Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A.M. Uhrmacher, eds

RUNTIME PERFORMANCE AND VIRTUAL NETWORK CONTROL ALTERNATIVES IN VM-BASED HIGH-FIDELITY NETWORK SIMULATIONS

Srikanth B. Yoginath,
Kalyan S. Perumalla

Computational Sciences and Engineering Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6085, USA

Brian J. Henz

ATTN: RDRL-CIH-C
Computing and Computational Sciences Division
U.S. Army Research Laboratory
APG, MD 21005, USA

ABSTRACT

In prior work (Yoginath and Perumalla, 2011; Yoginath, Perumalla and Henz, 2012), the motivation, challenges and issues were articulated in favor of virtual time ordering of Virtual Machines (VMs) in network simulations hosted on multi-core machines. Two major components in the overall virtualization challenge are (1) virtual timeline establishment and scheduling of VMs, and (2) virtualization of inter-VM communication. Here, we extend prior work by presenting scaling results for the first component, with experimental results on up to 128 VMs scheduled in virtual time order on a single 12-core host. We also explore the solution space of design alternatives for the second component, and present performance results from a multi-threaded, multi-queue implementation of inter-VM network control for synchronized execution with VM scheduling, incorporated in our NetWarp simulation system.

1 INTRODUCTION

Lately, network simulators are being built by integrating virtualization technologies with discrete event simulations of packet-level network models. In order to efficiently make use of virtual machines in such network simulations/emulations, new mechanisms are needed in hypervisors to maintain a global *simulation time-line* and also ensure a time ordered execution of events. The mechanisms must minimize or eliminate event causality errors wherein the events of the future (in terms of virtual simulation time-line) are prevented from affecting the events from the past on the simulation timeline. Unfortunately, almost none of the existing VM-based network simulators/emulators that employ VMs account for this requirement. Recently, incorrectness of simulations was quantitatively demonstrated (Yoginath and Perumalla 2011) when VM-based network simulations are executed without explicit simulation-specific support in conventional hypervisor schedulers. A solution was also proposed that maintains a separate simulation time clock at the level of each virtual CPU core (VCPU), and evaluated by experiments on a small scale (dual-core machine, three VMs). In follow-on work (Yoginath, Perumalla and Henz 2012) scaled results were reported on up to 64 VMs on larger-sized hardware (12-core machine) with multiple benchmarks exercising complex network behaviors. Here, two additional facets of VM-based network simulation are evaluated: (1) execution time characteristics for virtual time advances among VMs, and (2) discrete event-based network control for time-managed inter-VM communication. The runtime characteristics of our VM-based network simulator, NetWarp, are presented with experiments on twice the number of VMs than previously reported. Also discussed are the design, development and performance evaluation of virtualized communication between VMs with user-chosen virtual latencies, integrated with virtual timelines maintained at the VPCU-level.

978-1-4673-4781-5/12/$31.00 ©2012 IEEE

We use the Xen (Chisnall 2008) hypervisor in all our implementation and experimentation. Hence, we use its terminologies through out our paper, although the concepts apply with more generality. 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 efficiently and dynamically mapping (multiplexing) all the VCPUs of multiple VMs onto the available physical processor cores (PCPUs). We refer to the default scheduler of Xen as the Credit Scheduler of Xen (CSX) and our simulation-specific scheduler as NetWarp Scheduler for Xen (NSX).
In section 2, we discuss our approach in virtual time management that reduces multiple VCPU time-lines into a single simulation time-line, not including virtual inter-VM communication control, and also present the results from the benchmark runs to compare the time-order error and run times using CSX and NSX schedulers. In section 3, we introduce inter-VM communication control, and discuss its mechanisms, features, design challenges and implementation as the NetWarp Network Control (NNC) subsystem, and present the performance results. We summarize and conclude the paper with a brief of ongoing and future work in the section 4.

