Optimized Hypervisor Scheduler for Parallel Discrete Event Simulations on Virtual Machine Platforms

Srikanth B. Yoginath and Kalyan S. Perumalla

Computational Sciences and Engineering Division,
Oak Ridge National Laboratory, USA
yoginathsb@ornl.gov, perumallaks@ornl.gov

ABSTRACT

With the advent of virtual machine (VM)-based platforms for parallel computing, it is now possible to execute parallel discrete event simulations (PDES) over multiple virtual machines, in contrast to executing in native mode directly over hardware as is traditionally done over the past decades. While mature VM-based parallel systems now offer new, compelling benefits such as serviceability, dynamic reconfigurability and overall cost effectiveness, the runtime performance of parallel applications can be significantly affected. In particular, most VM-based platforms are optimized for general workloads, but PDES execution exhibits unique dynamics significantly different from other workloads. Here we first present results from experiments that highlight the gross deterioration of the runtime performance of VM-based PDES simulations when executed using traditional VM schedulers, quantitatively showing the bad scaling properties of the scheduler as the number of VMs is increased. The mismatch is fundamental in nature in the sense that any fairness-based VM scheduler implementation would exhibit this mismatch with PDES runs. We also present a new scheduler optimized specifically for PDES applications, and describe its design and implementation. Experimental results obtained from running PDES benchmarks (PHOLD and vehicular traffic simulations) over VMs show over an order of magnitude improvement in the run time of the PDES-optimized scheduler relative to the regular VM scheduler, with over 20× reduction in run time of simulations using up to 64 VMs. The observations and results are timely in the context of emerging systems such as cloud platforms and VM-based high performance computing installations, highlighting to the community the need for PDES-specific support, and the feasibility of significantly reducing the runtime overhead for scalable PDES on VM platforms.

Categories and Subject Descriptors

I.6.8 [**Simulation and Modeling**]: Types of Simulation – *discrete event, distributed, parallel*

General Terms

Algorithms, Measurement, Performance, Design, Experimentation

Keywords

Resource scheduling, hypervisor schedulers, virtual machines, cloud computing, discrete-event simulations, parallel and distributed simulations

SIMUTools 2013 March 6–7, Cannes, France.

1. INTRODUCTION

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

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

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

In PDES the simulation time advances with the processing of time-stamped simulation events. In general, the number of events processed in a PDES application varies dynamically during the simulation execution (i.e., across simulation time), and also varies across processors. This implies that the amount of computation cycles consumed by a processor for event computation does not have any specific, direct correlation with its simulation time. A processor that has few events to process within a simulation time window ends up consuming few computational cycles. It is not

ready to process events belonging to the simulation-time future until other processors have executed their events and advanced their local simulation time. However, a fair-share scheduler would bias the scheduling towards this lightly loaded processor (since it has consumed fewer cycles) and penalize the processors that do in fact need more cycles to process their remaining events within that time window. This type of operation works against the actual time-based dependencies across processors, and can dramatically deteriorate the overall performance of the PDES application. This type of deterioration occurs when conservative synchronization is used. Similar arguments hold for optimistic synchronization, but, in this case, the deterioration can also arise in the form of an increase in the number of rollbacks. The only way to solve this problem is to design a new scheduler that is aware of, and accounts for, the simulation time of each VM, and schedule them in a *least-simulation-time-first* order.

1.1 Related Work

Poor performance of certain high-performance computing applications has also been observed very recently [1] and customized solutions are being proposed, which are not applicable to PDES. The issue of executing PDES applications in cloud environments has been studied recently [2][3], addressing the performance issue at the application layers (e.g., cloud architecture). The Master-Worker approach to distributed (and fault tolerant) PDES [13] is also a related but complementary approach, different from our support for the traditional PDES execution view in which all processors are equal. We adopt a different approach by focusing at the lowest level, i.e., at the level of the hypervisor itself. Incidentally, the Time-Warp Operating System [12] of the 1980's is one of the earliest works that addressed PDES performance issues by realizing the simulation scheduler (and related functionality) at the bottom-most hardware levels; however, this was limited to a single operating system, as opposed to a hypervisor system.

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

While the concepts developed in this paper for VM-based PDES are sufficiently general, our implementation and experimentation are performed with the Xen® [4] hypervisor (a freely available, popular hypervisor), and the μsik [5] parallel/distributed simulation kernel (a high performance PDES simulator).

1.2 Xen hypervisor

The Xen [4] hypervisor is a popular open source industry standard for virtualization, supporting 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. In Xen terminology, each VM is referred to as a Guest Domain or simply as a DOM[1]. 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.

Each DOM has its own set of virtual devices, including virtual multi-processors called virtual CPUs (VCPUs). The hypervisor scheduling mainly deals with efficient mapping (multiplexing) of all the VCPUs of multiple VMs onto the available physical processor cores (PCPUs). The credit-based scheduler (**CSX**) is the default Xen scheduler, which schedules VCPUs on to PCPUs based on the principle of fair-share. CSX uses credits for every DOM, these credits are expended as the DOM's VCPUs are scheduled for execution. It provides a limited amount of control to the user to customize the scheduler configuration through parameters called weight and cap. The default weight value for all DOMs is 256 and the cap is 0, ensuring fair CPU allocation to all of the DOMs. This scheduler is very widely used, and works excellently for a very large variety of virtualization uses.

