# Efficiently Scheduling Multi-core Guest Virtual Machines on Multi-core Hosts in Network Simulation

Srikanth B. Yoginath and Kalyan S. Perumalla
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
yoginathsb@ornl.gov, perumallaks@ornl.gov

***Abstract*—Virtual machine (VM)-based simulation is a method used by network simulators to incorporate realistic application behaviors by executing actual VMs as high-fidelity surrogates for simulated end-hosts. A critical requirement in such a method is the simulation time-ordered scheduling and execution of the VMs. Prior approaches such as time dilation are less efficient due to the high degree of multiplexing possible when multiple multi-core VMs are simulated on multi-core host systems. We present a new simulation time-ordered scheduler to efficiently schedule multi-core VMs on multi-core real hosts, with a virtual clock realized on each virtual core. The distinguishing features of our approach are: (1) customizable granularity of the VM scheduling time unit on the simulation time axis, (2) ability to take arbitrary leaps in virtual time by VMs to maximize the utilization of host (real) cores when guest virtual cores idle, and (3) empirically determinable optimality in the tradeoff between total execution (real) time and time-ordering accuracy levels. Experiments show that it is possible to get nearly perfect time-ordered execution, with a slight cost in total run time, relative to optimized non-simulation VM schedulers. Interestingly, with our time-ordered scheduler, it is also possible to reduce the time-ordering error from over 50% of non-simulation scheduler to less than 1% realized by our scheduler, with almost the same run time efficiency as that of the highly efficient non-simulation VM schedulers.**

**Keywords—Virtual Machines, Virtual Clocks, Scheduling, Hypervisors, Network Simulation, Network Emulation**

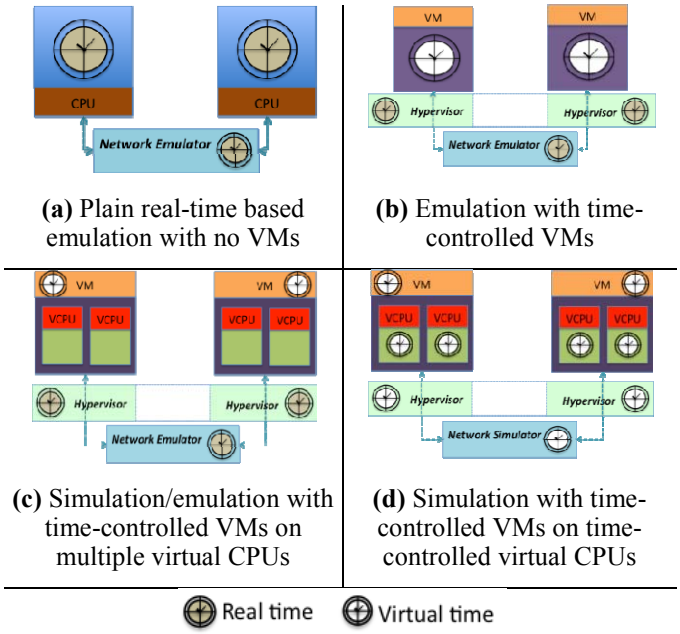
## I. INTRODUCTION

Historically, network simulators realized approximations of the application layer via software models of the same and integrating them with models of the lowers layers such as transport-, network- and physical layers. Since the software models were part of the network simulator timestamp-ordered discrete event simulation of their overall integrated execution was straightforward. Within the last decade, the application layer models have been greatly enhanced in fidelity via the use of virtual machine (VM) technologies to incorporate actual, complete operating system (OS) instances to serve as end-host behaviors. The integration of OS instances via VM platforms into the rest of the network simulator raised the issue of real- and virtual time integration, which was largely solved by the use of approaches such as time dilation. Such approaches have been adequate for single-processor virtual machines multiplexed on single-processor host machines. More recently, with the emergence of multi-core host systems, the issue of time control and time integration has resurfaced with the added dimension of multi-core considerations.

Consider a network simulator that employs multiple host (computing) nodes to host multiple guest (simulated) nodes. The guest nodes are realized via virtual machines that act as surrogates of simulated end-host nodes in the simulated scenario. The issue at hand is that both the host nodes as well as the guest nodes are multi-core systems, whose simulation time advances must be controlled and coordinated to deliver virtual time-ordered integrated execution with the rest of the network simulator. Native VM schedulers are clearly inappropriate because they are designed with processor utilization and fairness considerations that are fundamentally different from virtual time-ordered scheduling. While time dilation [1] approaches can be employed in the multi-core setting as well, they are fundamentally sub-optimal in their host processor utilization when guest virtual cores idle in the scenarios. This sub-optimality can become more pronounced when the number of cores in either the guest and/or the host system is large. Our present work explores VM integration into network simulators by generalizing the virtual time representations taken down to the level of each virtual core.

Figure 1 shows our approach in relation to a few past approaches: (a) represents free-running emulations with only real time-based clocks, (b) represents VMs integrated into emulations via virtual clocks controlled with approaches such as time dilation, (c) shows multi-core execution with one virtual clock per VM, and (d) shows a distinct virtual clock for every entity down to the level of each virtual core (VCPU). The scheme shown in Figure 1(d) is the focus of this paper.

