Multiverse: Dynamic VM Provisioning for Virtualized High Performance Computing Clusters

Jashwant Raj Gunasekaran*, Michael Cui†, Prashanth Thinakaran*, Josh Simons†, Mahmut T. Kandemir*, Chita R. Das*
* *Computer Science and Engineering, The Pennsylvania State University,* † *VMware Inc.*
{jashwant, prashanth, mtk2, das}@cse.psu.edu, {xiaolongc, simons}@vmware.com

arXiv:2006.12560v1 [cs.DC] 22 Jun 2020

***Abstract*—Traditionally, HPC workloads have been deployed in bare-metal clusters; but the advances in virtualization have led the pathway for these workloads to be deployed in virtualized clusters. However, HPC cluster administrators/providers still face challenges in terms of resource elasticity and virtual machine (VM) provisioning at large-scale, due to the lack of coordination between a traditional HPC scheduler and the VM hypervisor (resource management layer). This lack of interaction leads to low cluster utilization and job completion throughput. Furthermore, the VM provisioning delays directly impact the overall performance of jobs in the cluster. Hence, there is a need for effectively provisioning virtualized HPC clusters, which can best-utilize the physical hardware with minimal provisioning overheads.**

Towards this, we propose *Multiverse*, a VM provisioning framework, which can dynamically spawn VMs for incoming jobs in a virtualized HPC cluster, by integrating the HPC scheduler along with VM resource manager. We have implemented this framework on the *Slurm* scheduler along with the *vSphere* VM resource manager. In order to reduce the VM provisioning overheads, we use instant cloning which shares both the disk and memory with the parent VM, when compared to full VM cloning which has to boot-up a new VM from scratch. Measurements with real-world HPC workloads demonstrate that, instant cloning is 2.5× faster than full cloning in terms of VM provisioning time. Further, it improves resource utilization by up to 40%, and cluster throughput by up to 1.5×, when compared to full clone for bursty job arrival scenarios.

***Keywords*- HPC; virtualization; VM provisioning; cloning;**

I. INTRODUCTION

High performance computing (HPC) has evolved over the years, due to the application demands ranging from scientific computing to AI/ML-based applications [1]. HPC system stack is changing rapidly to keep up with the performance demands of such applications. It is an important constraint for the HPC cluster administrators to improve the cluster throughput and utilization, without sacrificing the application performance. Therefore, in the past these HPC applications have been traditionally deployed on bare-metal hardware [2], due to the performance guarantees offered by native execution.

On the other hand, with the advent of high throughput computing jobs [3], HPC clusters require massive scaling of resources while accessing large volumes of data. Towards this, there have been several advancements for rapid HPC provisioning such as OpenStack [4], which enables software-defined HPC infrastructure, that saves the time wasted on manual configuration and cluster provisioning. This leads to better infrastructure manageability [5], while ensuring the native bare-metal performance. However, this performance guarantee comes at the cost of abysmal bare-metal cluster utilization, due to lack of fine-grain support in sharing resource like CPU, memory and network across multiple tenants [6].

Recent advancements in hypervisor and virtualization technology can improve the cluster utilization without sacrificing the performance while supporting multi-tenant execution. Through virtualization, HPC workloads can also benefit from supporting resource heterogeneity, performance isolation, improved security, etc., [7]. Moreover, the performance overheads of virtualized HPC clusters with respect to job execution times, have less than 5% overheads for throughput and non-I/O intensive applications, when compared to bare-metal clusters [8], [9]. Combined with these numerous benefits, nowadays virtualization of HPC environments is becoming more prevalent [10], [11]. For instance, the Johns Hopkins University Applied Physics Laboratory, transformed their existing bare-metal cluster to a virtual cluster (vGRID) [12], which led to drastic improvements in resource utilization by upto 19%, with less than 4% performance degradation.

Although, virtualization guarantees near-native job execution time, private virtualized HPC clusters still face challenges in terms of cluster management such as, (i) efficient scalability of the VMs in the cluster, (ii) efficient cluster utilization with respect to dynamic job arrivals at the HPC scheduler, and (iii) minimizing the VM provisioning delays. This is because, the native HPC job schedulers like *Slurm [13] or Torque [14] do not interact with the hypervisor management layer* [15], [16]. Therefore, HPC cluster administrators/providers need to statically provision and manage Virtual Machines (VMs) for every incoming job from the HPC scheduler. Motivated by these observations, we argue that, there is a need for a framework to dynamically provision virtualized HPC clusters, which ensures efficient cluster utilization, with minimal provisioning overheads.

To overcome some of the challenges mentioned above, several frameworks have been proposed to integrate traditional HPC schedulers [13], [14], along with VM resource managers [4], [17], to enable seamless and self-managed VM provisioning for HPC clusters. Still, a majority of these frameworks lack support to dynamically provision

VM for every job and requires the cluster administrator to manually intervene during scale-out phase [1] and mapping the job requirements from the HPC scheduler to the VMs. Furthermore, in an attempt to reduce the VM provisioning delays, several optimizations such as image sharing, taking snapshots of VM [18], [19], etc,. have been proposed. Despite these efforts, new VMs still take hundreds of seconds for provisioning [20], [21]. There have been recent efforts to reduce VM provisioning latencies using *instant clone technology [22]*. Instant clones uses a copy-on-write [23] feature to drastically reduce the provisioning time when compared to full clones [24], which has to boot-up a new VM from scratch. To the best of our knowledge, there are very few existing works, which have support for dynamic VM provisioning [20], [21], [25], however, they do not utilize techniques such as instant clone to reduce provisioning overheads in a virtualized HPC environment.

To holistically address the shortcomings of existing works, we build a dynamic VM provisioning framework, called *Multiverse*, which can spawn new VMs for incoming jobs using instant cloning in a virtualized HPC cluster. This enables more flexible and cluster utilization-aware VM provisioning. In summary, we make the following **contributions**:

- We have designed *Multiverse*, a generic framework that *integrates* HPC scheduler with VM orchestrator[2] for dynamic VM provisioning on a per-job basis. We have implemented a prototype of the mentioned framework using Slurm [13] as the HPC job scheduler and VMware vSphere [26] as the VM orchestrator.
- We incorporate an admission control system and a dynamic load balancer using sqlite3 database [27], which ensures efficient VM placement decisions in cluster.
- We have characterized the performance and scalability of the *Multiverse* framework for two types of VM cloning mechanisms on a 220 core HPC cluster.
- Our experimental analysis using different job arrival scenarios shows that, instant cloning is 2.5× - 7.2× better than full cloning, in terms of VM provisioning time. Further, our results show that instant cloning can provide up to 40% better resource utilization, and 1.5× better cluster throughput, when compared to full cloning.

II. BACKGROUND AND MOTIVATION

A. Why Virtualization for HPC?

Despite the fact that virtualization having proved to be cost effective, scalable and reliable in majority enterprise infrastructures, HPC applications are still executed on bare-metal, non-virtualized clusters (for most cases), to achieve maximum performance. This suffers from major challenges such as (i) lack of support for dynamic load balancing and migration, and (ii) lack of isolation and security among multi-tenant workloads. However, virtualization can transform HPC infrastructure by overcoming these challenges by enabling (i) proactive VM migrations during failure and load imbalance, (ii) micro-segmentation using network virtualization for fine-grain isolation.

