Efficient execution of the WRF model and other HPC applications in the cloud

Article *in* Earth Science Informatics · March 2016

DOI: 10.1007/s12145-016-0253-7

5 authors, including:

Hector A. Duran-Limon
University of Guadalajara
60 PUBLICATIONS **912** CITATIONS
SEE PROFILE

Jesus Flores-Contreras
University of Guadalajara
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Nikos Parlavantzas
Institut National des Sciences Appliquées de …
62 PUBLICATIONS **1,052** CITATIONS
SEE PROFILE

Efficient Execution of the WRF Model and other HPC Applications in the Cloud

Hector A. Duran-Limon[1], Jesus Flores-Contreras
Information Systems Department
CUCEA, University of Guadalajara
Zapopan, Jalisco, México. Tel. +52 33 3770 3352
hduran@cucea.udg.mx, jesus.flores@cucea.udg.mx

Nikos Parlavantzas
Université Européenne de Bretagne
INSA, INRIA, IRISA, UMR 6074
F-35708 Rennes, France
Nikos.Parlavantzas@irisa.fr

Ming Zhao
School of Computing and Information Sciences
Florida International University
Miami, FL, USA
ming@cs.fiu.edu

Angel Meulenert-Peña
Institute of astronomy and meteorology
CUCEI, University of Guadalajara
Guadalajara, Jalisco, México
ameulene@astro.iam.udg.mx

Abstract— **There are many scientific applications that have high performance computing (HPC) demands. Such demands are traditionally supported by cluster- or Grid-based systems. Cloud computing, which has experienced a tremendous growth, emerged as an approach to provide on-demand access to computing resources. The cloud computing paradigm offers a number of advantages over other distributed platforms. For example, the access to resources is flexible and cost-effective since it is not necessary to invest a large amount of money on a computing infrastructure nor pay salaries for maintenance functions. Therefore, the possibility of using cloud computing for running high performance computing applications is attractive. However, it has been shown elsewhere that current cloud computing platforms are not suitable for running some of these kinds of applications since the performance offered is very poor. The reason is mainly the overhead from virtualisation which is extensively used by most cloud computing platforms as a means to optimise resource usage. Furthermore, running HPC applications in current cloud platforms is a complex task that in many cases requires configuring a cluster of virtual machines (VMs). In this paper, we present a lightweight virtualisation approach for efficiently running the Weather Research and Forecasting (WRF) model (a computing- and communication-intensive application) in a cloud computing environment. Our approach also provides a higher-level programming model that automates the process of configuring a cluster of VMs. We assume such a cloud environment can be shared with other types of HPC applications such as mpiBLAST (an embarrassingly parallel application), and MiniFE (a memory-intensive application). Our experimental results show that lightweight virtualisation imposes about 5% overhead and it substantially outperforms traditional heavyweight virtualisation such as KVM.**

Keywords: WRF model; high performance computing; cloud computing; virtualisation; lightweight virtual machines

I. INTRODUCTION

High performance computing (HPC) applications expand several scientific fields such as meteorology, astronomy, chemistry, and bioinformatics among others. Examples of these applications include weather forecasting, materials science, quantum chemistry calculations, accelerator modelling, and astrophysical simulations (Antypas et al. 2008). These kinds of applications have huge demands of

[1] Corresponding author. Address.- Departamento de Sistemas de Información Centro Universitario de Ciencias Económico Administrativas (CUCEA), Universidad de Guadalajara, Periferico Norte #799,Núcleo Belenes, CP 45100, Zapopan, Jalisco, Mexico.

computing resources, which are traditionally supported by cluster- or Grid-based systems e.g. (Fernández-Quiruelas et al. 2009). Cloud computing (Mell et al. 2011), which has experienced a tremendous growth recently, emerged as an approach to provide on-demand access to computing resources. The cloud computing paradigm offers a number of advantages over other distributed platforms. For example, the access to resources is flexible and cost-effective since it is not necessary to invest a large amount of money on computing infrastructure nor pay salaries for maintenance functions. Therefore, the possibility of using cloud computing for running high performance computing applications is attractive. However, it has been shown elsewhere (Jackson et al. 2010; Wang et al. 2010; Dai et al. 2013; Huang et al. 2006; Martinez et al. 2009; Ekanayake et al. 2009; Evangelinos et al. 2008; Ekanayake et al. 2011; Sun et al. 2011) that current cloud computing platforms are not suitable for running certain kind of HPC applications (e.g. communication- and memory-intensive applications) since the performance offered is very poor. The reason is most cloud computing platforms make extensive use of virtualisation as a means to optimise resource usage. One of the core challenges that faces virtualisation is: it has a performance penalty that is unsuitable for certain kinds of applications (Kroeker 2009). Furthermore, running HPC applications in current cloud platforms is a complex task that requires in many cases configuring a cluster of virtual machines (VMs).

In this paper, we present a lightweight virtualisation framework, called *Distributed Virtual Task Machine (D-VTM)*, which can significantly increase performance while retaining the benefits of cloud computing in terms of ease of management, cost reductions, resource usage optimisation, etc. Our approach also provides a higher-level programming model than current infrastructure as a service (IasS) cloud platforms, since such programming model automates the process of configuring a cluster of VMs. Within the context of this paper there are two kinds of lightweight virtualisation: operating system-level virtualisation (e.g. Linux-VServer (Linux-VServer 2013)) and operating system virtualisation facilities. Such virtualisation facilities can be either part of the kernel (e.g. Cgroups (Cgroup et al. 2013)) or implemented on top of the operating system (e.g. the approach defined in (Duran-Limon et al. 2011a)). In many cases, operating system-level virtualisation requires recompiling the kernel whereas virtualisation facilities commonly do not require such a recompilation. In these kinds of virtualisation the same operating system image is used for all virtual machine (VM) instances.

We rely on our previous work on lightweight virtualisation in middleware (Duran-Limon et al. 2011a), which focused on serial computations (e.g. transcoding a video file to MP4). We have extended this work here whereby lightweight virtualisation is applied to distributed computations (e.g. MPI applications), hence, making it suitable for HPC environments.

In another previous work (Duran-Limon et al. 2011b) we focused on applying the framework to the Weather Research and Forecasting (WRF) model (WRF 2013), a challenging communication-intensive, high performance computing application. We have extended this work in three ways. First, our previous prototype only supported virtualisation of CPU resources. Now, our current prototype supports also virtualisation of memory and network bandwidth resources. Second, we have extended the evaluation to other two types of HPC applications: mpiBLAST (Darling et al. 2003), an embarrassingly parallel application and MiniFE (MiniFE 2013), a memory-intensive application. Third, we have evaluated our approach against KVM; in our previous work we compared our prototype with VMware (VMware 2006). KVM is a competitive hypervisor as demonstrated by some benchmarks (SPECvirt_sc2010 2013; SAP 2013). Furthermore, some research (Younge et al. 2011) suggests that the KVM hypervisor is the optimal choice for executing HPC applications in the cloud. KVM supports I/O para-virtualisation by using virtio (Russell 2008). Also, KVM is part of the mainline Linux kernel since 2.6.20.

Rather than being tied to any particular virtual machine (VM) implementation, our focus is on developing an open-ended framework for efficiently running communication-intensive, HPC applications, such as WRF, in cloud computing environments. We present a particular implementation of the framework on Linux; however, this implementation is only presented as a means to evaluate the framework. Our experimental results show that our lightweight virtualisation approach imposes about 5% overhead, hence, making our approach suitable to run WRF with other HPC applications in a cloud computing environment. The results also show that D-VTM substantially outperforms traditional heavyweight virtualisation such as KVM.

The paper is structured as follows. Section II presents the motivation behind our work. Section III presents some related work. Our virtualisation framework is presented in section IV. The experimental results are shown in section V. Finally, some concluding remarks are given in section VI.

II. APPLICATION AND MOTIVATION

We approach the problem of mitigating natural disasters, which involve hurricanes, forest fires, and air pollution. For example, a number of hurricanes have had a devastating impact in the U. S. as well as in the north and south part of Mexico. Therefore, a better prediction capacity is required to generate better evacuation plans when this type of disasters happens.

The Weather Research and Forecasting (WRF) model (WRF 2013) is gaining acceptance worldwide as one of the best models to carry out weather prediction. H-WRF (HWRF 2015) is an extension of WRF to support the prediction of hurricanes. Importantly, WRF can also be used to predict the evolution of forest fires as well as to determine and predict air pollution. However, the numeric model used by WRF demands a large amount of CPU power. Such demand can be increased dramatically if WRF is used to model a big geographical area with a high-resolution level (for example <1 km). Satisfying the high computational demands of WRF requires putting together large numbers of computing resources through infrastructures such as clusters, Grids, or clouds.

Within the DiViNE (Disaster mItigation on VIrtualised eNvironmEnts) project (DiViNE 2015) we have developed a WRF Web portal able to carry out on demand runs of WRF (i.e. interactive runs of WRF) in cloud computing environments. A screenshot of our WRF Web Portal is shown in Figure 1. Before our WRF Web Portal system[2] was developed, Mexico did not possess a user-friendly system whereby Mexican meteorologists had open and remote access to perform on demand runs of WRF. In the best case, some meteorologists had access to a local computer infrastructure where on demand runs can only be carried out with the assistance of highly trained computer engineers. Some other meteorologists did not even have access to enough computing power to run WRF in a reasonable amount of time. Furthermore, to the best of our knowledge there is not a system, worldwide, able to offer on demand runs of WRF in cloud environments using lightweight virtualization.

As said earlier, HPC applications are traditionally run in cluster and Grid infrastructures. One of the main disadvantages of such infrastructures is that a job queue is normally involved. As a result, the user sometimes has to wait for long periods for his applications to be launched. In contrast, in a cloud environment the computing resources are always available, hence, the user does not experience such long waiting periods. As it is normal in a cloud environment, we assume that several applications (in our case HPC applications) can run at the same time.

