

Efficient Parallel Discrete Event Simulation on Cloud/Virtual Machine Platforms

SRIKANTH B. YOGINATH, Oak Ridge National Laboratory
KALYAN S. PERUMALLA, Oak Ridge National Laboratory

Cloud and Virtual machine (VM) technologies present new challenges with respect to performance and monetary cost in executing parallel discrete event simulation (PDES) applications. Due to the introduction of overall cost as a metric, the traditional use of the highest-end computing configuration is no longer the most obvious choice. Moreover, the unique runtime dynamics and configuration choices of Cloud and VM platforms introduce new design considerations and runtime characteristics specific to PDES over Cloud/VMs. Here, an empirical study is presented to guide an understanding of the dynamics, trends, and trade-offs in executing PDES on Cloud/VM platforms. Performance and cost measures obtained from multiple PDES applications executed on the Amazon EC2 Cloud and on a high-end VM host machine reveal new, counterintuitive VM-PDES dynamics and guidelines. One of the critical aspects uncovered is the fundamental mismatch in hypervisor scheduler policies designed for general cloud workloads versus the virtual time ordering needed for PDES workloads. This insight is supported by experimental data revealing the gross deterioration in PDES performance traceable to VM scheduling policy. To overcome this fundamental problem, the design and implementation of a new deadlock-free scheduler algorithm are presented, optimized specifically for PDES applications on VMs. The scalability of our scheduler has been tested up to 128 VMs multiplexed on 32 cores, showing significant improvement in the runtime relative to the default Cloud/VM scheduler. The observations, algorithmic design, and results are timely for emerging cloud/VM-based installations, highlighting the need for PDES-specific support in high performance discrete event simulations on Cloud/VM platforms.

Categories and Subject Descriptors: C.2.2 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Protocols*; C.5.1 [**Computer Systems Organization**]: Computer System Implementation—*Large and Medium Computers*; I.6.1 [**Computing Methodologies**]: Simulation and Modeling—*Discrete*; I.6.8 [**Computing Methodologies**]: Simulation and Modeling—*Types of Simulation (Discrete Event, Parallel)*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Parallel Discrete Event Simulation, Time Warp, Global Virtual Time, Virtual Machines, Scheduler

ACM Reference Format:

Yoginath, S. B., Perumalla, K. S., 2014. Efficient Parallel Discrete Event Simulation on Cloud/Virtual Machine Platforms ACM Trans. Model. Comput. Simul. 0, 0, Article 0 (2014), 26 pages.
DOI = 10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

1. INTRODUCTION

1.1. Cloud/VMs as PDES Execution Platform

Parallel computing platforms based on virtualization technologies, such as commercial Cloud offerings, have emerged lately, and are seen as a good alternative to native ex-

Author's addresses: S. B. Yoginath and K. S. Perumalla, Computational Sciences and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1049-3301/2014/-ART0 \$10.00

DOI 10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

ecution 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.

1.2. Issues and Challenges

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

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

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

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

Scheduling Problem The conflict arises due to the hypervisor scheduler: the default schedulers designed for general Cloud workloads are a mismatch to PDES work-

loads. The hypervisor is a critical component of the virtualized system, enabling the execution of multiple VMs on the same physical machine. To support the largest class of applications on the Cloud, a fair-sharing scheme is employed by the hypervisor for sharing the physical processors among the VMs. The concept of fair sharing works best either when the VMs execute relatively independently of each other, or when the concurrency across VMs is fully realized via uniform sharing of computational cycles. This property holds 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.

Virtual Time-based Scheduling In PDES the simulation time advances with the processing of time-stamped simulation events. In general, the number of events processed in a PDES application varies dynamically during the simulation execution (i.e., across simulation time), and also varies across processors. This implies that the amount of computation cycles consumed by a processor for event computation does not have any specific, direct correlation with its simulation time. A processor that has few events to process within a simulation time window ends up consuming few computational cycles. It is not ready to process events belonging to the simulation-time future until other processors have executed their events and advanced their local simulation time. However, a fair-share scheduler would bias the scheduling towards this lightly loaded processor (since it has consumed fewer cycles) and penalize the processors that do in fact need more cycles to process their remaining events within that 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.

A final consideration is that a scheduling algorithm based solely on least-simulation-time-first-order is susceptible to deadlocks which need to be resolved and implemented in a scalable manner with respect to the number of VMs multiplexed by the hypervisor. Thus, a new, deadlock-free, scalable hypervisor scheduling algorithm is needed to deliver the most efficient execution of PDES on Cloud/VM platforms.

1.3. Organization

The rest of the paper is organized as follows. In Section 2, a brief background is provided on the major concepts used in the work. In Section 3, the design of the PDES-specific VM scheduler algorithm is presented, followed by the details of its implementation in Section 4. A detailed performance evaluation is presented in Section 5. Related work is covered in Section 6. The results are summarized in Section 7 followed by conclusions and future work in Section 8.

2. BACKGROUND

In this section, a brief overview is provided on virtual machines, cloud computing, the Xen hypervisor, hypervisor scheduling, and PDES execution.

2.1. Virtual Machines

The concept of virtualization has been realized at different levels of the computer systems architecture. At the hardware-level, two methodologies are used for virtualiza-

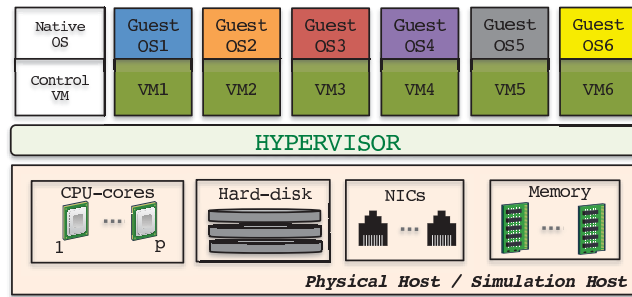


Fig. 1: Xen Hypervisor

tion, namely, *full-virtualization* and *para-virtualization*. Although they are similar in functionality, they differ in the means to realizing virtualization. Both the methodologies run on the top of the hardware by pushing the OS above them and make use of highly configurable virtual machines comprising virtual peripheral I/O components. Para-virtualization differs from full-virtualization in that it requires the modification of guest OS kernel, while the full-virtualization can host any OS without modifications.

The VMware ESX Server, the Xen Hypervisor, and the Microsoft Hyper-V hypervisor are examples of popular VM systems. The VMware ESX Server hypervisor was principally designed to support full-virtualization. While the Xen [Chisnall 2007; Matthews et al. 2008] hypervisor started with the concept of para-virtualization but currently also supports full virtualization. The Microsoft Hyper-V hypervisor also supports full-virtualization. The concepts developed in this paper apply equally well to all these VM systems because all of them share the fundamental concept of multiplexing many virtual resources on fewer physical resources; it is the multiplexing of virtual processor cores over real processor cores which creates a fundamental runtime problem of correctness and runtime efficiency of PDES over VMs.

Currently, applications based on virtualization technology span from single-user desktops to huge data centers. On the lower end, virtualization allows desktop users to concurrently host multiple OS instances on the same hardware. On the larger scale, VMs can be moved from one hypervisor (or device) to another even while the VMs are actively running. This capability for mobility is used to support many advantageous features of Cloud computing, including load balancing, fault tolerance, and economical hosting of computing/storage services.

2.2. Cloud Computing

A very attractive product for both business operators and users alike arose from tapping the virtualization technology through the Internet services, which famously came to be known as the Cloud computing. The Infrastructure-as-a-Service (IAAS), Platform-as-a-Service (PAAS), Software-as-a-Service (SAAS) and Network-as-a-Service (NAAS) are prominent among the types of services offered currently by the Cloud computing service vendors [Mell and Grance 2011]. Apart from IAAS, the other services are completely unaware of the physical hardware on which they are executing. Exploiting the unlikelihood of 100% resource utilization from all clients at all times, the Cloud operators multiplex VMs on limited resources and hence are able to provide easy accessibility to large compute resource at an impressively competitive price.

2.3. Xen Hypervisor

The Xen hypervisor is a popular open source industry standard for virtualization, supporting several architectures including x86, x86-64, IA64, and ARM, and guest OS types including Windows, Linux, Solaris and various versions of BSD OS. Figure 1 shows a schematic of guests running on the Xen hypervisor. Xen refers to VMs as Guest Domains or DOMs. Each DOM is identified by its DOM-ID. The first DOM, DOM-0, possesses special hardware privileges. Such privileges are not provided to other user DOMs, which are generically referred to as DOMUs. Each DOM has its own set of virtual devices, including virtual multi-processors called virtual CPUs (VCPUs). System administration tasks such as suspension, resumption, and migration of DOMs are managed via DOM0.

2.4. Hypervisor Scheduling

Among the shared resources multiplexed by the hypervisor, the physical processor cycles are especially important for PDES. To seamlessly share the physical CPU (PCPU) resources among the virtual CPUs (VCPUs), the hypervisor contains a scheduler that allots PCPU cycles to the VCPUs using a scheduling policy. The hypervisor's scheduling is distinct from multi-threading in operating systems hosted in the VMs over the hypervisor [Chisnall 2007]. Essentially, there exist three scheduling tiers in Xen:

- A user-space threading library schedules user-space threads over OS-level (kernel) threads within a VM
- Every guest OS schedules its kernel threads to VCPUs
- The hypervisor schedules the VCPUs over the PCPUs.