Here, we are interested in supporting the capability to simultaneously schedule multiple single-core and/or multi-core VMs in simulation-time order on native multi-core hardware. Unlike schedulers that maintain only one virtual clock per VM, our scheduler supports a separate virtual clock for each virtual core (VCPU) of a multi-core VM, maintained with its own simulated timeline as a Logical Process (LP) in the simulation framework, making it amenable for incorporation into a discrete-event based simulation system.




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

**(b)** Emulation with time-controlled VMs

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

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

**Figure 1: Real-time vs. virtual-time representations in a few simulation/emulation approaches**

### A. *Organization*

The rest of the article is organized as follows. A brief overview of virtualization, its related terminology, and its use in network simulation are provided in the next section. The implementation details of our scheduler are documented in Section III, followed by a test setup description in Section IV, and a performance analysis in Section V. The findings are summarized in Section VI, related work discussed in Section VII, and future work is presented in Section VIII.

## II. BACKGROUND AND TERMINOLOGY

### A. *Virtualization*

Virtualization replaces the OS to be the lowest level software running directly over the hardware. By doing so it segregates the hardware and the OS by virtualizing the hardware components to the OS, and relieves the tight hardware-OS coupling. As an important consequence, OS instances from different vendors co-exist and run on the same physical resource, interacting with the virtual counterparts of the physical hardware. The thin multiplexing layer between the hardware and the OS instances is called the *hypervisor*.

Generally there are two types of virtualization: Full-Virtualization (FV), and Para-Virtualization (PV). In FV, the hypervisor supports any unmodified guest OS to run. In PV, the hypervisor provides hypercalls for accessing any hardware service and this requires the modification of a guest OS.

### B. *The Xen Hypervisor*

The Xen hypervisor [2] is a popular, powerful open source industry standard for virtualization, supporting a wide range of architectures including x86, x86-64, IA64, and ARM, and guest OS types including Windows®, Linux®, Solaris® and various versions of BSD OS [3]. Although Xen started as a PV, FV is supported using QEMU, a generic machine emulator and virtualizer[4].

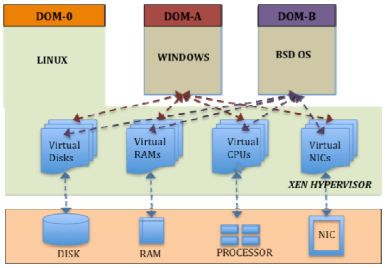



**Figure 2: Xen architecture**

Figure 2 shows a schematic of guests running on the Xen hypervisor. The guest OS in Xen terminology is referred to as Guest Domain or simply DOM. Each DOM has a unique identifier called its Domain ID (DOM-ID). The first DOM called DOM0 is a privileged one with special management privileges. System administration tasks such as suspension, resumption, and migration of DOMs are managed via DOM0.

In a multi-core system, a software interface called Virtual Central Processing Unit (VCPU) is supported by Xen to provide a virtual equivalent of a processor core. Another interface called Physical CPU (PCPU) maps to an actual hardware core. Thus, all PCPUs are contained within Xen, and the number of PCPUs, $Cp$, is equal to the number of physical cores (across all processor sockets on the machine). Each DOM$i$ uses a number of VCPUs, $Ci$, as configured by the user. In network simulations, it is desirable to have $Cp << \sum Ci$, i.e., the number of host cores is much smaller than the total number of virtual cores being multiplexed, in order to be able to host a large number of simulated end-hosts on relatively small simulation test-bed hardware. Also, $Cp \geq Ci$ for every DOM $i$, i.e., the number of virtual cores in any individual DOM is smaller than the number of physical cores available, because the test-bed hardware is usually a high-end system with many cores than an end-host in a typical simulated network scenario.

### C. *NetWarp*

NetWarp, is a scalable computer network simulation framework being developed at our institution to utilize a large number of multi-VCPU VMs sustained on multi-core nodes interoperating with a packet-level backbone network simulation also executing in parallel. The execution architecture thus presents two distinct components of time-controlled execution, namely, the control of intra-node pacing among VMs within a multi-core node, and the control of inter-node pacing for virtual time coordination across multiple multi-core nodes. In this article, we focus on the first component of NetWarp, namely, time-controlled execution of multi-VCPU VMs on multi-core host nodes.

### D. *Design Principles*

The problem of scheduling multi-VCPU DOMs on a multi-core system is defined by the following considerations when used in network simulation.

The DOMS are bound together as one simulation application requiring that the simulation time must be global across all the DOMs, and the simulation time must account for the idle-VCPU time within a DOM. A strict time-order in scheduling the VCPUs for execution must be maintained. The scheme must also allow addition and removal of DOMs

dynamically in the simulation without adversely affecting the performance of the running simulation.

In Xen, the scenario network setup, interaction with other DOMs and the monitoring of their execution are performed by DOM0. Also, the device of communication (bridge or router) to which all the other DOMs connect is serviced by DOM0. Hence, DOM0 must be given sufficient cycles to ensure that interactivity is not compromised. Yet, DOM0 must not starve other DOMs.

A *tick* is the time allotted for a particular VCPU when it is scheduled for execution. The chosen tick duration must ensure low run time: the smaller the tick size, the better is the ability to interleave execution among VCPUs. However, this also increases the number of scheduling operations, which can impose larger overhead.