One of the main research challenges we are addressing involves using lightweight virtualisation as a means to make cloud computing environments suitable for efficiently running WRF altogether with other HPC applications.

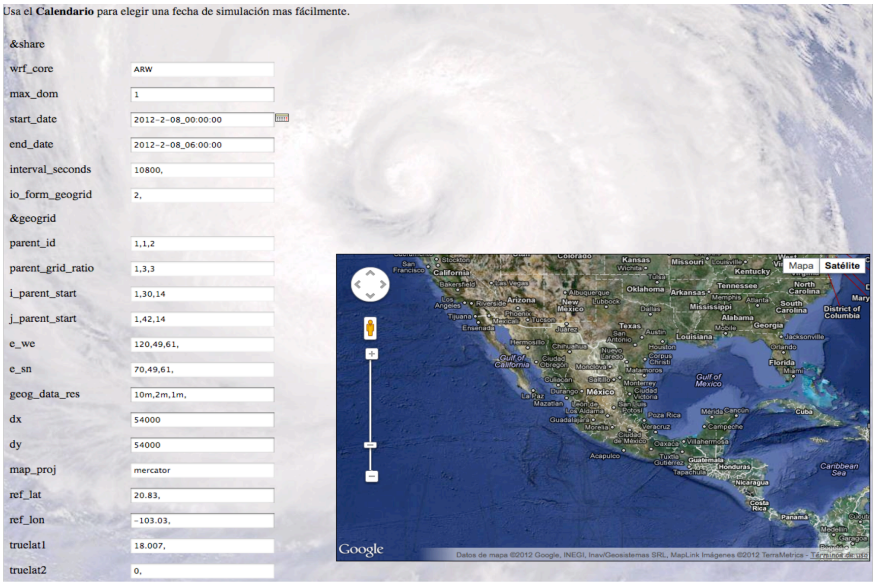

Fig 1. A screenshot of The WRF Web Portal

III. Related Work

Much research work has focused on evaluating the suitability of virtualised environments for executing HPC applications. Jackson et al. (Jackson et al. 2010) and Ekanayake et al. (Ekanayake et al. 2009) presented comprehensive studies of the performance of parallel applications on the Amazon EC2 cloud platform and on a Eucalyptus cloud system, respectively, using a set of typical HPC applications. Younge et al. (Younge et al. 2011) examined popular virtualisation technologies, focusing on their applicability to HPC applications. Zhai et al. (Zhai et al. 2011) compared the performance and cost-

[2] Currently, access to our WRF Web Portal is only provided to Mexican meteorologists that request an account.

effectiveness of running tightly-coupled parallel applications on Amazon Cluster Compute Instances and on an in-house cluster. Related cloud performance studies also considered several specific HPC applications including climate modelling (Evangelinos et al. 2008), bioinformatics (Ekanayake et al. 2011), lattice optimisation (Sun et al. 2011), and astronomy workflows (Hoffa et al. 2008). These results all indicate that cloud execution incurs a significant performance overhead, particularly high for communication-intensive applications.

Wang et al. (Wang et al. 2010) concentrated on the networking performance of Amazon EC2 and reported that the underlying Xen-based virtualisation approach causes significant throughput instability and delay variations. Exposito et al. (Expósito et al. 2013a) evaluated the networking performance of Amazon EC2 Cluster Compute instances and assessed their impact on the scalability of communication-intensive applications. Mauch et al. (Mauch et al. 2013) presented an approach relying on the use of InfiniBand (InfiniBand 2007) in a high performance cloud computing environment. The authors reported the communication latency of the virtual environment using InfiniBand was near to the latency obtained by the native operating system using InfiniBand in a cluster system. Such a good performance is due the fact that the communication over InfiniBand bypasses both the virtualisation layer and the operating system, thus, having direct access to the host channel adapter. However, the InfiniBand technology is not directly supported by current hypervisors. Exposito et al. (Expósito et al. 2013b) evaluated the I/O storage subsystem on the Amazon EC2 Cluster Compute platform and showed significant performance differences among the available cloud storage devices. An experimental study of running WRF on large-scale virtualised environments is presented in (Martinez et al. 2009). While the overhead of using VMs instead of physical machines is relatively low for single-cluster environments, the overhead becomes prohibitive when using multiple clusters.

Considerable effort has been made to address the performance limitations of virtualised environments. Performance studies using general-purpose benchmarks show that traditional, heavyweight virtualisation approaches, such as Xen (Barham et al. 2003) and VMware (VMware 2006), incur a performance penalty from 10-20% compared to native OS environments (XenSource 2007; VMware 2007). The performance penalty is higher when there is a large amount of I/O operations. Techniques that have been proposed to improve the performance of I/O virtualisation include VMM-bypass I/O (Huang et al. 2006) and self-virtualised devices (Raj et al. 2007).

Operating system-level VMs have also been proposed as a way to reduce the virtualisation overhead. Xavier et al. (Xavier et al. 2013) evaluated the performance and isolation of three operating system-level virtualization systems (Linux-VServer (Linux-VServer 2013), OpenVZ (OpenVZ 2013) and LXC (LXC 2013)), and compared them with Xen. The results show that these virtualization systems achieve near-native performance but poor isolation for resources other than CPU. Docker (Docker 2015) is a platform that allows portable deployment of LXC VMs (as well as *libvirt* VMs, a Docker's technology for operating system-level virtualisation) across different Linux servers. Nova (Nova 2015) is a tool of OpenStack (a cloud platform) (OpenStack 2015), which allows the user to manage both traditional heavy-weight VMs and lightweight VMs (the latter being managed with the aid of Docker). Nova's managing functions include instantiating and running VMs. Our approach is complementary to Nova since the D-VTM framework focus on offering a higher-level programming model whereby it is automated the process of configuring and instantiating a cluster of lightweight VMs for running a distributed computation. In fact, our approach can be implemented on top of Nova and Docker.

Lange et al. (Lange et al. 2011) proposed a low overhead virtualisation solution for supercomputing by embedding the virtual machine monitor inside of a lightweight kernel. However, this solution relies on the existence of the lightweight kernel which supports only limited applications tailored for supercomputing. In contrast, our lightweight virtual machine framework does not impose such kind of assumptions. Dai et al. (Dai et al. 2013) presented a lightweight VM monitor, named OSV, which achieves similar performance to native Linux. Unlike standard VM monitors, OSV statically allocates processor cores and memory to VMs and cannot support dynamic resource reallocation to deal with changing application demands.

IV. The Lightweight Virtual Machine Framework for HPC Environments

The D-VTM framework is based on using lightweight virtualisation, which involves either operating system virtualisation facilities or operating system-level virtualisation. As mentioned before, the D-VTM framework relies on the VTM framework (Duran-Limon et al. 2011a), our earlier work on lightweight virtualisation. Briefly, the VTM framework includes three main elements, namely *resource factories*, *resource managers,* and *abstract resources*. Factories are in charge of creating resources

whereas managers are responsible for managing them. Abstract resources explicitly represent system resources. There may be various levels of abstraction in which higher-level resources are constructed on top of lower-level resources. At the lowest-level are the physical resources such as CPU, memory, and network resources. Higher abstraction levels then include the representation of more abstract resources such as virtual memory, team of threads, network connections and more generic type of resources. Top-level abstractions are *virtual task machines (VTMs)*, which encompass lower level resources (e.g. processes and memory) and provide the execution environment of serial computations (e.g. transcoding a video file to MP4).

In this paper, we have extended the concept of VTMs to composite and distributed VTMs. A *D-VTM* provides the execution environment of a distributed computation (e.g. a WRF run) involving multiple processes running in parallel on different nodes. In addition, the framework was also extended with a number of VM management functions, as shown in Figure 2. The *Job Scheduler* is in charge of receiving job requests accompanied with the desired number of VM instances. The *VTM Scheduler* keeps track of resource usage and determines if the required resources are available. Also, in case of resource contention, the amount of resources given to the D-VTMs can be reallocated by the VTM Scheduler. When the requested resources are available, a resource reservation is performed and the Job Scheduler asks the *VTM Factory* to create instances of D-VTMs. Hence, the life-cycle of D-VTMs (i.e. creation and deletion of D-VTMs) is controlled by the VTM Factory. Finally, the *Job Monitor* is in charge of monitoring task execution on the D-VTMs. In case a task fails (e.g. the task halts or has an abnormal exit), the Job Monitor informs the Job Scheduler about this situation, which in turns asks the VTM Scheduler to stop the execution of the D-VTM associated with the failing task. Afterwards, the VTM Scheduler releases the resources of the D-VTM involved. A notification is sent to the user of such a failure. The Job Monitor also keeps track of resource usage for billing purposes. Further details of each of these modules are presented below.