1.1 Related Work

Time dilation, introduced by (Gupta et al. 2006) 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 (Apostolopoulos and Hasapis 2006), DieCast (Gupta et al. 2008), VENICE (Liu, Raju, and Ming 2010), dONE (Bergstrom et al. 2006), and Time-Jails (Grau et al. 2008), allowing some flexibility in configuring the emulation setup. While these 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. Resource conservation from virtualization affects the simulation time ordered execution of the VMs. 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 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 take the onus of defining and maintaining the concept of a intra-node simulation timeline and also ensure the simulation time-order of VM execution *within* each multi-core host node. Simulation time across nodes can be synchronized using standard parallel discrete event simulation techniques, such as using lower bound on time stamp (LBTS) or global virtual time (GVT) algorithms (Fujimoto 2000). None of the aforementioned simulation systems provides necessary mechanisms to address these issues. Some newly emerging network simulators such as PRIME (Liu, Yue, and Ying 2009) account for simulation virtual time across nodes executing packet-level (discrete event) network models integrated with VM-based end-host models; however, they do not yet explicitly address intra-host ordering needed across VMs *within* a host node.

2 VIRTUAL TIME MANAGEMENT WITHOUT NETWORK CONTROL

Virtual time-ordered execution of DOMs can be efficiently supported by replacing the hypervisor scheduler for simulation, as demonstrated by (Yoginath and Perumalla 2011), in which each VCPU maintains a virtual clock that is advanced based on the number of physical CPU cycles the VCPU uses. This local time maintained by the VCPU is referred to as VCPU-LVT. Each Physical CPU-core (PCPU) maintains

a queue of VCPUs. A VCPU is enqueued in a PCPU queue dynamically chosen during execution. Each DOM, in addition the VCPU-LVT of its VCPUs also maintains a DOM-LVT variable, which is maximum value of all its VCPU-LVTs, is computed periodically.

To ensure time-ordered execution, the NSX scheduler employs a Least-LVT-first (LLF) policy, according to which the VCPU with the least VCPU-LVT value (among all the VCPUs across all DOMs) is scheduled for execution first on the availability of PCPU. Due to the presence of multiple PCPU queues, the scheduling involves searching all PCPU queues for the least VCPU-LVT and subsequent migration of the selected VCPU from its original queue to the currently active PCPU queue. Employing LLF scheduling ensures virtual time-ordered progress of DOM-LVTs within the host node.

2.1 Virtual Timeline Evolution

Since the time accounting by every VCPU results in the staggering of the VCPU timelines, it affects the DOM timelines as the simulation progresses. To demonstrate the staggering of unsynchronized DOM timelines, we use a parallel program (testing algorithm) that is logically designed to introduce imbalanced load on the parallel processes.

```
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;
      }
   }
}
```

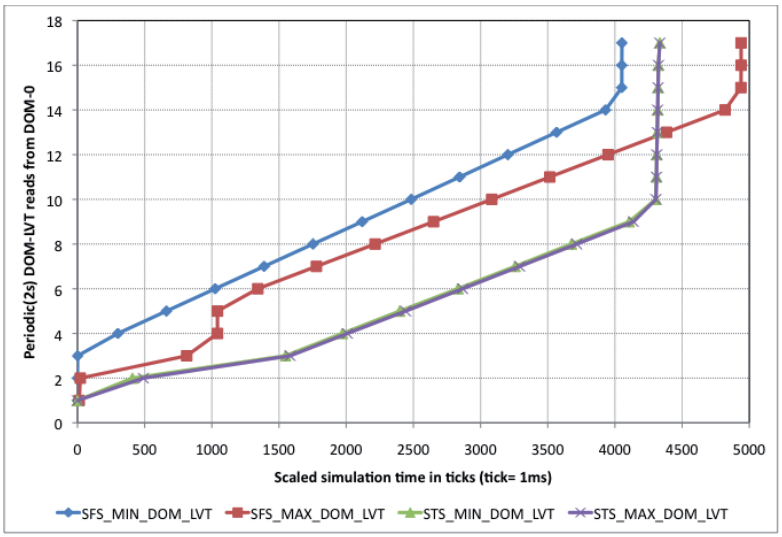

Figure 1: Algorithm to test staggering virtual timelines (left); Experiment results comparing simulation time divergence in SFS and STS (right).