To accommodate these considerations, we create a new scheduler for Xen hypervisor.

## III. IMPLEMENTATION

We refer to the NetWarp scheduler for Xen as NSX and the native scheduler of Xen, called the Credit Scheduler[2,5,7], as CSX. In this section, we describe the implementation details of NSX to meet our needs as outlined in the preceding section, and provide a comparison to the internals of CSX.

### A. *Scheduler Structure in Xen*

Scheduling in Xen shares some concepts with an operating system that provides an *N:M* threading library. In such a system, the operating system kernel schedules *N* threads (typically one per physical context). Upon each OS thread, a user space library multiplexes into *M* user-space threads. In Xen, the OS threads are analogous to the VCPUs and the user-space threads represent processes within the domain [5]. Essentially, there exist three scheduler tiers in Xen:

(a) User-space threading library maps the user-space threads to kernel threads (with in a DOM)
(b) Guest DOM OS maps its kernel threads to VCPUs
(c) Hypervisor maps each VCPU to a physical CPU (PCPU) dynamically at run time.

Both CSX and NSX are ultimately concerned with mapping VCPUs to PCPUs. A new scheduler such as NSX can replace the CSX relatively easily due to the modular design of Xen. The Xen design prescribes a set of functions that needs to be implemented and interfaced by the new scheduler. Hence, the CSX implementation can be viewed separately from Xen's internals and schedulers with different strategies for accounting can be realized in a similar way.

By reassigning the relevant function pointers to the scheduler's interface variables, our scheduler functionality is integrated into Xen. For example, the *init_domain* variable is a function pointer to *csched_dom_init*, which is used for the initialization of a DOM. After integrating our scheduler, Xen invokes our routines for decisions on scheduling any VCPU onto a PCPU.

### B. *NetWarp Scheduler Data structures*

Figure 3 shows the static object created for the NSX to interface with Xen. Similar to CSX, we maintain four different data-structures in NSX, namely,

a. *nw_pcpu* – correspond to the PCPUs
b. *nw_vcpu* – correspond to the VCPUs
c. *nw_dom* – corresponds to a DOM
d. *nw_private* – is global data shared by all DOMs.

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

**Figure 3: NSX scheduler interface**

The *nw_pcpu* data-structure as shown in Figure 4, comprises a VCPU queue called *runq*, a *timer*, and a counter named *tick* that keeps a count of the number of ticks used. The number of instances of the *nw_pcpu* data structure corresponds to the number of processor cores in the physical hardware. The next VCPU to schedule on a particular physical core is chosen by the Xen scheduler from the *runq* list of the *nw_pcpu* object corresponding to that physical core.

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

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

**Figure 4: CSX's PCPU vs. NSX's PCPU structures**

Figure 5 compares data-structures *csched_vcpu* and *nw_vcpu* (some variables concerned with keeping a log of VCPU status in *csched_vcpu* data-structure are not included in Figure 5 for clarity purposes). As can be seen most of the variables in *nw_vcpu* are the same as in *csched_vcpu*, except that the *credit* and *pri* (priority) variables of the *csched_vcpu* are replaced by the *nticks* variable of *nw_vcpu* and the *ref_time*. The *nticks variable* keeps track of a number of ticks that the VCPU has used up, and the *ref_time* gives a reference-time from which *nticks* are tracked. The *ref_time* along with the *nticks* gives the Local Virtual Time (*LVT*) as $LVT = ref\_time + (nticks \times tick\_size)$ for a VCPU.

```
struct csched_vcpu {
   struct list_head runq_elem;
   struct list_head active_vcpu_elem;

   struct csched_dom *sdom;
   struct vcpu *vcpu;
   atomic_t credit;
   uint16_t flags;
   int16_t pri;
      …
};
```

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

**Figure 5: CSX's VCPU vs. NSX's VCPU structures**

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

```
struct csched_dom {
   struct list_head active_vcpu;
   struct list_head active_sdom_elem;

   struct domain *dom;
   uint16_t active_vcpu_count;
   uint16_t weight;
   uint16_t cap;
};
```

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

**Figure 6: DOM variables in CSX vs. NSX**

Figure 6 shows the CSX's data-structure *csched_dom* and NSX's *nw_dom* for domains. The *nw_dom* comprises a list named *active_vcpu*, which is a list of *nw_vcpu* belonging to this DOM. It contains a member-variable named *active_sdom_elem*, which is initialized to point to self, and aids in the functioning and operation of the *active_sdom* queue in the *nw_priv* global object of type *nw_private*. The *nw_priv* holds all the active DOMs in this *active_sdom* queue.

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

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

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

**Figure 7: CSX and NSX private data**

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

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

The variable named *idlers* keeps track of the idling *VCPUs* and the variable *lock* is used for updating this global object. Figure 7 compares the global data-structure that is *csched_private* of CSX and *nw_private* of NSX. Most of the CSX variables namely, *credit*, *weight*, *balance* and *runq_sort* of the *csched_private* data-structure are removed, while keeping the remaining others unchanged.

C. *Implementation of Features*

To fulfill simulation time ordering requirement, each VCPU keeps track of its utilized ticks (time units) of its execution using variables *nticks* and *ref_time*, the reference time from which *nticks* has elapsed. These two values are used to calculate the LVT of the VCPU.