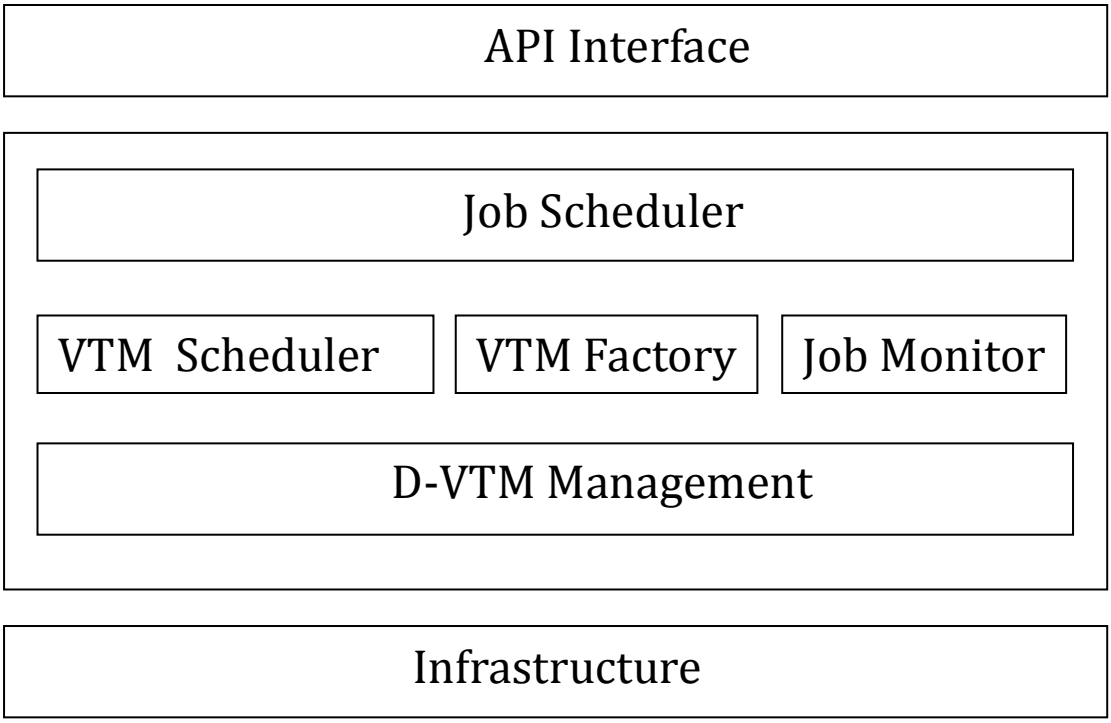

Fig 2. Overall architecture of the D-VTM Framework

A. Distributed VTMs

A D-VTM represents a VM encapsulating a pool of resources allocated for the execution of its associated tasks. The UML model in Figure 3 illustrates details concerning the relationships between tasks and D-VTMs. C*omposite tasks* are constituted by a number of independent sub-tasks, which can be executed in parallel. Sub-tasks are not further partitioned and are called *primitive tasks*. In addition, primitive tasks are only related to one address space. However, *distributed tasks* involve two or more address spaces. It should be noted that a composite task is not necessarily distributed.

Similarly, VTMs not containing other VTMs as lower-level resources are named *primitive VTMs*. There is a one-to-one relationship between primitive VTMs and primitive tasks. A composite VTM may encompass various local primitive VTMs. In this case, the primitive VTMs represent the virtual cores of

a VM (i.e. a composite VTM). Importantly, a distributed VTM is a specialisation of composite VTMs whereby two or more address space are involved. Hence, a distributed VTM includes various local and remote primitive VTMs. Finally, the VTM scheduler and VTM factory are defined on a per-address space basis.

The operation for running a job is the following:

```
run(program_name, dir_input_data, input_files, dir_output_data, output_files)
```

This operation starts the execution of the given program, which involves multiple parallel processes. The process IDs (pids) of such processes are evenly attached to the VMs contained within the D-VTM.

We used Control Groups (Cgroups) (Cgroup et al. 2013) to implement D-VTMs. Cgroups is a generic process-grouping framework, which is part of the Linux kernel since version 2.6.24. The amount of resources used by processes can be controlled per group of processes. A portion of the resources can be allocated to each group; for instance, a portion of the CPU can be assigned to each group. The processes belonging to a group share the group's resources and their resource usage is limited to the resources owned by the group. In our prototype, a VM instance (i.e. a primitive VTM or composite VTM) has allocated 50% of a CPU core. Nevertheless, a different share can be configured. Hence, two VMs are allocated per physical core. Also, primitive VTMs run a single process whereas composite VTMs run one process per virtual core. Although, for our prototype we employed the existing lightweight virtualisation facilities provided by the operating system (i.e. Cgroups), our approach can also be implemented on top of operating system-level virtualisation.

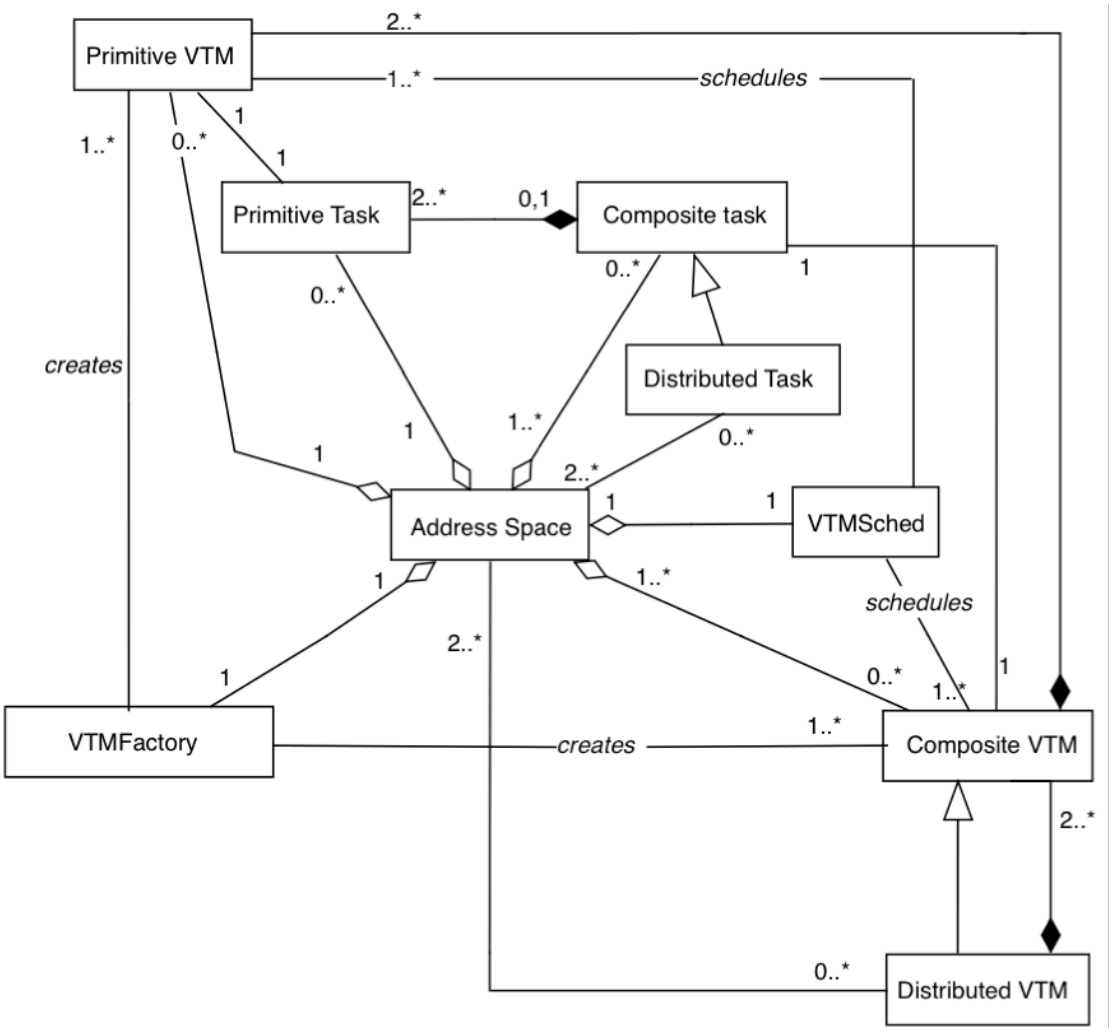

Fig 3. UML Diagram of Relationships between VTMs and Tasks

Virtual cores are implemented as sub-groups, i.e. groups belonging to a group. In this case, the resources used by the sub-groups are constrained by the amount of resources allocated to the group they belong to. Therefore, a D-VTM involves multiple VMs running across different nodes. Each VM may be either uni-core or multi-core. Our prototype was implemented in C++ and it currently has control

over CPU, memory, and network bandwidth resources. Memory allocations are also controlled by Cgroups making it possible to isolate the use of an amount of memory, which is exclusively shared by the processes that are part of a VM. In the case of network resources, Cgroups mark the packets that are generated within a VM. In conjunction with Cgroups, TC (Hubert et al. 2013) is used to control the traffic. This is implemented by using Diffserv based on a hierarchical token bucket (HTB) queuing discipline (Devera 2013). This scheduling algorithm is used to limit the rate of sent network traffic. Such an algorithm is based on class services whereby traffic can be classified and different assured rates can be associated with each class.

B. The Job Scheduler

Job requests are attended by the Job Scheduler. Clients are able to send requests by invoking the `schedule()` operation. A number of parameters are involved in a request, which mainly specify the virtual execution environment and the data related to the program to be executed. The operation for scheduling a job is the following:

```
schedule(no_vms, no_vcores, memory, bandwidth, program_name, dir_input_data,
input_files, dir_output_data, output_files, job_id)
```

The parameters that specify the virtual execution environment (in terms of resource requirements) are the former four and define the desired number of VMs, the number of virtual cores per VM, the amount of memory per VM, and the amount of network bandwidth per VM, respectively. We assume the user has previously performed a benchmarking of the HPC application. As a result, the user knows the amount of resources required for running the HPC application within an acceptable window-time. We also assume that such an "acceptable window-time" is based on subjective criteria of the user.

The HPC program related data involves the program name, the directory where the input data is placed and the input file names. In addition, the directory of the output data and output file names are provided. Upon successfully execution of the `schedule()` operation, the job ID is obtained as an output parameter.