Staggering Timelines Test Algorithm

Figure 1 (left) gives the algorithm of our test program in which the execution involves hosting every individual process involved in a parallel computing task on a separate DOM. 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 represents the number of processes involved 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 that is satisfied after it gets a message from LRP, while the process with r=1 receives (p-2) messages and, sends (p-1) messages to LRP. Consequently, the load (both computation and communication) on the process ranks decreases with increasing rank. We refer to this test as *Staggering Timelines Test* (STT).

A program to collect the DOM-LVT of a required DOM from the control-DOM or DOM-0 was developed using *libxcutil* . Before launching the test, the shell-script that collects LVT by querying the

DOMs (sequentially) for the DOM-LVT is started on DOM-0. Via this script, the collection of the DOM-LVT is periodical every 2 seconds.

Timeline Evolution Experiment Results

Two experiment cases, namely, *Scheduling Free of Synchronization* (SFS**)** and *Scheduling with Time Synchronization* (STS) are 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 is equal to (*NVCPUS* × *tick_size*), where *NVCPUS* is the total number VCPUs in the test environment (including DOM-0 VCPUs), and *tick_size* is 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 a DOM-0 with 24 VCPUs will be ((64+24) × 100μs)= 8.8ms . The test application run comprising 64 single VCPU user DOMs and the DOM-0 supported 24 PCPUs (equivalent to number of cores supported by the machine) was considered. The *minimum* (MIN) and *maximum* (MAX) of sampled DOM-LVTs collected from all DOMs periodically for both SFS and STS scenarios.

Except for the startup time, which is an artifact of parallel job launching, the DOM-LVT values are observed to increase at uniform rate. At the end of the simulation run the simulation time remains constant seen as almost vertical lines toward the end of test run, as shown in Figure 1 (right). While two distinct curves representing MIN and MAX DOM-LVT values can be observed in SFS case, the same two curves overlap in the STS, showing the need and the solution, respectively, for timeline synchronization. After the completion of the test 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 simulation time value in STS falls in between the MIN and MAX of SFS simulation time values, and closer to the MIN time-stamp value. 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 1 suggests that the runtime of the STS method is better than that of the SFS method. We also ran experiments to compare the time-order execution errors in the STS and SFS setups and found the errors in STS were far lesser than the SFS and so was the runtime. Thus, all the experiments discussed in the following sections of this article use STS.

2.2 Performance Benchmarks

In this section we present the results from the Message Passing Interface (MPI)-based benchmarks and cyber-security benchmarks, proposed and described in detail in (Yoginath, Perumalla and Henz 2012). The MPI benchmarks comprise two scenarios, namely, Constant Network Delay (CND) and Varying Network Delay (VND). With CND, we evaluate the performance of NSX and CSX scheduler support for time-ordered event execution when the communication structure and dynamics across the DOMs is deterministic and only the observed message generation order differs. With VND, 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 results presented for MPI benchmarks here are obtained by averaging results from 50 independent runs for all scenarios except for the 128 DOM scenario which are from 30 independent runs. In the cyber-security benchmark we emulate the behavior of worm-infection and its subsequent propagation across the service hosts in an interacting multi-server and multi-client scenario. The propagation in the system proceeds as a simple instance of the well-known "SI" epidemic model. We refer the interested reader to (Yoginath, Perumalla and Henz 2012) for more details on these benchmarks.

The performance data presented here are from twice the number of DOMs (128) than previously reported in (Yoginath, Perumalla and Henz 2012). In addition, we also present new results with CSX enhanced to use smaller *tick* size during scheduling, for higher accuracy. The results for the CSX runs previously presented used default *tick-size* (10ms) and *time-slice* (30ms), which are here reduced to 100μs (at the cost of longer execution time) to match the *tick-size* settings of NSX.