The utilization of the VCPUs increases as the interconnected DOMs execute any application and as they do, the corresponding VCPU tick values increments. Hence the simulation time increases as it is dependent on the LVT values of VCPUs. The least value of the VCPU can be safely considered as the elapsed simulation time in a single core machine. In a multi-core system, we consider the maximum LVT among the VCPUs of a DOM as the *dom_lvt* and the minimum of the *dom_lvt*s is considered as the simulation time.

A periodic timer named *ticker* corresponding to each core (in a multi-core processor) is maintained in the *nw_pcpu* data-structure. This periodic timer goes off at every *tick* and when this goes off, the accounting (charging tick time) is performed to the *current* runnable VCPU. Then a soft-interrupt for scheduling this VCPU is raised on the physical core, which is selected either based on the provided VCPU-PCPU affinity-map or on the criteria that the VCPU has most idling neighbors in its grouping. This code in NSX is exactly same as that found in CSX. Thus at every tick, generated by the *ticker* timer from either of the nw_*pcpu* object corresponding to a cpu-cores a VCPU is scheduled to run on a CPU-core.

Another global timer named *master_ticker* is made to go off for every 10 ticks. This calculates the LVT of the DOM, which is the maximum of the LVT values among all its VCPUs. This is carried out for every DOM and in each case the *dom_lvt* is recorded in its corresponding *nw_dom* object. While this doesn't have much functional significance, it allows us to achieve better performance and it is also used to artificially increment the LVT of the VCPU with lagging LVT to its *dom_lvt* value, periodically, as discussed next.

*Accounting for the effect of idle-VCPU cycles*

If left unchecked the VCPUs run asynchronously. Even, within a single DOM, there could be differences in the LVT values of its VCPUs, as this is dependent on the task assignment of the guest OS onto its VCPUs. To limit these differences, and their subsequent propagation, between the LVT values of the VCPUs within a single DOM, we adjust the LVT values of all the VCPUs to the maximum of the LVT values among them (i.e. *dom_lvt*). This adjustment is carried

out at regular intervals, which we chose to be a minute. A periodic timer *master_lvt_ticker* is used for this purpose. When this timer goes off, the *Adjust_lvt* routine is called and this function adjusts the LVTs of all the VCPUs to the *dom_lvt*.

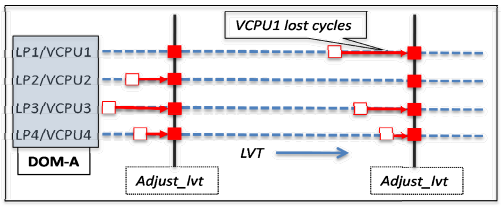



**Figure 8: Accounting for VCPUs with lagging LVTs**

The NSX scheduler enables us to capture the effect of lost cycles due to the idle cycling of the peer-VCPU cores while using the multi-core processors. Note that the idle VCPUs are not charged and hence their LVT value remains stationary, while the LVTs of the active VCPUs increase. This results in a staggering of LVT values for different VCPUs within the same DOM as shown in the Figure 8. Since the VCPUs being idle or active is dependent on the guest OS's scheduling mechanism, the lagging VPCUs can be tracked with periodic updates. To illustrate, Figure 8 shows a DOM with 4 cores and the LVT values of each VCPUs staggered and being pulled to the *dom_lvt* value during periodic updates serviced by the *Adjust_lvt* routine.

*Accounting the effect of Idle Doms in network*

Similarly, a *master_all_lvt_ticker* is used to adjust the LVTs of the DOMs (i.e. all the VCPUs) to the *sys_lvt_next* value after some long time, as shown in Figure 9. The only difference is that the *Adjust_all_lvt* routine that services this ticker checks if any of the DOMs are lagging beyond certain difference period (*NW_MAX_DIFF*) from *sys_lvt* before pushing the LVTs of all the VCPUs from all DOMs to *sys_lvt_next*. In our runs the *NW_MAX_DIFF* was set to 10 ticks.

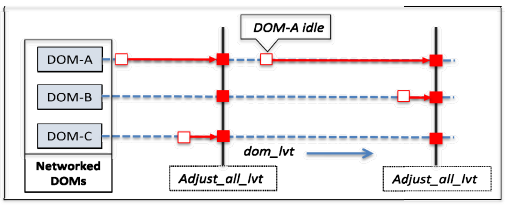



**Figure 9: Accounting for Idle DOM time**

This updating procedure of LVTs of all VCPUs to *sys_lvt_next* accounts for the time of idle-DOMs during the network simulation. All global timers are maintained in *nw_priv*. Thus the LVT corresponding to each VCPU, along with the help of intermediate VCPU-specific timers and global timers, keep track of the simulation time.

The VCPU with the least LVT must always be scheduled for next execution. This is ensured during the insertion of VCPU in to the run-queue, which happens either during *nw_schedule* is called or when the VCPU awakes from sleep (*nw_wake*). Since, the LVT value of VCPUs is always non-decreasing and never changes in between updates, it is not necessary to sort the queue in between as required in the CSX. However, this criterion is not sufficient to ensure the required property of "least-LVT scheduled first" on machines with multi-core processors, and, as such, the scheduling needs to be extended further, as described next.