When a job request is received, the Job Scheduler executes the script shown below. First, the Job Scheduler asks the VTM Scheduler to perform an admission test, whereby it is determined if the requested resources are available. In case the admission test is successful (line 1), the VTM Scheduler performs a resource reservation (line 2) and the scheduling parameters are obtained, which involve the specific physical cores that will be used as well as the CPU share given to each VM (line 3). This schedule information is used by the VTM Factory to create the requested VMs (line 4). As a result, a job id and a D-VTM are obtained. Afterwards, the D-VTM performs the `run()` operation, which is in charge of starting the execution of the requested program within the created VM environment (line 5).

```
1. if ( vtmSched.admit(no_vms, memory, bandwidth) )
2.     vtmSched.reserve(no_vms, memory, bandwidth)
3.     vtmSched.schedule(no_vms, no_vcores, memory, bandwidth, schedParam)
4.     vtmFactory.newResource(schedParam, job_id, d_vtm)
5.     error = d_vtm.run(program_name, dir_input_data, input_files,
dir_output_data, output_files)
6.     if error
7.           return error
8. else
9.     return error
```

C. The VTM Scheduler

The `admit()` operation examines whether there are enough physical resources to satisfy the demands of the number of VMs requested. The operation returns true in a successful case, otherwise returns false. Resource reservation is achieved by the operation `reserve()`, which updates a register of both allocated and unused resources. Conversely, once the execution of a job has finished, the resources (i.e. the VMs) used can be released by the `release()` operation. This operation updates the register of unused resources. The current implementation keeps track of physical cores that are available as well as the information of which physical cores are using each running job.

The `schedule()` operation receives as input the number of VMs that will be created, the number of virtual cores for each VM, as well as the requested memory and bandwidth for each VM. This operation produces as output the scheduling parameters (placed in `schedParam`) that will be used to run the VMs. The `schedParam` argument defines the specific physical cores that will be used as well as the assigned CPU, memory, and network share for the VMs. This argument involves a two-dimensional

array of integers m_{ij} in which *i+1* is the number of VMs that will be created. m_{j0} represents the core id where the VM will be allocated. We defined a *physical core registry*, which maps a core id to a node, a processor, and a physical core. m_{j1} involves the CPU share value for each VM, m_{j2} represents the number of virtual cores that the associated VM will include. The number of virtual cores can be either 0 or a multiple of 2. m_{j3} defines the amount of memory allocated for each VM, and m_{j4} refers to the allocated amount of output network bandwidth for each VM. Since defining a load balancing policy is not the main focus of this work, we are currently using a simple load balancing approach. The execution load of a D-VTM is evenly allocated to free resources as follows. Each VM belonging to a specific D-VTM is allocated to a single physical core owned by a different processor. In case there are not enough free resources to apply this policy, a D-VTM may have two or more VMs running in a same processor.

Jobs can be finished by the operation `finish()`. This operation can be used, for example, when a job has overcome the maximum allowed execution time. The implementation of this operation basically kills all the processes belonging to the job and invokes the `release()` operation to release the resources used by the job.

D. The VTM Factory

The VTM Factory is responsible for creating D-VTMs. The following operation is used for this purpose.

```
newResource(schedParam, job_id, d_vtm)
```

The VTM Factory receives `schedParam` as an input parameter. As a result of the operation invocation, a job id and a D-VTM are obtained. The VTM Factory was implemented as follows. Based on the information provided by the `schedParam` argument, the VTM Factory first creates the needed Cgroups (each one representing a single VM) assigning the requested amount of CPU, memory, and network bandwidth. Such Cgroups are bound to the specified physical core[3]. Following, the processes of the requested HPC application are launched on the corresponding physical cores according to the specification of the `schedParam` argument. Finally, the processes are bound to their associated Cgroups. In case, zero virtual cores are defined in a VM, a single process is bound to a Cgroup[4].

Our approach provides a higher-level programming model rather than a software library to automate the process of configuring a cluster of VMs. Such a process is achieved as follows. In the case of operating system virtualisation facilities (e.g. Cgroups), the same operating system image is used for all VMs; therefore, no further configuration is required to: a) allow the communication among the VMs, b) define how files are shared among the VMs. The reason is, once the physical cluster is configured to allow communication and file sharing, all VM instances will share the same configuration. In the case of operating system-level virtualisation (e.g. LXC) in which a different file system can be embedded in a VM, the configuration process of a cluster of VMs is not as simple as in the former case. Here, our framework delegates the implementation of such a configuration process to the programmer of the D-VTM API. However, the implementation details of such an API must be transparent to the cloud application programmer. As said earlier, our approach is an open-ended framework, which defines a higher-level programming model that is not tight to any particular implementation.

E. The Job Monitor

The Job Monitor is in charge of monitoring the execution of jobs. In case a job finishes its execution, the monitor detects and informs of this situation to the Job Scheduler, which in turns asks the VTM Scheduler to release the resources allocated to the job. The Monitor also detects when a job has overcome the maximum execution time allowed and informs the Job Scheduler of such a situation. In this case, the Job Scheduler asks the VTM Scheduler to stop the execution of the job and release the resources associated with the job. Other operations supported by the Job Monitor are the following.

```
getStatus(job_id, status)
getExecutedTime(job_id, exec_time)
```

---

[3] The core id m_{j0} is looked up in the physical core registry to obtain the node, processor, and core number.

[4] This situation does not hold in some of our experimental scenarios in which some Cgroups have bound more than two processes in order to perform CPU stress tests. Also, there are certain HPC applications that require running more than one process per VM. For instance, mpiBLAST requires at least 3 processes per VM in our experimental setup.

The former reports the status of a job, which can be running, suspended, or finished. The latter retrieves the execution time that a job has consumed.

V. EVALUATION

We present a performance evaluation of the C++ prototype and compare it with KVM. We use virtio-compatible guests (Russell 2008; Motika et al. 2012). The proposed framework is evaluated with different kinds of applications: CPU-intensive, embarrassingly parallel, memory-intensive, and communication-intensive applications. Our experiments were run in a private cloud setup based on the *Nadimit* cluster located at CUCEA, University of Guadalajara. The cluster contains 64 cores, which involves 8 nodes where each node includes two quad-core Xeon 5500 2.0 GH and 16 GB of main memory. The cluster runs Linux Centos 6.2, 64 bit, kernel 2.6.32-220 and uses Intel MPI 3.2.2, which is based on MPICH2.

The experimental scenarios involve the execution of three different types of applications: WRF, mpiBLAST, and MiniFE. The Weather Research and Forecasting (WRF) model (WRF 2013) is a challenging CPU- and communication-intensive application. WRF is gaining acceptance worldwide as one of the best models to carry out weather prediction. However, the numeric model used by WRF demands a large amount of CPU power. Such demand can be increased dramatically if WRF is used to model a big geographical area with a high-resolution level (for example <1 km). Satisfying the high computational demands of WRF requires putting together large numbers of computing resources through infrastructures such as clusters, grids, or clouds. The experimental scenarios involving WRF had an independent run of WRF 3.3.1. In the case of Job 1, we used a 7.5 Km by 7.5 Km domain decomposition with 20 km resolution, which contains 120 x 70 grid points; and a simulation time of 24 hours. A larger simulation time was used in the rest of the jobs to make sure Job 1 finishes its execution before the other jobs.

mpiBLAST (Darling et al. 2003) is an embarrassingly parallel application. mpiBLAST is a distributed implementation of the Basic Local Alignment Search Tool (BLAST) (Altschul et al. 1990). BLAST is an algorithm that finds regions of similarity between biological sequences by calculating the statistical significance of the matches. That is, a BLAST search compares a query sequence with a database of sequences, and identifies the sequences in the database that better match (according to a threshold criterion) the query sentence. Both the query sequences and the database were obtained from (NCBI 2013). As the query sequences we used the file "alu.n" of 96 KB. In the case of the database we used a 1.7 GB file named "est_mouse". mpiBLAST employs *n-2* worker processes since two control processes are needed. One process is used as the "master" scheduler, and another process is used to handling the results. For each run, the database was fragmented into *n-2* pieces, where *n* is the number of processes run. A single VM always runs three processes, hence, the first created VM involves two control processes and one worker process whereas the rest of the VMs include three worker processes. In the experimental Series 2 (see below), each VM has allocated locally (in its virtualised disk space) the fragments needed i.e. three fragments. In Series 3 (see below), the fragments are not local to the VMs; instead they are located in a file system shared via NFS.

MiniFe (mini finite element) (MiniFE 2013) is an MPI implementation of a mini-application designed to mimic the core functionality of an implicit finite element method application in 1,500 lines of C++. The computation of MiniFe is dominated by sparse matrix-vector multiplications using a conjugate gradient iterative solver with no preconditioning, whose performance is commonly known to be greatly affected by the availability of memory bandwidth.

All runs of Job 1 were performed three times and the average of these runs was considered. We found a relative standard deviation of about 5%. We are using KVM 2.6.32-220. KVM VMs are configured with one CPU, 1 GB RAM (except in Series 4 in which VMs have 8GB), and 8 GB of disk. These VMs run Linux Centos 6.2 with kernel 2.6.32.

In the context of this work the term 'overhead' is used to mean the ratio whereby a virtualisation approach is slower than the native approach. The overhead is calculated as follows: $overhead = ((100*e_i)/e_j) - 100$, where e_i is the execution time of a virtualisation approach and e_j is the native execution time. We carried out 5 series of experiments each one including one or more experimental scenarios. The following experimental scenarios were defined:

Scenario 1.- The first experimental scenario starts running 5 jobs namely, Job 2 to Job 6, in which each job runs 6 lightweight VMs. The execution time of Job 1 is measured for a different number of VMs. In this scenario, all lightweight VMs and KVM VMs have a 50% share of a physical core and run as few processes as necessary depending on the application. WRF and MiniFE can work with one process whereas mpiBLAST requires 3 processes at minimum. In addition, each of the VMs belonging to the same job runs on a different processor.

Scenario 2.- The second experimental scenario is similar to the first scenario. The difference is that the measurements are taken on Job 1 while no other jobs are executing. In the case of KVM, VMs are given 100% of the physical core whereas lightweight VMs receive only 50% of the physical core.