The focus of this paper is on the lowest layer, namely, the hypervisor-level mapping of VCPUs to PCPUs.

2.5. Credit Scheduler of Xen

The credit-based scheduler of Xen (CSX) is Xen's default hypervisor scheduler based on the principle of fair sharing. CSX uses a concept of *credits* for every DOM; these credits are expended as the DOMs VCPUs are scheduled for execution. It provides control to the user to alter the configuration of scheduling through parameters called *weight* and *cap*. While the *weight* value determines the share of PCPU cycles a DOM's VCPU gets with respect to the VCPUs of other DOMs, the *cap* value restricts the utilizable PCPU cycles by a DOMs VCPU. By default, the *weight* value for all DOMs is 256 and *cap* is 0, providing a fair CPU allocation to all of the DOMs. This scheduler is very widely used, and works excellently for a large variety of virtualization uses. However, under overloaded conditions (number of VCPUs > number of PCPUs) and when the loads on DOMs are non-uniform (as is the case in PDES), this fairshare scheduling algorithm detrimentally affects the overall performance of the parallel application.

2.6. PDES Model Execution

In PDES, the model is divided into distinct independent virtual timelines referred to as logical processes (LP). Each LP typically encapsulates a set of state variables of modeled entity. The timelines of LPs within and across processors are kept synchronized by the PDES simulation engine. PDES engines may support optimistic synchronization or conservative synchronization or a combination. The runtime environment allows multiple LPs to be hosted per simulation loop (SL) and, each SL is mapped to a single processor core. When executed on VM, this implies that each SL is mapped to a single VCPU. Thus, the scheduling problem for PDES on VMs becomes one of (indirectly) scheduling SLs over PCPUs.

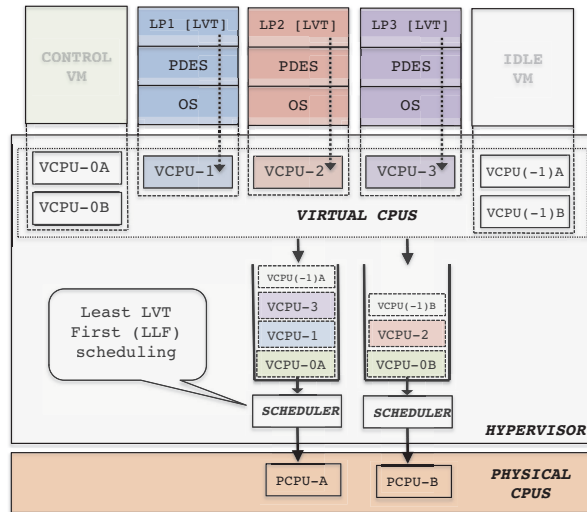


Fig. 2: PDES scheduler design

3. PDES SCHEDULER DESIGN

A hypervisor scheduler time-multiplexes (many) VCPUs and on to (few) PCPUs. To accomplish this, each PCPU maintains a run-queue, which is a list of VCPUs that share the PCPU resources, as shown in Figure 2. The hypervisor scheduler follows a system-defined scheduling policy for dynamically mapping the VCPUs onto PCPUs. A fair-share policy is usually adopted by the default hypervisor schedulers to suit a wide range of workloads.

In PDES, the LP with the least value of *local virtual time* (LVT) affects the progress of the entire simulation. Hence, by prioritizing the VCPU that hosts the SL with the least LVT value, the runtime performance can be optimized. To achieve this, the SLs on different VMs need to communicate the LVT values to their corresponding VCPUs, so that the hypervisor scheduler can use that information to schedule the VCPUs onto the PCPUs.

Figure 2 shows the system architecture of a hypervisor-based execution platform customized for PDES applications. In the figure, for clarity, only a single LP is shown per SL and a single SL is shown per VCPU. However, any number of LPs per SL and any number of SLs per VCPU can be supported by our approach with simple modifications. As illustrated in Figure 2, the LVT of an LP is passed to the VCPU in its VM. The VCPU records the LVT inside the hypervisor's scheduler data structures as the VCPU LVT. The hypervisor scheduler uses the LVT values of the VCPUs in a Least-LVT-First (LLF) strategy during scheduling. With the LLF scheduling policy, the scheduler gives the highest priority to the VCPU with least LVT value.

The special VMs (DOM0 and Idle-DOM) of Xen do not participate in the PDES simulation. The DOM0 is the privileged DOM used to manage and control VMs, while the Idle-DOM is a Xen mechanism to ensure that the PCPU run-queues are never empty. These are hypervisor-specific artifacts that do not affect the fundamental PDES-specific VM scheduling problem.

3.1. Deadlocks from Purely LVT-based VM Scheduling

The VCPU scheduling based on a purely LLF-based policy leads to deadlock of PDES applications when the number of hosted VCPUs is greater than the available PC-

PU. This is because the VCPUs with smaller VCPU LVT values (i.e., having a higher scheduling priority) would prevent the VCPUs with a higher VCPU LVT value from being scheduled for execution. Consequently, some of the SLs (those executing in VMs with higher LVT values) would never get a chance to participate in inter-SL time synchronization operations such as Global Virtual Time (GVT) or Lower Bound on incoming Time Stamp (LBTS) computation. Since GVT or LBTS constrain the progress of the simulation, the execution deadlocks. Note that this deadlock can arise for both conservative or optimistic synchronization. In conservative operation, simulation progress is critically dependent on global time guarantees, and hence no processor will be able to go past the most recently computed LBTS. Optimistic operation, although resilient to an extent to slow GVT progress rates, can also deadlock if/when the processor with the least LVT runs out of memory because fossil collection gets stalled when GVT computation does not progress.

3.2. Deadlock Condition

In this section we reason the presence of deadlocks that arise in a purely LLF-based scheduler. For a system to deadlock, four conditions are necessary and sufficient [Coffman et al. 1971]:

- *Mutual exclusion*: Tasks claim exclusive control of the resources
- *Hold and wait condition*: Tasks hold resources already allocated to them while waiting for additional resources
- *No preemption*: Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion
- *Circular wait*: A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain

All these conditions arise with PDES over pure LLF hypervisor scheduler as follows.

Mutual Exclusion. At runtime, only one VCPU can execute at any given time on a PCPU. When a VCPU is scheduled, no other VCPU can get computational cycles. The scheduled VCPU always has exclusive control of the PCPU resource. This satisfies the mutual exclusion criterion for deadlock.

Hold and Wait. As long as the scheduled VCPU has the least LVT, it is always assured of coming back to the PCPU even after its timeslice is exhausted. If the LVT of such a VCPU (with least LVT) does not progress, that VCPU holds up the PCPU resource (HOLD). The PCPU resource is indefinitely held up by this least-LVT VCPU (WAIT) because it is waiting for another processor to participate in GVT/LBTS computation. This will not be resolved because at least one of the SLs (with a larger LVT) does not participate in the GVT computation due to non-availability of PCPU compute cycles.

No-Preemption Condition. Since the hypervisor scheduler strictly follows least-LVT first order of scheduling, it never preempts the least-LVT VCPU with other VCPUs.

Circular Wait Condition. The LVT values to each of the VCPUs are passed from the PDES application. To ensure the progress of the simulation, each of the SLs need to regularly exchange their LVTs with their peer SLs to compute the GVT. For the GVT computation SLs need PCPU cycles. When the number of VCPUs is greater than number of PCPUs, some VCPUs do get PCPU cycles. Hence, The least-LVT VCPU waits for the GVT message from its peer, while its peer VCPUs (with higher LVT value) wait on the least-LVT VCPU to release the PCPU resource (to participate in GVT computation). Thus, a circular wait condition emerges.

3.3. Deadlock Resolution

The deadlock does not occur when at least one of the aforementioned conditions is violated. Hence, a simple way to break the deadlock is through preemption. In the hypervisor scheduling process this can be done by occasionally ignoring the application-supplied LVT and increasing the LVT value of the VCPU to a large value. One simple algorithm to achieve this is to make the VCPU LVTs toggle between the actual LVT and a very large LVT value in regular scheduling intervals. By such toggling, the VCPUs self-preempt themselves momentarily. This simple algorithm breaks the deadlock and ensures that all the SLs get the necessary PCPU cycles to participate in GVT or LBTS algorithms, as demonstrated in [Yoginath and Perumalla 2013b]. However, this method is inefficient because of the significantly large number of self-preemptions. This simple algorithm can be optimized by relaxing it into a *counter-based* approach, wherein every VCPU maintains a counter in addition to its VCPU LVT. This counter in the VCPU is incremented whenever they are bypassed during scheduling by another VCPU with a lower LVT. When the counter of any VCPU reaches a pre-specified threshold (empirically set), this VCPU preempts others in the run queue to break the deadlock.

3.4. Counter-based Algorithm to Resolve Deadlock

Algorithm 1 gives the pseudo-code for our new deadlock-free PDES specific hypervisor scheduler algorithm. As mentioned previously, the hypervisor scheduler inserts the currently executed VCPU v_c into the run-queue and picks a new VCPU v_n for scheduling onto the PCPU. The hypervisor scheduler servicing the interrupt of the PCPU performs this scheduling action continuously. The Algorithm 1 starts executing on PCPU p with v_c as input and determines v_n as the output. The process of selecting the v_n follows the LLF principle for PSX, modified for deadlock avoidance and for minimizing inter-processor synchronization (locking).

