The Art of CPU-Pinning: Evaluating and Improving the Performance of Virtualization and Containerization Platforms

Davood GhatrehSamani[+], Chavit Denninnart[+], Josef Bacik[*], Mohsen Amini Salehi[+],
[+]High Performance Cloud Computing (HPCC) lab, University of Louisiana at Lafayette, USA
{davood.ghatrehsamani1,chavit.denninnart1,amini}@louisiana.edu
[*]Software Engineer at Facebook Inc.
josef@toxicpanda.com

***Abstract*—Cloud providers offer a variety of execution platforms in form of bare-metal, VM, and containers. However, due to the pros and cons of each execution platform, choosing the appropriate platform for a specific cloud-based application has become a challenge for solution architects. The possibility to combine these platforms (*e.g.,* deploying containers within VMs) offers new capacities that makes the challenge even further complicated. However, there is a little study in the literature on the pros and cons of deploying different application types on various execution platforms. In particular, evaluation of diverse hardware configurations and different CPU provisioning methods, such as CPU pinning, have not been sufficiently studied in the literature. In this work, the performance overhead of container, VM, and bare-metal execution platforms are measured and analyzed for four categories of real-world applications, namely video processing, parallel processing (MPI), web processing, and No-SQL, respectively representing CPU intensive, parallel processing, and two IO intensive processes. Our analyses reveal a set of interesting and sometimes counterintuitive findings that can be used as best practices by the solution architects to efficiently deploy cloud-based applications. Here are some notable mentions: (A) Under specific circumstances, containers can impose a higher overhead than VMs; (B) Containers on top of VMs can mitigate the overhead of VMs for certain applications; (C) Containers with a large number of cores impose a lower overhead than those with a few cores.**

***Index Terms*—Virtualization, Container, performance overhead, CPU pinning.**

I. Introduction

Hardware virtualization in form of virtual machines (VMs) is an indispensable part of cloud computing technology that offers isolation, manageability, consolidation, and reliability [1] to cloud-based applications. However, performance overhead, resulted from several abstraction layers in the hypervisor [2]–[5], has historically been a side-effect of the virtualization technology. More recently, a lightweight virtualization technology, known as containerization, that provides abstraction at the application layer has gained popularity in the cloud era. Numerous cloud services, such as serverless computing [6], [7] (*e.g.,* AWS Lambda [8], Azure Service Fabric [9]), are offered based on containers. However, we note that containerization is even making a deeper shift in application deployment, such as those used to manage critical layers of IT infrastructure. For instance, containers are being utilized in storage appliances (*e.g.,* EMC Unity [10]) to reduce the fail-over time and improve their availability. In addition to negligible imposed overhead, containers are more storage-efficient, have shorter cloning and application deployment time, faster scale out, and offer Continuous Integration/Continuous Delivery (CI/CD) [11]. Nonetheless, conflicts between processes sharing the kernel and lack of cross middleware portability are known drawbacks of the containerization [12].

The pros and cons of each virtualization technology in addition to the possibility of deploying an application on bare-metal (*e.g.,* Dedicated EC2 instances in AWS) has offered cloud solution architects a range of *execution platforms* (*i.e.,* bare-metal, VMs, containers, and containers on top of VMs) to deploy a certain application on the cloud. In particular, each application has its own characteristics that can reap the benefits of a certain type of execution platform and undergoes a different overhead. For instance, database servers are known to take advantage of bare-metal platforms, due to high volume of disk operations [13], whereas web servers benefit from *virtualized platforms* (*i.e.,* VMs, containers, and containers within VMs) that offer a higher portability [12], [14], [15].

It is challenging for a cloud solution architect to efficiently deploy a certain application via choosing a proper execution platform. In a multi-tier application, this can potentially lead to choosing distinct execution platforms for deployment of each application tier [16]. The challenge becomes further complicated when we know that the overhead of each execution platform can remarkably vary, depending on the way the execution platform is configured on the underlying hardware resources. Specifically, CPU provisioning for virtualized platforms can be configured either through CPU-quota or CPU-set (a.k.a *CPU pinning*) [17]. In the former, at each scheduling event, the middleware of the host machine decides about allocating the proper CPU core(s) to each VM/container, whereas, in the latter, certain CPU cores are statically bound to each VM/container by the solution architect. Our hypothesis is that CPU pinning can drastically reduce the overhead of virtualized platforms. However, it is noteworthy that extensive CPU pinning incurs a higher cost and makes the host management more challenging.

The *goal* of this study is to unveil the imposed overhead of each virtualization platform for different application types commonly deployed on the cloud. Further, we study the impact

of various CPU provisioning configurations for the underlying hardware resources.

To achieve the goal, we conduct an extensive performance evaluation on the following four application types that exhibit different processing characteristics and are commonly used in the cloud: (A) *FFmpeg* [18] is a video transcoding application that exhibits a CPU-bound behavior; (B) *MPI* [19] applications that represent parallel processing behavior; (C) *WordPress* [20] is a web-based system representing many short tasks with IO-bound behavior; (D) *Apache Cassandra* [21] is a NoSQL database management system representing an extensive IO-bound behavior within a single large process.

Each application type is deployed under various resource configurations (a.k.a instance type) on our private cloud. Specifically, we evaluate the imposed overhead of different execution platforms using different number of CPU cores and under two circumstances—when CPU pinning is in place and when it is not. In summary, the *contributions* of this paper are as follows:

- Measuring and analyzing the imposed overhead of different execution platforms for widely-used cloud-based applications.
- Analyzing the impact of altering resource configurations, including number of CPU cores and CPU pinning, on the imposed overhead of different execution platforms.
- Proposing a set of best practices for deploying different application types in different virtualization platforms.

This paper is structured as follows. Section II provides an architectural view of virtualization platforms and describes CPU provisioning models. Section III describes the applications, our testbed, and analyzes the overhead of different platforms under various configurations. In Section IV, we perform a cross-application overhead analysis of the imposed overhead across different application types. Section V presents the most relevant works in the literature for benchmarking virtualization platforms and the use of pinning. Then, in Section VI, we summarize the lessons learnt and provide a set of best practices to efficiently configure cloud platforms.

II. Background

A. Overview

Virtualization platforms emulate and isolate compute, storage, and network resources within a host. Current virtualization platforms are categorized based on the level of abstraction they provide. In particular, VMs provide a hardware layer abstraction, known as *hardware virtualization*, whereas containers enable abstraction from the operating system (OS) layer, known as *OS virtualization* [22].