Scenario 3.- The third scenario is also similar to the first scenario. The difference is that the VMs of Job 2 to Job 6 have four processes per VM as a means to stress the use of the resources, while Job 1 keeps running a single process per VM.

In both Scenario 1 and Scenario 3, one out of four cores per processor remains idle. An active core runs two VMs, each VM belonging to a different Job. The execution load of VMs is evenly allocated to the active cores as follows: Core 0 runs Job 1 and Job 2, Core 1 runs Job 3 and Job 4, and Core 2 runs Job 5 and Job 6.

Series 1 and Series 5 involve the three experimental scenarios whereas Series 2 and Series 3 include only Scenarios 1 and 3, respectively. Finally, Series 4 encompasses Scenario 4, which is described below. Table 1 describes these series.

Series	Resource Isolated	Type of App	Job 1	Competing Jobs
Series 1	CPU	CPU-intensive	WRF	WRF
Series 2	CPU	Embarrassingly parallel	mpiBLAST	WRF
Series 3	CPU, Memory	Embarrassingly parallel	mpiBLAST	MiniFE
Series 4	CPU, Memory	Memory-intensive	MiniFe	MiniFe
Series 5	CPU, Network Bandwidth	Communication-intensive	WRF	WRF

Table 1. Series of Experiments

In such series, lightweight VMs provide resource isolation of CPU, memory, and network bandwidth whereas heavyweight VMs only offer resource isolation of the former two. In all series the execution of Job 1 is measured whereas Job 2 to Job 6 are competing jobs. In Series 1 and Series 5 the measured job is WRF, a CPU- and communication-intensive application. mpiBLAST, which is an embarrassingly parallel application, is used in Series 2 and Series 3. Finally, in Series 4, we test our framework with MiniFe, a memory-intensive application.

A. Experimental Results of Series 1 - CPU Isolation on a CPU-intensive Application

This series includes Scenario 1, Scenario 2, and Scenario 3 in which the performance of a CPU-intensive application is evaluated when CPU isolation is carried out. In addition, each of the VMs belonging to the same job runs on a different processor in the case of Scenario 1 and Scenario 3. All jobs involve the execution of WRF. Series 1 includes the following three experimental scenarios. The first experimental scenario starts running 5 jobs namely, Job 2 to Job 6, in which each job runs 6 lightweight VMs. In this scenario, all lightweight VMs and KVM VMs run a single process and have a 50% share of a physical core. The execution time of Job 1 is measured for a different number of VMs, as shown in Figure 4. The same scenario is repeated for the case of using KVM VMs. Finally, this scenario is repeated on the native Linux, i.e. without the use of any virtualisation. In this case, Job 2 to Job 6 run 6 processes, each one on a different processor. Afterwards, Job 1 increases the number of the processes it runs. This is equivalent to running the VMs in terms of the number of processes run.

The lightweight VM approach obtains a much better performance than KVM VMs, as shown in Figure 4. Only about 5% overhead is imposed by the former whereas the latter has an overhead that ranges from 21% to 67%.

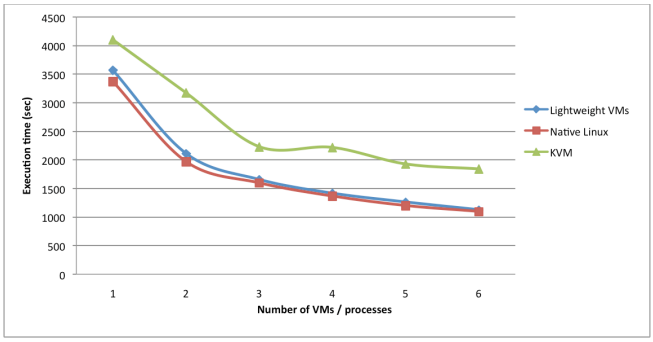

Fig 4. Series 1 – Scenario 1 (the execution time of Job 1 while Job 2 to Job 6 are executing).

The second experimental scenario is similar to the first scenario. The difference is that the measurements are taken on Job 1 while no other jobs are executing. Also, in the case of KVM, VMs are given 100% of the physical core whereas lightweight VMs receive only 50% of the physical core. In this case, a single core is shared by two VMs of Job 1.

In Figure 5 we can observe that despite of the fact that the lightweight VMs are in this disadvantageous condition, they clearly outperform the KVM instances.

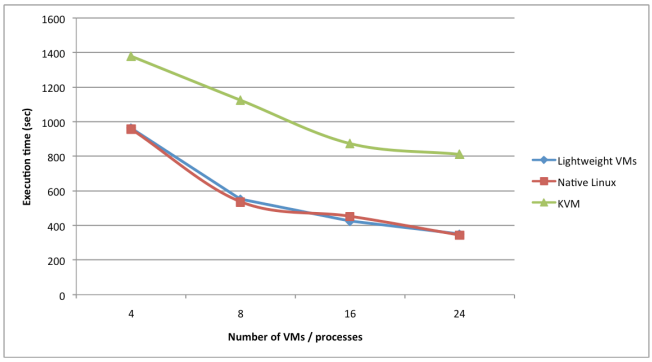

Fig 5. Series 1 – Scenario 2 (the execution time of Job 1 while no other jobs are executing).

Finally, the third scenario is also similar to the first scenario. The difference is that the VMs of Job 2 to Job 6 have 4 processes as a means to stress the use of CPU, while Job 1 keeps running a single process. The performance of KVM VMs is clearly degraded as the number of VMs increases, as shown in Figure 6.

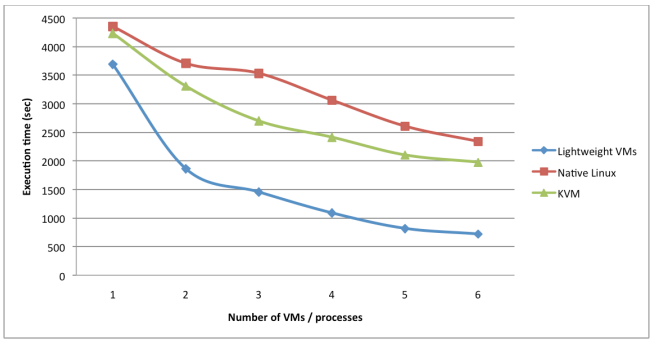

Fig 6. Series 1 – Scenario 3 (the execution of Job 1 while the other jobs stress the CPU).

It should be noted that the lightweight VMs obtain a better performance than the native Linux. The reason is the latter does not have any control on the amount of CPU resources given to Job 1 while the CPU is stressed by other processes. In contrast, the lightweight VMs isolate the amount of resources used by Job 1. Furthermore, the lightweight approach maintains a fairly similar level of resource isolation even in overload conditions. This assertion is corroborated in Figure 7 where scenario 1 and scenario 3 are compared in the case of lightweight VMs. KVM instances present a similar behaviour as shown in Figure 8.

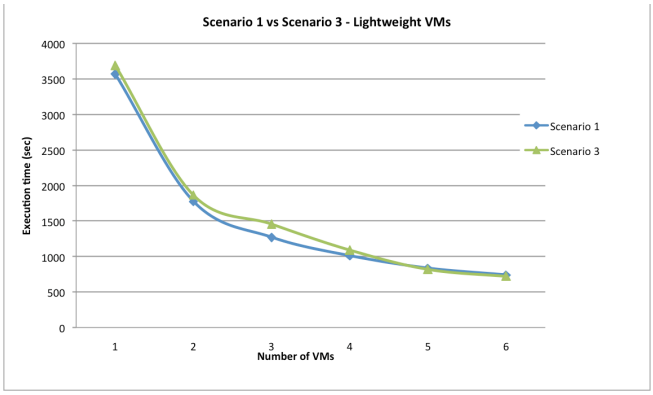

Fig 7. Comparing the performance of lightweight VMs in scenario 1 and scenario 3

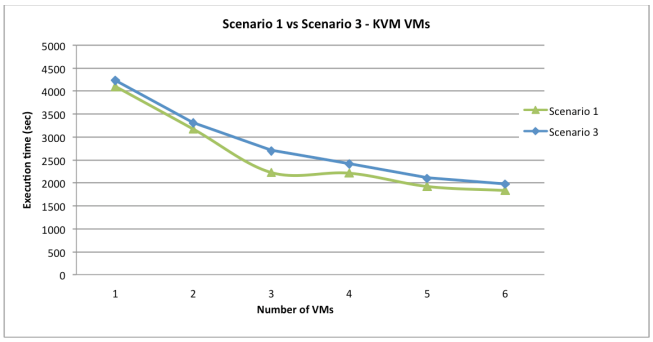

Fig 8. Comparing the performance of KVM VMs in scenario 1 and scenario 3

B. Experimental Results of Series 2 - CPU Isolation on an Embarrassingly parallel Application

This series includes Scenario 1 in which the performance of an embarrassingly parallel application is evaluated when CPU isolation is carried out. Job 1 involves the execution of mpiBlast whereas Job 2 to Job 6 regard the execution of WRF. In this case each VM of Job 1 involves 3 processes whereas Job 2 to Job 6 have only one process per VM. Therefore each core runs 4 processes (3 processes running mpiBlast and 1 process running WRF). In this particular case we have customised Scenario 1 so that Job 1's VMs are given 75% of the CPU as a means to give 25% of CPU to each process. This way the performance comparison of the VMs with native Linux is fairer.

The execution time of Job 1 is measured for a different number of VMs, as shown in Figure 9. The same scenario is repeated for the case of using KVM VMs and native Linux. We can observe, in Figure 9, that both lightweight VMs and KVM exhibit a better performance in the case of the first run which involves running a single VM. This is due to the fact that the first VM runs two control processes and a single worker process whereas the rest of the VMs run three worker processes. Thus, for the first run, in the case of native Linux, the WRF process takes more than 25% of the CPU as the control processes are not CPU intensive. In contrast, in the virtualised cases the portion given to WRF does not goes higher than 25%.