1.3 Parallel Discrete Event Simulation

In PDES, the model is divided into distinct independent virtual timelines referred to as Logical Process (LP). Each LP typically encapsulates a set of state variables of a modeled entity. The timelines of LPs within and across processors are kept synchronized by the simulation engine. Two distinct synchronization mechanisms namely, conservative and optimistic synchronizations, are used in PDES. In conservative synchronization, all event processing is always strictly performed in virtual time order at every LP. On the contrary, in optimistic synchronization, any LP is allowed to temporarily violate simulation-time order of its events but uses a rollback mechanism (event cancellations and check-pointing/reverse computation) to correct the errors committed, and achieve eventual correctness. More details on PDES and its synchronization mechanisms can be found in [6]. The μsik parallel/distributed simulation kernel [5] built upon micro-kernel architecture is used for our experimentations. The runtime architecture of μsik supports execution of models in which one or more LPs can be mapped to each VM, and it supports both conservative and optimistic synchronization mechanisms.

In general, one or more LPs can be mapped on to each μsik *federate* (simulation process), each such federate maps to a VCPU, each VM contains one or more VCPUs, and multiple such VMs are hosted by a hypervisor.

For the purposes of this research, the μsik engine is made to interoperate with the Xen hypervisor scheduler. We will refer to the PDES-specific scheduler for Xen in this implementation as **PSX**.

1.4 Organization

The rest of the article is organized as follows. The important design considerations in the development of PDES-specific VM scheduler are described in Sections 2, followed by implementation details in Section 3. In Section 4, the PDES benchmarks and scenarios used for performance evaluation are described. A performance study is presented in Section 5 along with the results

[1] Hence, we will use the terms VM and DOM interchangeably in the rest of the article.

from detailed experiments. The findings are summarized and concluded in Section 6.

2. DESIGN

In PDES, since LPs (and consequently, VMs) can have widely differing event loads, they exhibit different ratios of simulation time to wall clock time. Event load imbalance can arise across VMs, which is not only inherent but also hard to predict due to its dynamic nature. Fundamentally, this dynamic, scenario-specific variation of the ratio of simulation time to wall clock time is the critical factor that must be accounted for in the design of the PDES-specific VM scheduler.

However, in PDES we do know that the LP with the lowest value of local virtual time (LVT) affects the progress of its peers and hence the entire simulation application. Hence, if the LVT of the LP were used as the criterion in allocating processor time to VMs by the hypervisor (i.e., LPs with lower LVT values are prioritized over those with higher LVT values), then the runtime performance can be optimized. This can be achieved if the LPs running on different VMs are able to communicate their LVT values to the hypervisor, and the hypervisor in turn uses this information during the scheduling of VCPUs on to PCPUs. An additional aspect in relation to global virtual time computation also becomes an important design consideration. These design considerations are described next.

2.1 Issues and Challenges

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

To communicate the LVT of an LP from the application layer to the hypervisor, the LVT must first pass from the user-space of the PDES process to the kernel-space of the guest-OS and then to the hypervisor data regions. One way to accomplish this is by adding a system call to the guest-OS to enable the transit of user-space data to kernel-space. However, to make this data accessible to the Xen hypervisor the guest kernel uses a shared memory page named *shared_info*, which is used by the Xen hypervisor through out its runtime to retrieve information about the global state [4]. The *shared_info* contains information that is dynamically updated as the system runs. In fact the Xen hypervisor uses the *shared_info* for time-keeping functionality of its para-virtual guest-OS. The LVT value from the guest-OS kernel-space is written into the *shared_info*, thus making it available to the hypervisor.

Next, we need to implement the hypervisor scheduler that employs a *least-simulation-time-first* policy instead of the default credit-based fair scheduling strategy. Implementing the Xen hypervisor scheduler for the application-specific requirements has been previously accomplished [7][8]. Each PCPU maintains a *runq* (priority-queue) in which the VCPUs requiring clock-cycles are en-queued. The scheduler inserts the VCPUs into the PCPU *runqs* based on the LVT value. Hence, every VCPU of a DOM that hosts the LP is required to maintain a variable (VCPU-LVT) representing LVT value of the LP. Based on the VCPU-LVT value, the VCPUs are inserted in the *runq* of the PCPU. With a *least-simulation-time-first* policy, the VCPU that the scheduler picks for allotting PCPU cycles will have the lowest LVT among all its peers.

In addition to these two major requirements, it is also necessary to ensure that the LPs receive sufficient number of computational cycles to participate in Global-Virtual-Time (GVT) computation regardless of its LVT priority in relation to other LPs. In the absence of this special treatment required for GVT computation, the simulation would deadlock. This is because the VCPUs with lower VCPU-LVT values (i.e., having a higher scheduling priority) would not allow the VCPUs with a higher VCPU-LVT to be chosen for scheduling. This results in blocking the GVT computation at the application level, as some of the LPs (with a higher LVT value) would never get a chance to respond during GVT computation. This problem can be solved by ensuring that some PCPU cycles are periodically and *unconditionally* provided to the VCPUs regardless of their VCPU-LVT values.

