedulers to ensure sensible results from high fidelity network simulations. To achieve accuracy in high-fidelity VM-based network simulation models in next generation network simulators, these results serve as important findings.

We defined a new quantitative error metric, termed *e*unit, and the associated pseudo code of benchmarks, which are designed for easy sharing and independent evaluation (and independent duplication of results) of virtual time ordering concepts by other researchers in the community.

The number of VMs (64) tested per node is the largest that we are aware of in a virtual time-based network simulator. Many of the runs with CSX (native credit scheduler) become unstable or fail on larger runs of simulations. NSX (our virtual time scheduler) on the other hand continues to be very stable and is observed to scale extremely well with the number of VMs.

Additional work is needed to extend the experiments and techniques to scheduling across multiple nodes. Since distributed virtual time synchronization techniques are well understood, we do not see any major impediments in scaling the experiments to more DOMs distributed across multiple nodes.

ACKNOWLEDGEMENTS

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

REFERENCES

[1] Fall, K. (1999). Network emulation in the VINT/NS simulator. Proceedings IEEE International Symposium on Computers and Communications: 244-250.

[2] OPNET: www.opnet.com

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

[4] White, B., J. Lepreau, et al. (2002). An Integrated Experimental Environment for Distributed Systems and Networks. Fifth Symposium on Operating Systems Design and Implementation. http://www.emulab.net/.

[5] Vahdat, A., K. Yocum, et al. (2002). Scalability and Accuracy in a Large-Scale Network Emulator. Operating System Design and Implementation (OSDI).

[6] Peterson, L., Muir, S., Roscoe, T., and Klingaman, A. PlanetLab Architecture: An Overview. Tech. Rep. PDN-06-031, PlanetLab Consortium, May 2006. http://www.planet-lab.org/

[7] GENI: http://www.geni.net/

[8] D. Gupta, et al. To Infinity and Beyond: Time- Warped Network Emulation. In Proceedings of the 3rd Sym- posium on Networked Systems Design and Implementation (NSDI'06), pages 87–100, San Jose, CA, USA, May 8–10 2006.

[9] Apostolopoulos, G. and C. Hasapis (2006). V-eM: A Cluster of Virtual Machines for Robust, Detailed and High-performance Network Emulation. 14th IEEE International Symposium on Modeling, Analysis and Simulation of Computing and Telecommunication Systems.

[10] Gupta, D., K. Vishwanath, et al. (2008). DieCast: Testing Distributed Systems with an Accurate Scale Model. 5th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA.

[11] Jason Liu, Raju Rangaswami and Ming Zhao. Model-Driven Network Emulation with Virtual Time Machine. In Proceedings of the 2010 Winter Simulation Conference (WSC) 2010.

[12] C. Bergstrom, et al., "The Distributed Open Network Emulator: Using Relativistic Time for Distributed Scalable Simulation," presented at the Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation, 2006.

[13] A. Grau, et al., "Time Jails: A Hybrid Approach to Scalable Network Emulation," in 22nd Workshop on Principles of Advanced and Distributed Simulation, Rome, Italy, 2008, pp. 7-14.

[14] R. M. Fujimoto, Parallel and Distributed Simulation Systems, Wiley Interscience, 2000.

[15] Jason Liu, Yue Li and Ying He. A large scale real-time network simulation study using PRIME, Proceedings of the 2009 Winter Simulation Conference (WSC), 2009.

[16] Yoginath, S.B.; Perumalla, K.S.;, "Efficiently Scheduling Multi-Core Guest Virtual Machines on Multi-Core Hosts in Network Simulation," *Principles of Advanced and Distributed Simulation (PADS), 2011 IEEE Workshop on* , vol., no., pp.1-9, 14-17 June 2011.

[17] David Chisnall, The Definitive Guide to the Xen Hypervisor, ISBN 978-013-234971-0, Prentice Hall, 2008.

[18] W. Gropp. E. Lusk, and A. Skjellum. "*Using MPI: Portable Programming with Message Passing Interface - 2nd edition*". The MIT Press, 1999.

[19] *XENSTORE*: http://aseemsethi.wordpress.com/article/learning-xenstore-29fizhrip655z-4/

[20] Henz, B.J.; Parker, T.; Richie, D.; Marvel, L.;, "Large scale MANET emulations using U.S. Army waveforms with application: VoIP," *MILCOM 2011* , vol., no., pp.2164-2169, 7-10 Nov. 2011.

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

[22] Clausen, T.; Jacquet, P.; Optimized Link State Routing Protocol (OLSR). RFC 3626.

[23] Olsrd: www.olsr.org

[24] Rosenberg, J,; Schulzrinne, H.; Camarillo, G.; Johnston, A.; Peterson, J.; Sparks, R.; Handley, M.; 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.

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

[26] PJSIP. Open source SIP stack and media stack for presence, im/instant messaging, and multimedia communication, http://www.pjsip.org, [Online; accessed December 2010].

[27] Armbrust, M., et al. Above the clouds: A Berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, U.C. Berkeley, Feb 2009.

[28] Fujimoto R.M., Malik A.W., and Park A.J. (2010). "Parallel and Distributed Simulation in the Cloud." Simulation Magazine, Society for Modeling and Simulation, Intl., 1(3).