Even though embarrassingly parallel applications running on virtualised environments have a similar performance when running in the native operating system, we can observe in Figure 9 that KVM instances have some performance overhead that goes up to 76%. The reason is, Job 2 to Job 6, which are running WRF, have a negative effect on the performance of mpiBLAST.

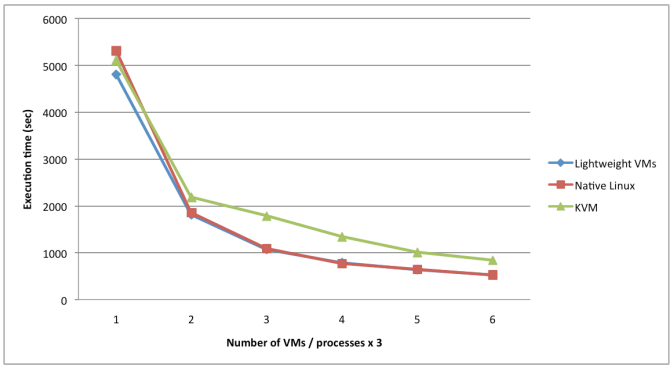

Fig 9. Series 2 – Scenario 1 (competing jobs). CPU Isolation on an Embarrassingly Parallel Application.

C. *Experimental Results of Series 3 – CPU and Memory Isolation on an Embarrassingly Parallel Application*

This series includes Scenario 3 in which the performance of an embarrassingly parallel application is evaluated when CPU and memory isolation is performed. The competing jobs run MiniFE receiving as input the vector *x=300, y=300, z=400*. Job 1 involves the execution of mpiBlast whereas Job 2 to Job 6 regard the execution of MiniFE. Each VM of Job 1 involves 3 processes whereas Job 2 to Job 6 have 4 process per VM. The execution time of Job 1 is measured for a different number of VMs, as shown in Figure 10. The same scenario is repeated for the case of using KVM VMs and native Linux.

We can observe in Figure 10 that lightweight VMs isolating CPU and memory resources obtain the best performance. Even KVM achieve some improvement over native Linux and lightweight VMs not performing memory isolation. The reason is that the execution of WRF in Job 1 is negatively affected by the exhaustive memory consumption carried out by MiniFE. Therefore, carrying out memory isolation helps to improve the performance.

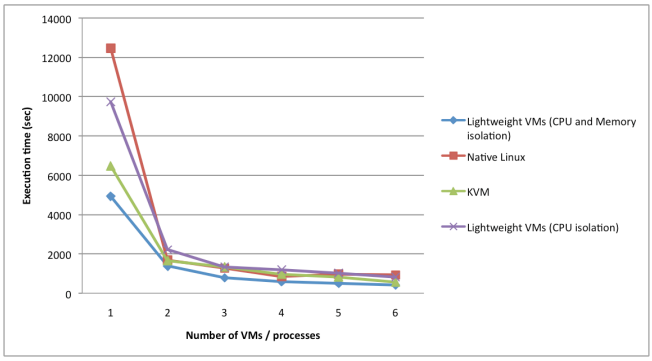

Fig 10. Series 3 – Scenario 1 (competing jobs). Memory Isolation on an Embarrassingly Parallel Application.

D. *Experimental Results of Series 4 – CPU and Memory Isolation on a Memory-intensive Application*

This series only involves Scenario 4 in which the performance of a memory-intensive application is evaluated when CPU and memory isolation is carried out. In this scenario we have only Job 1 and Job 2. The former is the measured job whereas the latter is the competing job. Both jobs run MiniFE. Job 1 has the following input vector: *x=200, y=200, z=300* whereas the input vector for Job 2 is *x=300, y=300, z=500*. The VMs have 100% of the CPU core and 8 GB of RAM. Each VM runs only one process. Also, every VM is executed on a single core of one processor whereby the rest three cores remain idle. Hence, a node runs one VM of Job 1 and one VM of Job 2, each one on a different processor of the node. Each subsequent run involves adding a new node running one VM more for Job 1 and another VM more for Job 2, each one on a different processor of the node. The execution time of Job 1 is measured for a different number of VMs, as shown in Figure 11. The same scenario is repeated for the case of using KVM VMs and native Linux.

We can observe in Figure 11 that the performance of the VMs is far better than that obtained by native Linux. This is due to the fact that Job 1 is greatly affected by the consumption of memory of the

competing job. Also, native Linux provide no guarantees to avoid the execution of Job 1 accessing part of the neighbouring memory, in which case an extra overhead is imposed. Importantly, lightweight VMs have an improvement in performance compared to KVM VMs that goes from 25% to 42%. We can see that as more nodes are included in the runs the performance of native Linux considerably improves. The reason is adding more nodes provides more memory to the execution of Job 1. Job 1 obtains 8 GB more of memory for each node that is included.

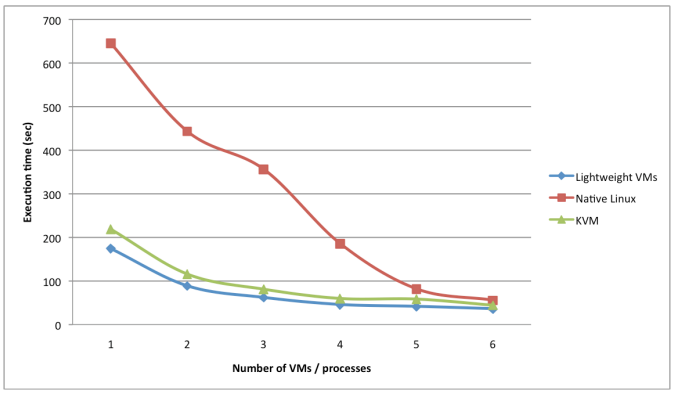

Fig 11. Series 4 – Scenario 4 (nodes added as new VMs are instantiated). Memory Isolation on a Memory-intensive Application.

E. Experimental Results of Series 5 – CPU and Network Bandwidth Isolation on a Communication-intensive Application

This series includes Scenario 1, Scenario 2 and Scenario 3 in which the performance of a communication-intensive application is evaluated when network bandwidth and CPU isolation is carried out. The cluster nodes are interconnected by a 1 Gbit/s switch. A job has two VMs in a node, one VM per processor. Each VM of Job 1 has assigned a class with 350 Mbit/s. The remaining bandwidth is allocated as follows. VMs of Job 2 to Job 6 have associated a class with 30 Mbit/s each. The traffic generator JTG (JTG 2013) is used to overload the network by injecting 1 Gbit/s of traffic. The traffic is generated from other nodes than the ones that are running the VMs. Node 1 sends traffic to node 5, node 2 sends traffic to node 6, and node 3 sends traffic to node 7.

In Scenario 1, Job 1 to Job 6 regards the execution of WRF. The execution time of Job 1 is measured for a different number of VMs, as shown in Figure 12. The same scenario is repeated for the case of using KVM VMs and native Linux.

Figure 12 shows that lightweight VMs clearly outperform KVM instances. We can observe that not only CPU isolation brings benefits but also network isolation. This assertion is corroborated by the fact that lightweight VMs with CPU and network isolation outperform both native Linux and lightweight VMs with only CPU isolation. Lightweight VMs improve the performance of native Linux from 3% to 20%.

In the second experimental scenario there are no other jobs executing apart from Job 1. KVM VMs are given 100% of the physical core whereas lightweight VMs receive only 50% of the physical core. Figure 13 shows that lightweight VMs perform better than KVM's VMs, even though the former are in a disadvantageous position at having 50% of CPU in contrast with the 100% of CPU obtained by KVM instances.

Finally, in the third scenario VMs of Job 2 to Job 6 have 4 processes as a means to stress the use of CPU, while Job 1 keeps running a single process. In Figure 14 we can observe that lightweight VMs clearly outperform both native Linux and KVM instances. Lightweight VMs obtain 50% to 66% gain in performance compared to native Linux. KVM has the worst performance of the three and some fluctuations can be observed in its performance. This is due to the fact that KVM VMs were severely affected by the traffic generated by both the process stressing the CPU and the traffic injector. KVM suffers from a performance loss in the cases of 3 and 5 VMs. In both cases, a VM is added to run in another node. That is, such a node runs only a single VM of Job 1. Hence, the communication burden imposed with the other node is higher than the extra CPU resources obtained.

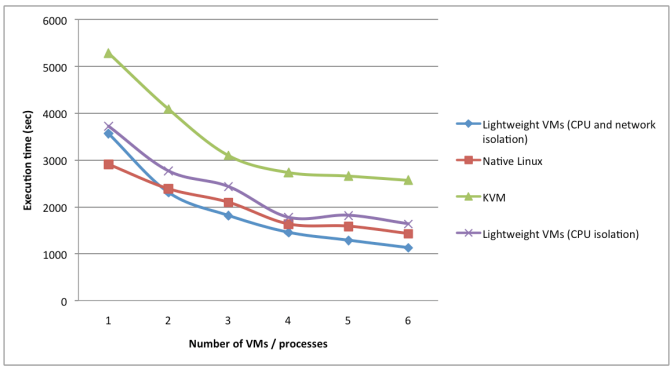

Fig 12. Series 5 – Scenario 1 (competing jobs). Network Bandwidth Isolation on a Communication-intensive Application.