The scheduler maintains a run queue for every physical core. When the *nw_load_balance* routine is used, it safely steals the least LVT valued VCPU from its peer PCPU's *nw_pcpu's runq* and schedules it for execution. This functionality is very similar to that carried out by the CSX. The only difference is that CSX looks for a VCPU with higher-priority value while the NSX looks for the least-LVT value. This should ensure the least-LVT valued VCPU is always scheduled first.

## New DOM Boot Process Considerations

Not all the DOMs participating in the simulation can be expected to start at the same time. Whenever a new DOM needs to be added to the existing simulation scenario, it needs to start at the current simulation time, so that its execution is in coherence with other DOMs. This is ensured in the *vcpu_init* routine, which assigns the *sys_lvt* value to all the VCPUs of the newly active DOM. Thus ensuring that the arbitrarily added DOM works in coherence with the other DOMs participating in the simulation.

Also, since the *sys_lvt* computations are carried out only across the active DOMs, we ensure that a sudden departure of any DOM in the middle of the simulation would not affect the rest of the simulation. These features provide an opportunity to arbitrarily grow or shrink the number of DOMs simulated during the simulation.

## Scheduling DOM0

The DOM0 is not only an interface for interacting and monitoring the other DOMs but also the one where a link (bridge or a router) exists that interconnects the user DOMs. In our example network used for testing, we create a bridge in DOM0 and all user DOMs are connected to each other via this bridge. Consequently, all the communications between the user DOMs passes through DOM0. Hence, sufficient number of compute cycles must be provided to DOM0, so that the user interactivity doesn't suffer and also other necessary tasks for the functioning of the communicating DOMs happen without delay. In a similar way, we also need to ensure that the DOM0 accedes to other DOMs, so that they are not starved.

To achieve this, the VCPU's of DOM0 are always maintained at *sys_lvt* of the DOMs. Note that *sys_lvt* is the minimum of *dom_lvt* and the *dom_lvt* is the maximum of all the VCPUs LVT values in that DOM. This essentially says that the DOM0 VCPUs accede to the VCPU with the least LVT, if there is one that lags behind the *sys_lvt* value. The updates to *dom_lvt* and *sys_lvt* are performed periodically after every '*t*' ticks. By increasing or decreasing this tick value, we were able to vary the number of CPU-cycles provided to DOM0. In our testing we assigned *t =10* ticks.

## Small Time-slice

The credit scheduler of Xen uses a default time-slice value of 30ms, which is much larger than the network link latencies

required in our simulation scenarios. Hence we made modifications to CSX so that the time slice can be reduced much below 30ms in our experiments.

Every time Xen needs to schedule a new VCPU, it invokes the *nw_schedule* routine. This routine picks up the least-LVT valued VCPU to execute and provides a *time-slice* (number of *ticks*) for the VCPU to execute. Xen measures time slices in milliseconds, which we changed to microseconds for finer time granularity control. The number of ticks in a *time-slice* was kept 1. Thus, by changing the tick size in the pre-processor statements of the scheduler code, we were able to alter the tick size as needed.

### D. *Overall Scheduling Process in NSX*

*Schedule service routine*: Xen's scheduling framework ensures that there is always at least one VCPU in the *run-queues* of each PCPU. When the scheduler is interrupted, the service routine in NSX inserts the *current* (currently being serviced VCPU) into the run-queue of the PCPU (cpu-core) that was interrupted. Then the VCPU with least LVT is searched in all the *run_queues* of the existing PCPUs (cpu-cores). This VCPU is removed from the *run_queue* and would be scheduled to run next. The time-slice (number of ticks) that this VCPU should run is also provided in this routine. The selected VCPU and the *time-slice* are returned back from this service routine of NSX to Xen. Xen, during context switching process, assigns the selected VCPU as *current* and the *current* VCPU is executed on the actual physical core.

*VCPU accounting*: The *master_ticker*, that goes off periodically at every *tick* (same as *time-slice* in our implementation) increments the LVT of the *current* VCPU by *tick* size. Then it raises a *schedule* software interrupt on the core that has the most idling peer VCPUs in its DOM. This *schedule* interrupt is serviced by the schedule service routine.

## IV. EXPERIMENTAL SETUP

### A. *Test Scenario*

We implemented a simple parallel program using the Message Passing Interface (MPI) library to test and also to compare NSX against CSX. The parallel execution is distributed across the guest DOMs such that exactly one process of the parallel execution is run on one guest DOM. Three domains, DOM-A, DOM-B and DOM-C, hosts three MPI processes of rank-0, rank-1 and rank-2, respectively, which participate in multiple *messaging rounds*. In a *messaging round*, the rank-0 process (in DOM-A) sends a message to rank-1 process (in DOM-B) and a then message to rank-2 (in DOM-C). The rank-1 process that was waiting for message from the rank-0 process receives it and then sends out a message to rank-2 process immediately. The message sent out by rank-0 is an integer 0 and the message sent out by rank-1 is an integer 1. We use *MPI_ANY_TAG* and *MPI_ANY_SOURCE* while receiving an MPI messages to ensure that the MPI_Recv permits reception in any order for incoming messages.