The algorithm starts by updating the VCPU LVT $T[v_c]$ value of the v_c before inserting it into the interrupted PCPU p 's run-queue ($RQ[p]$) following the LLF principle (lines 2 to 4). A GVT counter ($g[1 : V]$) is maintained for each VCPU. During insertion of v_c into the local run queue, the GVT counters of all VCPUs being preceded in the local queue $RQ[p]$ are incremented (that is, those VCPUs whose LVT is greater than v_c 's (lines 5 to 7)).

Next, the VCPU to be scheduled next (v_n) is determined as follows. The run queue $RQ[p]$ is first searched to find if any VCPU is starving for cycles. This is indicated by a GVT counter value exceeding the GVT threshold G . If any VCPU satisfying this pre-emption condition is found, that VCPU is selected as v_n (lines 9 to 14), and scheduled next after resetting its GVT counter $g[p]$ to zero (line 37). If no such VCPU is found, then an effort is made to find a VCPU with a lower LVT across all the PCPU run queues (lines 15 to 36). In this code segment, first the least LVT VCPU of $RQ[p]$ is picked as the default candidate v_n and its LVT is compared with that of the VCPUs at the head of run queues of other PCPUs. If this search is unsuccessful, the local candidate is returned as v_n . Otherwise, the VCPU v_r stolen from the peer PCPU run-queue ($RQ[p']$) is returned as v_n . Note that code-segment in between (line 32 to 35) increments the GVT counter of all the VCPUs in $RQ[p]$ on successfully stealing the VCPU v_r . Not incrementing the GVT counter after stealing of lower LVT VCPU v_r results in scenarios wherein the lower LVT VCPUs may be continuously exchanged between the PCPU run-queues without altering the GVT counters of the VCPUs. Hence, not incrementing GVT counter of all VCPUs in $RQ[p]$ after successfully stealing makes the deadlock persist. Finally, the GVT counter of the v_n is reset before it is scheduled on the PCPU.

ALGORITHM 1: Counter-based deadlock-free hypervisor scheduler algorithm

Input: p, v_c
Output: v_n
Data:

p PCPU currently being serviced by this scheduler algorithm
 v_c VCPU that is currently vacating the PCPU p
 v_n VCPU to be picked next for execution on PCPU p
 P Total number of PCPUs (constant)
 V Total number of VCPUs (constant)
 G GVT counter *threshold* (constant)
 $RQ[1..P]$ Each element $RQ[p]$ is a list of VCPUs $\langle v \rangle$ ready to run on PCPU p
Each list is ordered by $T[v]$ (ascending), then by $g[v]$ (descending)
 $L[1..P]$ Each element $L[p]$ is a lock for exclusive read-write access to $RQ[p]$
 $g[1..V]$ GVT *counter* variable for every VCPU to avoid deadlock; initialized to 0
 $T[1..V]$ LVT value for every VCPU dynamically supplied by application; initialized to 0

```

1  $g[v_c] \leftarrow 0$ 
2 Obtain latest LVT  $t_c$  of VCPU  $v_c$  from application
3  $T[v_c] \leftarrow t_c$ 
4 Insert  $v_c$  in  $RQ[p]$  behind all  $v$  in  $RQ[p]$  whose  $T[v] \leq T[v_c]$ 
5 for every VCPU  $v$  in  $RQ[p]$  whose  $T[v] > T[v_c]$  do
6   Increment  $g[v]$ 
7 end
8  $v_n \leftarrow -1$ 
9 for every VCPU  $v$  from head to tail of  $RQ[p]$  do
10   if  $(g[v] > G)$  then
11      $v_n \leftarrow v$ 
12     break out of loop
13   end
14 end
15 if  $(v_n = -1)$  then
16    $v_n \leftarrow$  head of  $RQ[p]$  /*local VCPU with least LVT*/
17   /*See if at least one other PCPU  $p'$  has a VCPU  $v_r$  with a smaller LVT*/
18    $v_r \leftarrow -1$ 
19   for (each PCPU  $p' \neq p$ ) do
20     Attempt to obtain lock  $L[p']$ 
21     if lock  $L[p']$  is successful then
22        $v_0 \leftarrow$  head of  $RQ[p']$ 
23       if  $T[v_0] < T[v_n]$  then
24          $v_r \leftarrow v_0$ 
25       end
26       unlock  $L[p']$ 
27       if  $v_r \neq -1$  then
28         break out of loop
29       end
30     end
31   end
32   if  $(v_r \neq -1)$  then
33     Increment  $g[v]$  for all VCPUs  $v$  in  $RQ[p]$ 
34      $v_n \leftarrow v_r$ 
35   end
36 end
37  $g[v_n] \leftarrow 0$ 
38 return  $v_n$ 

```

```

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;
};

```

Fig. 3: PSX shared_info data structure

4. PDES SCHEDULER IMPLEMENTATION

To realize the PDES optimized hypervisor scheduler, we require (a) each μsik kernel instance to independently communicate its LVT value to the Xen scheduler, and (b) a new Xen hypervisor scheduler that utilizes the LVTs to optimize the compute resource sharing specifically for PDES applications. Also, note that the PDES specific hypervisor scheduler should be active only during the execution of the PDES application. To realize this, we need at least two different modes of hypervisor scheduler operations. We refer the mode of operation during PDES execution as *simulation mode*, which otherwise is referred to be in a *normal mode* of operation.

4.1. Communicate LVT to Hypervisor Scheduler

To efficiently communicate LVT to the hypervisor, necessary support from the guest-OS kernel is expected. The guest-OS kernel should also impart the obtained application-level information to the hypervisor. Further, the PDES μsik library should also be modified to communicate the change in hypervisor scheduler mode of operation, and the LVT value, to the guest-OS kernel.

4.1.1. Linux Kernel Modifications. Each guest-OS maintains a *shared_info* page (Figure 3), which is dynamically updated as the system runs. We added two fields namely, *simtime* and *switch_scheduler* are added to the *shared_info* data-structure. The *simtime* is used to record the LVT from the SL and the *switch_scheduler* flag indicates the switch between different modes of scheduler operation. To send the LVT information from the application level, we implemented a system call for our guest-OS (Linux). This *system call* allows the LVT information to transit from user-space to kernel-space and here the LVT value is written into the *simtime* of *shared_info* data-structure of the host DOM. This *shared_info* data-structure is accessed by the hypervisor during scheduling.

4.1.2. Modifications to the PDES Engine μsik . Every μsik SL 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 [Perumalla 2005]. 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 simulation time value was used in all our experiments.

During simulation initialization the μsik library sets *switch_scheduler* in *shared_info* of its host DOM to *true*, using our *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*, suggesting the scheduler

```

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

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

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

```

Fig. 4: PSX VCPU, PCPU and global ps_private data structures

to revert back to its normal-mode of operation. The same *system call* is used to by the μ sik library to write its SL's LVT value to *simtime* of the *shared.info* of host DOM.

4.2. LVT-based PDES Hypervisor Scheduler

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 [Yoginath and Perumalla 2011]. *ps_private* (Figure 4) is the structure of the global data-structure object that the PSX scheduler maintains. The *switch_sched* variable of *ps_private* object is updated during VCPU scheduling after corresponding value is read from *shared.info* by the VCPU from its relevant DOM. Setting the *switch_sched* in the PSX's *ps_private* global variable enables PSX to switch from *normal mode* to *simulation mode* and viceversa.

4.2.1. Scheduling in Normal-mode. The scheduler is referred to be in normal-mode if the *switch_sched* (of *ps_private* data-structure Figure 4) is false. This corresponds to the mode in which the VMs are booted and operational, but no PDES run is active (and hence LVT-based scheduling is undefined). In this mode of operation, PSX maintains the *sim_time* (VCPU data-structure Figure 4) of all DOM0 VCPUs lower than all the DOMUs. In the normal-mode all the guest-DOM VCPUs have their *sim_time* initialized to a constant 1, while DOM0 VCPUs have their *sim_times* initialized to a constant 0. Only after the *switch_sched* is set true by PDES SL, 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.

Table I: EC2 VM specifications

| | |
|-------------------|--|
| <i>m1.small</i> | 1-core VM with compute power of 1 ECU with memory of 1.7 GB |
| <i>m1.medium</i> | 1-core VM with compute power of 2 ECUs with 3.7 GB memory |
| <i>m1.large</i> | 2-core VM with compute power of 4 ECUs with memory of 7.5 GB |
| <i>m1.xlarge</i> | 4-core VM with compute power of 8 ECUs with memory of 15 GB |
| <i>m3.2xlarge</i> | 8-core VM with compute power of 26 ECUs with memory of 30 GB |
| <i>hs.8xlarge</i> | 16-core VM with compute power of 35 ECUs with memory of 117 GB |

4.2.2. *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 *RQ* (priority-queue) as shown in Figure 4 and, in the simulation-mode PSX en-queues the VCPUs to be scheduled in LLF priority. We use the LVT as the VCPU priority the lower the *sim.time* (VCPU data-structure Figure 4), the higher is its priority in the *RQ* and, hence the earlier it is picked by PSX to allocate compute resource. The scheduler allots a *tick* amount of PCPU cycles for the selected VCPU. Also, 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 PSXs mode of operation. This guarantees that DOM0 VCPUs are always preferred over the other VCPUs, which in turn ensures a good and responsive user-interactivity with DOM0 before, during and after PDES executions.