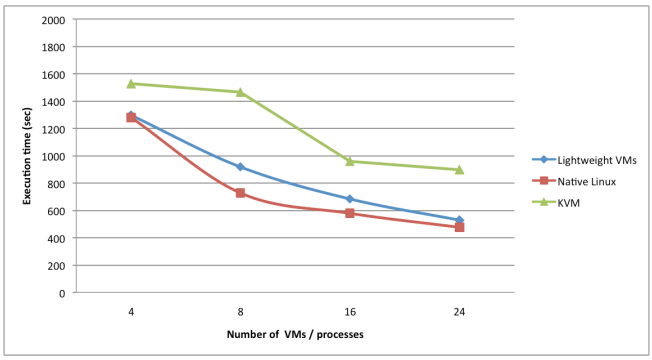

Fig 13. Series 5 – Scenario 2 (no competing jobs). Network Bandwidth Isolation on a Communication-intensive Application.

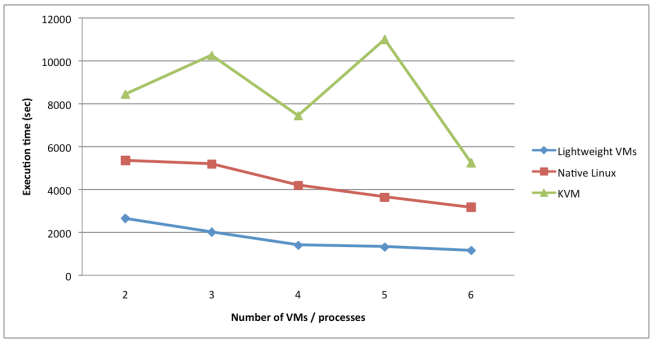

Fig 14. Series 5 – Scenario 3 (competing jobs stress resources). Network Bandwidth Isolation on a Communication-intensive Application.

In order to test the scalability of our lightweight VMs we present further experiments in the next subsection.

F. Evaluating the scalability of the approach

We carried out two further experiments to test the scalability of the lightweight VMs. The experiments were performed in a private cloud setup based on the *Zazil* cluster also located at CUCEA, University of Guadalajara. The cluster contains 64 cores, which involves 4 nodes where each node includes two 8-core processors Xeon E5-2640 v2 with 32 GB of main memory, and the nodes are interconnected with InfiniBand. The cluster runs Linux Centos 6.4, 64 bit, kernel 2.6.32-573 and uses Intel MPI 3.2.2. In this experimental setup we used a 75 Km by 75 Km domain decomposition with 20 km resolution, which contains 120 x 70 grid simulation time of 24 hours.

In the first experiment we tested the scalability of the lightweight VMs with WRF in a free workload environment. We carried out a run of Job 1, involving the execution of WRF for up to 128 lightweight VMs and the performance of the VMs was compared with that of native Linux, as shown in Figure 15. In the case of lightweight VMs, we employed two VMs per core, each VM with 50% of CPU, whereas for native Linux we used two processes per core. It can be observed that the performance of the lightweight VMs is pretty close to the performance obtained with native Linux even in the case of a larger number of VMs (e.g. 96, 112, and 128 VMs).

In the second experiment we tested the scalability of the lightweight VMs with WRF in a non-free workload environment. We tested the performance of Job 1 involving a run of WRF while Job 2 and Job 3 were executing, each of them also involving a run of WRF. In the case of lightweight VMs, each Job was executed with one VM per core and at a time three VMs were running in the same core (each VM belonging to one of the three jobs). VMs of Job 1 were given 50% of CPU whereas VMs of Job 2 and Job 3 were given 25% each. In the case of native Linux, three processes were running at a time per core used (each process belonging to one of the three jobs). We can observe that the performance of lightweight VMs is far better than the performance obtained with native Linux. The reason is the CPU isolation gives a share of 50% to Job 1 in the case of lightweight VMs, whereas with native Linux Job 1 equally shares the CPU with the other competing jobs, namely Job 2 and Job 3.

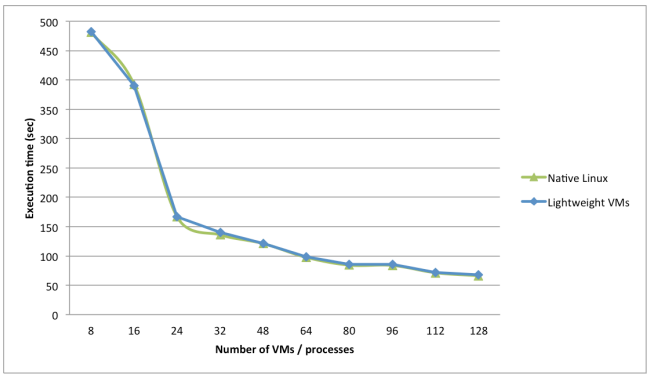

Fig 15. Scalability of Lightweight VMs in a free workload environment (the execution time of Job 1 while no other jobs are executing).

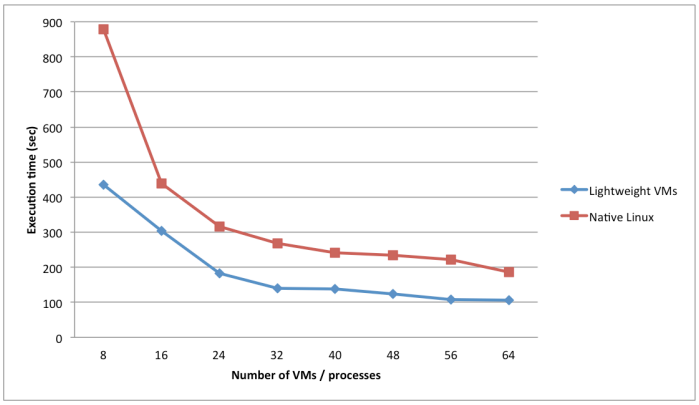

Fig 16. Scalability of Lightweight VMs in a non-free workload environment (the execution time of Job 1 while Job 2 and Job 3 are executing).

G. Discussion

Lightweight VMs have shown to be a suitable mechanism to run WRF and other HPC applications efficiently while retaining the benefits of cloud computing such as resource usage optimisation. The total overhead imposed by the lightweight VM approach is about 5%. In contrast, KVM VMs have a much higher overhead that go up to 303% in the case of WRF.

WRF was the application that was more severely affected by the performance of KVM. An overhead of 67% is reached when there are competing jobs sharing the same resources (Series 1, Scenario 1). The overhead is even worse, i.e. 303%, when traffic is injected into the network (Series 5, Scenario 3). The reason this application was more negatively impacted than the other ones is WRF is a communication-intensive application. The performance of KVM is greatly affected when a huge amount of I/O operations are involved.

Embarrassingly parallel applications, such as mpiBLAST, perform well in virtualised environments due to the fact that the worker jobs interchange a small amount of messages. However, in the case of KVM, mpiBLAST obtained an overhead up to 76% when competing with WRF jobs (series 2, scenario 1). Therefore, the performance of mpiBLAST is negatively impacted when sharing resources with a communication-intensive application.

In the case of MiniFE, a memory-intensive application, both KVM VMs and lightweight VMs obtain much better performance than native Linux. However, lightweight VMs outperform KVM from 20% to 30%. The experimental results have also shown that lightweight VMs scale well as no performance degradation can be observed as the number of VMs increases to a higher number of VMs.

Importantly, the D-VTM framework provides a higher-level programming model than current infrastructure as a service (IasS) cloud platforms (e.g. Amazon EC2, OpenStack, etc). Running an HPC application in such platforms is in many cases a complex task as several steps need to be carried out manually (or coding a script that can be complex i.e. such a script is not trivial in any way). Some of these steps are: instantiating the required VMs, configuring a single VM as a master node and configuring a set of VMs as a cluster of VMs working as slaves computing nodes able to communicate among each other. In addition, a mechanism to share files among the cluster of VMs, such as NFS, needs to be configured. In contrast, in our approach none of these steps have to be carried out by the cloud application programmer, instead such steps are automated by the D-VTM programming model. Amazon offers CC2 instances for HPC applications, which alleviates part of such a complexity as a single CC2 instance includes two octa-processors. Hence, there may not be a need to configure a cluster of VMs since a single CC2 instance might provide enough computer power in some cases. However, CC2 instances are much more costly that non-HPC instances. Therefore, there can be cases in which it is not affordable to pay the cost of CC2 instances. Moreover, in other cases, more than two octa-processors can be needed to run an HPC application within a specific window-time, thus, needing to configure a cluster of CC2 instances.

Although, our lightweight VMs cannot allow different operating systems to run on top of the host operating system, there are certain environments in which having a single operating system running is enough to cover the demands. This is the case of our WRF Web Portal application and of the other HPC applications evaluated in this paper, namely mpiBLAST and MiniFE. Hence, various Linux applications can benefit from achieving resource isolation from a VM and still share the same operating system image for working properly.

VI. Concluding Remarks

We have presented the D-VTM framework, which is a lightweight virtualisation approach to make cloud computing suitable for running WRF as well as other HPC applications. Our framework is based on lightweight virtualisation. At the core of the framework we have D-VTMs, which represents a distributed VM encapsulating a pool of resources and provide the execution environment of distributed computations. A D-VTM involves multiple lightweight VMs running in different nodes in charge of executing a parallel application. In addition, a number of VM management functions are provided. The Job Scheduler attends job requests accompanied with the desired number of VM instances. The VTM Scheduler is able to determine the availability of resources as well as performing resource reservations. Instances of D-VTMs are created by the VTM Factory. Lastly, the Job Monitor is responsible for monitoring the execution of jobs.

We have evaluated our approach with three different kinds of HPC applications. The experimental results have shown that our lightweight VM approach involves about 5% overhead in all cases. In contrast, KVM VMs are slower and the performance obtained is prohibitive when the hardware is shared with other CPU- and communication-intensive applications, such as WRF. Even mpiBLAST, an embarrassingly parallel application, was negatively affected when sharing resources with a communication-intensive application. Only MiniFE, a memory-intensive application, showed no degradation, compared to native Linux, when KVM VMs were used. Nonetheless, the performance of KVM VMs were clearly outperformed by lightweight VMs. Therefore, our approach is much more efficient than KVM VMs for running these kinds of HPC applications while retaining the benefits from cloud computing.