Error Metric

We use the time-order error metric of (Yoginath, Perumalla and Henz 2012) called *eunits* calculated as $E = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} \left| X_{ij} - O_{ij} \right|$ eunits, where, E is the error per simulation run, 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.

CND Benchmark Performance

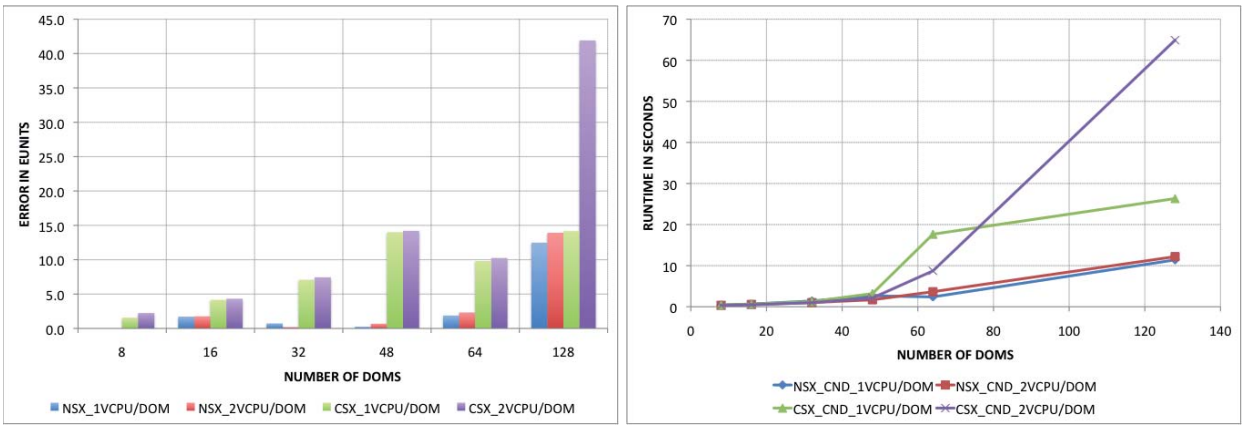

Figure 2: CND benchmark error plots (left); CND benchmark runtime plots (right).

As seen from the error graph in Figure 2 (left), the NSX scheduler for both 1 VCPU/DOM and 2 VCPU/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 2 (right) 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 as presented in (Yoginath, Perumalla and Henz 2012). The 2 VCPU/DOM case with CSX performs poorly both in terms of errors and runtime.

The DOM-0 VCPUs are maintained at the mininum 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 its 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 DOM-0) before the VCPUs time-order priorities are flattened due to periodic synchronization.

VND Benchmark Performance

In the VND tests, the NSX scheduler for 2VCPU/DOM scenario perfectly keeps both the error rates and the run times low, even as the number of DOMs in the test scenario increases as shown in Figure 3. On the other hand, NSX for 1 VCPU/DOM varies quite a bit in controlling 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 (Yoginath, Perumalla and Henz 2012) in terms of controlling errors, however the runtime suffers in both 1 VCPU/DOM and 2 VCPU/DOM scenario.