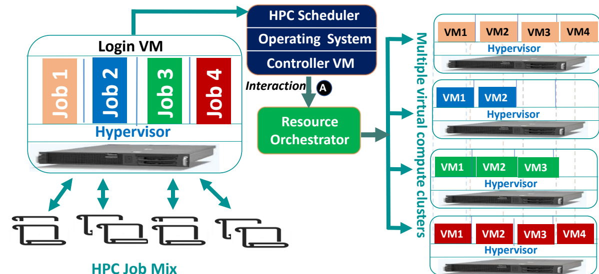


Figure 1: Overview of a Virtualized HPC framework.

B. Overview of Virtualized HPC framework

Figure 1 illustrates the architecture of virtualized HPC framework. All physical nodes in the cluster are virtualized using a VM hypervisor (VMware ESXi or kvm [28]). The hypervisor which directly runs on the physical nodes in privileged mode provides abstraction for VMs to run while mapping host resources such as CPU, memory, storage, and network to each VM. The original login node, master node for job scheduling, and compute nodes now run as VMs on the existing login, controller, and compute nodes, respectively, within the same cluster. In addition, a VM orchestrator (VMware vCenter or Opennebula) [17], [26] runs inside a VM on one controller node and provides centralized management of the hosts and VMs, coordinating resources for the entire cluster. In this paper, we propose a dynamic VM provisioning framework for allocating VM(s) for every incoming job in such a virtualized HPC cluster.

C. Challenges in Dynamic VM Provisioning

In case of a large HPC cluster with hundreds to thousands of nodes, where many jobs can arrive within a short span of time, cluster administrators face fundamental challenges to statically allocate and manage VMs for those jobs. Dynamic provisioning would be better in such scenarios and also lead to better resource utilization, because it avoids over-provisioning of VMs. Typically, VM provisioning is handled by resource managers such as VMware vSphere, OpenStack or KVM [17], [26], [28]. However, traditional HPC schedulers do *not* have support to provision and manage VMs (as shown in Figure 1 A). Therefore, there is a need to integrate HPC schedulers to work collectively with VM resource managers and make such integration transparent to the user. We explain the challenges in such a design and how we achieve this integration in Section IV.

[1] Scale-out indicates that, we have more incoming jobs than available VMs.

[2] We use the terms orchestrator and resource manager interchangeably.

D. VM Cloning Types

Fast and agile dynamic provisioning of VMs is directly impacted by the time taken to provision a new VM. Typically, virtualized environments use cloning [24], to provision new VMs. A clone is a copy of an existing VM, which is called the parent of the clone. There are two well known types of clone (i) *full clone*, which is an independent copy of a virtual machine that shares nothing with the parent virtual machine after the cloning operation, and (ii) *linked clone*, which is a copy of a virtual machine that shares virtual disks with the parent virtual machine.

Recent interest in container-based resource provisioning has led to the developments of a new cloning technique called *instant clone* [22]. Instant clone uses rapid in-memory cloning of a running parent VM, and leverages copy-on-write to rapidly deploy VMs. They are much faster when compared to full and linked clones but a significant amount of time needs to be spent to configure and customize the network. We explore the opportunity to use instant clone and compare its performance with full clones for different inter-arrival times of jobs. The results of the characterization are presented and discussed in Section VI.

III. Overall Design of *Multiverse*

In this section, we first discuss the challenges in our proposed design to integrate HPC schedulers with VM orchestrators. Subsequently, we explain the in detail the design of *Multiverse* framework that effectively addresses the challenges.

A. Design Challenges

It is not trivial to enable the integration of HPC schedulers along with VM resource managers due to the following reasons. First, we need to hide the virtualization integration from the user. Second, multiple jobs might share the same parent VM/image while cloning new VMs. This leads to potential issues related to disk management (snapshots) and concurrency in cloning. Third, VMs have to be customized as specified in the job requirements file submitted to the HPC scheduler. This requires support for availability of multiple different VM images when using instant clone. Finally, the resource orchestrator needs to have policies for dynamic load balancing and admission control of VMs. Typically this would be handled by the HPC scheduler for all jobs, since we override the scheduler functionalities to support integration with VM resource orchestrators, it is the responsibility of resource orchestrator to handle load balancing and admission control.

B. Design Choices

We explain the design of *Multiverse* with respect to overcoming the challenges mentioned in the previous section. First, to retain the original job submission behaviour of users, we need to customize the job submission workflow in the HPC scheduler. Typically, jobs go through different stages within the scheduler starting from job submission to resource selection to job allocation. We modify each of these stages as follows. During the job submission, we parse all the job requirements so that, later we can *dynamically provision* a new VM based on the requirements. In the resource selection stage, we make the job wait until the VM for the job has been provisioned and, finally, in the job allocation stage, we ensure that the job is allocated to the corresponding VM.

Second, to support launching VMs for concurrent job submissions, our system needs to be thread safe. Though, the schedulers themselves are thread safe for multiple job submissions, the changes we make to every stage in the scheduler should ensure the same. Our design makes use of an *explicit state machine*, which is thread safe, for maintaining the different states of a job, as shown in Figure 2. Once a job is submitted, it enters into queued state ❶, and continues to exist in that state until a VM spawn[3] process is initiated for that job. Then, it moves to spawning state ❷, while the VM is being spawned and configured. Once the spawning is complete, the job moves to a spawned state ❸. Next the job should be scheduled on the newly-spawned VM. However, the scheduler is unaware about which VM to allocate the job, because there could be multiple VMs with similar configuration that can satisfy for a job. Hence, we need a mechanism to enforce the scheduler to allocate a job on to a specific VM, which was spawned for the job. To do so, we make use of job-feature parameters (predominantly supported in all HPC schedulers) within every job and VM, to enable a unique job-to-VM mapping. After the job is allocated on the VM, it moves to the allocated state ❹. We explain in detail about the implementation of the state machine in Section IV.

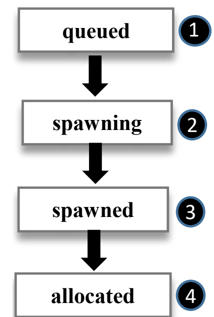


Figure 2: State machine used in our *Multiverse* design.

Apart from being thread-safe, we also need to ensure that, multiple VM clones from the same parent image does not add additional overheads on disk management. This in turn leads to clone failures. To mitigate such clone failures, we make use of a *rate-limiter mechanism*, which can limit the number of clones per parent VM for a given time. Through our characterization study, we set the rate-limiter at 15 clones per minute and 200 clones per second for full clone and instant clone, respectively. Note that instant clone supports more concurrent clones as it shares both disk and memory with the parent VM.

Third, to support different customizations of the VM in

[3]We use the terms launch and spawn interchangeably.