Note that the interacting parallel processes are running on different DOMs. Hence for an individual messaging round, if we were to consider the network delay experienced by the communication message a constant, and, if the DOMs are being scheduled in the strict time-order, then the causal order of arrival of messages received by rank-2 (in DOM-C) must be 0-1. In other words, rank-2 must first receive the message from rank-0, before it receives the message from rank-1. Further, the blocking send and receive MPI calls used in realizing the test-program largely reduces the time gap between the direct message from rank-0 and the relayed message from rank 1, at rank-2. Hence, this makes the test fine grained and tight.

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

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

For the CSX readings, the *weights* for all the DOMs were kept same (at 256) and none of them was capped, thus ensuring maximum fairness in the working of CSX.

### B. *Test Setup*

The test setup comprised a Xen hypervisor v3.4.2, with Linux running as its DOM0 and all the other three guest-DOMs (DOM-A, DOM-B and DOM-C). Each of the guest-DOMs was assigned a static IP address.

In DOM0, a bridge named *privatebr*, for establishing network connections between the guest-DOMs was created using the *brctl* tool. The DOM0 itself does not connect to *privatebr* and hence remains separate from the network of guest-DOMs. However, since the *privatebr* exists in DOM0, all the communication messages from the guest-DOMs pass through DOM0.

### C. *Test Machine*

The setup and the test runs were carried out on a MacBook-Pro with Intel® Core™ 2 CPU T7600 @ 2.33 GHz, with 3 GB memory, running OpenSUSE 11.1 Linux over Xen 3.3.1.

The source code of Xen 3.4.2 distribution was used for making scheduler modifications. The resulting boot image xen-3.4.2.gz was used to boot the hypervisor, and the boot-loader grub configuration was changed to enable the selection of modified Xen during the boot up. All the guest-DOMs were configured to run OpenSUSE 11.1 Linux and they were installed as para-virtualized DOMs for performance reasons.

The test program was written in the C programming

language using MPI (OpenMPI v1.4.1) library.

## V. PERFORMANCE RESULTS

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

### A. *Case 1: Single VCPU per DOM*

Thirty runs with each run performing 1000 *messaging rounds* were carried out. The mean-percent-error and runtime are shown in the Figure 10 and Figure 11. A table of 95% confidence intervals (CI) is provided for CSX and NSX for the top two curves.

For NSX, we see a dramatic reduction in the *mean-error* with the increase in the tick-size. In the CSX the *mean-error* drops by a small amount initially and steeply increases later with the increase in the tick size. From Figure 10, we see that as the tick size increases from 30µs to 30ms, the mean error percentage decreases from 4% to 0%, whereas CSX increases from 3% to 56%.

Figure 11 shows almost same runtime for both NSX and CSX. However, the runtime of NSX suffers at lower tick sizes. The increase in the runtime with increase in the tick size is expected because larger than necessary tick-size is allotted for every VCPU, which essentially is wasted.

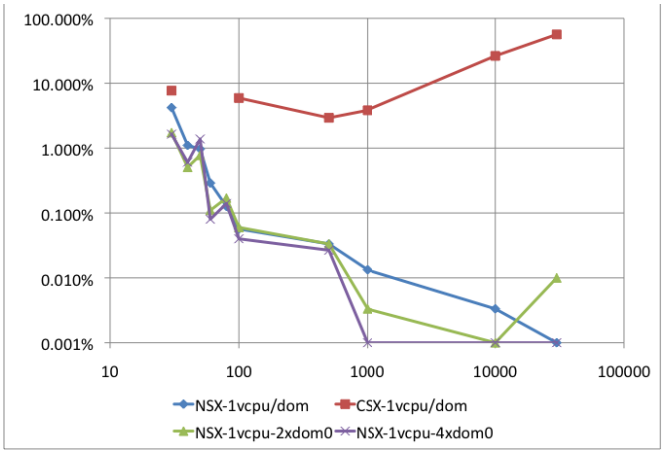
### B. *Case 2: Multiple VCPUs per DOM*

A similar reduction in the *mean-error* is observed in Figure 12. However, unlike in the 1-VCPU scenario, it does not go down to absolute zero, but stays around 0.04%. Similar to the 1-VCPU scenario, an increase in CSX's mean-error with increase in the tick size is observed. At 30ms tick-size, the error was at 49%.

The runtime plot in Figure 13 shows a steep increase in the NSX runtime with increasing tick size, which is decreased by increasing the DOM0 tick size, as discussed shortly.

### C. *Analysis of Results*

Causal error in CSX: The CSX caps the *over-scheduled* VCPUs and chooses *under-scheduled* VCPU for scheduling. In doing so, it constantly re-orders the *run_queue* based on priority, and, when all VCPUs become over-scheduled, the credits are re-assigned. This process continues over the simulation.