2.2 System Architecture

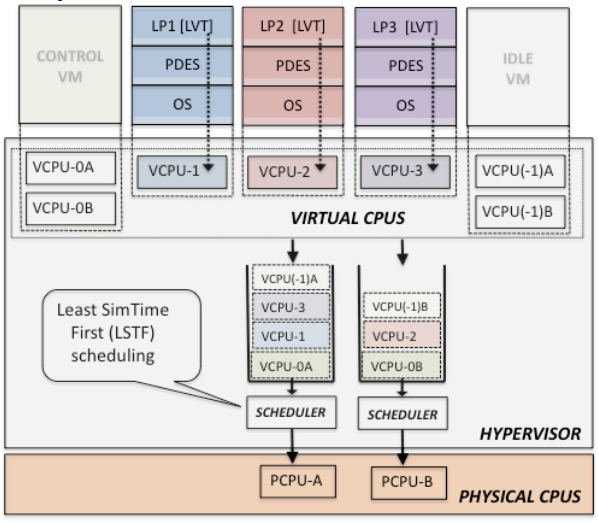

Figure 1 The design of the PDES-customized scheduler

Figure 1 shows the system architecture of a hypervisor-based parallel computing environment with a scheduler optimized for PDES execution. For simplicity of explanation, let us assume that a single LP is hosted on each PDES federate and each VM has a single VCPU (note that this is *not* a requirement or a limitation of our system, but it simplifies understanding). As illustrated in Figure 1, the LVT of an LP is passed to the VCPU of its DOM. The scheduler that performs the task of multiplexing VCPUs onto PCPUs uses the VCPU-LVT and employs *least-simulation-time-first* scheduling. With a *least-simulation-time-first* order, the scheduler gives the highest priority to the VCPU with least VCPU-LVT value, as opposed to the default fair distribution of compute cycles across all the DOMs. The special VMs (DOM0 and Idle-DOM) in Xen do not participate in the PDES simulation. The DOM0 is the privileged DOM, and the Idle-DOM is a Xen mechanism to ensure that the PCPU run-queues are never empty.

3. IMPLEMENTATION

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

3.1 Communicating LVT to Xen Scheduler

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

Linux Kernel Modifications

To send the LVT information from the application level which is a *μsik* federate, we defined and implemented a new system call for the Linux® OS to be able to invoke it from the *μsik* library from within the simulation loop. This system call allows the LVT information to transit from user-space to kernel-space; once reaching the kernel-space, the LVT value is written into the *shared-info* data structure of the host DOM, which in turn can be accessed by the hypervisor at runtime.

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

Figure 2 Modified *shared_info* data-structure with emboldened newly added variable names

However, the para-virtual guest-OS kernel has to be re-built after the addition of a new system call and incorporating the changes to the *shared_info* data-structure (Figure 2) in correspondence to its modification in the hypervisor. Two fields namely, *simtime* and *switch_scheduler* are added to the *shared_info* data-structure. Each guest-OS maintains a *shared_info* page, which is mapped on to memory by the hosting DOM, during its creation. While *simtime* is used for holding the LVT of the federate mapped on to this DOM, the *switch_scheduler* is a flag that indicates the switch between two different modes of the scheduler operation namely, *normal-mode* and *simulation-mode* (described later in greater detail).

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

μsik Library Modifications

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

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

Every μsik federate maintains a variety of simulation times based on its event processing state at any given moment. They are distinctly classified into four classes, namely, committed, committable, processable and emittable [5]. We can transmit any of these LVT values to the hypervisor. In practice, we observed that the use of the "earliest-committable-time-stamp" resulted in better performance than the others, and hence, this is the simulation time value used in all our experiments.

3.2 Xen Scheduler Implementation

The PDES Scheduler for Xen (PSX) scheduler replaces the default Credit Scheduler of Xen (CSX) in scheduling the virtual CPU (VCPUs) onto the physical cores of CPU (PCPU). The strategy that we take to replace the scheduler is similar to the one presented by [8].

PSX data-structures

The *switch_sched* (corresponding to *switch_scheduler* in *shared_info*) is a field of global *ps_priv* global variable, which is an instance of *ps_private* data-structure (shown in Figure 3) and, by default the value of *switch_sched* is *false* (*normal-mode*). The scheduler regularly checks the *shared_info* associated with the guest-DOM of the VCPU it services. Hence, when the *switch_scheduler* value the *shared_info* of any guest-DOM is updated, the scheduler reads it from the *shared_info* and, writes it to *switch_sched* field of *ps_priv* variable. The scheduler uses spin-locks in this process to avoid any un-desirable race conditions during its SMP execution. Each VCPU reads the LVT value from the *shared_info* into its *sim_time* variable. Figure 3, shows the *sim_time* and *switch_sched* variables in the PSX's VCPU and *ps_private* data-structures, respectively.

```
struct ps_vcpu
{
   struct list_head runq_elem;
   struct list_head active_vcpu_elem;
   struct nw_dom *sdom;
   struct vcpu *vcpu;
   uint16_t flags;
   uint64_t prior_sim_time;
   atomic_t vcpu_ticks;
   s_time_t vcpu_ref_time;
   s_time_t sim_time;
};
```

```
struct ps_private
{
   spinlock_t lock;
   struct list_head active_sdom;
   uint32_t ncpus;
   uint32_t nvcpus;
   cpumask_t idlers;
   int switch_sched;
};
```

Figure 3 VCPU data-structure and ps_private global data-structure in PSX, respectively

The *vcpu_ticks* and *vcpu_ref_time* in the VCPU data-structure are used to record PCPU time allotted for each VCPUs. The usage of

prior_sim_time is to instrument PDES specific requirement, which will be discussed shortly.

Scheduling in Normal-mode

The scheduler is referred to be in *normal-mode* if the *switch_sched* (*ps_private* data-structure Figure 3) is false. This corresponds to the mode in which the VMs are booted and operational, but no PDES run has been started (and hence LVT-based scheduling is undefined). PSX by default maintains the *sim_time* (VCPU data-structure Figure 3) of all DOM0 VCPUs lower than all the DOMUs. In the *normal-mode* all the guest-DOM VCPUs will have their *sim_time* initialized to 1, while DOM0 VCPUs have their *sim_time*s initialized to 0. Only after the *switch_sched* is set true by PDES federate the *sim_time* value of the relevant VCPU is updated after reading the *shared_info*. However, the *sim_time* of VCPUs of DOM0 continues to be 0 even after switching to *simulation-mode*.