5. PERFORMANCE EVALUATION

5.1. Hardware

5.1.1. *Local Test Platform (LTP).* is our custom-built machine with a Supermicro H8DG6-F motherboard supporting two 16-core (32 cores in total) AMD Opteron 6276 processors at 2.3 GHz, sharing 256GB of memory, Intel Solid State Drive 240GB and a 6TB Seagate constellation comprising 2 SAS drives configured as RAID-0. Ubuntu-12.10 runs with Linux 3.7.1 kernel runs as DOM0 and DOMUs, over Xen 4.2.0 hypervisor. All DOMUs are para-virtual and networked using a software bridge in DOM-0. DOM-0 is configured to use 10GB of memory and the guest DOMs were configured to use at least 1GB memories each, which were increased as necessitated by the application benchmarks. Each guest DOM uses 2GB of LVM-based hard disk created over SAS drives, while the DOM-0 uses an entire Solid State Drive (SSD). OpenMPI-1.6.3 (built using gcc-4.7.2) was used to build the simulation engine and its applications. A machine-file listing the IP addresses of the VMs was used along with mpirun utility of OpenMPI to launch the MPI-based PDES applications onto VMs. The 1ms tick size was used with both CSX and PSX schedulers.

5.1.2. *Amazon EC2.* We also ran our benchmarks on Amazons EC2 Cloud platform. We built a cluster of para-virtual VM instances of Ubuntu 12.04 LTS. Table I lists the VMs using which the clusters were built to run the performance benchmarks. In Table I, the term ECU refers to a EC2 Compute Unit, which is an abstraction defined and supported by Amazon as a normalization mechanism to provide a variety of virtual computation units independent of the actual physical hardware support that they use/maintain/upgrade without user intervention. OpenMPI-1.6.3 was built on the virtual instance, which was used to build the simulation engine and all the PDES applications. A machine-file listing the DNS names of the allotted instances was used to launch the MPI-based PDES applications using mpirun.

Table II: Abbreviations used by PDES applications

| | |
|-------------|---|
| <i>CSX</i> | Credit Scheduler of Xen |
| <i>PSX</i> | PDES Scheduler for Xen |
| <i>CONS</i> | PDES using conservative synchronization scheme |
| <i>OPT</i> | PDES using optimistic synchronization scheme |
| <i>NLP</i> | Number of LPs per simulation loop (SL) |
| <i>NMSG</i> | Number of messages generated per LP |
| <i>LOC</i> | Percent of traffic generated for local LPs (within same SL) |
| <i>LA</i> | Lookahead |

5.2. Software

Three μ sik library based application benchmarks, namely, PHOLD [Fujimoto 1990] (a synthetic PDES application generally used for performance evaluation), PDES-based Disease Spread simulation [Perumalla and Seal 2012], and SCATTER [Yoginath and Perumalla 2008] [Yoginath and Perumalla 2009] (a PDES-based vehicular traffic simulation application) were used for our performance studies. Various abbreviations used by the PDES applications and performance graphs have been consolidated in the Table II.

5.2.1. PHOLD Benchmark. 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 μ sik implementation of PHOLD allows exercising a wide variety of options in its execution. The NLP, NMSG, LA, LOC and synchronization techniques comprising CONS and OPT, were varied to realize a range of simulation scenarios.

5.2.2. Disease Spread Benchmark (DSB). This is an epidemiological disease spread PDES application [Perumalla and Seal 2012] that uses a discrete event approach to model the propagation of a disease in a population of individuals across locations and across regions (aggregates of locations). Each region is mapped to a SL and each location is housed in an LP. Multiple individuals are instantiated at each location, and they not only interact with individuals within the same location but also periodically (conforming to an individual-specific time distribution function) move from one location to another within and across regions. The scenario configuration parameters for this application are same as PHOLD, where NLP refers to number of locations in a region (SL), NMSG refers to population/location, and LOC refers to percentage of population movements within same region.

5.2.3. SCATTER Benchmark. This application is a discrete-event formulation and a parallel execution framework for vehicular traffic simulation. 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. Dijkstras shortest-path algorithm is used to direct a vehicle to its destination.

5.3. CSX Performance Characteristics

Before presenting the performance comparison of the PSX over the default CSX, we touch upon certain interesting aspects of CSX performance. Some information presented here is borrowed from our prior detailed study [Yoginath and Perumalla 2013a].

5.3.1. Native vs. VM. In Figure 5 we compare the PDES performance of PHOLD and DSB on a native Linux and VMs over the same hardware platform. Performance on VM which can be either privileged (DOM0) and non privileged (DOMU) is also presented.

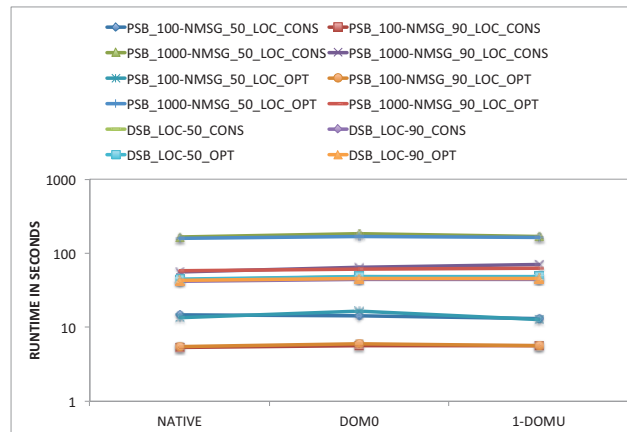


Fig. 5: Native, DOM0 and DOMU performance comparison on LTP

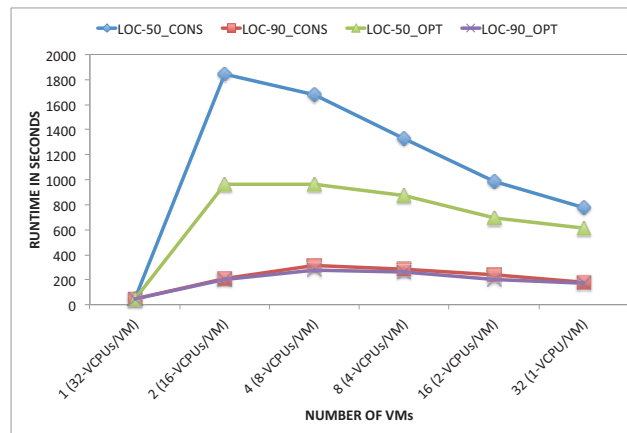


Fig. 6: Performance comparison with increase in the number of DOMs using DSB on LTP

The PHOLD scenarios were configured to use 100 LPs/SL, 100 and 1000 messages/LP, with 32 SLs for 50% and 90% LOC values. The DSB scenario involved simulation of disease spread across 320 locations among a population of 320000 for 7 days of simulation time using 32 regions (SLs), each with 10 locations (LPs) and each location with a population of 1000 (event messages). The LOC of 50% and 90% in DSB suggest that 50% and 90% of the trips the population perform are within the same region, respectively, which also suggest that the rest of the trips performed are across the regions (SL). Figure 5 shows the benchmark runtimes three setups namely, (a) native without hypervisor, (b) DOM0 with 32 VCPUs and, (c) DOMU with 32 VCPUs along with DOM0 without computational load. The runtime results across all the three setups were found almost identical for all the benchmarks. This result is significant because it demonstrates that when the virtual resources are exactly match the physical resources the performance of native and VM platforms are almost identical, suggesting a very-low overhead due to the presence of the hypervisor.

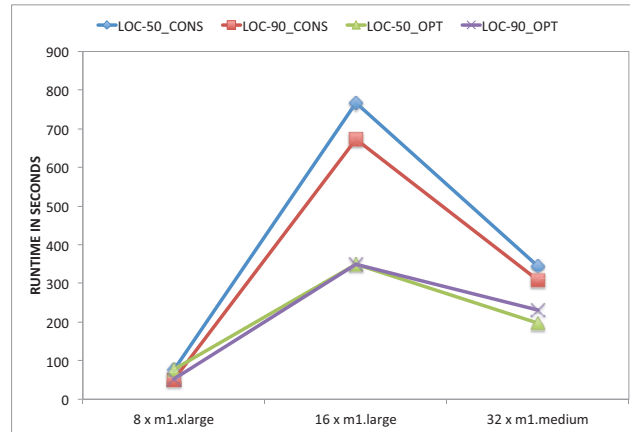


Fig. 7: Runtime performance of DSB on Amazon EC2 cloud

5.3.2. Configurability Options. With a hypervisor, the physical platform can host VMs of different capacities concurrently. This flexibility provides various options to the user for VM selection. For example: a physical platform with 32 CPU-cores can host a single VM with 32 VCPUs or 2 VMs each with 16 CPU-cores or 4 VMs each with 8 cores and so on. The Cloud platforms utilize this feature and offer the user with VMs of different capacities and the VM usage cost is usually directly proportional to its compute capacity. Hence, it becomes necessary to empirically evaluate the performance of PDES applications across a range VM configurations. Figure 6 and Figure 7 presents the runtime performance trends of DSB with the increase in number of VMs on LTP and Amazon’s EC2 Cloud platform, respectively. Note that even though the number of hosted VMs are different, the aggregate compute resource in each of our test VM configurations exactly match the physical compute resource of the LTP. An interesting and common trend in both these graphs are that the performance degrades with fewer VMs beyond 1, and the performance gets better with increase in the number of VMs. This trend was also observed with PHOLD benchmarks. More information on this counter-intuitive behavior can be found in [Yoginath and Perumalla 2013a]. This trend affects the monetary cost in VM utilization, as the cost of a VM on Cloud is predominantly determined by its compute capacity (number of VCPUs).