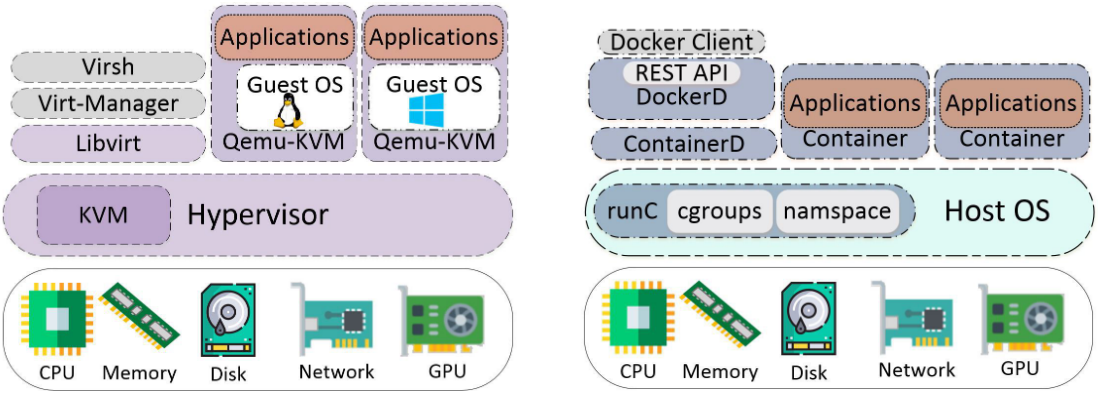
B. Hardware Virtualization (VM)

Hardware virtualization operates based on a hypervisor that enables one or more isolated guest operating systems (VMs) on a physical host [23], [24]. KVM [4] is a popular open-source hypervisor extensively used by cloud providers. For instance, AWS developed a KVM-based hypervisor called *Nitro* [25] and uses it for its `C5` VM types. Many datacenter management platforms have been developed around KVM hypervisor. For instance, Hyper Converged Infrastructure (HCI) platforms [26] (*e.g.,* Nutanix [27], Maxta [28], and Cloudistics [29]) that enable integrated software defined datacenters have adopted KVM as their underlying hypervisor.

KVM hypervisor, depicted in Figure 1a, is part of the host kernel and works directly with hardware virtualization features (*e.g.,* `Intel-VT` and `AMD-v` [30]) to share the hardware resources across multiple fully isolated VMs. KVM interfaces with the user space (*e.g.,* via `QEMU`) and executes guest OS commands via `ioctls` kernel module. Qemu-KVM emulates hardware devices for VMs and can para-virtualize [30] IO devices, such as disk and NIC, by using specific drivers that expedite accessing them. `Libvirt` is a library to help external services to interact with KVM/QEMU using its APIs. `virsh` and `virt-manager` are command-line and GUI interfaces for Libvirt.

C. OS Virtualization (Container)

Container is a lightweight and portable virtualization solution in which the host OS kernel is shared across multiple



(a) Modules of KVM hypervisor. Each VM, called Qemu-KVM, has a full-stack of the deployed applications and an operating system. Libvirt provides necessary APIs for managing KVM.

(b) Main modules of Docker. Containers are coupling of `namespace` and `cgroups` modules of the host OS kernel. Docker daemon interacts with Container daemon (`ContainerD`) and `runC` kernel module to manage containers.

Fig. 1: High level architecture of different virtualization platforms.

isolated user-space environments. In contrast to VMs, containers are transparent from the host OS perspective. That is, the processes created by a container are visible to the host OS as native processes, however, the container itself is not a process. All processes created via a container have access to the same set of resources and libraries.

Technically speaking, a container is an abstraction created by the coupling of *namespace* and *cgroups* modules of the host OS. A namespace that is assigned to a container creates an abstraction for the processes of the container and offers them an isolated user space, such as network configurations, storage space, and software packages.

Control Groups (a.k.a `cgroups`) is a kernel module that enforces and monitors resource usage limitations for a given group of processes [31] [32]. Cgroups module of the host OS is in charge of enforcing resource constraints designated for a container instance. As an example, for a container instance with two CPU cores, cgroups oversees its usage not to go beyond the two cores across the entire host. It is noteworthy that the way cgroups enforces constraints is a decisive factor from the performance overhead perspective.

Docker [14] is the most widely adopted container technology in the cloud era, hence, we consider it as the container platform in this study. However, we believe that our findings can be extrapolated to other containerization techniques that operates based on cgroups (*e.g.,* Singularity). Figure 1b illustrates the architecture of the Docker container. Docker Engine (`DockerD`) receives container management requests via its APIs. The engine is in charge of creating container instances and enforcing their specifications via `containerD` service. ContainerD utilizes `runC` module of the OS kernel to create namespace and cgroups for each container instance.

D. CPU Provisioning for Virtualized Platforms

The host OS generally uses time sharing scheduling policies (*e.g.,* Completely Fair Scheduler (CFS) [33]) that does not specify a processing unit (*e.g.,* core) to each process. That is, a VM or a container-based process is assigned to a different set of CPU cores of the host machine in each quantum, during their life cycle. In this study, a platform that has its provisioned CPU cores in this default manner is called to be deployed in the *vanilla* mode.

Alternatively, a user can choose the *pinning* mode, to manually set the CPU cores allocated to a given process (either a VM or a container) and override the default host OS scheduler. We explained the technical details of how to perform CPU pinning in a web-blog post [1]. In this case, the host OS scheduler allocates the pinned process only to the specified cores. Note that, unlike vanilla mode that utilizes all the host CPU cores to cumulatively offer the expected performance, for a pinned platform, only the designated cores are utilized and the rest are left idle. As such, the side-effect of pinning (*i.e.,* not using the host OS scheduling) can appear in a lower CPU utilization, hence, it should not be used carelessly for any application.

[1]How to perform CPU pinning: https://bit.ly/2XrENUM

III. Overhead Analysis of Different Application Types on a Variety of Virtualized Platforms

A. Evaluation Environment

In this section, we evaluate the performance of four cloud-based application types (detailed in Table I) on four popular execution platforms.