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

Scheduling in Simulation-mode

The hypervisor switches to the simulation mode after the PDES execution is started on all the VMs. Each PCPU maintains a *runq* (priority-queue) as shown in Figure 4 and, in the *simulation-mode* PSX en-queues the VCPUs to be scheduled in a prescribed priority.

```
struct ps_pcpu
{
   struct list_head runq;
   uint32_t runq_len;
   struct timer ticker;
   unsigned int tick;
};
```

Figure 4 PSX physical CPU-core specific data-structure maintained by PSX

We use the LVT as the VCPU priority – the lower the *sim_time* (VCPU data-structure Figure 3), the higher is its priority in the *runq* and, hence the earlier it is picked by PSX to allocate compute resource. Every PCPU schedules itself for every *tick* using the timer named *ticker*. The PCPU performs accounting for the VCPU currently being serviced by incrementing the *vcpu_ticks* and, updating the *sim_time* by reading the *shared_info*. The PCPU also generates a *schedule* interrupt for the VCPU being serviced on a less loaded PCPU. During scheduling the SMP scheduler enqueues the VCPU being serviced and picks the VCPU with least *sim_time* across all PCPU *runq*s to service. Our implementation of the scheduler allots a *tick* size (300 μs) of PCPU time for the VCPU picked to service.

Scheduling consideration for GVT progress

In addition to event processing, the LPs also need to participate in periodic GVT computation. This periodic computation is necessary to consolidate the independent LVTs of each LP into a global GVT. With *least-simulation-time-first* scheduling the federate with higher LVTs never get past the federate with lower LVTs and hence do not get any PCPU time to participate in GVT computation. Without successful GVT computation, LPs cannot determine which events can be processed next. Without a special consideration for GVT computations, a strict *least-simulation-time-first* based PSX does not allow completion of GVT computation, hence the PDES execution deadlocks.

To overcome this, it is necessary to periodically provide a few PCPU cycles for VCPUs with higher *sim_time*. We accomplish this by using the *prior_sim_time* variable of VCPU data-structure shown in Figure 3. The *prior_sim_time* is updated whenever *sim_time* is read from the *shared_info*. When the *sim_time* of a VCPU remains unchanged to its *prior_sim_time* value for a consecutive read of *shared_info*, then the *sim_time* is temporarily increased to a large value greater than simulation *end_time* say, *max_time* and, *prior_sim_time* is *not* updated. Hence, during the next consecutive *shared_info* read, the *prior_sim_time* value will be different from the temporary *max_time* value; the *sim_time* and *prior_sim_time* are updated regardless of whether the simulation time actually advanced or not. Such an operation based on temporarily flipping the *sim_time* to *max_time* ensures unhindered GVT computations.

4. Experimental Setup

To exercise the implementation and perform a quantitative study of runtime performance, we use a range of application scenarios, as described next.

4.1 Benchmark Applications

Two applications namely, PHOLD [10] (a synthetic PDES application generally used for performance evaluation) and SCATTER-OPT [11] (a reverse-computation-based vehicular traffic PDES application) are used in our performance studies.

PHOLD

This is a widely used synthetic benchmark for performance evaluation in the PDES community. This PDES application randomly exchanges a specified set of messages between the LPs. The μsik implementation of PHOLD allows exercising a wide variety of options in its execution. In all of our PHOLD benchmarks we use a lookahead of 1 and utilize combinations of the listed variants for performance evaluation.

a. Synchronization: optimistic (OPT) or conservative (CONS)

b. Number of LPs per Federate (NLP) [for example: 10 NLP = 10 LPs/federate]

c. Number of messages per LP (NMSG) [for example: 10 NMSG = 10 messages/LP]

d. Destination locality of the LP generated message (LOC) specifies the percentages of local and remote events. Values of 50, 90 and 100 suggest respectively that 50%, 90% and 100% of the messages generated by an LP are local to its federate. Hence, a value of 50% for LOC involves more LP message exchanges across the network and results in increased network traffic; a value of 90% results in a reasonable amount of inter-federate event traffic, and, 100% suggests an embarrassingly parallel PDES application involving little inter-federate interaction except for GVT computations.

SCATTER-OPT

This application is a discrete-event formulation and a parallel execution framework for vehicular traffic simulation. It uses the μsik library for parallel execution and, is amenable to both conservative (CONS) and optimistic (OPT) synchronizations. A simulation scenario is set up by reading an input file that specifies

the road-network structure, number of lanes, speed limit, source nodes, sink nodes, vehicle generation rate, traffic light timings and other relevant information. Dijkstra's shortest-path algorithm is used to direct a vehicle to its destination. This benchmark serves to exercise the hypervisor-based PDES performance using CSX and PSX schedulers with a more complex PDES application behavior and dynamics.

In all our experiments a single-federate is hosted on each DOM and such federate houses multiple LPs.

4.2 Benchmark Application Scenarios

Using the PDES applications listed in the previous section four benchmark applications namely, *Balanced Load Benchmark* (BLB), *Unbalanced Load Benchmark* (ULB), *High Load Benchmark* (HLB) and Vehicular-traffic Simulation Benchmark (VSB), were designed. In all the benchmarks using PHOLD a *lookahead* of 1 was used.

Balanced Load Benchmark

All BLB experiments have NLP as 1, indicating 1 LP/Federate and the NMSG is varied (10 and 100 messages/LP). With an experiment involving 24 DOMs (NUM_DOMS = 24) the number of messages PHOLD uses is ($NUM\_DOMS \times NLP \times NMSG$), i.e., 240 and 2400 messages for 10 and 100 messages/LP, respectively. These set of experiments are carried out for both OPT and CONS, for LOC values of 50, 90 and, 100 percent.