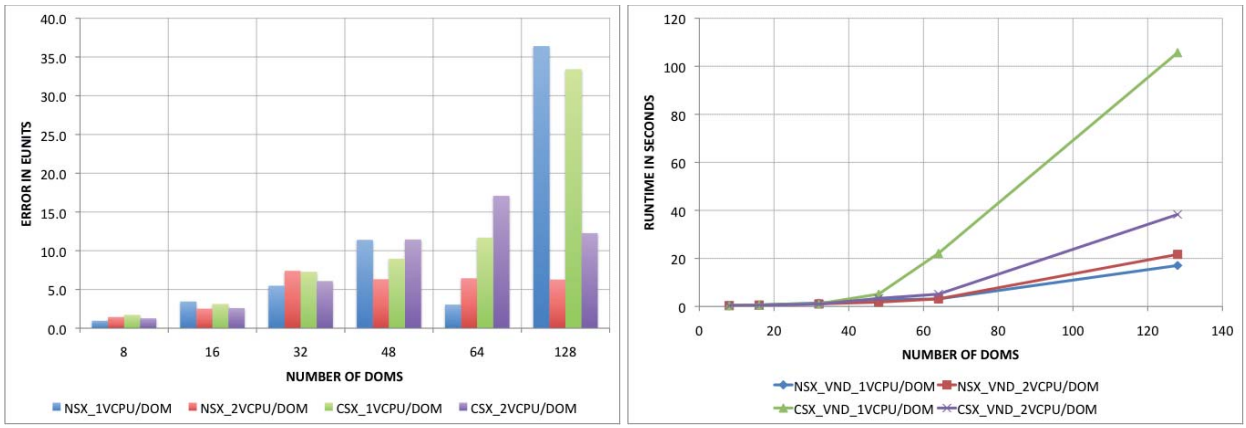

Figure 3: VND benchmark error plots (left); CND benchmark error plots (right)

Here too, the avalability 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 a strong evidence.

Cyber Security Benchmark Performance

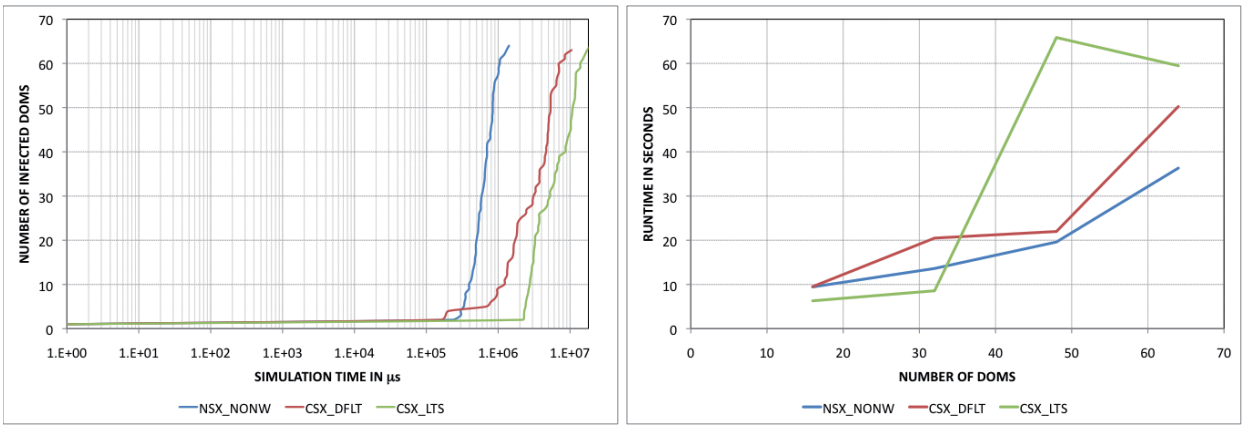

Figure 4: Infection propagation curve for 64-DOM (left) and runtime curves for 16, 32, 48 and 64 DOM scenarios (right) for cyber security benchmarks involving both CSX setups and, NSX without network control setup

The plots in Figure 4 (left) 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 4 (right), 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 4 (left).

This comparison reinforces the finding that the wall-clock time is not reliable, especially when large number 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 4 (right), especially in with large number of VMs.

3 VIRTUAL TIME MANAGEMENT WITH NETWORK CONTROL

In discrete-event network simulation, the *arrival* and *departure* of communication packets from and to the network at the end-hosts are modeled as *events*, since they determine the state-changes in the interacting nodes. Additionally, in contrast with network emulation, discrete-event simulation 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 by capturing the data, time-stamping it with virtual time, and delivering to the destination VM at the correct virtual time. Here, we describe our virtual network control (*i.e.*, controlling inter-VM network traffic) methodology, design and implementation in conjunction with the virtual timelines established in prior sections.