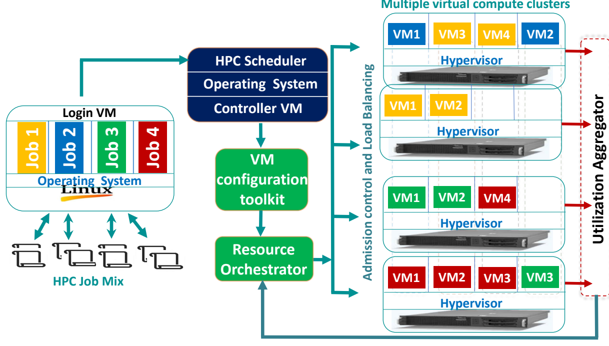


Figure 3: A high level overview of *Multiverse* framework. The VM configuration toolkit acts as an interface between scheduler and resource manager. Utilization aggregator (shown in red) sends real-time cluster utilization metrics to the VM resource manager (shown in green).

accordance to the job requirements, we provide a baseli VM image (based on the OS used) which is used for cloni of VMs. Any application-specific libraries which might required by the users can be made available on a shared f systems (like NFS) and this file system is mounted on the cloned VMs. We also have to enable the scheduler specific configurations such that, the new VM will be added to the scheduler's node pool. This is done by using customization scripts which are executed on the VM right after cloning.

Fourth, to enable admission control and load balancing, we design an *utilization aggregator* to store physical node-specific metrics like CPU, memory utilization, number of active VMs, etc,. This database can be queried by the resource orchestrator using custom APIs, to check current resource utilization against specific admission control policies. Also, by leveraging the resource utilization metrics from the database, we design two *load balancing policies*, to enable fair resource allocation in the cluster. The detailed implementation of the system is given in Section IV-C.

C. *Multiverse workflow*

The overall workflow of the *Multiverse* framework is shown in Figure 3. Users submit a job to a login node, which is extracted by the controller node. The controller node decides job scheduling and placement based on specified policies. As briefly discussed in the previous subsection, we override this functionality of the controller, to spawn a VM for every job submitted and allocate the job on the spawned VM. To achieve this design, we make use of custom scheduler plugins to extract the job requirements, spawn a new VM based on the requirements, allocate the job and delete the VM after job completion. For all the VM specific interactions, we need to use a VM configuration toolkit which can interface with the resource orchestrator. The resource allocation and management of the VMs on to physical nodes is handled by the resource orchestrator/manager by leveraging the cluster specific metrics, which are exposed to a database using our *utilization aggregator*.

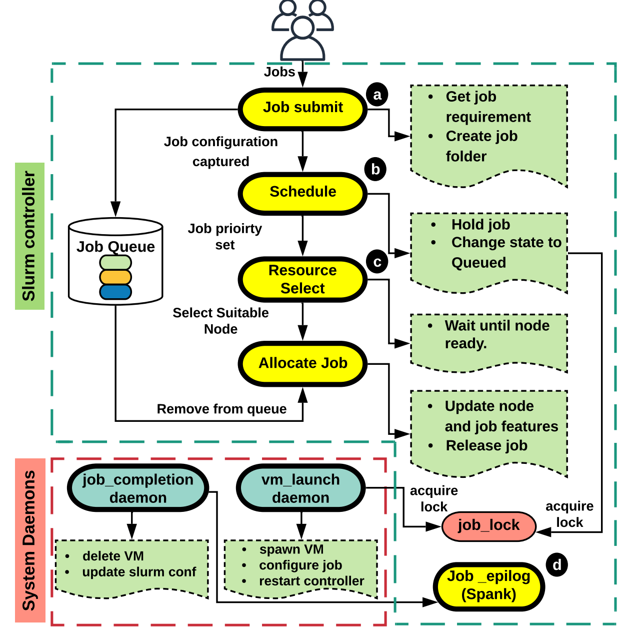


Figure 4: Plugin-based implementation of *Multiverse* framework, showing the various steps of interaction. The system daemons are shown in blue and the *Slurm* plugins are shown in yellow. Job lock (shown in red) is shared between the plugins and daemons.

IV. Implementation Methodology

We explain in detail about the implementation of our proposed *Multiverse* framework. While our implementation is with respect to *Slurm* scheduler and *vSphere* resource manager, *Multiverse* is generic to be extended to work with most HPC schedulers and VM resource managers.

A. *Augmenting Slurm*

As shown in Figure 4, we make use of four custom *Slurm* plugins to enable support for dynamic VM provisioning. The details of these plugins are given below.

*1) **Job Submit Plugin**:* Job submit ⓐ is called by *Slurm* controller right after submitting the job and before scheduling resource allocation. This plugin can override the existing `job_submit` method, to give user-defined controls and change the configuration parameters of the job. We use this plugin to create a job config file and copy the following information to it: job name, number of CPUs, required memory, minimum number of nodes, submit time and other job related metrics. The job config file has a uniquely identifiable name, which is a concatenation of job name with submission timestamp.

*2) **Scheduler plugin**:* Scheduler plugin ⓑ is called after the `job_submit` and before resource (VM/node) selector. We override the `slurm_sched_p_initial_priority` function using scheduler plugin, which sets the initial priority for the job. We set the priority value such that, all incoming

jobs will be on hold and will not be eligible to schedule (we name this as `sched_hold`). This is essential, because the VMs for allocating the jobs are spawned only after job submission. Also, in the same function, we update the job information (job name and *Slurm* generated `job_id`) to a file named `queued_jobs`. We employ a locks to ensure atomic writes to the file by multiple jobs.

*3) **Resource Select Plugin**:* The resource select plugin ⓒ is invoked after the `job_submit` and scheduler plugin to check if there are any available resources (VMs) to run the job. Since the VMs for the job are spawned after submission, they might not be ready during this phase of scheduling. Therefore, we modify this plugin to return true for VMs selection by default, though there are no available resources.

*4) **Spank Plugin**:* The spank plugin ⓓ is used to define any user-defined functions which is called during various steps in *Slurm* execution depending upon the context during which it is called. We use this plugin in `job_epilogue` context to call a cleanup function. This function will notify the controller node that the job has been complete. The state of compute VM (the newly spawned VM for executing the job) is marked as "down" to prevent future jobs to be scheduled on this VM. Also, the job output and error logs are copied to master and login node.

B. Custom system daemons

Apart from customizing the *Slurm* plugins, we also design and implement two custom system daemons which will handle the tasks of VM creation, job launch and VM deletions. They are described in detail below.[4]

*1) **VM launch daemon**:* The primary purpose of this daemon, is to initiate a VM launch for all submitted jobs to *Slurm*. The daemon is designed to work like a state machine (as shown in Fig 2) where every job can be in one of the following four states.