Unbalanced Load Benchmark

All ULB experiments have NMSG as 1.5, while NLP is varied (10 LPs/Federate and 100 LPs/Federate). To elaborate on 1.5 NMSG, let us consider experiments with 24 federates, the total number of messages PHOLD uses will be 360 (24×10×1.5) and 3600 (24×100×1.5) messages for 10 LPs/federate and 100 LPs/federate, respectively. The distribution of total messages at the start of the simulation is performed in *round-robin* pattern among all the LPs. Hence, the first 12 DOMs hosting 1200 LPs in the 100 LPs/federate scenario, are initialized to handle 2400 messages (2 messages/LP), with each federate or DOM handling 200 messages, while the remaining 1200 LPs are initialized with remnant 1200 messages (1 message/LP) with each federate or DOM handling 100 messages. Thus creating a load imbalance among DOMs, with the first half of DOMs essentially starting the simulation with twice the number of messages than the second half of DOMs.

High Load Benchmark

HLB is used study the impact of increase in federates (over-subscription) from 16 to 64 in a highly loaded PHOLD experiment scenario. We run 100 LPs/federate, 100 messages/LP with 24, 32 and 64 federates with 90% LOC. Hence, as the number of federates increase so does the problem size. For example: in the 16 federate scenario 1600 LPs hosted on 16 DOMs simulate exchanges of 160000 messages over 100 seconds of simulation time, while the 64 federate scenario involves simulation of 640000 message exchanges among 6400 LPs hosted on 64 DOMs.

Vehicular-traffic Simulation Benchmark

VSB simulates the evacuation of 163,840 vehicles emanating from 128 sources (right and left) and moving towards 128 (top and bottom) sinks through a road-network grid with 64×64 (4,096 intersections), as shown in Figure 5. As shown in Figure 5, the sources and sinks are always equally divided among all federates. Each source randomly (uniform random distribution) generates 10 vehicles/hour/destination for an hour. The intersections are connected to one another by a pair of oppositely directed and a kilometer long single-lane roads. The traffic lights are provided with 8-second cycle (red-green-red) time, the time required by a vehicle to cross an intersection (which is used as *lookahead*) 1 second.

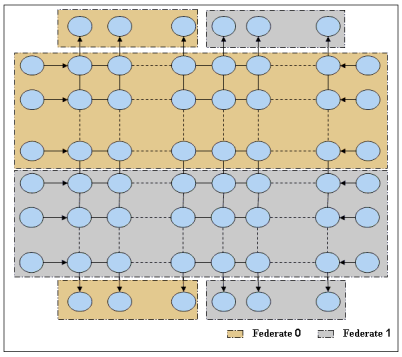

Figure 5 VSB road-network

4.3 Hardware and Software

The experiments were performed on a high end Mac-Pro server with two hex-core Intel® Xeon processors at 2.66 GHz, 6.4 GT/s processor interconnect speed with 32G of memory. With hyper-threading enabled, Xen sees 24 cores. With Xen creating 24 PCPUs to handle this, all our experiments view this system as a 24-core machine. OpenSUSE 11.1 with Xen-3.3.1 and Xen-3.4.2 source code was used on this hardware. Para-virtual DOMs running Linux® guest-OSs connected using a software bridge and sharing a file-based disk image (mounted as a loop-device in DOM0) using *Network File System* (NFS) [9] were used for experimentation. The Linux® guest-OSs were built using Linux-2.6.18 distribution modified to support a *system-call* to update the *shared_info* data-structure. Two versions of the μsik engine, one unmodified and the other having the capability to pass LVT values to the hypervisor, were installed on all the guest-OSs. All the CSX experiments used the un-modified version, while PSX used the modified version of μsik.

5. Performance Results

5.1 CSX Instrumentation

In the Xen hypervisor, all the inter-DOM network traffic passes through the DOM0. Hence, as the number of inter-federate events increases in a PDES application, and, if sufficiently high priority or weight is not given to the DOM0, the performance degrades. To identify this performance effect and to optimize the execution by taking this factor into account in our performance comparison with PSX, we performed experiments to determine the best configuration for the CSX native scheduler. We present the results from BLB and ULB using CSX with varying weights for DOM0.

By default, all DOMs including DOM0 are assigned a weight of 256. However, as mentioned earlier, CSX provides user the ability to alter the weight of any DOM. In our performance runs we increase the DOM0 weights in multiples of 256, while all others are maintained at 256. Hence 2x and 4x weights suggest twice (512) and four times (1024) the default weight of 256, respectively, giving proportionally more weight to DOM0 than the other DOMs.

The plots in Figure 6 and Figure 7 show the runtime results for BLB and ULB for varying weights of DOM-0, respectively, with 24 federates hosted on 24 DOMs. CSX BLB plot correspond to 1LP/federate and 100 messages/LP. While, the CSX ULB plot corresponds to 100 LPs/federate (2400 LPs) and 1.5 messages/LP

(3600 messages). From both plots two important points can be derived (a) special treatment for DOM-0 in terms of weight is absolutely needed for performance (b) increase in weight beyond certain factor neither yields better performance nor deteriorates it. Flat performance with increasing weights of DOM-0 is expected because CSX divides the unused credits of DOM-0 equally among other DOMs. Yet another interesting point to note is CSX with OPT performs better than CSX_CONS for the same load conditions, this is more clear in ULB.