5.3.3. Cost vs. Performance on EC2. While the cost-model for the utilization of Cloud compute resources makes economic sense, the options and related pricing of the offered VMs generally confounds the user. This is more true if the user intends the Cloud resource for a parallel computing application, like PDES application. In which case, the monetary cost of executing the same PDES application among various configurable options of VM resources provides necessary insight. In Figure 8, we plot the cost of executing various PHOLD simulation scenarios on cluster formed using various EC2 Cloud resources. In this graph, the left-most VM (on X-axis) is the costliest, while the right-most is the cheapest in terms of monetary cost. From this graph we observe that the costliest resource might not yield the best performance, while the cluster of VMs formed by single-VCPU (m1.medium) seems to provide a better value for the cost. In Figure 9, we plot the runtime and the cost of one of the dense simulation PHOLD scenarios using conservative synchronization (with better runtime than its optimistic counterpart) for comparison. A similar trend was observed with the DSB benchmarks

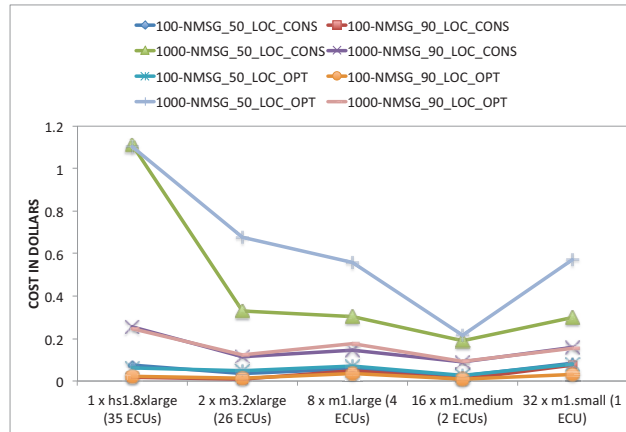


Fig. 8: Cost of running PHOLD benchmarks on Amazon EC2 cloud

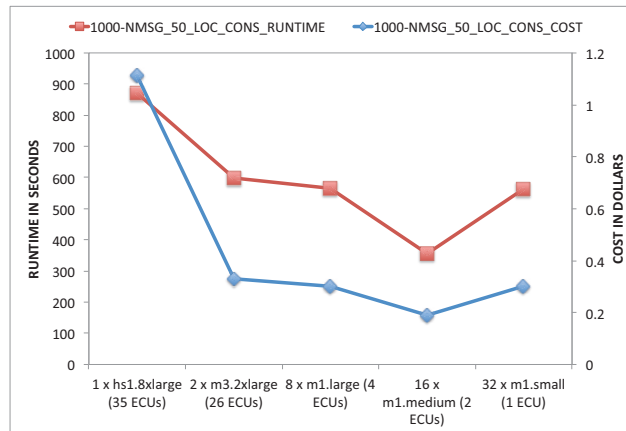


Fig. 9: Cost and runtime plots of a large-scale PHOLD benchmark scenario with NLP=100, NMSG=1000, LOC=50, LA=1 using conservative synchronization scheme on Amazon EC2 cloud

as well. We refer the interested reader to [Yoginath and Perumalla 2013a] for more information.

5.3.4. Higher Multiplexing ratios. Having learnt that a cheaper cluster of single-VCPU VM would yield better performance than a costlier multi-VCPU VM, we further investigated the impact on PDES application performance with increase in the compute resource multiplexing (virtual to physical) ratio. Note, that in all our previous performance discussions we maintained a 1:1 multiplexing ratio, i.e. the number of virtual resources hosted on hypervisor were equivalent to number of physical resources utilized. This is an important study characteristic because the Cloud computing scales economically based on this concept. The understanding and a fact that not every VM hosted on physical resource utilizes its compute resources continuously allows the vendors to host VMs whose aggregate compute resources surpass the actual number of physical compute resources. While, this over-subscription does not introduce any problems, when VMs are independent of each other or when the parallel tasks are

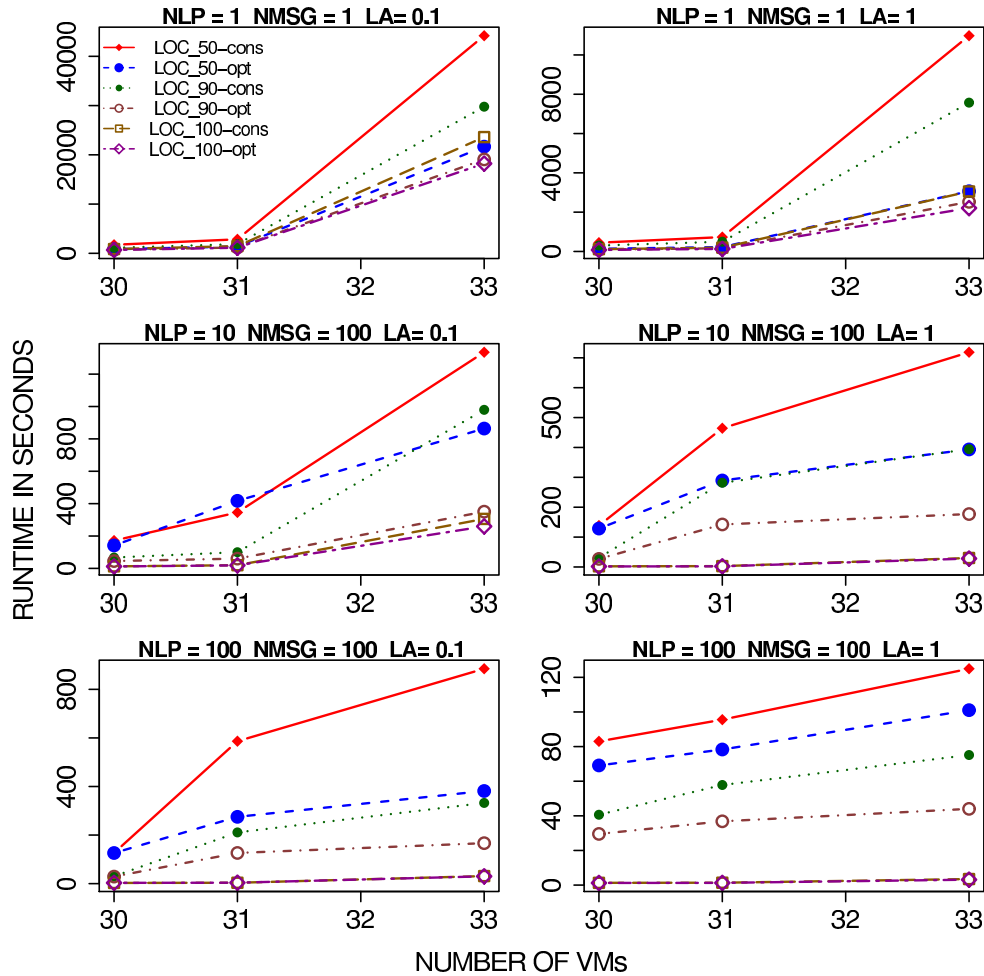


Fig. 10: CSX performance with increase in number of VMs

embarrassingly-parallel, it makes a highly negative impact on fine-grained PDES application's performance. This performance issue can be directly attributed to the virtual compute resource scheduling (hypervisor scheduling) strategy. Figure 10 captures this effect using PHOLD benchmarks over LTP and these plots assert the need for a PDES based hypervisor scheduler.

In this set of experiments, the DOM0 was setup to use two VCPU cores, thus at any time instance 30 remnant PCPUs are available for hosted VM or VMs. We launched single VCPU VMs equal to remnant PCPUs (30) and increase the number of VMs hosted until the number of VCPUs become 10% greater than number of PCPUs. The Figure 10 plots the runtimes for varying PDES loads for a mere 10% increase in the number of hosted VCPUs.