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 (DOM-0). By controlling the network, we would have the capability to introduce any simulated *delay* on any in-transit packet based on its source and destination addresses as required, and we directly dictate when a packet should arrive at its destination in virtual time order. Since, we are aware of the time of arrival of all the in-transit packets, by processing them in their *emit-time* order, we can induce a leap in *simulation 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:

a. Introduce a virtual delay without explicitly making a (byte-)copy of the packet buffer, or moving the transiting packets from kernel to user space
b. 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)
c. Minimal overhead while processing the trapped in-transit packets

3.1 Network Control Approach

To establish a control on an in-transit communication packet we should first trap the packets in transit. This can be achieved in the control domain as all communication packets traverse through it. One way of achieving this is by utilizing *netfilter* (Netfilter 2012), which is the packet filtering framework inside Linux® kernel services. The *iptables* rules can be used to redirect the in-transit packets to a specific *netfilter queue* (NFQ) maintained by the kernel packet filter, while the *libnetfilter_queue* function-APIs can be used to control the queued packets from the userspace.

The *libnetfilter_queue* function APIs allow 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. Using *netfilter* for the purpose of network contorl in network simulations, 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 acquiring the next in NFQ.

3.2 Network Control Design Alternatives

In this section, we discuss our experience in exploring the range of strategies for network control, from which the infeasible or inefficient approaches are discounted before arriving at the specific control mechanism we finally adopted.

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 DOM-0 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 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 temporarily hold packets while the packets' destinations have not reached the appropriate virtual times. We do not discuss Multi-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. While as many as 64K queues can be put to work at once using the *iptables* and *libnetfilter_queue* library, one can think of the following possible options based on the queues and service threads used.

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 *emit time* of the packet based its source and destination, then sets an NF_ACCEPT verdict when the simulation time catches up with the *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 is not correct for virtual network control purposes.

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, even though the processing ensures that a packet is released when the *emit time* catches up with *simulation time*, we cannot be sure of maintainance 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*.

Multithread processing using Multiple Queues (MT-MQ)

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 *emit time* of the processed packet can be greater than the *emit time* of the next packet, and hence, a service thread cannot *wait* for the *simulation time* to catch up with *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.

3.3 NetWarp Network Control (NNC) Architecture

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 (DOM-0) are set such that all the in-transit packets are routed to a single NFQ, with 0 *queue identifier* (*qid*). In Figure 5, this functionality is schematically presented using directional pointers suggesting the path of packet movement from a DOM-U 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.

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 5, 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.

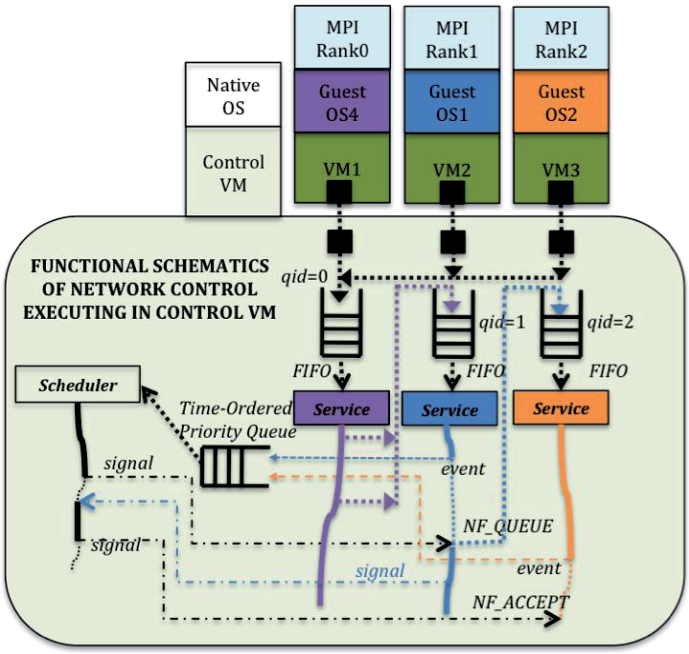

Figure 5: Functional schematics of NNC operation

The INT_DELAY mentioned previously refers to *intermediate-delay* 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* or *simulation 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 the signal from the signalled *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 perfect *emit time-order*.