- Queued ❶: The job is added to *Slurm* queue and is updated in queued_jobs file by the scheduler plugin. For jobs in this state, the daemon calls the corresponding function to start spawning a new VM as per the job requirements which were captured using the job_submit plugin. The new state of the job is changed to spawning.
- Spawning ❷: The daemon calls vm_launch script for a previously queued job. Now the job will be periodically queried to see if VM spawning is complete. The daemon takes necessary actions (re-spawn or cancel) if the spawning fails due to some reason.
- Spawned: If the VM spawning for the job is complete, the daemon changes the state of the job to spawned. For all jobs in spawned state ❸, the daemon updates the *slurm* config file to include details of the newly spawned VM. It releases the jobs from *Slurm* hold so that the job can be allocated ❹ to the corresponding VM for execution. Also, after adding any new nodes to *slurm* config file, the *Slurm* controller has to be restarted. This is due to the inherent design of *Slurm*. Hence, the daemon also restarts the controller, for the VM allocation to be successful for all spawned jobs.
- Pending: Since we use locks to ensure serialized write to the queued_job file, the job_lock has to be acquired by the *Slurm* scheduler plugin and the VM_launch daemon. Hence, in the *Slurm* scheduler plugin, if the job_lock is busy, it updates the job information to a pending_job file. The daemon constantly extracts jobs from this file and initiates the VM_launch function. Hence the pending state, is an auxiliary state used when the job_lock is busy.

*2) **Job completion daemon**:* This daemon, constantly monitors the state of all the compute VMs. Recall from our spank `job_epilog` plugin, the state of VMs after job completion is marked to be "down". For all such VMs, the job completion daemon calls a cleanup function which will ensure the following two steps. First, it clears the node information from *Slurm* config file. Second, it deletes all the job configuration details captured during submission and also deletes the VM which was spawned for running the job.

C. Admission control and load balancing

We developed a python-based api which can query the real-time information about all hosts in a cluster and maintain the information in a *sqlite* [27] database. We use this api in the VM_launch function to get a compatible host for cloning new VMs. This is a convenient API which can be extended to develop different admission control and load balancing schemes. The API exposes basic functionalities such as (i) initializing a database with existing cluster information, (ii) update the database based on new allocations/de-allocations, and (iii) get a compatible host for the new clone request Using these functionalities of the API, we designed an admission control and load balancing policy for the *Multiverse* framework.

1) Admission control: We enforce two types of admission control. If all the resources of the hosts are currently utilized or there is not enough room to accommodate a new request, the job waits in the queue until resources become available. If the required resources of the job are more than the physical capacity of the host, the job is revoked from execution. In the former case, to avoid starvation of the jobs due to unavailability of resources, we make sure that newly incoming jobs are queued behind the delayed job inside the internal queued_jobs file used by vhpc_launch daemon. There is still scope to improvise this policy by enforcing rules for starvation. For example we can set a limit on how many times the job can be re-queued, or how long the scheduler can run other smaller jobs until the bigger jobs keep waiting for resources.

2) Load balancing: We have two different policies for load balancing the VMs across the hosts. In the first pol-

[4]The circled annotations in each subsection are with respect to Figure 2.

icy, we chose the first compatible host by doing a linear search across all hosts. In this context, "compatible" means that the host has *enough resources* to be allocated for the VM requirements. However, there can be more than one compatible host in the cluster. Hence we implement a second policy, where we randomly select a host from the list of compatible hosts. This incurs additional overhead compared to the first_available policy but can ensure better load balancing across the cluster.

D. Software specifications

1) Scheduler and APIs: We use *Slurm* version 19.05 as our HPC scheduler. All the *Slurm* plugins are written in `C-language` and generated as shared library (.so) files, which are dynamically linked to the scheduler. The system daemons are `bash` files which are hosted as `systemctl` service in Linux. We also design our VM_launch and cleanup scripts as bash files. For implementing `mutex` locks we make use of `Flock` [29] Linux utility which is available in both bash and C. vHPC toolkit [30] was developed using Python which makes use of `pyVmoi` [31] and `pyVim` python packages that enable access to *vSphere* APIs. We use this toolkit for VM configuration based on job requirements.

2) Clone configuration: We need to generate a clone configuration file for spawning a new VM using vHPC toolkit. For full clone, the template VM can reside in any node in the cluster. But in the case of instant clone, we cannot instantiate clones on different hosts, other than the template VM. Therefore we have a template VM on every node of the cluster and based on the chosen host given by our load balancer, we initiate the instant clone on that host.

In addition, for instant clone the CPU and memory cannot be dynamically configured , because it uses VMFork [22] to fork off a new VM from the template. Essentially, the same hardware configuration of the template VM would be retained for the cloned VM. In the case of diverse jobs which have different memory and CPU requirements, we can have different-sized template VMs on each host and select a closest matching compatible template VM.

E. Assumptions and Limitations

As soon as a user submits a job to slurm, the job would be waiting to get scheduled until a VM is spawned for the job. Hence, the additional time incurred to clone new VMs is accounted along with the total waiting time of the user. However the cloning time would not affect other running jobs because it is implemented as background process that can be executed in parallel with other processes. Due to the inherent design of slurm, each time a new VM is added to the configuration file, we have to restart the *Slurm* controller daemon. This overhead can be avoided if we use other HPC schedulers like PBS [32] or Torque [14], wherein new nodes can be added online without restarting controller daemons.

Our proposed design changes to *Slurm* might affect its inherent job scheduling features. For instance, *Slurm* supports different scheduling policies like backfill, priority etc. Since the scheduling policies now depends on the VM scheduler used by *vSphere*, administrators can use the corresponding scheduling policies in *vSphere*, which reflect the same behaviour as *Slurm*. Moreover, the original scheduling policies of the HPC scheduler can be retained if we do not tag every job to its respective VM. Its easier to customize this change into other HPC schedulers, when compared to *Slurm*.

V. Experimental Setup

A. Hardware Configurations

1) Cluster: We use a cluster of five Dell PowerEdge R630 nodes, with 44 cores and 256 GB memory in each node The cluster has a 72 TB shared datastore with 5 physical network adapter of 10Gbps for each node. We use ESXi-6.7 hypervisor along with vSphere-6 resource manager.

2) Master node: *Slurm* controller is initiated from a master VM which is preallocated on the cluster. It runs on Centos 7 with 8 vCPUs and 16 GB memory which is large enough to handle our input job sequence.

3) Login node: Users submit jobs via login node which is also a VM configured with 2 vCPUs and 4 GB memory.

B. Workload Generator

We generate a workload for a job-sequence modeled after *Poisson* inter-arrival times using a mean job arrival rate of $\lambda = 10$ for 100 jobs. The jobs are associated with one of the three benchmarks which are (i) High-Performance Conjugate-Gradient (HPCG),a simple additive Schwarz, symmetric GaussâĂŞSeidel preconditioned conjugate-gradient solver [33], (ii) High Performance Linpack (HPL) [34] which measures the performance solving a dense linear equation system, and (iii) MPIRandomaccess [35] which measures peak capacity of the memory subsystem while performing random updates to the system memory.

We configure two types of jobs. First, we configure a short job which uses 2 vCPUs and 4 GB memory. Second, we configure a large job which uses 8 vCPUs and 16 GB memory. Both jobs only use 1GB of local storage (disk) space. We randomly sample from both the jobs for the sequence of 100 jobs. The HPCC and HPCG input files are modified accordingly to generate sufficient load for the given CPU and memory configuration. Each job has a running time ranging from 140s to 350s depending on the benchmark.

VI. Evaluation and Results

A. Evaluation Methodology

We perform our evaluations from two complementary angles. First, we compare the overall job completion time in terms of cloning time, other overheads (explained below) and actual job running time for two types

Table I: The overheads incurred by Multiverse framework for VM provisioning and allocation.