The top two graphs in Figure 10 plot performance runs with lowest possible computational load for varying communication loads and for varying lookaheads 0.1 (left) and 1.0 (right). These show the effect of VCPU scheduling in absence of significant computational load. These plots show several orders of magnitude of degradation in

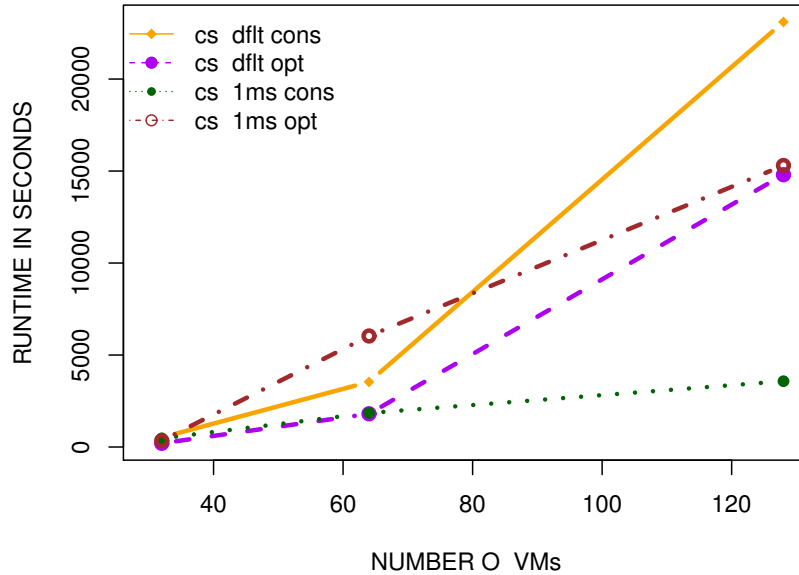


Fig. 11: Runtime performance of PHOLD for varying time slices

performance with negligible increase in load. These readings constitute some of the worst possible performance that can be expected on a cloud platform.

The bottom two graphs in Figure 10 plot performance runs with highest possible computational load for varying communication loads and for varying lookaheads of 0.1 (left) and 1.0 (right). These set of readings represents one of the best performances that PDES applications can expect on a cloud platform. Yet, based on the communication load, the performance varies significantly. At worst it is an order of magnitude slower, as shown by the plots for LOC=50% and 0.1 lookahead configuration.

The center two graphs in Figure 10 plot average computational load for varying communication loads and for varying lookaheads 0.1 (left) and 1.0 (right). These set of readings can be considered to represent an average behavior of most of the PDES applications. With the performance degrading by several folds with increase in VMs, especially with increase in the communication load highlights the high impact of scheduling on PDES application performance.

5.3.5. Lower Time-slice. We observed that the time-slice provided to each VCPU during scheduling significantly alters the performance of the PDES application. Changing the time-slices is made easy in the recent releases of Xen hypervisor using the *xl* tool. By default CSX provides a time-slice of 30ms in quanta of 10ms tick-size for each scheduled VCPU. The time-slice can at most be reduced to 1ms using the *xl* tool. Figure 11 compares the runtimes of PHOLD benchmark scenario (NLP=100, NMSG=100, LOC=95, LA=0.1 and endtime=1e3) for CSX with default time-slice with CSX with 1ms time-slice. As seen in the Figure 11, the conservative synchronization performs extremely well with reduced time-slice as evident in 128 VM and 64 VM scenarios. Close to an order-of-magnitude performance gain is in 128 VM scenario. However, the

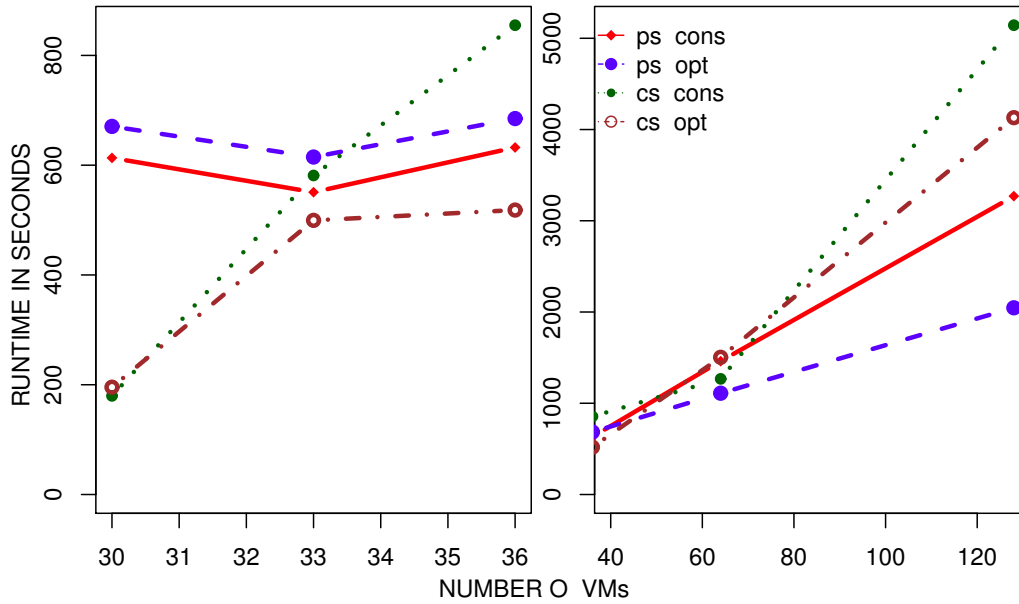


Fig. 12: PSX and CSX comparison PHOLD with LA=0.1

same is not true while using the optimistic synchronization case. In the 128 VM scenario using optimistic synchronization 1ms time-slice makes no difference in runtime when compared to default time-slice, and the performance suffers very badly in the 64 VM scenario. This is because high number of reversals (tens of millions) in case of 1ms time-slice runs compared to lower (few hundred thousands) number of reversals, while using default time-slice. In the absence of high reversals, optimistic synchronization can be expected to perform better than conservative. Hence, all following performance runs use a CSX with 1ms time-slice.

5.4. PSX Performance Comparisons

5.4.1. PSX and CSX Performance Comparison using PHOLD Benchmarks. For this set of performance results we use a PHOLD scenario with NLP=100, NMSG=100, LOC=95 and we host one μ sik SL on a VM. NLP=100, ensures that each VM/SL hosts 100 LPs. The NMSG=100, ensures that each LP exchanges 100 messages amongst its peers. Thus at any instance a simulation scenario with 128 VMs exchanges 1.28 million messages among 12800 LPs. The locality(LOC) is set to 95% suggesting 95% of the randomly generated messages are local while the 5% are sent to a random peer LP hosted on other VM. Locality of 95% ensures that the SL has enough local events to process at any instance, hence the affect of scheduling could be minimal as observed in Figure 10. For all the PSX runs the GVT.Threshold(GT) was kept at 10.

Figure 12 shows the plots of conservative and optimistic runs with lookahead of 0.1. As the number of VMs hosted on the physical machine increases, both optimistic and conservative runtimes of the PSX perform well in comparison to their CSX counterparts. While the PSX using conservative synchronization suffers slightly, the PSX

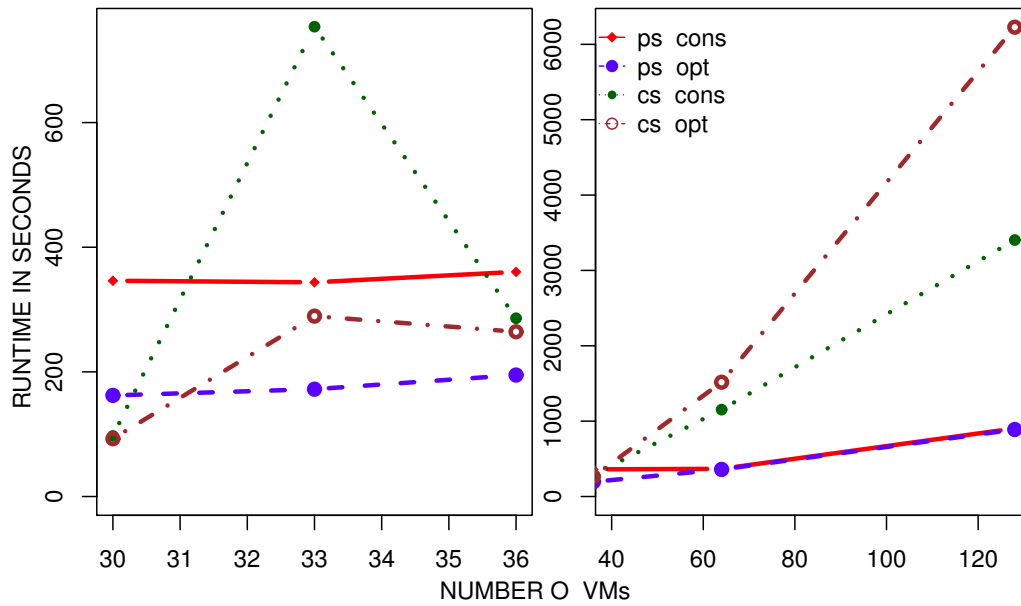


Fig. 13: PSX and CSX performance comparison DSB with LOC=50

execution using optimistic synchronization performs very well with the increase in the number of VMs.

The plot on left in 12 is magnification of the initial set of points. They demonstrate the behavior of PSX with respect to CSX when the multiplexing ratio of VCPUs on to PCPUs are low. As seen, CSX performs best when no mismatch between PCPUs and VCPUs exist and suffers significantly even due to a slight mismatch. In contrast, the PSX suffers in the absence of the mismatch due to unnecessary overhead of writing LVTs to hypervisor and performs better than CSX as the mismatch grows, as expected.