3.4 NNC Virtual Time Management and 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 DOM-0 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 like, 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 DOM-0.

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 can reset 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, not used with NNC.

3.5 NNC Performance Benchmarks

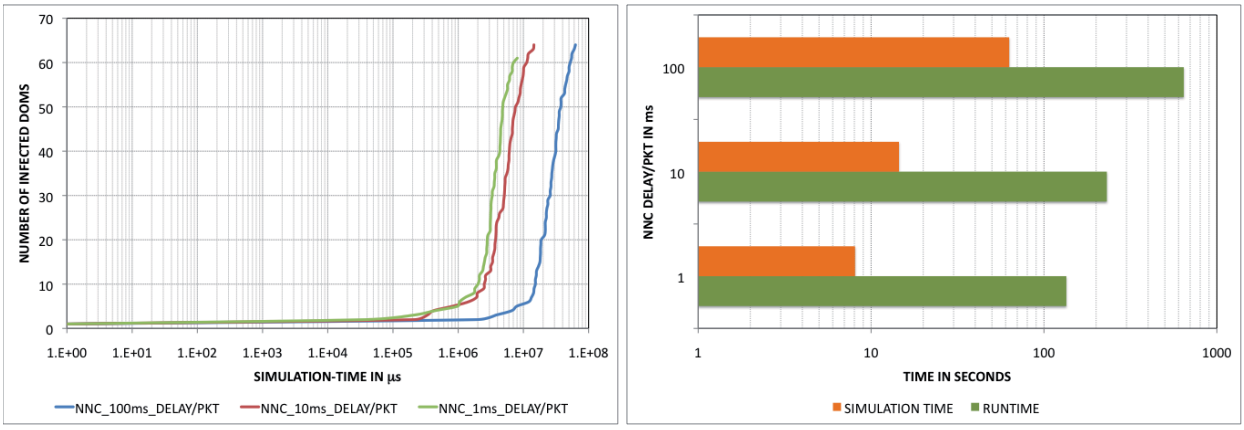

Figure 6: NNC verification and run time performance results with cyber-security application benchmark

To exercise the virtual network control and evaluate its performance, we use the cyber security (worm propagation) benchmark of (Yoginath, Perumalla and Henz 2012). The curves in Figure 6 (left) 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 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. In Figure 6 (right), 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 in-

volving 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*.

In Figure 7, 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.

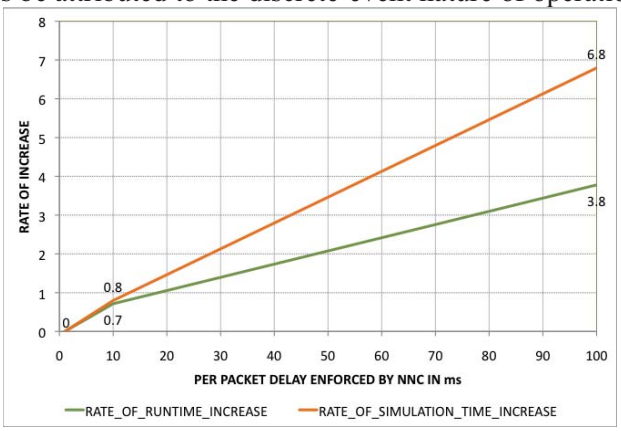

Figure 7: Rate of increase in the *runtime* and the *simulation time* with the increase in virtual packet delay

4 SUMMARY AND FUTURE WORK

Here, we presented performance results from scaled execution of benchmark scenarios containing 128 VMs hosted on a machine with 12 physical cores (24 cores with hyper threading enabled). The virtual time-ordered scheduler is observed to provide closely-evolving virtual timelines among all the VMs, approximating a single global virtual timeline among all VMs on the host machine, for use as simulation time in the larger network simulation framework. The issues and challenges in the design of a network communication controller for virtualization of inter-VM communication was presented. The implementation alternatives were explored, accounting for the concerns of scalability with the number of VMs and the timeliness of packet delivery in close match with virtual times of destination VMs. A multi-threaded, multi-queue scheme is identified as the best match that scales well with the number VMs. Results from an implementation of the scheme are presented, in which virtual time-ordered scheduling is executed in combination with virtualized communication, exercised and tested with a range of virtual latencies.