Overhead Type	Description
schedule_clone	Time taken to start the clone script by VM_Launch daemon.
get_host	Time taken to get a compatible host from load balancer.
network_configuration	Time taken to configure and customize the network.
slurmd_customization	Time taken to copy the Slurm config files and restart the slurmd node daemons.
slurm_restart	Time taken by VM Launch daemon to restart slurmctld after VM is ready.
slurm_schedule	Time taken to assign the job to the VM after restarting the controller.

of cloning techniques. Second, we individually characterize every overhead incurred in our framework. The overheads and their description are shown in Table I. We evaluate two different job arrival scenarios as shown in Figure 5, (a) The first 50 jobs of the Poisson distribution, such that the cluster is fully utilized (workload-1) and (b) all the 100 jobs of the Poisson distribution, with 2x CPU over-commitment enabled in the cluster (workload-2). Further to characterize the bottleneck with full clone with respect to concurrent cloning, we also use a constant inter-arrival time of 10s, for jobs such that they all don't arrive within a short span of time.

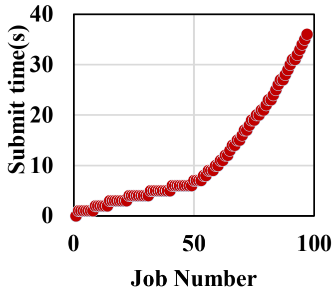


Figure 5: Poisson arrival rate fora sequence of 100 jobs with $\lambda = 10$**.**

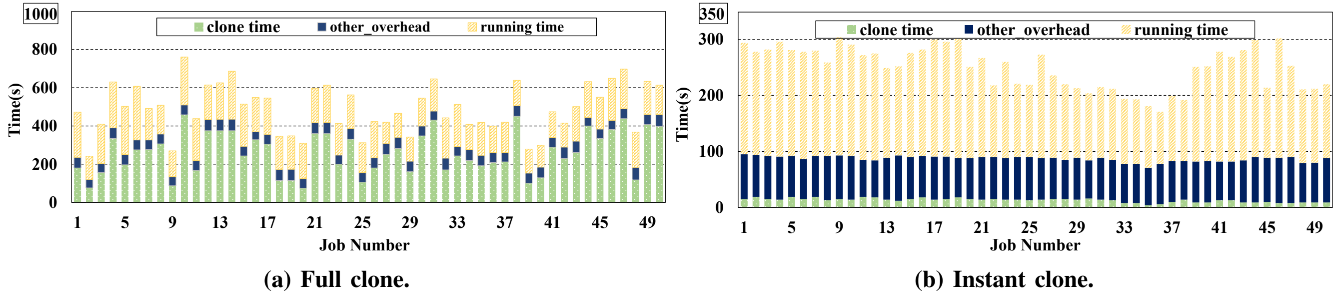
B. Results and Analysis

Figure 6a and 6b shows the breakdown of the overall job completion time for 50 jobs using full and instant clone respectively. It can be seen that instant clone is extremely fast in provisioning VMs with an average cloning time of 10s. On the other hand, full clone takes about 150s on average with a maximum clone time of 450s in some cases. This is because, full clones do not share any resources with the parent VM and require a lot more time for disk provisioning. Instant clone, on the other hand, are forked off of the parent VM and hence are very fast. However, from Figure 7b which shows the break down of individual overheads, the network configuration overhead is very high for instant clones. *Since they share the same network config as the parent VM after cloning, the network has to be reconfigured*. However, instant clones (36s on average) are still 7.2× faster compared to full clones (260s on average), with respect to overall VM provisioning time.

The overhead to restart the *Slurm* controller by the VM launch daemon is in the order of 20s. Majority of this time is spent within *Slurm* to complete the restart after being initiated by the daemon. Since a lot of jobs are running on the system, the scheduler spends significant amount of time for the restart. The other overheads apart from network_config and slurmctld_restart are minimal and very similar for both clone types (shown in Figure 7a and 7b). Figure 9 plots the job running time using both instant and full clone. The running times are fairly similar with a few variations. We can infer that running time does not get affected due to type of clone used.

Figure 8 shows the breakdown of job completion and other overheads for a constant inter-arrival (10s) between jobs. The total cloning time is within 75s, and the overall provisioning time including other overheads is within 140s for all the jobs. In this case, full clone performs very similar to instant clone in terms of overall job completion time. This is because, the number of concurrent clone operations handled by vSphere is significantly reduced. vSphere can handle up to 200 concurrent instant clones but it incurs higher latency for concurrent full clones. However, the overall provisioning time using instant clone (36s on average) is still 2.5× faster than full clone (87s on average). Note that, the job running times are much lower for constant arrival when compared to bursty arrival. This is due to the fact that, the cluster is not 100% utilized for a constant arrival, which consequently leads to lesser interference among VMs.

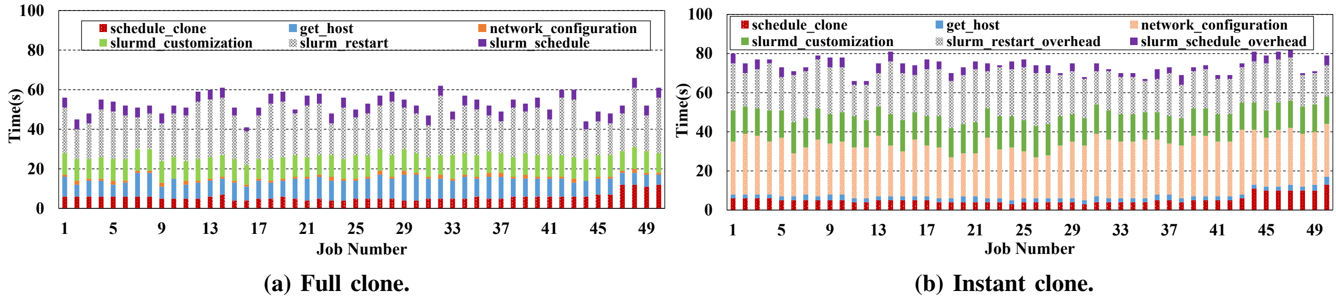
*1) **Scalability using CPU Over-commitment**:* We conduct another set of experiments where we use 2x over-commitment of CPU in the cluster (i.e), the cluster will be running jobs with total vCPUs equal to twice as many as available CPUs. We use a the same Poisson based arrival sequence, but for 100 jobs. Figure 10 plots the breakdown of job completion time for instant and full clone. It can be seen that, instant clone scales well for 100 jobs as the cloning time is well within 15s (Figure 10b) for all the jobs. For the last few jobs from 86 to 91, the overheads are higher because the cluster is already full with no more available vCPUs to allocate for VMs. This is shown in Figure 11b, that the get_host time for these jobs are very high. On the other hand, for full clone, the time taken to clone is very large (shown in Figure 10a). This is because concurrent full clones are very slow to handle. As described in Section III, we use a rate-limiter of 15 clones per minute. This can be seen in Figure 11a, where the schedule_clone increases in multiples of 15s. The performance degradation due to clone overheads is very large for jobs starting from 51, because more and more jobs are queued in vSphere to be cloned. Note that, for instant clone, some jobs (41, 42 etc shown in Figure 11b) take a longer time to complete than the average. This is because, the 2x CPU over-commitment causes a CPU



(a) Full clone.