5.4.2. PSX and CSX Performance Comparison using Disease Spread Benchmarks. This DSB benchmark scenario comprises μ sik SLs that correspond to region and LPs that correspond to locations. Each region (μ sik SL) hosts multiple locations (LPs). Each region or SL is hosted on a VM. Here, the DSB scenario comprises 100 locations per region and a population of 100 per each location. Thus with 128 VMs, we simulate the disease spread across a population of 1,280,000 people, across 128 regions, and each with 100 localities. The simulation scenario studies the disease spread among the population over a week.

Figure 13 shows the runtime plots for LOC=50%, suggesting that 50% of the population move across regions. Both optimistic and conservative runs with PSX scheduler perform extremely well with the increase in the VMs. We also experimented with higher LOC percentages (LOC=90%), which limits the population movement across regions to 10% and observed similar performance trends. Similar to the PHOLD benchmark runs, the DSB benchmarks also show that when the physical and virtual compute resources match the CSX performs slightly better than PSX.

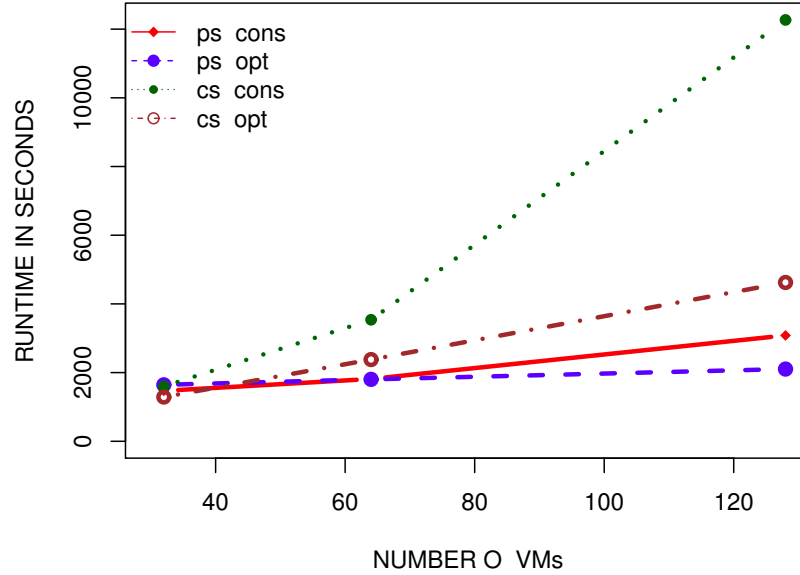


Fig. 14: PSX and CSX performance comparison SCATTER vehicular traffic simulation scenario

5.4.3. PSX and CSX Performance using SCATTER Benchmarks. As opposed to the two prior benchmarks that evaluated weak-scaling (increase in computational load with increase in number of VMs), this benchmark evaluates strong-scaling (computational load remains same across all scenarios varying in terms of number of VMs used) behavior. The SCATTER benchmark simulates the vehicular traffic evacuation scenario of 3.2 million vehicles originating from 256 sources. Each vehicle makes its way across 128×128 (16K) grid of intersections toward its destination (one of the 256 sinks), using the djikstra's shortest path algorithm. The vehicles were generated in source node at a rate of 50 vehicles/sink/hour for an hour. Vehicles injected by the sources placed on either side (left and right) move toward the sinks (top and bottom) across a road-network grid of 128×128 intersections. The same simulation scenario is executed on 32, 64 and 128 VMs. The intersections, sources and sinks are modeled as LPs of PDES. The spatial decomposition ensured that equal number of intersection LPs, source LPs and sink LPs were allotted to each μ sik SL hosted on a VM.

The corresponding performance plots are presented in Figure 14. The runtime of the optimistic plot for the PSX remains almost same with the increase in number of VMs. The PSX conservative runtime also shows similar trend, except when number of VMs hosted is 128, when its runtime slightly increases. In comparison the CSX runtime suffers as the number of VMs hosted increases. However, the CSX using optimistic synchronization is able to curtail the performance degradation significantly in comparison with its conservative synchronization.

5.4.4. PSX, CSX and Native Performance Comparisons. In this section, we compare the performance benchmarks on the VM platform with the native Linux platform, on the

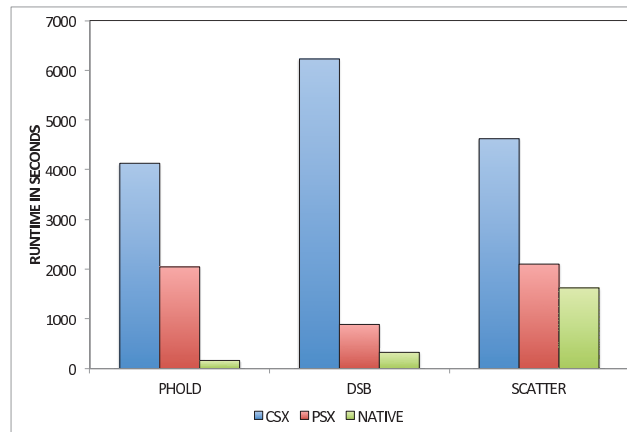


Fig. 15: PSX, CSX and Native Linux performance comparisons with PHOLD (LA=0.1), DSB (LOC=50) and SCATTER benchmarks

same hardware device. For a fair comparison, the number of μ sik SL processes equivalent to number of VMs used (in VM setups) were spawned on Linux. The executions involving 128 VMs using PHOLD, DSB and SCATTER were used for comparison. The best runtime results regardless of the PDES synchronization scheme exercised, was used for comparison.

As seen in Figure 15 the PHOLD benchmark runtime on Linux using 128 processes is several order-of-magnitude faster than results from CSX and PSX. Though PSX is able to alleviate the performance degradation to a certain extent, it still is inefficient because the PHOLD benchmark is very fine grained and the low lookahead (0.1) requires frequent synchronization. This is in spite of optimistic synchronization trying its best to keep the runtime lower. The native runs are almost over an order of magnitude faster than VM environment using PSX or CSX.

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

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

5.4.5. Variance in Performance. The data points of CSX showed high variance, when the number of VCPUs multiplexed were greater than the number of PCPUs. Figure 16 plots the variance of data points of PSX and CSX runtimes in PHOLD benchmark, with lookahead 1. In our previous plots, we had used the best runtimes obtained using CSX plots from multiple runs for comparison with the runtimes of PSX. The observed behavior can be expected from CSX because of the VCPU scheduling strategy it uses. In contrast to CSX, the PSX readings show a very low variance regardless of the number of multiplexed VMs.

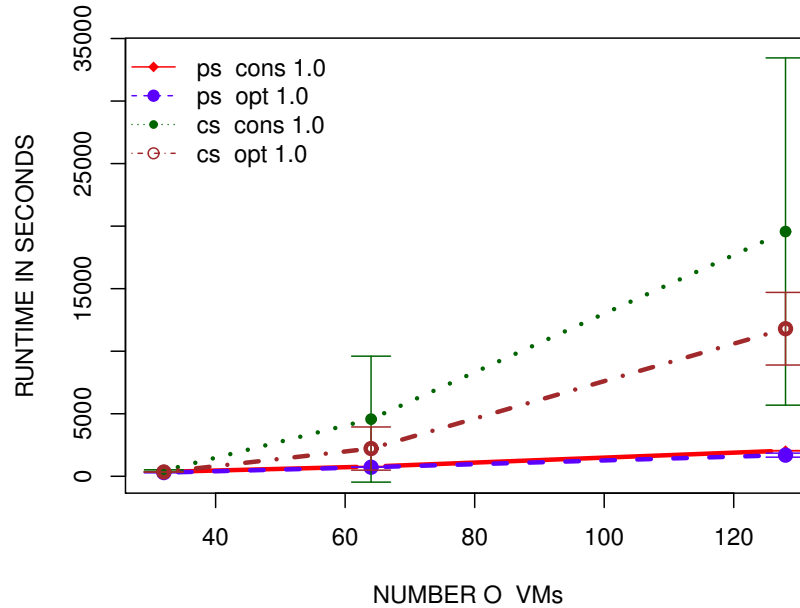


Fig. 16: PSX and CSX variance (with 95% confidence interval) using PHOLD with 1 lookahead

6. RELATED WORK

Evaluation of high performance computing applications on Cloud infrastructures has been reported in [Jackson et al. 2010]; these applications are largely non-PDES, scientific codes such as Community Atmospheric Model (CAM). Network performance on Amazon EC2 data-centers has been studied and an evaluation of the impact of virtualization on network parameters such as latency, throughput, and packet-loss was discussed in [Wang and Ng 2010], again in non-PDES context. There is a good overview and discussion of generic utilization of Cloud infrastructure for PDES applications stressing on advantages and challenges it poses [D’Angelo 2011], which serves as a good motivation and background for PDES on Cloud platforms. The Master-Worker approach to distributed (and fault tolerant) PDES [Park 2009] and optimistic cloud-based execution [Malik et al. 2009; 2010; Fujimoto et al. 2010] are also a related but complementary approaches, different from our support for the traditional PDES execution view in which all processors are equal. We also adopt a unique approach by focusing at the lowest level, namely at the level of the hypervisor itself. Recently, [Vanmechelen et al. 2012] reported evaluation of a set of conservative synchronization protocols on EC2 and suggesting conservative-algorithms that could perform better in the Cloud infrastructure. Overall, the area is nascent, and much additional research is needed to explore the space opened by the new metrics beyond raw speed of PDES execution.