Type	Version	Characteristic
FFmpeg	3.4.6	CPU-bound workload
Open MPI	2.1.1	HPC workload
WordPress	5.3.2	IO-bound web-based workload
Cassandra	2.2	Big Data (NoSQL) workload

TABLE I: Specifications of application types used for evaluation.

The performance metric we measure in the evaluations is the *execution time* of each application type. Also, to quantify the overhead of a certain virtualized platform, we define *overhead ratio* as the average execution time offered by a given virtualized platform to the average execution time of bare-metal. Table II describes the configuration of instance types used for the evaluation. The host server is a DELL PowerEdge R830 with 4×Intel Xeon E5-4628Lv4 processors with 112 homogeneous cores, 384 GB memory (24×16 GB DRAM), and RAID1 (2×900 GB HDD) storage. Each processor is 1.80 GHz with 35 MB cache and 14 processing cores (28 threads).

Instance Type	No. of Cores	Memory (GB)
`Large`	2	8
`xLarge`	4	16
`2xLarge`	8	32
`4xLarge`	16	64
`8xLarge`	32	128
`16xLarge`	64	256

TABLE II: List of instance types used for evaluation.

The four studied execution platforms include bare-metal, which imposes the minimum overhead and is used as the baseline, in addition to three variations of virtualized platforms commonly used in the cloud (*i.e.,* VMs, containers, and containers within VMs). Figure 2 provides a schematic view of the four execution platforms and Table III elaborate on the specifications of each platform. The abbreviations mentioned in the table are used henceforth to represent each execution platforms. Note that each execution platform can be instantiated using any instance type of Tabel II.

Abbr.	Platform	Specifications
BM	Bare-Metal	Ubuntu 18.04.3, Kernel 5.4.5
VM	Virtual Machine	Qemu 2.11.1, Libvirt 4 Ubuntu 18.04.3, Kernel 5.4.5
CN	Container on Bare-Metal	Docker 19.03.6, Ubuntu 18.04 image
VMCN	Container on VM	As above

TABLE III: Characteristics of different execution platforms used in the evaluations. First column shows the abbreviation of the execution platform used henceforth in the paper.

Bare-metal (BM) execution platform only includes the host OS and the application. In VM platform, one Ubuntu VM instance is created (based on KVM hypervisor) to process requests of the application type in question. Similarly, in container platform (CN), one Docker container is instantiated

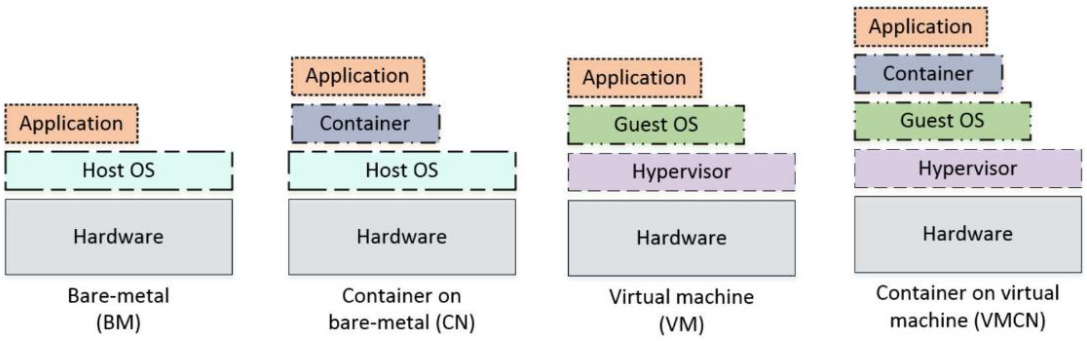


Fig. 2: The four execution platforms used for performance evaluation of different application types.

on bare-metal from an Ubuntu image. Lastly, VMCN platform refers to an execution platform where a Docker container is instantiated within a VM (with the aforementioned configurations).

Resource contention between coexisting processes in a host can potentially affect the tasks' execution times, hence, introducing noise in our overhead measurement objective. To avoid such noises, we assure that each application type is examined in isolation. That is, during the evaluation process, there is no other coexisting workload in the system.

As for the performance monitoring tools employed in this study, we used basic Linux utilities, such as `top`, `htop`, `iostat` and `perf`, alongside with `BCC` (BPF Compiler Collection [34]) as a profiling tool to perform kernel tracing and to track the execution status of processes running inside the OS. In particular, we used `cpudist` and `offcputime` to monitor and profile the instantaneous status of the processes in the OS scheduler.

Irrespective of the execution platform, the host OS scheduler is the ultimate decision maker in allocating processes to CPU cores [33]. A process (*e.g.,* `FFmpeg`) can be potentially assigned to a different set of cores at each scheduling event. It is important to note that even VMs are considered as processes from the host OS perspective.

As the scheduling events happen frequently and there are numerous cores in a typical cloud host, migrating processes from one core to another at each event is expected to induce a remarkable overhead in the cloud hosts. Specifically, among other reasons, migrating a given process induces overheads for redundant memory access due to cache miss, reestablishing interrupts for IO operation, and context switching [15], [33]. Even more overheads are involved in migrating virtualized platforms, *e.g.,* for resource usage accounting activities.

We need to measure and verify the significance of the induced overhead of different execution platforms on the overall performance. We envisage that overriding the host OS scheduler, via *CPU pinning* techniques, limits process migrations to a smaller set of CPU cores, hence, reduces the imposed overhead. As such, to verify the impact of CPU pinning, we evaluate each execution platform for different workloads with and without CPU pinning. Note that the virtualized platforms offer built-in pinning ability (*e.g.,* via *Qemu* configuration file for each VM). For BM, we modelled pinning via limiting the number of available CPU cores on the host using GRUB configuration in Linux [35].

B. Application-Specific Overhead Analysis

In the following subsections, we concentrate on the characteristics of each application type across different execution platforms. Later, in Section IV, we provide a collective cross-application overhead analysis on the root causes of the imposed overhead.

1) Video Processing Workload Using FFmpeg

FFmpeg offers a wide variety of video transcoding functions, such as those to change video resolution, bit-rate, frame rate, and compression standard. Changing the compression standard (a.k.a *codec*) is known to be the most CPU-intensive transcoding operation [36], [37] with small memory footpring (around 50 MB in our observations). Hence, we employ it in this study to represent a CPU-intensive workload. This also makes the overhead of the execution platform more visible and makes it easier to harvest. FFmpeg is a multi-threaded application and can utilize up to 16 CPU cores to transcode a video. Hence, for this evaluation, we do not allocate more than 16 cores (*i.e.,* `4×large`) to each execution platform.