(b) Instant clone.

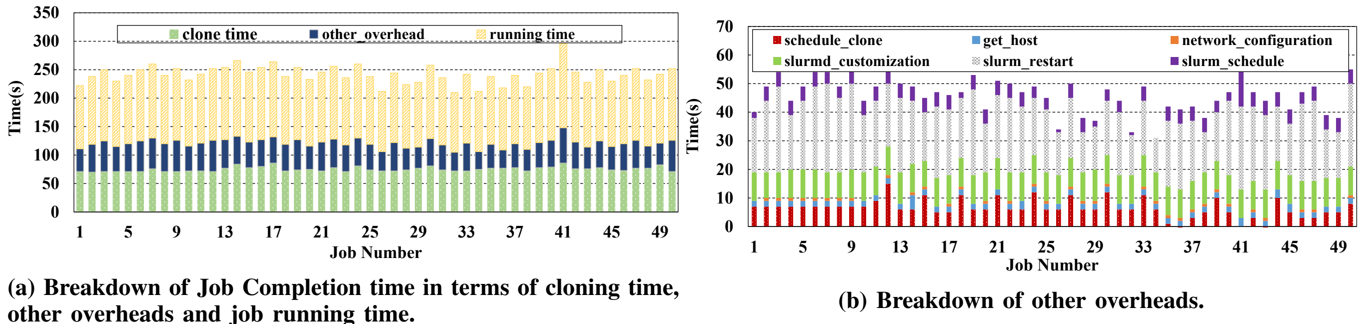
Figure 6: Workload-1: Breakdown of Job Completion time in terms of cloning time, other overheads and job running time.



(a) Full clone.

(b) Instant clone.

Figure 7: Workload-1: Breakdown of other overheads for 50 jobs.



(a) Breakdown of Job Completion time in terms of cloning time, other overheads and job running time.

(b) Breakdown of other overheads.

Figure 8: Constant job arrival for 50 jobs using full clone.

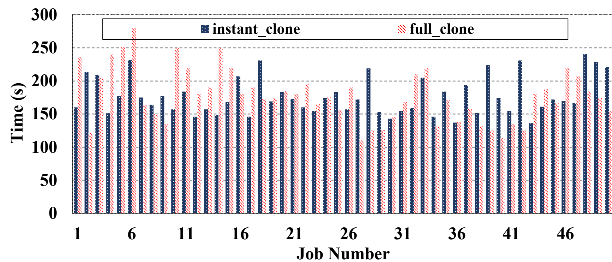
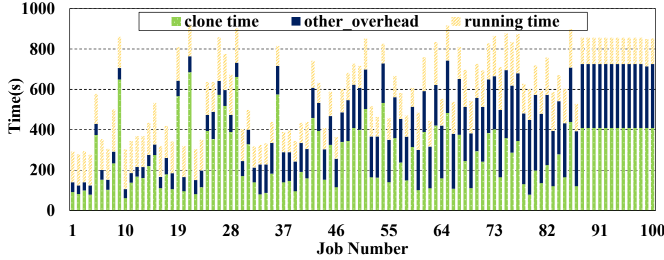


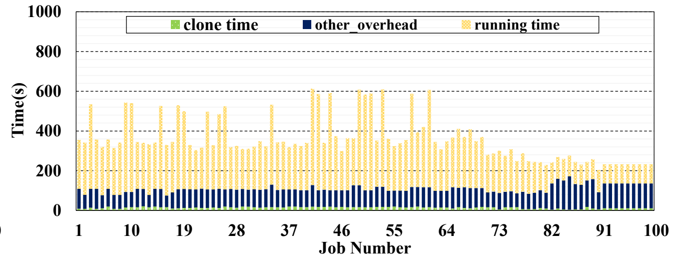
Figure 9: Workload-1: Comparison of Job running time for full and instant clone for 50 jobs.

pressure on the physical hosts as we have more allocated virtual CPUs than available CPUs. We repeat the experiment for same job configuration but using a constant inter-arrival time of jobs for full clone. As seen in Figure 12a, the cloning time is much faster compared to workload-2 and the overall job completion is very similar to instant clone. As stated earlier, this is because increasing the inter-arrival time between jobs leads to fewer concurrent clones executing in the cluster. Also there are no straggler jobs towards the end as in instant clone because, all the 100 jobs are equally spaced out in the cluster and do not run concurrently. We can conclude that, instant clone is best suited in case of bursty job arrivals as opposed to full clone, which would be suitable for a constant job arrival rate. Furthermore, we can build an mixed system that can use a combination of instant and full clones, depending on the difference in job arrival rate over time.

*2) **Cluster Utilization and Throughput**:* Figure 13 plots the CPU utilization for workload-2 using both full and instant clones. The utilization numbers are

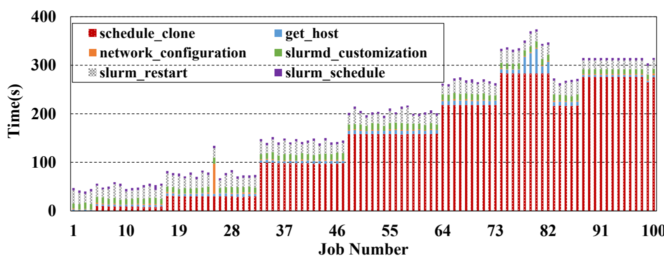


(a) Full clone.

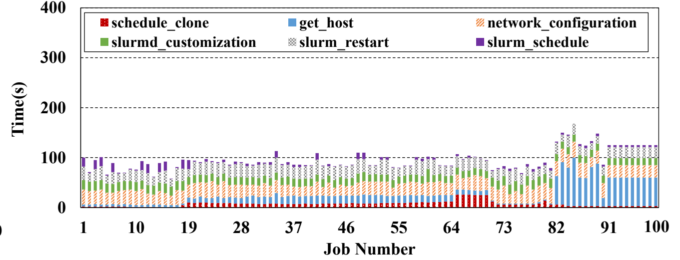


(b) Instant clone.

Figure 10: Workload-2: Breakdown of Job Completion time in terms of cloning time, other overheads and job running time with 2x CPU over-commitment enabled in the cluster.

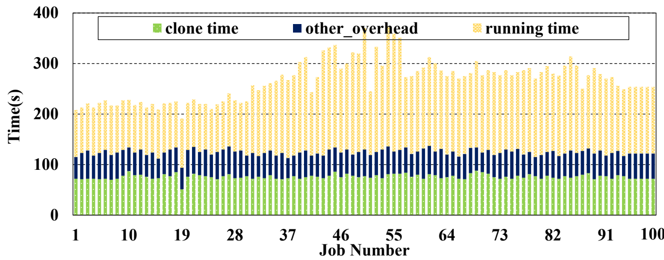


(a) Full clone.