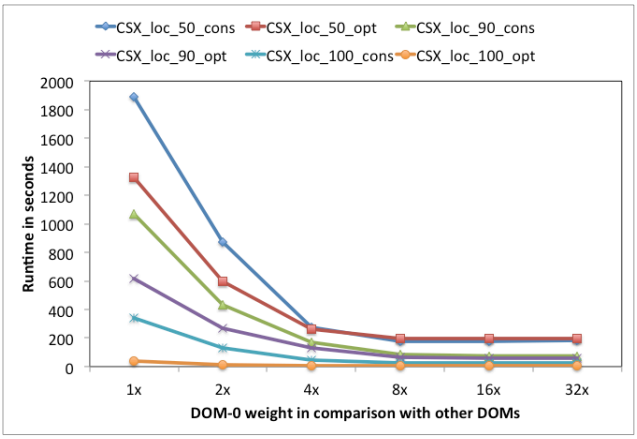

Figure 6 CSX with balanced workload

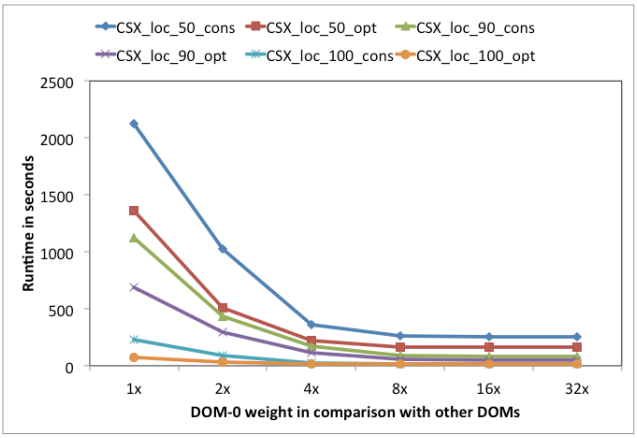

Figure 7 CSX with unbalanced workload

Based on the results, it is clear that performance gains are not so pronounced after reach 8x. However, to be conservative, we use a DOM0 weight of 32x (8192, which provides the maximum gain) for CSX in all the following comparative performance study with PSX. Thus, all PSX (PDES-optimized scheduler) runs are in fact judged against this already optimized CSX (native, fairness-based VM scheduler).

5.2 Speedup under Balanced and Unbalanced Workloads

In this section we compare the runtime performance of PSX and CSX using BLB and ULB. Two experimental setups are considered (a) Virtual compute resources closely match physical resources (b) Virtual compute resources are slightly over-subscribed.

Virtual and Physical compute resources match

We use 24 DOMUs in this case; hence this setup more closely matches the virtual compute resources with physical compute resources. Even though DOM0 infringes into the resource pool, it is not involved in the simulation. Note the CSX runs are not burdened to write into and read from *shared_info*. However, PSX requires that LVT be written into the *shared_info*, which is read during scheduling. Note that all BLB benchmarks have NLP=1, while NMSG and LOC values vary. Similarly, all ULB benchmarks have NMSG=1.5, while NLP and LOC values vary.

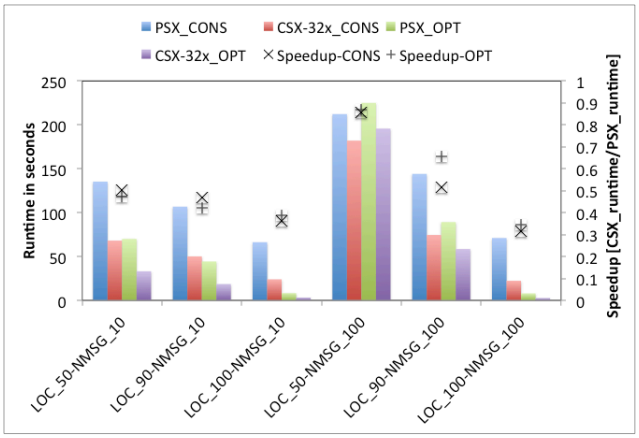

Figure 8 Speed-up of PSX over CSX-32x in 24-DOM with balanced workload

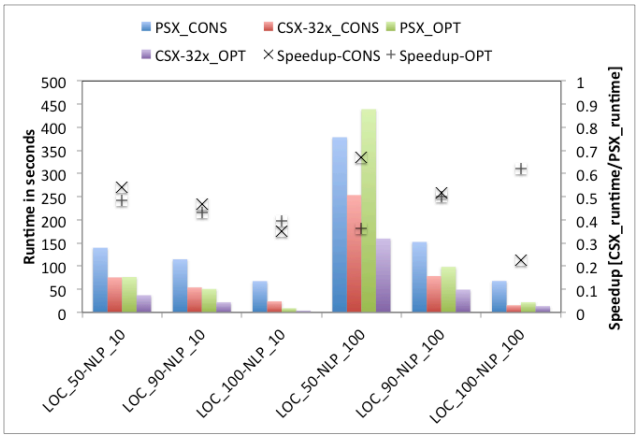

Figure 9 Speed-up of PSX over CSX-32x in 24-DOM with unbalanced workload

Figure 8 and Figure 9 plot the observed speed-up of PSX over CSX for BLB and ULB, respectively. It is observed from both BLB and ULB plots that when the physical resources match virtual resources CSX performs better than PSX. This performance loss in PSX can be attributed to the overhead involving the *shared_info* writes and reads. One can see that as the simulation load (in-terms of number of messages exchanged) increases PSX's performance gets better and catches up with that of CSX. The only exception to this observation can be seen in ULB plot in Figure 9, where the runtime and hence the corresponding speedup of the OPT run in *LOC_50-NLP-100-NMSG_1.5* scenario suffers, this can be attributed to the reversals in the OPT run.