We examine a source video segment[2] that has a large codec transcoding time. The reason that we examine one video segment is to concentrate on the overhead resulted from the execution platform and remove any uncertainty in the analysis, caused by the video characteristics. The source video segment is 30 MB in High Definition (HD) format. The codec is changed from AVC (`H.264`) to HEVC (`H.265`). The evaluation was conducted 20 times and the mean and confidence interval of the results were collected.

Results of the evaluation is shown in Figure 3 where the vertical axis shows the mean execution time for each experiment and the horizontal axis shows different instance types. We note that the confidence interval in many cases were negligible.

Specific observations and analysis of Figure 3 are enumerated in the following list. Note that, we defer further analysis of these observations to Section IV where we conduct a comparative study across all application types.

i. VMCN imposes the highest overhead and pinning it cannot reduce the overhead remarkably. Alternatively, CN platforms (particularly, pinned CN) are shown to impose the minimal overhead with respect to BM. Importantly,

[2]The video file is free-licensed and is publicly available in the following address: https://peach.blender.org/download/

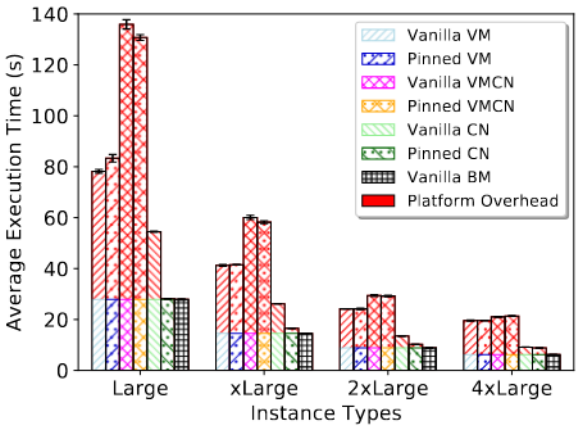


Fig. 3: Comparing execution time of FFmpeg on different execution platforms under varying number of CPU cores. Horizontal axis indicates the number of CPU cores in form of different instance types.

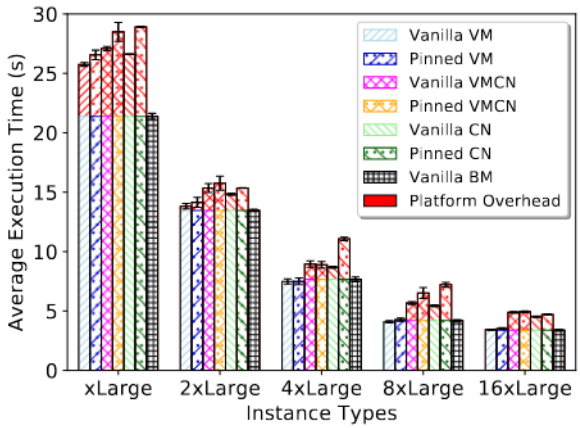


Fig. 4: Comparing execution time of MPI_search on different execution platforms. Horizontal axis represents the number of CPU cores in the form of different instance types. Vertical axis shows the mean execution time (in seconds).

we observe that as the number of cores increases, the overhead of vanilla CN and both VMCN platforms decrease.

ii. The imposed overhead of VM platforms (vanilla or pinned) across all instance types is remarkable to the extent that causing the execution times to remains at least twice as much as BM. Unexpectedly, pinning does not mitigate the imposed overhead for VMs when FFmpeg application is deployed.

iii. By adding the containerization layer on top of VM (*i.e.,* VMCN), even a larger performance overhead is imposed. The maximum and minimum imposed overhead ratios are 4 and 1, respectively. However, as the number of CPU cores allocated to the VMCN increases, the overhead is mitigated drastically, such that for `4×Large`, the overhead imposed by VMCN is almost the same as VM. Furthermore, overhead ratio of VM and VMCN (vanilla and pinned) remains almost the same across all instance types.

iv. Unlike VM and VMCN, pinning CN significantly reduces the overhead, particularly when containers are allocated with fewer processing cores (*e.g.,* `Large`). This suggests that pinned CN is a suitable virtualization platform for CPU-bound applications, such as FFmpeg.

2) Parallel Processing Workload Using MPI

Message Passing interface (MPI) [19] is a widely-used high performance computing (HPC) platform to develop parallel programs. For the analysis, we examined two MPI applications, namely *MPI_Search* (for parallel searching of an integer value) [38] and *Prime_MPI* [39] (to find all prime numbers within a given range). In these applications, the communication part dominates the computation part. This is to enable us concentrating on the impact of various virtualization platforms on the overall overhead imposed, in circumstances where intensive communication occurs between cores of the same virtualized platform. As our observations for both of the MPI applications were alike, to avoid redundancy, we only report the results for MPI_Search. To remove any randomness in the results, the evaluations were conducted 20 times and the mean and confidence interval of the execution times are reported.

Result of this evaluation is shown in Figure 4. Our observations and analyses are as follows:

i. Overhead imposed by VMCN platforms (both vanilla and pinned) is slightly more than VM-based platforms. Surprisingly, the overhead of CN (vanilla and pinned) even exceeds the VMCN platforms. A closer observation reveals that, for the MPI applications, the overhead of any containerized platform exceeds the ones for the VM platforms. Although the difference in the imposed overheads is reduced for larger number of cores, the overhead ratio remains the same.

ii. From `2×Large` onward, we notice that the overhead of VM platforms (vanilla and pinned) becomes negligible and the execution times become almost the same as BM. The reason is that, as we move towards larger instances, the bottleneck shifts from computation part to the communication part. Because the hypervisor (KVM) provides an abstraction layer to facilitate inter-core communication between VM's cores, the imposed overhead diminishes and their execution times approach BM. This is unlike communications within cores of a container that involves host OS intervention, thus, implies a higher overhead [2]. It is noteworthy that for smaller instance types, the bottleneck of VM platforms is in the computation part that, due to several abstraction layers, incurs a significant overhead.

3) Web-based Workload Using WordPress

WordPress is a PHP-based content management system (CMS) that uses Apache Web Server and MySQL database in the back-end. It is broadly deployed in clouds and known to be IO intensive [20], because each HTTP request to a website implies at least network (to read/write from/to the socket) and disk IO (to perform file/database operations). As such, we consider it as a representation of an IO intensive application,

in which each process receives at least three IO interrupts.