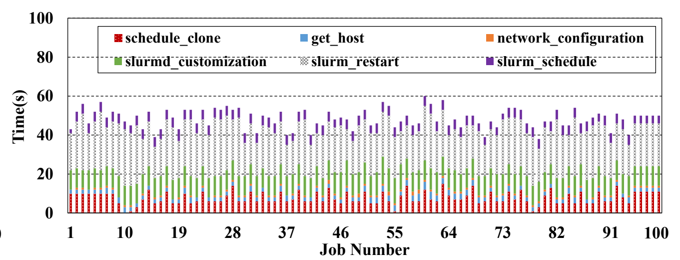


(b) Instant clone.

Figure 11: Workload-2: Breakdown of other overheads with 2x CPU over-commitment enabled in the cluster for both clone types.



(a) Breakdown of Job Completion time in terms of cloning time, other overheads and job running time.



(b) Breakdown of other overheads.

Figure 12: Constant job arrival with 100 jobs using full clone.

collected periodically every 10s for the entire workload execution time. It can be seen that the average CPU utilization for instant clone, initially is 60%; but, starting from time-step 21, it is very high ranging from 80-100%. This clearly indicates that all the 100 jobs are scheduled as soon as they arrive and are executing concurrently in the cluster. On the other hand for full clone, the maximum cluster CPU utilization never goes beyond 50%.

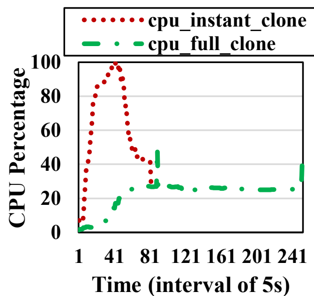


Figure 13: Average CPU utilization.

This is because most of the jobs spend a lot of time to get a cloned VM before they can start executing. This reduces the number of concurrent jobs in the system. Further, the system throughput using instant clone is 1.5× better than using full clone. This is because, the total time taken for job completion is 581s (end of dotted-red line) for instant clone, when compared to 868s (end of dashed-green line) for full clone.

*3) **Comparison with bare-metal deployments**:* To ensure that, the job performance is not affected due to virtualization overheads, we conduct another set of experiments by executing the jobs in a bare-metal cluster managed by *Slurm*. Figure 14 shows the comparison of job running times for both bare-metal and virtualized deployments. The running/execution times are fairly similar without significant variations. Hence, we can conclude that virtualization can

deliver near-native bare-metal performance for jobs.

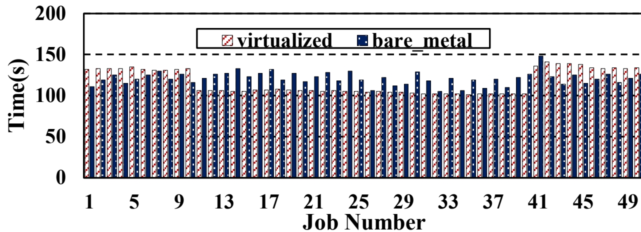


Figure 14: Job running times of virtualized deployment compared to bare-metal deployment.

C. Overheads of Multiverse

We explain in detail the overheads incurred with respect to the different components of multiverse.

*1) **Clone Overheads**:* Despite instant clone being faster than full clone, the network overhead incurred by instant clone is significantly high. This is because, they replicate the same network configuration as the parent VM. We need to manually set the IP address (or use DHCP) and reconfigure the network of the cloned VM. This incurs about 10-20s of the total VM start-up time.

*2) **Controller Restart Overheads**:* As explained in Section VI-B, restarting the slurm controller takes around 20s on average. This overhead can be significantly minimized by (i) increasing the size (CPU and memory allocations) for the slurm controller VM, and (ii) multiple slurm controllers can be used in parallel to share the load of managing the slave VMs. We also mention in Section IV-E that restarting the controller can be avoided if we use other HPC schedulers like PBS or Torque.

*3) **Job Concurrency Overheads**:* Our Workload-1 does not have any interference from jobs because, it consists of stream of 50 jobs that can entirely fit in the physical capacity (CPU and memory) of the 220-core cluster. However, in Workload-2 the overall job requires 2x of available physical capacity (2x over-commitment). Therefore there is certain degree of performance degradation in terms of job running times (Figure 10b vs Figure 6b). We can further characterize the tolerance of over-commitment ratio with respect to job performance, but that is beyond the scope of this paper. For constant job-arrival, there is no impact of interference because the jobs are equally spaced without any contention for resources.

VII. Related Work

Dynamic VM Provisioning for HPC: With the prevalence of virtualization into the HPC community, some prior works have attempted to integrate a dynamic VM provisioning model using HPC scheduler like Slurm and Torque [20], [25]. However, these frameworks are neither robust nor eliminate manual user intervention. The most relevant work to *Multiverse* is proposed by Zhange et al [21] called v-slurm. However, they do not characterize the other overheads apart from VM provisioning. We provide a detailed characterization of all overheads associated with the *Multiverse* framework. Moreover, *Multiverse* is the first work to employ instant clone based rapid dynamic VM provisioning in a HPC environment.

Agile VM Provisioning: There are several prior works which have proposed quick and agile VM provisioning mechanisms. Some of them require live VMs [22], while the rest try to minimize the VM disk size [36]. However, none of these have been adopted by a majority of mainstream HPC private clusters. There has recently been an increased interest in running containers such as Docker inside VMs [37]. One of the major benefits of such an approach is fast environment startup – in the order of seconds. Containers, however, have dependency on their hosting OS, due to their process-based nature. This makes container migration difficult. On the other hand, instant clones used in *Multiverse* are very fast, and comparable to container based provisioning times.

HPC in public cloud: There has been significant advancements in HPC provisioning by the union of cloud system stack along with HPC, enabling IaaS-based provisioning of HPC infrastructures. In this context, many dynamic VM provisioning schemes have been proposed using Microsoft Azure, AWS EC2, etc, [38]. However, HPC clusters are largely hosted in private datacenters for security, tractability, and fault tolerance. Our work primarily focuses on mitigating the bottlenecks of virtualized HPC in a private setting. Also, some of the ideas proposed in the context of public cloud, [39]–[42] can also be leveraged by *Multiverse*, as our framework is a generic implementation by setting up a platform for further enhancements.

VIII. Conclusion and Future Work

In this paper, we identify several challenges in developing a dynamic VM provisioning framework for virtualized HPC clusters. We design and implement *Multiverse*, which integrates an HPC scheduler with a VM orchestrator, to dynamically spawn VMs for incoming jobs in a virtualized HPC cluster. This enables more flexible and cluster utilization aware VM provisioning. We further explore the potential of using instant cloning compared to full cloning in terms of VM provisioning overhead, resource utilization and cluster throughput. Experimental results with HPC workloads indicate that instant cloning on an average is $2.5\times$ - $7.2\times$ faster than full cloning in terms of VM provisioning time. Further, instant cloning improves cluster utilization by up to 40% and cluster throughput by up to $1.5\times$, when compared to full clones for bursty job arrivals. On the other hand, full cloning is comparatively better for constant job arrivals with large inter-arrival times. In our future work, we plan to compare instant clone based VM provisioning against a container-

based provisioning. Towards this, we plan to integrate a docker hypervisor with the Slurm scheduler and analyze the job completion times with our *Multiverse* framework.