The poor performance of certain high-performance computing applications has also been observed [Jackson et al. 2010] and customized solutions are being proposed, which are not applicable to PDES that fundamentally relies on virtual time order.

Incidentally, the Time-Warp Operating System [Jefferson et al. 1987] of the 1980s 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. There is also a superficial semblance with 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 users simulation model.

While our implementation and experimentation have been performed in the context of the Xen [Chisnall 2007] hypervisor and the μ sik [Perumalla 2005] parallel/distributed simulation kernel, the concepts developed in this paper for VM-based PDES are sufficiently general, and can be applied to other hypervisors and parallel discrete event simulators.

7. SUMMARY

We started by listing out the advantages of using Cloud platforms for PDES applications. We found that the runtime performance (one of core reason for PDES execution) of a PDES application suffered terribly on the Cloud platforms. To understand and unravel the performance issues of PDES application payloads on the Cloud platforms, we undertook an extensive PDES performance study on a custom built hardware with a hypervisor capable of hosting hundreds of VMs. Several performance behaviors like, (a) almost similar runtime performance of PDES application over VM and native platform, when virtual and physical resources match, (b) a counter-intuitive performance trend while using various compositions of VM compute resources for PDES execution and, (c) monetary implications of observed performance (d) severe performance degradation with slight increase(10%) in the virtual to physical multiplexing ratio, and (e) better performance of PDES using smaller time-slices in VCPU scheduling, were discovered.

As a generic guideline when executing PDES application on Cloud, if the number of processor-cores of the physical system on which VMs are hosted is known than use a single VM with number of VCPUs equal to number of processor cores. In the absence of this details the best performance for the monetary cost involved can be obtained by using a cluster of VMs, each with a single VCPU. Note that this guideline is valid when the virtual compute resources exactly match the physical compute resources. Even a slight increase (less than 10%) in the virtual to physical multiplexing ratio yields poor performance. We discovered that the poor performance of the PDES application was related to the hypervisor scheduling policy.

After recognizing the scheduling policy mismatch in Cloud platforms as the reason for poor PDES performance, we designed, implemented and extensively evaluated the performance of a new PDES specific scheduler. We used different scenario configurations of synthetic PHOLD, disease spread simulation and vehicular traffic simulation benchmarks to evaluate the performance. We demonstrated a significant speedup using PSX scheduler over CSX scheduler across all the benchmark runs. Note that all the performance comparisons were done with CSX using small time-slices and hence, the speedup against CSX with default time-slice configurations can be expected to be much higher (order-of-magnitude). We also compared the runtime performance of our three simulation benchmarks involving 128 VMs with 128 processes hosted a Linux machine on the same hardware, and demonstrated that a tremendous reduction in performance degradation can be achieved in VM based execution platforms. We also

demonstrated the high-variance of PDES application runtime with CSX, and very low variance with PSX execution setups.

8. CONCLUSION AND FUTURE WORK

With the proliferation of Cloud and VM-based platforms for parallel computing, it is now possible to execute parallel discrete event simulations (PDES) over multiple VMs, in contrast to executing in native mode directly over hardware as has been traditionally done over the past decades. However, while most VM-based platforms are optimized for general workloads, PDES execution exhibits unique dynamics significantly different from other workloads. Here we present results that identify the gross deterioration of the runtime performance of VM-based PDES simulations when executed using traditional VM schedulers, quantitatively showing the bad scaling properties of the scheduler as the number of VMs is increased. The mismatch is fundamental in nature in the sense that any fairness-based VM scheduler implementation would exhibit this mismatch with PDES runs.

To solve this mismatch, a new algorithm has been presented for PDES-specific scheduling of VMs by a hypervisor. The algorithm schedules VMs primarily by their local virtual time (LVT) order, and incorporates mechanisms that prevent deadlocks and livelocks that are otherwise possible in a purely LVT-based scheduling. The new scheduler has been implemented and exercised in an actual hypervisor system (Xen) that is popularly used in major Cloud platforms worldwide.

Experimental results have been documented from detailed experiments with multiple discrete event models over a range of scenarios (with different lookahead values, inter-processor event exchange frequencies, and conservative and optimistic synchronization), all of which show (a) the high variability and sub-optimality of the default credit-based VM scheduler that is PDES-agnostic, and (b) the well-behaved scalability and significantly faster execution of our new algorithm.

The study and results presented here are among the first to evaluate the characteristics of Cloud and VM-based PDES in detail, and the first to propose a deadlock- and livelock-free PDES-customized hypervisor scheduler.

The results are timely due to the great appeal of commercial Cloud offerings that many find to be very user-friendly and convenient to access and manage. 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. Using CPU-Pools support of Xen to get our PDES specific scheduler into the realms of current Cloud computing infrastructure, and thus make it accessible by more PDES application developers and users. PDES runs may also benefit from the inclusion of a network metric in the specification of the abstract computational unit for VMs, the absence of which leaves the computation highly sensitive to the vagaries of virtual network devices. Cloud-specific synchronization algorithms may also be needed to be resilient to variations in virtual network latencies.

REFERENCES

- CHISNALL, D. 2007. *The definitive guide to the xen hypervisor*. Pearson Education.
- COFFMAN, E. G., ELPHICK, M., AND SHOSHANI, A. 1971. System deadlocks. *ACM Computing Surveys (CSUR)* 3, 2, 67–78.
- D'ANGELO, G. 2011. Parallel and distributed simulation from many cores to the public cloud. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. 14–23.
- FUJIMOTO, R. M. 1990. Performance of time warp under synthetic workloads.
- FUJIMOTO, R. M., MALIK, A. W., AND PARK, A. 2010. Parallel and distributed simulation in the cloud. *SCS M&S Magazine* 3, 1–10.

- JACKSON, K., RAMAKRISHNAN, L., MURIKI, K., CANON, S., CHOLIA, S., SHALF, J., WASSERMAN, H. J., AND WRIGHT, N. 2010. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. 159–168.
- JEFFERSON, D., BECKMAN, B., WIELAND, F., BLUME, L., AND DILORETO, M. 1987. Time warp operating system. *SIGOPS Oper. Syst. Rev.* 21, 5, 77–93.
- MALIK, A., PARK, A., AND FUJIMOTO, R. 2009. Optimistic synchronization of parallel simulations in cloud computing environments. In *Cloud Computing, IEEE International Conference on*. 49–56.
- MALIK, A., PARK, A., AND FUJIMOTO, R. 2010. An optimistic parallel simulation protocol for cloud computing environments. *SCS M&S Magazine* n4 (Oct), 1–9.
- MATTHEWS, J. N., DOW, E. M., DESHANE, T., HU, W., BONGIO, J., WILBUR, P. F., AND JOHNSON, B. 2008. *Running Xen: a hands-on guide to the art of virtualization*. Prentice Hall PTR.
- MELL, P. AND GRANCE, T. 2011. The nist definition of cloud computing (draft). *NIST special publication 800*, 145, 7.
- PARK, A. J. 2009. *Master/worker parallel discrete event simulation*. ProQuest.
- PERUMALLA, K. S. 2005. μ sik-a micro-kernel for parallel/distributed simulation systems. In *Principles of Advanced and Distributed Simulation, 2005. PADS 2005. Workshop on*. IEEE, 59–68.
- PERUMALLA, K. S. AND SEAL, S. K. 2012. Discrete event modeling and massively parallel execution of epidemic outbreak phenomena. *Simulation* 88, 7, 768–783.
- VANMECHELEN, K., DE MUNCK, S., AND BROECKHOVE, J. 2012. Conservative distributed discrete event simulation on amazon ec2. In *IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing. CCGRID '12*. IEEE Computer Society, Washington, DC, USA, 853–860.
- WANG, G. AND NG, T. S. E. 2010. The impact of virtualization on network performance of amazon ec2 data center. In *Proceedings of the 29th Conference on Information Communications*. INFOCOM'10. IEEE Press, Piscataway, NJ, USA, 1163–1171.
- YOGINATH, S. B. AND PERUMALLA, K. S. 2008. Parallel vehicular traffic simulation using reverse computation-based optimistic execution. In *Principles of Advanced and Distributed Simulation, 2008. PADS'08. 22nd Workshop on*. IEEE, 33–42.
- YOGINATH, S. B. AND PERUMALLA, K. S. 2009. Reversible discrete event formulation and optimistic parallel execution of vehicular traffic models. *Intl. Journal of Simulation and Process Modelling* 5, 2, 104–119.
- YOGINATH, S. B. AND PERUMALLA, K. S. 2011. Efficiently scheduling multi-core guest virtual machines on multi-core hosts in network simulation. In *Principles of Advanced and Distributed Simulation (PADS), 2011 IEEE Workshop on*. IEEE, 1–9.
- YOGINATH, S. B. AND PERUMALLA, K. S. 2013a. Empirical evaluation of conservative and optimistic discrete event execution on cloud and vm platforms. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. SIGSIM-PADS '13. ACM, New York, NY, USA, 201–210.
- YOGINATH, S. B. AND PERUMALLA, K. S. 2013b. Optimized hypervisor scheduler for parallel discrete event simulations on virtual machine platforms. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*. SimuTools '13. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 1–9.

Received January 2014; revised TBD 2014; accepted TBD 2015