We configured the same WordPress website on all of the execution platforms. Apache Jmeter [40] is a tool to apply workload and measure the performance of a web server. We configured it to generate 1,000 simultaneous web requests (*i.e.,* processes) on each execution platform and then, we calculated the mean execution time (a.k.a *response time*) of these web processes. We note that Jmeter itself is a resource intensive application that can affect our intended performance evaluation. Therefore, we configured it to run on a dedicated server in the same network. To remove any possible environmental randomness, we evaluated the workload six times on each instance type. Then, we report the mean and 95% confidence interval of response time of all web requests.

Results of this evaluation are shown in Figure 5. The vertical axis shows the mean execution time of 1,000 web processes (in Seconds) and the horizontal axis shows different instance types. Our observations and analyses of the results are enumerated in the following list:

i. Unlike pinned CN that imposes the lowest overhead, vanilla CN imposes the highest overhead across all execution platforms—twice as much as BM for `Large` instance type. However, by increasing the number of CPU cores, this behavior is changed such that the mean execution time offered by vanilla CN approaches BM. As a similar behavior is observed for other application types, we defer analysis of this observation to Section IV.

ii. Even though VMCN platforms (vanilla and pinned) include one more layer of virtualization in compare with VM platforms, they impose a slightly lower overhead. Further analysis within VM platforms shows that the pinned VM consistently imposes a lower overhead than the vanilla VM. Building upon these two observation, we hypothesize that, for IO intensive applications, both pinning and containerization are decisive factors in mitigating the imposed overhead of the virtualized platform. In curtail, the reason that pinning remarkably mitigates the execution time overhead is a more efficient use of the cache and the possibility to pin virtualized platforms on CPU slots based on their IO affinity [41]. Alternatively, in a non-pinned (*i.e.,* vanilla) scenario, at each time slot, the virtualized platform is allocated on a different set of processing cores by the scheduler that may not favor IO affinity and implies reestablishing the cache. We elaborate this analysis further in Section IV with respect to other application types as well.

4) NoSQL Workload using Apache Cassandra

Apache Cassandra [21] is a distributed NoSQL database extensively used to handle Big Data in the cloud. We evaluate it in this study as an application type that demands compute, memory, and IO. We configured Cassandra exclusively on one execution platform and used its native stress tool, *Cassandra-stress* [42] [21], to submit 1,000 synthesized database operations within one second. A set of 100 threads, each one simulating one user, were spawned by Cassandra-stress. To make the imposed overhead stand out, we put Cassandra under extreme pressure by forcing a quarter of the synthesized requests as the *write* operations and the rest as the *read* operations. Then, we calculated the average execution time (a.k.a response time) of all the synthesized operations. To capture the randomness in the results, caused by the execution platforms, we conducted the experiment 20 times for each instance type and the mean and 95% confidence interval of the results are reported.

Results of this evaluation are shown in Figure 6. Note that, for the `Large` instance type, the system is overloaded and thrashed and the results are out of range. As such, to be able to concentrate on the imposed overhead of other instance types, we excluded the results of the `Large` type. Our observations and analysis for this experiment are as follows:

i. Vanilla CN imposes the largest overhead—3.5 times or more with respect to BM. This overhead is even higher than the similar phenomenon we observed for

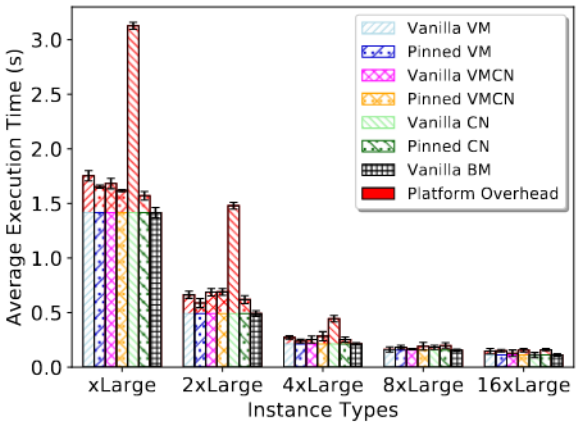


Fig. 5: Comparing mean response time (aka execution time) of 1,000 web processes on different execution platforms (WordPress evaluation). The horizontal axis represents the number of CPU cores in the form of different instance types and the vertical axis shows the mean execution time (in seconds).

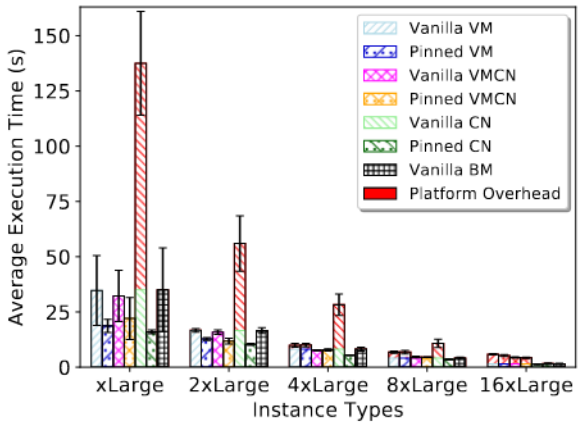


Fig. 6: Comparing mean execution time (aka response time) of Cassandra workload (in seconds) on different execution platforms. Horizontal axis represents the number of CPU cores in the form of different instance types. Note that the execution time for the `Large` instance type is out of range and unchartable.