ACKNOWLEDGMENTS

We are indebted to Na Zhang, Anup Sarma and Cyan Mishra for their insightful comments on several drafts of this paper. This research was partially supported by NSF grants #1931531, #1629129, #1763681, #1629915, #1908793, #1526750 and we thank NSF Chameleon Cloud project CH-819640 for their generous compute grant. We also thank Mohan Potheri for providing us with a compute cluster from VMware to conduct all the experiments.

REFERENCES

[1] Y.-T. Tsai, "An overview of machine learning and hpc in open sources for bioinformatics," in *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*.

[2] P. Rad, A. Chronopoulos, P. Lama, P. Madduri, and C. Loader, "Benchmarking bare metal cloud servers for hpc applications," in *2015 CCEM*.

[3] R. Arora, *Conquering Big Data with High Performance Computing*. Springer, 2016.

[4] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, 2012.

[5] I. Kureshi, C. Pulley, J. Brennan, V. Holmes, S. Bonner, and Y. James, "Advancing research infrastructure using openstack," *International Journal of Advanced Computer Science and Applications(IJACSA), Special Issue on Extended Papers from Science and Information Conference 2013*.

[6] W. Huang, J. Liu, B. Abali, and D. K. Panda, "A case for high performance computing with virtual machines," in *ICS*, 2006.

[7] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: High performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, 2015.

[8] M. Potheri, J. Ling, N. Zhang, shawn Kelly, and J. Simons, "Virtualizing High-Performance Computing (HPC) Environments," VMware, Tech. Rep., 2018.

[9] N. Zhang and J. Simons, "Running HPC and Machine Learning Workloads on VMware vSphere," VMware, Tech. Rep., 2018.

[10] Q. Ali, V. Kiriansky, J. Simons, and P. Zaroo, "Performance evaluation of hpc benchmarks on vmwareâĂŹs esxi server," in *European Conference on Parallel Processing*. Springer, 2011.

[11] A. Gupta, O. Sarood, L. V. Kale, and D. Milojicic, "Improving hpc application performance in cloud through dynamic load balancing," in *CCGrid*, May 2013.

[12] J. Simons, E. DeMattia, and C. Chaubal, "Virtualizing HPC and Technical Computing with VMware vSphere," VMware, Johns Hopkins University Applied Physics Laboratory, Tech. Rep., 2017.

[13] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003.

[14] D. Klusáček, V. Chlumskỳ, and H. Rudová, "Planning and optimization in torque resource manager," in *HPDC*, 2015.

[15] D. Beserra, F. Oliveira, J. Araujo, F. Fernandes, A. AraÃžjo, P. Endo, P. Maciel, and E. D. Moreno, "Performance evaluation of hypervisors for hpc applications," in *2015 IEEE International Conference on Systems, Man, and Cybernetics*, Oct 2015.

[16] H. A. Hassan, S. A. Mohamed, and W. M. Sheta, "Scalability and communication performance of hpc on azure cloud," *Egyptian Informatics Journal*, 2016.

[17] D. Milojičić, I. M. Llorente, and R. S. Montero, "Opennebula: A cloud management tool," *IEEE Internet Computing*, 2011.

[18] J. Li, Y. Zhang, J. Zheng, H. Liu, B. Li, and J. Huai, "Towards an efficient snapshot approach for virtual machines in clouds," *Information Sciences*, vol. 379, 2017.

[19] Y. Zaslavsky, O. Frenkel, and M. Kolesnik, "Creating a virtual machine from a snapshot," Jun. 2 2015, uS Patent 9,047,238.

[20] C.-H. Li, T.-M. Chen, Y.-C. Chen, and S.-T. Wang, "Formosa3: A cloudenabled hpc cluster in nchc," *World Academy of Science, Engineering, and Technology Journal*, 2011.

[21] J. Zhang, X. Lu, S. Chakraborty, and D. K. D. Panda, "Slurm-v: Extending slurm for building efficient hpc cloud with sr-iov and ivshmem," in *European Conference on Parallel Processing*. Springer, 2016.

[22] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: rapid virtual machine cloning for cloud computing," in *Eurosys*. ACM, 2009.

[23] J. Bhimani, Z. Yang, M. Leeser, and N. Mi, "Accelerating big data applications using lightweight virtualization framework on enterprise cloud," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017.

[24] "Vm cloning," "https://www.vmware.com/support/ws5/doc/ws_clone_overview.html", 2019.

[25] K. Meier, G. Fleig, T. Hauth, M. Janczyk, G. Quast, D. Von Suchodoletz, and B. Wiebelt, "Dynamic provisioning of a hep computing infrastructure on a shared hybrid hpc system," in *Journal of Physics: Conference Series*. IOP Publishing, 2016.

[26] N. Marshall, M. Brown, G. B. Fritz, and R. Johnson, *Mastering VMware VSphere 6.7*. John Wiley & Sons, 2018.

[27] M. Owens, *The definitive guide to SQLite*. Apress, 2006.

[28] H. D. Chirammal, P. Mukhedkar, and A. Vettathu, *Mastering KVM virtualization*. Packt Publishing Ltd, 2016.

[29] "Flock," https://linux.die.net/man/1/flock.

[30] "Virtualized hpc configuration toolkit," "https://github.com/vmware/vhpc-toolkit", 2019.

[31] F. Liu and Z. Yang, "Design of vmware vsphere automatic operation and maintenance system based on python," in *ICAMechS*. IEEE, 2018.

[32] H. Feng, V. Misra, and D. Rubenstein, "Pbs: a unified priority-based scheduler," in *SIGMETRICS*, 2007.

[33] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, 2016.

[34] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[35] S. Saini, R. Ciotti, B. T. Gunney, T. E. Spelce, A. Koniges, D. Dossa, P. Adamidis, R. Rabenseifner, S. R. Tiyyagura, and M. Mueller, "Performance evaluation of supercomputers using hpcc and imb benchmarks," *Journal of Computer and System Sciences*, 2008, performance Analysis and Evaluation of Parallel, Cluster, and Grid Computing Systems.

[36] K. Razavi, G. Van Der Kolk, and T. Kielmann, "Prebaked μvms: Scalable, instant vm startup for iaas clouds," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, 2015.

[37] J. Zhang, X. Lu, and D. K. Panda, "Performance characterization of hypervisor-and container-based virtualization for HPC on SR-IOV enabled infiniband clusters," in *IPDPS Workshops 2016*.

[38] S. K. Garg, A. N. Toosi, S. K. Gopalaiyengar, and R. Buyya, "Sla-based virtual machine management for heterogeneous workloads in a cloud datacenter," *Journal of Network and Computer Applications*, 2014.

[39] C. Kotas, T. Naughton, and N. Imam, "A comparison of amazon web services and microsoft azure cloud platforms for high performance computing," in *ICCE*, 2018.

[40] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in *IEEE CLOUD*, 2019.

[41] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, “Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters,” in *IEEE CLUSTER*, 2019.

[42] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, “Phoenix: A constraint-aware scheduler for heterogeneous datacenters,” in *ICDCS*. IEEE, 2017.