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

**Figure 10: Mean-error (y-axis in %) vs. tick-size (x-axis in microseconds) for the NSX, the CSX, the NSX with 2x and 4x DOM0 tick sizes. Mean error is obtained from 30 runs with each run of 1000 messaging rounds in a 1-VCPU per DOM scenario**

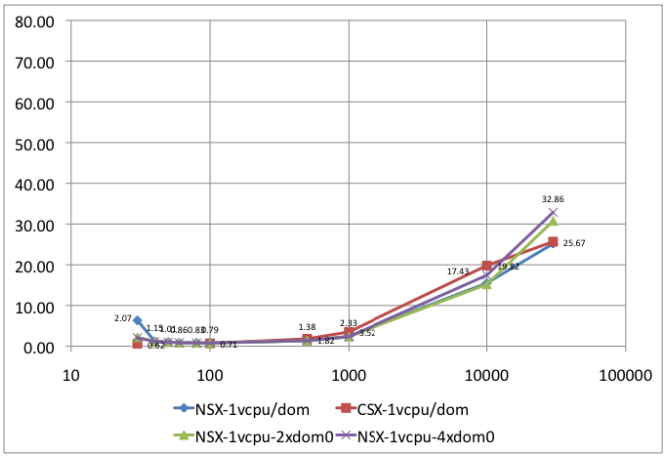
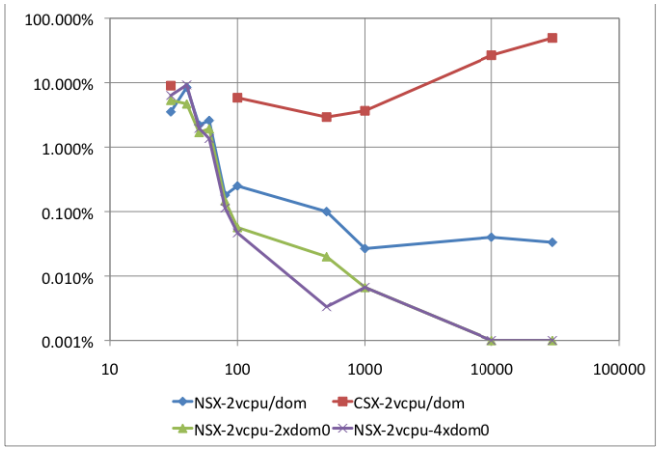



**Figure 11: Single VCPU/DOM case - runtime curves of *mean-error* runs, for the NSX, the CSX, the NSX with 2x and 4x DOM0 tick size**

The CSX maintains fairness (in a non-simulation sense) but causality is affected. This is because the program execution and communications are asynchronous and, the load on the processes in the parallel program is usually not equal. Capping one VCPU while scheduling others increases the probability of committing causal error. For example, capping the VCPUs of DOM-A holds back DOM-A from sending message to DOM-C in time as expected; the under-scheduled DOM-B VCPU, given more-cycles, will be able to send message to DOM-C before DOM-A does, hence creating a causal error.




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

**Figure 12: Tick size vs. error plots similar to Figure 10, but with 2 VCPUs per DOM**

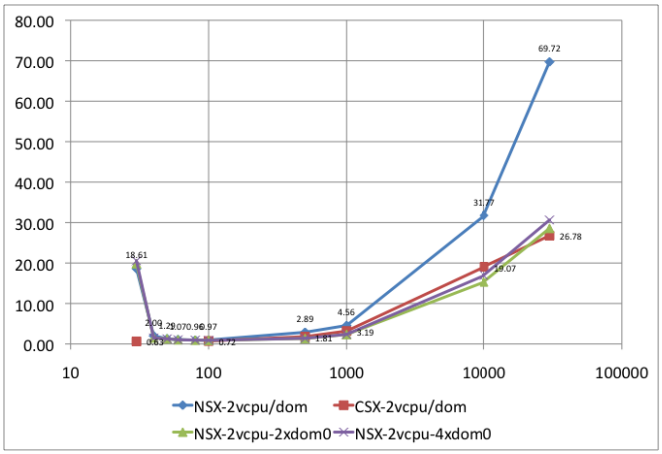



**Figure 13: Tick size vs. runtime similar to Figure 11, but with 2 VCPUs per DOM**

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

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

Since DOM0 manages the bank-end drivers and the network-bridge, and also serves as interaction interface, its starvation due to a higher context switching frequency aggravates the error-rate. The error-rate can be significantly alleviated as seen in Figure 10 and Figure 12, by providing larger tick sizes to the DOM0. In our experiments, we provided twice (2x) and four times (4x) tick sizes to DOM0. This also reduces the runtime as seen in Figure 11 and in Figure 13, and, the impact of this change is clearly evident. As the DOM0's LVT does not affect the *sys_lvt* (the simulation time), this change does not adversely impact the error in the test-scenario.

*Mean-error and Runtime at lower tick sizes*: The smallest-tick size is very essential to simulate low-latency and high-bandwidth network. Hence, it is interesting to detect the earliest point at which a continued reduction in tick size gives acceptable error without significant increase in the runtime. At 20µs tick size, both CSX and NSX pose interactivity problem. Thus the least tick-size with which we could run the experiments was 30µs.

For the 1-VCPU per DOM scenario, Figure 10 and Figure 11, with 2x DOM0 indicate the best error reduction and at 60µs, the error reduces to 0.1% and the runtime 0.86 seconds. From Figure 12 and Figure 13, we see that, in the 2 VCPU/DOM scenario, the 2x DOM0 case gives an overall best error reduction and a better run time. At 80µs, the error reduces to 0.18% and the run time by 0.95 seconds.