WordPress (in Figure 5). The reason for this behavior is the higher volume of IO operations in Cassandra rather than WordPress. However, the CN overhead diminishes for instances with larger number of cores. As we had similar observations for other application types, we defer further analysis of this phenomenon to Section IV.

ii. In contrast to vanilla CN, pinned CN imposes the lowest overhead. This confirms our earlier observations on the positive impact of pinning on IO intensive applications. Surprisingly, we see that for `×Large`—`4×Large` instance types, pinned CN can even beat BM. The reason is the BM scheduler is oblivious to IO affinity and the extreme volume of IO operations makes BM perform worse than the pinned CN. For the same reason, we can see that offering a lower execution time is not limited to only pinned CN, but it is also noticed in any pinned virtualized platform.

iii. By increasing the number of cores, the impact of pinning is diminished to the extent that for `8×Large` and `16×Large` instance types, there is no improvement between the pinned virtualized platforms and their vanilla counterparts. We believe this is because in virtualized platforms with larger instance types: (A) there are fewer scheduling options within the host machine, hence, the scheduling overhead is mitigated; (B) cache and IO affinity are improved, because in each scheduling time slot, it is likely that a virtualized platform is assigned to the same set of processing cores.

iv. For all VM-based platforms whose instance type is `8×Large` and beyond, there is an increased overhead with respect to BM. As noted in the previous point, for larger instances, the overhead of IO diminishes, hence, the execution time is dominated by the CPU processing time. This makes the situation similar to CPU intensive application types (*e.g.,* FFmpeg) where VM-based platforms impose a considerable overhead.

IV. Cross-Application Overhead Analysis

Building upon application-specific observations conducted in the previous section, in this part, we further analyze the root causes of the imposed overhead by various virtualization platforms across different application types. We also carry out additional experiments to verify our findings.

By comparing the results and studying overhead behavior across all application types and execution platforms, the following two categories of the imposed overhead can be distinguished:

1) Platform-Type Overhead (PTO)

This kind of overhead pertains to the type of virtualized platform and its ratio remains constant, irrespective of the instance type it is deployed on. For instance, in both FFmpeg (Figure 3) and Cassandra (for instance types greater than `4×Large` in Figure 6), the overhead ratio of VM remains the same across various instance types. This type of overhead is caused by the abstraction layers of the virtualized platforms and are reflected when CPU intensive applications (*e.g.,* FFmpeg) are deployed. As the source of this type of overhead pertains to virutalization layers, pinning cannot mitigate it drastically. This type of overhead is a known issue and has been studied in prior works [4], [43], [44].

2) Platform-Size Overhead (PSO)

This type of overhead is diminished by increasing the number of cores assigned and is specific to vanilla CN platform. PSO is observed throughout all the studied application types, particularly, when a container is assigned a small portion of the host CPU cores. Our hypothesis is that the ratio of the container cores to the host cores is a decisive factor on the magnitude of the imposed overhead. This behavior of containers has also been identified and reported for Docker by IBM [45]. In particular, for IO-bound applications, the overhead even exceeds VM platforms. As an instance, for `Large` instance type, the overhead ratio of vanilla CN to vanilla VM for WordPress application is 2.4 (see Figure 5) and it is 3.7 for Cassandra (see Figure 6). Importantly, pinning can considerably mitigate this type of overhead. Since this type of overhead has not been investigated before, we elaborate on its root causes in the next subsections.

A. The Impact of Container-to-Host Core Ratio (CHR) on PSO

To analyze the impact of container size on PSO, for a given container, we define *Container-to-Host Core Ratio* (CHR) as the ratio of its assigned cores to the total number of host cores. To evaluate the impact of CHR, we choose FFmpeg as the application type, because it does not impose additional IO overhead and our analysis is concentrated on PSO. We configure a CN platform of `4×Large` type on two homogeneous hosts, with 16 and 112 cores, respectively. Then, we measure the mean execution time of the FFmpeg workload (described in Section III) on these configurations.

Results of this experiment are shown in Figure 7. The first set of bars represent CHR=1 and the second set represent CHR=0.14. In addition to the CN platform, we report the result for the BM platform with 16 cores. This enables us

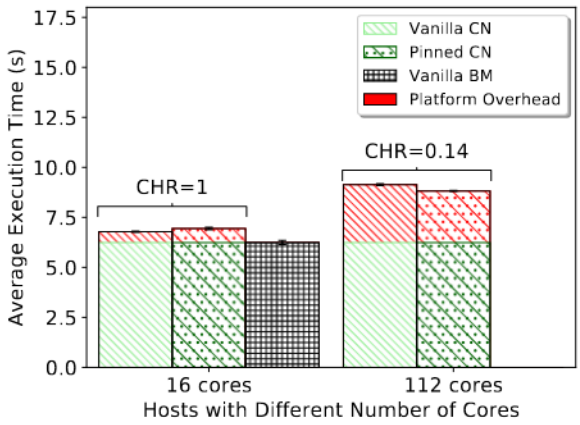


Fig. 7: Evaluating the impact of CHR on the overhead of a vanilla and a pinned CN platform on two homogeneous hosts with 16 and 112 cores. The vertical axis shows the mean execution time (in seconds) and the horizontal axis shows the host's number of cores.

to study the overhead of a CN platform that is as big as the host machine.

In this figure, we observe that although the CN platforms have the same number of cores, on the larger host (with 112 cores) they impose a higher overhead in compare with the case that they are deployed on the smaller host. We can conclude that a container with a lower value of CHR imposes a larger overhead (PSO) to the system. As illustrated in the figure, pinning does not imply any statistically significant difference in the imposed overhead.

Throughout Section III, we repeatedly observed the impact of increasing CHR on mitigating PSO across different application types. That is, for larger CN platforms, the execution time approaches BM, even more rapidly than other virtualized platforms.

The question arises in this case is that, for a given container that processes a certain application type, how to know the suitable value of CHR? In this work, we address this question based on our observations for different application types. However, we note that answering this question theoretically is an interesting future research work. In WordPress, the PSO starts to vanish when the CN is configured in the range of [4×Large , 8×Large] that implies $0.14 < CHR < 0.28$. A similar observation was made in the IBM report for their proprietary IBM WebSphere web server [45]. Similar analysis for FFmpeg and Cassandra indicate suitable CHR should be in the range of $0.07 < CHR < 0.14$ and $0.28 < CHR < 0.57$, respectively. From this analysis, we conclude that IO intensive applications require a higher CHR value than the CPU intensive ones.

The estimated ranges of CHR can be used by cloud administrators to configure containers such that their imposed overhead is minimized. This can potentially benefit both cloud users (by reducing their incurred cost and response time) and cloud providers (by lowering their energy consumption).

B. The Impact of Container Resource Usage Tracking on PSO

Although CHR explains the diminishing PSO for containers with larger instances, it does not explain the high PSO of small vanilla CNs (*i.e.,* those with low CHR) that we observed for all task types. Recall (from Figures 3, 5, and 6) that pinning remarkably mitigate the PSO for low-CHR CNs. Accordingly, our hypothesis is that the high PSO is attributed to CPU provisioning model.