ACKNOWLEDGMENTS

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

Apostolopoulos, G. and C. Hasapis. 2006. “V-eM: A Cluster of Virtual Machines for Robust, Detailed and High-performance Network Emulation”. *In Proceedings of the 2006 14th IEEE International Symposium on Modeling, Analysis and Simulation of Computing and Telecommunication Systems*.

Bergstrom, C., et al., 2006. "The Distributed Open Network Emulator: Using Relativistic Time for Distributed Scalable Simulation." *In the Proceedings of the 2006 20th Workshop on Principles of Advanced and Distributed Simulation*, Singapore.

D. Chisnall. 2008. *The Definitive Guide to the Xen Hypervisor*, Prentice Hall.

Gupta, D., et al. 2006. "To Infinity and Beyond: Time- Warped Network Emulation." *In Proceedings of the 2006 3rd Symposium on Networked Systems Design and Implementation* (NSDI'06), San Jose, CA, USA.

Gupta, D., et al. 2008. "DieCast: Testing Distributed Systems with an Accurate Scale Model." *In Proceedings of 5th ACM/USENIX Symposium on Networked Systems Design and Implementation* (NSDI), San Francisco, CA, USA.

Grau, A., et al., 2008. "Time Jails: A Hybrid Approach to Scalable Network Emulation." In Proceedings of the 2008 22nd Workshop on Principles of Advanced and Distributed Simulation, Rome, Italy.

Liu, J., R. Raju, and Z. Ming. 2010. "Model-Driven Network Emulation with Virtual Time Machine." *In Proceedings of the 2010 Winter Simulation Conference* (WSC) 2010, Baltimore, MD, USA. Edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan and E. Yücesan.

Liu, J., L. Yue, and H. Ying. 2009. "A Large Scale Real-time Network Simulation Study using PRIME." *In Proceedings of the 2009 Winter Simulation Conference* (WSC), Austin, TX, USA. Edited by M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls.

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

R. M. Fujimoto. 2000. *Parallel and Distributed Simulation Systems*, Wiley Interscience.

Yoginath, S.B., and K.S. Perumalla. 2011. "Efficiently Scheduling Multi-core Guest Virtual Machines on Multi-core Hosts in Network Simulation." *In Proceedings of the 2011 Principles of Advanced and Distributed Simulation*, Nice, France.

Yoginath, S.B., K.S. Perumalla, and B.J. Henz. 2012. "Taming Wild Horses: The Need for Virtual Time-based Scheduling of VMs in Network Simulations." *In Proceedings of the 2012 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Washington D.C, USA.

AUTHOR BIOGRAPHIES

SRIKANTH B. YOGINATH is a Research Staff Member at the Computational Sciences and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, TN. He is also a PhD candidate at the School of Computaional Sciences and Engineering, Georgia Institute of Technology, Atlanta, GA. His email address is yoginathsb@ornl.gov.

KALYAN S. PERUMALLA founded and leads the High Performance Discrete Computing Systems team at the Oak Ridge National Laboratory. He is a Senior R&D Manager in the Computational Sciences and Engineering Division at the Oak Ridge National Laboratory, and an Adjunct Professor at the Georgia Institute of Technology. He holds a PhD in Computer Science (1999, Georgia Institute of Technology). His email address is perumallaks@ornl.gov.

BRIAN J. HENZ is a Research Staff Member in the Computational and Informational Sciences Directorate at the U.S. Army Research Laboratory, Aberdeen Proving Ground, MD. He holds a PhD in Mechanical Engineering (2009, University of Maryland, College Park). His email address is brian.j.henz@us.army.mil.