## VI. SUMMARY

The following table summarizes the results.

| | |
|---|---|
| *Simulation-time*<br>Global time<br>Idle-VCPU accounting<br>Idle-DOM accounting | <br>Achieved by *sys_lvt*.<br>Is done periodically.<br>Is done periodically. |
| *Causality*<br>Time-order<br>Errors | <br>Ensured in scheduling<br>Errors reduced to <1% |
| *New-DOM needs*<br>DOM add/remove<br>No detrimental effect<br>Scenario modification reflected in simulation | <br>Supported<br>Works as expected<br>VCPU of a new DOM initialized to *sys_lvt* ensures addition |
| *DOM0 needs*<br>Enough CPU-cycles<br>Lack of CPU-cycles to DOM0 might introduce causal errors<br>Starvation of DOMs by DOM0 | DOM0 is always maintained at sys_lvt<br>Is addressed by changing the tick-size just for DOM0<br>Absence of starvation implicit from results. |
| *Small tick slice*<br>Smallest tick size<br>Error less than 1%<br><br>Achievable runtime for required error bound | 1VCPU/DOM- 60µs<br>2VCPU/DOM- 80µs<br>1VCPU/DOM- 0.1%<br>2VCPU/DOM- 0.18%<br>1VCPU/DOM- 0.86s<br>2VCPU/DOM- 0.95s |

## VII. RELATED WORK

Within the area of network simulation research, there have been several prominent advancements realized in the form of simulators and testbeds such as Emulab, GTNetS, PDNS, SSF, DaSSF, PRIME, and Qualnet, to name just a few. Among the systems that employed virtualization for higher fidelity, the well-known ones include time dilation [1], time jails[9],

virtual time[10], real-time interface optimizations[8,11,12], and others [13-16]. However, we are not aware of prior work on simulation-specific schedulers for simulating multi-core guests on multi-core hosts. It may be possible to integrate our scheduler into one of the current simulation frameworks such as Qualnet, NS3 or PRIME, which we intend to explore as future work.

## VIII. CONCLUSION AND FUTURE WORK

In the evolution of network simulators with higher fidelity, virtual machine-based behaviors clearly play an important role in the future. This entails supporting multi-core guest VMs and multi-core host nodes executed in simulation time order to control VM interaction within a host node. Traditional fairness-oriented (or utilization-oriented) VM schedulers are intuitively inappropriate for simulation needs. Our experiments corroborate this mismatch empirically, showing the need for simulations-specific schedulers. To address the gap, we developed a new simulation-focused scheduler called NetWarp scheduler in our NetWarp simulation system for multi-core VMs simulated using multi-core hosts. We realized a working implementation within the Xen hypervisor framework. Preliminary experimental results from our experiments have shown here that simulation-oriented schedulers such as ours can indeed dramatically reduce time-ordering inaccuracies while still incurring minimal runtime overhead compared to non-simulation schedulers optimized for processor utilization. Additional research is needed to evaluate on larger platforms. Host nodes with dozens of cores are now available. We are presently exploring this approach on a host with 24 cores (four hex-core sockets), and hope to have performance data on this platform in the near future.

## 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] D. Gupta, et al. "To Infinity and Beyond: Time-Warped Network Emulation," in Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06), pp. 87-100, San Jose, CA, USA, 2006.

[2] Jenna N. Mathews et al., "Running Xen, A Hands-On Guide to the Art of Virtualization," ISBN 978-0-132-34966-6, Prentice Hall, 2008.

[3] http://www.xen.org/products/xenhyp.html.

[4] QEMU: http://wiki.qemu.org/Main_Page

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

[6] "Xen Wiki: Credit-Based CPU Scheduler," http://wiki.xensource.com/xenwiki/CreditScheduler.

[7] http://book.xen.prgmr.com/mediawiki/index.php/Scheduling

[8] Jason Liu, Yue Li and Ying He, "A Large Scale Real-time Network Simulation Study Using PRIME," in Proceedings of the Winter Simulation Conference (WSC), 2009.

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

[10] Y. Li, et al., "Towards Scalable Routing Experiments with Real-time Network Simulation," in 22nd International Workshop on Principles of Advanced and Distributed Simulation (PADS) , pp. 23-30, Rome, Italy, 2008.

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

[12] Elias Weingrtner, et al., "Synchronized Network Emulation: Matching Prototypes with Complex Simulations," SIGMETRICS Performormance Evaluation Review, vol. 36, pp. 58-63, 2008.

[13] S. Maier, et al., "Scalable Network Emulation: A Comparison of Virtual Routing and Virtual Machines," in IEEE Symposium on Computers and Communications, pp. 395-402, Aveiro, Portugal, 2007.

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

[15] G. Apostolopoulos and C. Hasapis, "V-eM: A Cluster of Virtual Machines for Robust, Detailed and High-performance Network Emulation," in Proceedings of the 14th IEEE International Symposium on Modeling, Analysis and Simulation of Computing and Telecommunication Systems, pp. 117-126, Monterey, CA, USA, 2006.

[16] C. Kiddle, et al., "Improving Scalability of Network Emulation through Parallelism and Abstraction," in Proceedings of the 38th Annual Simulation Symposium, pp. 119-129, 2005.