We observed in our experiments that, for small vanilla CN (those with 2 cores), OS scheduler allocates all available CPU cores of the host machine (112 cores) to the CN process. In this circumstance, cgroups has to assure that the cumulative CPU usage of the process does not exceed its designated quota. This means that in each scheduling event, vanilla CN undergoes the overhead of both OS scheduling (that implies process migration) and cgroups (for resource usage tracking). We realized that cgroups is an atomic (kernel space) process [31] [32] and each invocation of it implies one transition from the user-mode to the kernel-mode, which incurs a considerable overhead. Furthermore, we noticed that, in amortizing the process of a small container across all available CPU cores, the process footprint on each core is a tiny fraction of the whole process. Specifically, for small containers, we observed that the overhead of cgroups tasks reaches to the point that it dominates the container process. In particular, because cgroups is an atomic process, the container has to be suspended, until tracking and aggregating resource usage of the container is complete.

In contrast to the vanilla mode, CPU pinning avoids the overhead of host OS scheduling and cgroups. In the presence of pinning, as the allocated set of processing cores do not change at each scheduling event, there is less demand for cgroups invocation, hence, the imposed overhead is mitigated.

C. The Impact of IO Operations on PSO

The comparison of FFmpeg (particularly, ×Large in Figure 3) against WordPress and Cassandra (particularly, ×Large in Figures 5 and 6) illustrates that, in both WordPress and Cassandra, the PSO of vanilla CN (*i.e.,* the overhead part, colored in red) is drastically higher than FFmpeg. As both WordPress and Cassandra are IO-bound applications, our hypothesis is that the extra overhead pertains to performing the IO operations.

As a CPU-bound application, FFmpeg has a predictable behavior and fully utilizes their CPU quota in each time slot. In contrast, the IO-bound applications often do not make a complete use of their CPU quota, because they are interrupted and switch to the *pending* state, until they complete their IO operation. For example, in WordPress, each web request triggers at least three Interrupt Requests (IRQs): to read from the network socket; to fetch the requested HTML file from disk; and to write back to the network socket. Each IRQ implies the overheads to accomplish a set of scheduling actions (to enqueue, dequeue, and pick the next task) and transitioning to the kernel mode (to perform the IO operation).

Once an interrupt is served, to avoid cache line bouncing and reestablishing IO channels, OS scheduler makes its best effort to resume the interrupted tasks on the same set of cores. However, in the event that the process is assigned to a different set of cores, a significant overhead is imposed to reload L1 and L2 caches and establish new IO channels. As noted in the previous section, vanilla CN instances with smaller CHR are more prone to be allocated on a different set of cores, hence, experiencing even a more significant overhead, when compared with larger instances.

D. The Impact of Multitasking on PSO

Analysis of the overhead across different application types, in particular FFmpeg versus WordPress, brings us to the hypothesis that the number of processes increases the imposed PSO. To verify the hypothesis and figure out the importance of this factor, in this part, we conduct an experiment to analyze the impact of number of FFmpeg processes on its PSO. The reason that we compare FFmpeg with itself is to eliminate the impact of differences in application characteristics and, instead, single out the impact of number of processes.

For this experiment, we examined FFmpeg on 4×Large CN instance types to change the codec of the video file used in Section III-B1. We studied two scenarios: (A) the source video

file is a 30-second (large) video; and (B) splitting the same source video into 30 video files of the same size (one-second each) and process them in parallel.

Comparing the results in Figure 8 approves our hypothesis and shows that the number of processes results in increasing the PSO of CN platforms. This is because, a higher degree of multitasking increases the overhead imposed by OS scheduler and cgroups to collect resource usage of CNs.

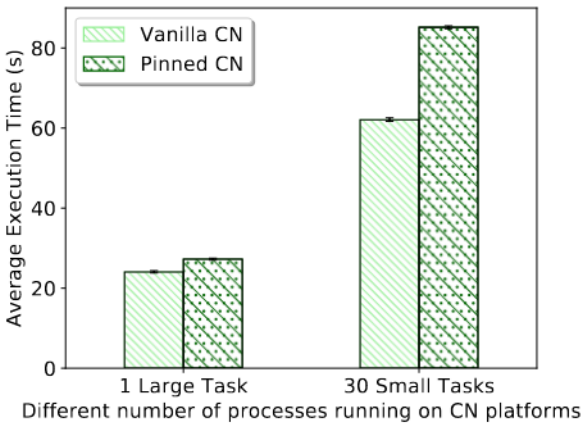


Fig. 8: Comparing the impact of number of processes on the imposed overhead of `4×Large` CN instance. The vertical axis shows the mean executing times (in Seconds) and the horizontal axis shows processing of a source video file in two cases: one large video versus partitioning it into 30 small videos.

V. Related Work

Container technology has gained a major attention over the past few years, thus, several research works have been undertaken to evaluate their performance in contrast to VMs. Xavier *et al.* [46] compared the Linux VServer, OpenVZ, and Linux Containers (LXC) with Xen hypervisor using synthetic workloads such as LINPACK, STREAM and IOzone. They observed that containers can provide a near-native performance for the mentioned workloads. However, this work does not consider popular cloud applications and platforms. In another study, Shirinbab *et al.* [42] compare the performance of Docker, VMware ESXi, and Bare-Metal. They evaluated the performance of these platforms for Apache Cassandra database. Their work shows that in terms of disk I/O, containers incur a performance penalty, specially for disk write operations.

In [47], authors evaluated the network overhead of Docker and Singularity containers using HPCG and miniFE workloads. Rudyy *et al.* [48] also compared three different container technologies on computational fluid dynamics (CFD) and concluded Singularity is the suitable container solution for HPC workloads that provide the same execution time as Bare-Metal. Mavridis and Karatza [49] studied containers on VMs configuration. Similar to [46], they evaluated Docker on top of KVM VMs using well-known benchmarking tools and explored the overhead in comparison with Bare-Metal. They concluded adding another virtualization layer that helps for easier management and system maintenance, but incurs a performance cost.

Several research works compared containers versus VMs and Bare-Metal [2]–[5], [42], [46]–[48]. Only few of these studies explored VMCN configuration, which is a popular platform in the cloud [49]. In addition, majority of the comparisons are performed using synthetic workloads and benchmarking tools such as LINPACK, STREAM, and NetPerf that exhibit different behavior than real cloud-based workloads.