Virtual compute resources are slightly over-subscribed

When the physical resources are over-subscribed the effects of PSX becomes noticeable. Figure 10 shows the speed-up plot for BLB. It is noticed from the plot that apart from the scenarios with LOC is 100 (inter-federate message passing is 0) where the gains

are small all the others show a significant speed-up (about 6× increase from CSX). The best speed-up is observed in the plots of ULB runs shown in Figure 11, where the speed-up gains are over an order (close to 20×). The ULB scenarios are slightly more loaded and imbalanced in their load distribution among LPs compared to BLB. Hence, PSX greatly excels in these sets of experiments. The speed-up achieved by PSX using CONS is always greater than its OPT counterpart. This is partly because CSX tends to perform better with OPT. However, when only runtime is taken into consideration the PSX with OPT is the best in most of the cases.

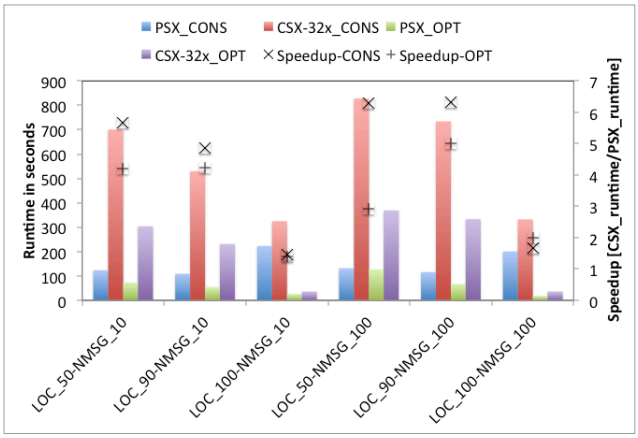

Figure 10 Speed-up of PSX over CSX-32x in 32-DOM with balanced workload

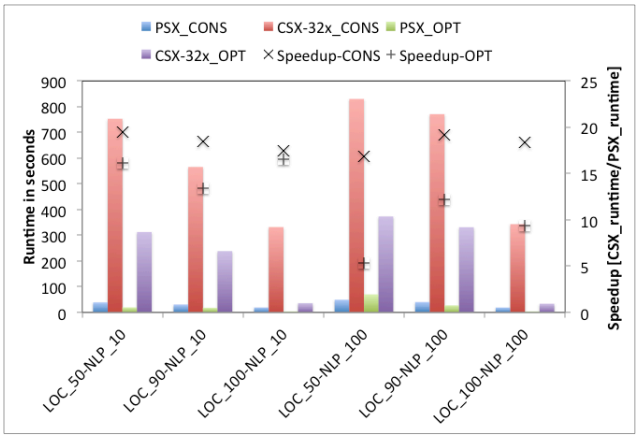

Figure 11 Speed-up of PSX over CSX-32x in 32-DOM with unbalanced workload

5.3 Scaling to many DOMs

This scenario of experiments exercise even higher simulation loads compared to BLB and ULB. With these benchmark runs, we observe the performance implications of running large PDES simulation scenarios on under-subscribed and over-subscribed physical resource. The plots in Figure 12 show an increase in the speed-up gains with increased over-subscription, thus suggesting efficient utilization of physical resources.

The worst speedup in PSX (with both CONS and OPT) is observed when the resources are under-subscribed. When the virtual compute resources match the physical resources the speed-up is slightly greater than 1. The speed-up increases with the increase in the over-subscription of the physical compute resources.

We observe that CSX with OPT is always better than CSX with CONS and this difference is more apparent when number of DOMs is 64. On the contrary the runtime of the PSX varies seldom with CONS and OPT. However, PSX with CONS does slightly well in benchmarks with 48 DOMs. Similar to previous observations the speedup gained in CONS is better than that gained in OPT. This weak-scaling benchmark with its performance numbers clearly demonstrates the significance of efficiently allocating of compute resources.

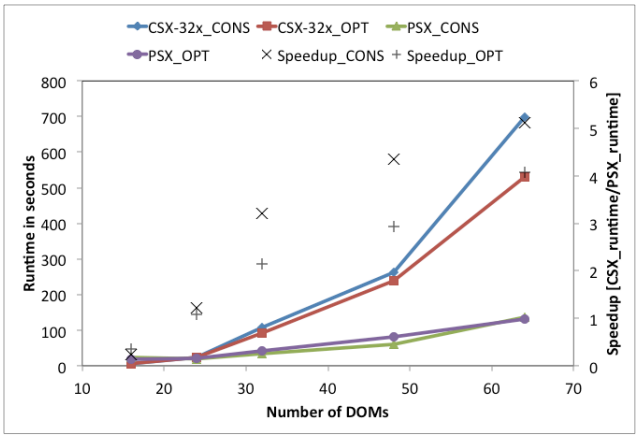

Figure 12 CSX-32x and PSX runtime plots (weak scaling)

5.4 Runtime variation of event load

Note that in the PDES applications, compute cycles consumed directly corresponds to the number of processed simulation events. In such scenarios the capability of a parallel simulation execution environment to grow or shrink its physical compute resources as necessitated by the varying workload is extremely beneficial in efficient utilization of available resources. Figure 13 plots the results from VSB experimental setup comprising 64 DOMs where, number of events processed by all federates are plotted against the runtime. This figure also pictorially shows the potential points (based on some number-of-processed-events threshold) at which simulation execution environment could grow and shrink.

A means to achieve the capability to *grow* and *shrink* in physical resource utilization is via *oversubscribing* the compute resources at the beginning of the simulation and, *growing* using *process-migration* or *shrinking* by *oversubscribing* again. Hence, it becomes necessary to optimize the performance of the hypervisor when it is oversubscribed.