Current work is looking at developing a load balancing approach to both increase the performance of HPC applications and improve resource usage in the cloud. Future work will look into the issue of developing a performance prediction model to tell the user of the WRF Web platform the estimated

execution time of a WRF run request. Finally, we will also consider dynamic D-VTM resource management, which allocates resources to lightweight VMs on demand according to application performance and resource utilisation objectives.

ACKNOWLEDGMENT

Hector A. Duran-Limon would like to thank the State Council of Science and Technology of the State of Jalisco (COECYTJAL) (grant 495-2008), IBM (Faculty Award 2008), RedesClim-CONACYT, and the Mexican's Public Education Ministry (grant PROMEP-Thematic Networks of Collaboration, call 2011) for supporting this work. Ming Zhao's research is sponsored by National Science Foundation under grant CCF-0938045 and Department of Homeland Security under grant 2010-ST-062-000039. This work is part of the Latin American Grid (LA Grid) initiative (LA Grid 2013). We also thank Erick Corona for his valuable work to carry out some of the experiments. The authors are also thankful to the anonymous reviewers for their useful comments. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

Altschul S, Gish W, Miller W, Myers E, Lipman D (1990). Basic local alignment search tool. Journal of molecular biology, Vol. 215, No. 3. (5 October 1990), pp. 403-410, doi:10.1006/jmbi.1990.9999.

Antypas K, Shalf J, and Wasserman H (2008). NERSC-6 workload analysis and benchmark selection process. LBNL, Tech. Rep.

Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebar R, Pratt I, and Warfield A (2003). Xen and the Art of Virtualization. In ACM Symposium on Operating Systems Principles (SOSP). 2003.

Cgroup (2013). http://www.mjmwired.net/kernel/Documentation/cgroups/. Accessed 9 May 2013.

Dai Y, Qi Y, Ren J, Shi Y, Wang X, and Yu X (2013). A lightweight VMM on many core for high performance computing. In Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '13). ACM, New York, NY, USA, 111-120.

Darling A, Carey L, and Feng W (2003). The Design, Implementation, and Evaluation of mpiBLAST. In the 4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with ClusterWorld Conference & Expo, June 2003.

Devera M (2013). Hierarchical Token Bucket. http://luxik.cdi.cz/~devik/qos/htb/. Accessed 9 May 2013.

DiViNE (2015). Disaster mItigation on VIrtualised eNvironmEnts. http://maestro.cucea.udg.mx/~hduran/project/NaturalDisasterMitigation/v2_NaturalDisasterMitigation_project.html. Accessed 15 December 2015.

Duran-Limon H, Siller M, Blair G, Lopez A, and Lombera-Landa J (2011). Using lightweight virtual machines to achieve resource adaptation in middleware. IET Softw. 5, 229 (2011).

Duran-Limon H, Silva-Bañuelos L, Tellez-Valdez V, Parlavantzas N, Zhao M (2011). Using Lightweight Virtual Machines to Run High Performance Computing Applications: The Case of the Weather Research and Forecasting Model. In Proceedings of the 4th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2011), Melbourne, Australia, December 2011.

Docker (2015). https://www.docker.com/. Accessed 15 December 2015.

Ekanayake J and Fox G (2009). High Performance Parallel Computing with Clouds and Cloud Technologies. In 1st International Conference on Cloud Computing (CloudComp09).

Ekanayake J, Gunarathne T, and Qiu J (2011). Cloud Technologies for Bioinformatics Applications. IEEE Transactions on Parallel and Distributed Systems, Vol. 22, Issue 6, June 2011.

Evangelinos C and Hill C (2008). Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In Proceedings of Cloud Computing and Its Applications.

Expósito R, Taboada G, Ramos S, Touriño J, Doallo R (2013a). Performance analysis of HPC applications in the cloud. Future Generation Computer Systems, Volume 29, Issue 1, January 2013, Pages 218-229.

Expósito R, Taboada G, Ramos S, González-Domínguez J, Touriño J, Doallo R (2013b). Analysis of I/O Performance on an Amazon EC2 Cluster Compute and High I/O Platform. The Journal of Grid Computing , (2013).

Fernández-Quiruelas V, Fernández J, Baeza C, Cofiño A, and Gutiérrez, J (2009). Execution management in the GRID, for sensitivity studies of global climate simulations. Earth Science Informatics, Springer-Verlag, (2009) Vol. 2, pp. 75-82.

Hoffa C, Mehta G, Freeman G, Deelman E, Keahey K, Berriman B and Good J (2008). On the Use of Cloud Computing for Scientific Workflows. SWBES 2008.

Huang W, Liu J, Abali B, and Panda D (2006). A case for high performance computing with virtual machines. In Proceedings of the 20th annual international conference on Supercomputing (ICS '06). ACM, New York, NY, USA, 125-134.

Hubert B, Maxwell G, van Mook R, van Oosterhout M, Schroeder P, Spaans J, Larroy P (2013). Linux Advanced Routing & Traffic Control HOWTO. Accessed 9 May 2013. http://lartc.org/howto/

HWRF (2015). The Hurricane Weather Research and Forecast System. http://wwwt.emc.ncep.noaa.gov/?branch=HWRF. Accessed 15 December 2015.

InfiniBand (2007). InfiniBand Architecture specification volume 1, Release 1.2.1, InfiniBand Trade Association, 2007.

Jackson K, Ramakrishnan L, Muriki K, Canon S, Cholia S, Shalf J, Wasserman H and Wright N (2010) .Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud Amazon Web Services Cloud. CloudCom 2010.

JTG (2013). Jugi's Traffic Generator. Accessed 9 May 2013. http://www.netlab.tkk.fi/~jmanner/jtg.html

Kroeker K (2009). The evolution of virtualization. *Commun. ACM* 52, 3 (Mar. 2009), p. 18-20.

Martinez J, Wang L, Zhao M, and Masoud S (2009). Experimental study of large-scale computing on virtualized resources. In Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing (VTDC '09). ACM, New York, NY, USA, 35-42.

Mauch V, Kunze M, Hillenbrand M (2013). High performance cloud computing, Future Generation Computer Systems, Volume 29, Issue 6, August 2013, Pages 1408-1416, ISSN 0167-739X, http://dx.doi.org/10.1016/j.future.2012.03.011.

Mell P and Grance T (2011). *The NIST Definition of Cloud Computing* (800-145). National Institute of Standards and Technology (NIST) National Institute of Standards and Technology (NIST).

MiniFE (2013). http://www.nersc.gov/systems/trinity-nersc-8-rfp/draft-nersc-8-trinity-benchmarks/minife/. Accessed 9 May 2013.

Motika G and Weiss S (2012). Virtio network paravirtualization driver: Implementation and performance of a de-facto standard. Comput. Stand. Interfaces 34, 1 (January 2012), 36-47.

Nova (2015). http://docs.openstack.org/developer/nova/. Accessed 15 December 2015.

NCBI (2013) National Center for Biotechnology Information (NCBI). ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/. Accessed 9 May 2013.

LA Grid (2013). Latin American Grid. http://latinamericangrid.org/. Accessed 9 May 2013.

Lange J, Pedretti K, Dinda P, Bridges P, Bae C, Soltero P, Merritt A (2011). Minimal Overhead Virtualization of a Large Scale Supercomputer. Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011), March, 2011.

(Linux-VServer 2013)Linux-VServer (2013). http://www.linux-vserver.org. Accessed 9 May 2013.

LXC (2013). The LXC Linux Containers. http://lxc.sourceforge.net. Accessed 9 May 2013.

OpenStack (2015). http://www.openstack.org/. Accessed 15 December 2015.

OpenVZ (2013). http://www.openvz.org. Accessed 9 May 2013.

Raj H and Schwan K (2007). High Performance and Scalable I/O Virtualization via Self-virtualized Devices. In Proceedings of the 16th International Symposium on High Performance Distributed Computing, Pages 179-188, 2007.

Russell R (2008). virtio: Towards a De-Facto Standard For Virtual I/O Devices. ACM SIGOPS, Operating Systems Review, 42(5), July 2008, pp 95-103.

SAP (2013). SAP SD Standard Benchmark Application Results, Two-Tier Internet Configuration. http://www.sap.com/solutions/benchmark/sd2tier.epx. Accessed 9 May 2013.

SPECvirt_sc2010 (2013). Results Published by SPEC. http://www.spec.org/virt_sc2010/results/specvirt_sc2010_perf.html. Accessed 9 May 2013.

Sun C, Nishimura H, James S, Song K, Muriki K, Qin Y (2011). HPC Cloud Applied To Lattice Optimization. Proceedings of 2011 Particle Accelerator Conference, New York, NY, USA.

VMware (2006). VMware Infrastructure Architecture Overview. Technical paper. 2006.

VMware (2007). A Performance Comparison of Hypervisors, 2007.

Wang G and Ng T (2010). The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. INFOCOM 2010.

WRF (2013). The Weather Research and Forecasting Model. http://wrf-model.org. Accessed 9 May 2013.

Xavier M, Neves M, Rossi F, Ferreto T, Lange T, De Rose C (2013). Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp.233, 240, Feb. 2013.

XenSource (2007). A Performance Comparison of Commercial Hypervisors, 2007.

Younge A, Henschel R, Brown J, von Laszewski G, Qiu J, Fox G (2011). Analysis of Virtualization Technologies for High Performance Computing Environments. 2011 IEEE International Conference on Cloud Computing (CLOUD), pp.9-16, 4-9 July 2011.

Zhai Y, Liu M, Zhai J, Ma X, and Chen W (2011). Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications. In State of the Practice Reports (SC '11). ACM, New York, NY, USA, Article 11, 10 pages, 2011.