The VSB plots in Figure 14 show a speedup of 1.5 for both CONS and OPT runs in 32 DOMs experimental setup and, a speedup over 4.5 and 8 for CONS and OPT, respectively, in experimental setup with 64 DOMs. More interesting than the speedup is that the runtime of simulations using PSX remains nearly constant even as the oversubscription intensity is doubled. This aspect of scalability with the number of VMs makes PSX better suited for parallel execution environments requiring the capability to grow and shrink in physical resources over runtime.

6. SUMMARY AND CONCLUSION

We identified the performance degradation of native scheduling as a potential drawback of existing virtualization-based parallel computing installations. When this performance problem is solved, the numerous other benefits such as serviceability and cost

effectiveness offered by the new platforms can be reaped for PDES applications. We addressed the performance problem by first tracing it to the mismatch of virtual time-based dependencies of PDES across VMs and the time sharing-based operation of VM platforms for typical applications. Based on this insight, a new PDES-optimized scheduler was designed, developed and implemented. Several benchmarks, with different event workload patterns, load imbalance levels and application behaviors were used in a detailed performance study. Results from all benchmarks demonstrate excellent speedup, scalability and suitability (for growing and shrinking in compute resources) of the PDES-optimized hypervisor.

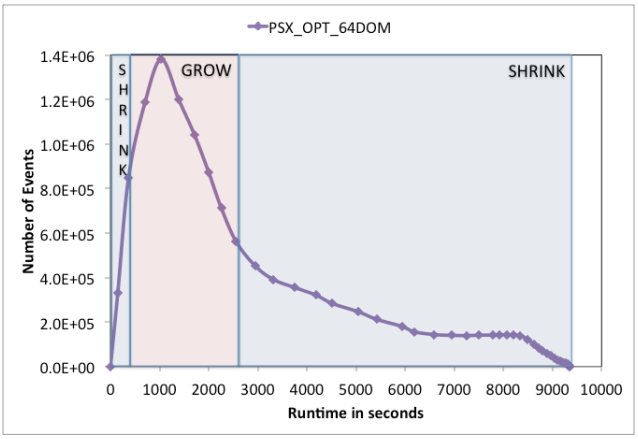

Figure 13 Number of processed events with respect to runtime in VSB using PSX with 64 DOMs

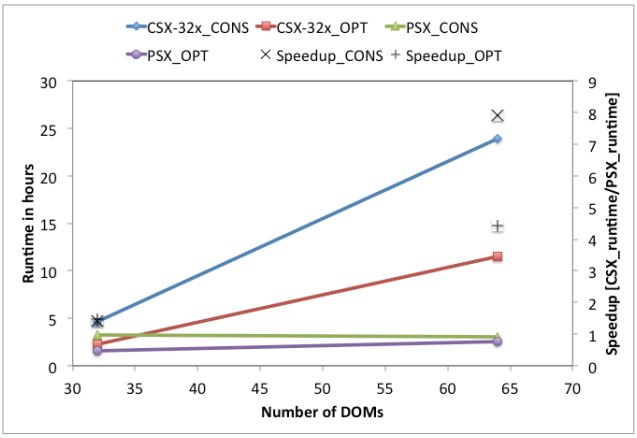

Figure 14 VSB performance plots for experimental setups comprising 32 DOMs and 64 DOMs (strong scaling)

Overall, it is clear from the results that virtualization-based parallel computing platforms should be required to provide facilities for application-specific scheduling on compute node instances in order to avoid the performance degradation in specialized applications such as PDES.

Future work of interest includes incorporating and benchmarking the support for dynamic growth and shrinkage of physical processors allocated to a PDES run dynamically during its execution.

ACKNOWLEDGMENTS

This research was performed as part of a project sponsored by the U.S. Army Research Laboratory. 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 US Government purposes.

REFERENCES

[1] Jackson, K.R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H.J., Wright, N.J., "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud," *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom),* 2010

[2] Malik, A., Park, A., Fujimoto, R., "Optimistic Synchronization of Parallel Simulations in Cloud Computing Environments," *IEEE International Conference on Cloud Computing,* 2009

[3] Fujimoto, R.M., Malik A. W., Park, A. J., "Parallel and Distributed Simulation in the Cloud," *SCS Modeling and Simulation Magazine, Society for Modeling and Simulation International*, Vol. 1, No. 3, 2010

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

[5] Perumalla, K. S. "µsik - A Micro-Kernel for Parallel/Distributed Simulation Systems," *IEEE Workshop on Principles of Advanced and Distributed Simulation*, 2005

[6] Fujimoto, R. M., "Parallel and Distributed Simulation Systems," Wiley-Interscience, 2000

[7] Gu, Z. and Q. Zhao, "A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization," *Journal of Software Engineering and Applications*, 2012

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

[9] Callaghan, B., Pawlowski, B., and Statubach, P, "NFS Version 3 Protocol Specification," *RFC 1813*

[10] Fujimoto, R. M., "Performance of Time Warp Under Synthetic Workloads," *Distributed Simulation Conference*, 1990

[11] Yoginath, S.B. and Perumalla, K.S., "Parallel Vehicular Traffic Simulation using Reverse Computation-based Optimistic Execution," *IEEE Workshop on Principles of Advanced and Distributed Simulation*, 2008

[12] Jefferson, D., Beckman B., Wieland, F., Blume, L., and Diloreto, M. "Time warp operating system" *Proceedings of the ACM Symposium on Operating systems principles*, 1987

[13] Park, A., "Master/Worker Parallel Discrete Event Simulation" PhD thesis, Georgia Institute of Technology, 2008