VI. Summary and Best Practices

In this study, the performance overhead imposed by different virtualized platforms commonly used in the cloud was studied. Four popular application types, namely video processing and parallel processing (MPI), web processing, and No-SQL were studied. The study revealed that: **(A)** application characteristic (*i.e.,* IO-bound versus CPU-bound) is decisive on the imposed overhead of different virtualization platforms. **(B)** CPU pinning can reduce the overhead of virtualized platforms, particularly for IO-bound applications running on containers. **(C)** The proportion of container cores to the host cores (we named it CHR) plays a significant role on the overhead of containers. Greater CHR values reduce the overhead. **(D)** Although containers are known to impose a lower overhead than VMs, the opposite was observed for containers with very low CHRs. **(E)** Containers on top of VMs (called VMCN) impose a lower overhead for IO intensive applications. **(F)** Increasing the level of multitasking amplifies the overhead of containers.

In addition, this study provides the following set of **Best Practices** that can help cloud architects to efficiently configure cloud systems based on their application types:

1) Avoid instantiating small vanilla containers (with one or two cores) for any type of application.
2) For CPU intensive applications (*e.g.,* FFmpeg), pinned containers impose the least overhead.
3) If VMs are being utilized for CPU-bound applications, do not bother pinning them. It neither improves the performance, nor decreases the incurred cost.
4) For IO intensive applications, if pinned container is not a viable option, then use container within VM (VMCN). It imposes a lower overhead than a VM or a vanilla container.
5) To minimize the overhead of containers, for CPU intensive applications configure them with $0.07 < CHR < 0.14$ and for IO intensive applications use $0.14 < CHR < 0.28$. If the application is ultra IO intensive (*e.g.,* Cassandra), even a higher CHR ($0.28 < CHR < 0.57$) is suggested.

In the further, we plan to extend the study to incorporate the impact of network overhead. In addition, we plan to provide a mathematical model to measure the overhead of a given

virtualization platforms based on the isolation level it offers.

Acknowledgment

We thank reviewers of the manuscript. The research was supported by the Louisiana Board of Regents under grant number LEQSF (2017- 20)-RD-B-06, Perceptive Intelligence LLC, and Amazon Cloud (AWS) research credit.

References

[1] M. Parashar, M. AbdelBaky, I. Rodero, and A. Devarakonda, "Cloud paradigms and practices for computational and data-enabled science and engineering," *Computing in Science & Engineering*, vol. 15, no. 4, pp. 10–18, Jul. 2013.

[2] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, "Performance overhead comparison between hypervisor and container based virtualization," in *Proceedings of the 31st IEEE International Conference on Advanced Information Networking and Applications*, ser. AINA '17, Mar. 2017.

[3] R. Morabito, J. Kjllman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *Proceedings of the IEEE International Conference on Cloud Engineering*, Mar. 2015.

[4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Proceedings of the IEEE international symposium on performance analysis of systems and software*, ser. ISPASS '15, Mar. 2015, pp. 171–172.

[5] R. K. Barik, R. K. Lenka, R. K. Rahul, and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," in *Proceedings of the IEEE International Conference on Computing, Communication and Automation*, ser. ICCCA '16, Apr. 2016.

[6] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the USENIX Annual Technical Conference 2018*, ser. USENIX ATC 18, Jul. 2018, pp. 133–146.

[7] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proceedings of the IEEE International Conference on Cloud Engineering*, ser. (IC2E '18), Apr. 2018, pp. 159–169.

[8] Amazon web services. [Online]. Available: https://aws.amazon.com/lambda/

[9] Azure service fabric, https://azure.microsoft.com/en-us/services/service-fabric/. [Online]. Available: https://azure.microsoft.com/en-us/services/service-fabric/

[10] "DELL EMC Unity XT All-Flash Unified Storage," Accessed on 2020 Mar 14. [Online]. Available: https://www.delltechnologies.com/en-us/storage/unity.htm

[11] A. F. Nogueira, J. C. Ribeiro, M. Zenha-Rela, and A. Craske, "Improving la redoute's ci/cd pipeline and devops processes by applying machine learning techniques," in *Proceedings of the 11th International Conference on the Quality of Information and Communications Technology*, ser. QUATIC 18, Sep. 2018, pp. 282–286.

[12] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in iot context: Horizontal and vertical linux container migration," in *Proceedings of the Global Internet of Things Summit*, ser. GIoTS '17, Jun. 2017, pp. 1–4.

[13] J. Zhang, G. Wu, X. Hu, and X. Wu, "A distributed cache for hadoop distributed file system in real-time cloud services," in *Proceedings of the 13th ACM/IEEE International Conference on Grid Computing*, ser. GRID '12, Sep. 2012, pp. 12–21.

[14] C. Yu and F. Huan, "Live migration of docker containers through logging and replay," in *Proceedings of the 3rd International Conference on Mechatronics and Industrial Informatics*, ser. ICMII '15, Oct. 2015.

[15] Y. Qiu, C.-H. Lung, S. Ajila, and P. Srivastava, "Lxc container migration in cloudlets under multipath tcp," in *Proceedings of the 41st IEEE Annual Computer Software and Applications Conference*, ser. COMPSAC '17, vol. 2, Jul. 2017, pp. 31–36.

[16] J. Bi, H. Yuan, W. Tan, M. Zhou, Y. Fan, J. Zhang, and J. Li, "Application-aware dynamic fine-grained resource provisioning in a virtualized cloud data center," *IEEE Transactions on Automation Science and Engineering*, vol. 14, no. 2, pp. 1172–1184, Apr. 2017.

[17] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency," in *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 1–10.

[18] H. Zeng, Z. Zhang, and L. Shi, "Research and implementation of video codec based on ffmpeg," in *Proceedings of 2nd International Conference on Network and Information Systems for Computers*, ser. ICNISC '16, Apr. 2016, pp. 184–188.

[19] W. Gropp, R. Thakur, and E. Lusk, *Using MPI-2: Advanced features of the message passing interface*. MIT press, 1999.

[20] S. K. Patel, V. Rathod, and J. B. Prajapati, "Performance analysis of content management systems-joomla, drupal and wordpress," *International Journal of Computer Applications*, vol. 21, no. 4, pp. 39–43, May 2011.

[21] V. Abramova and J. Bernardino, "Nosql databases: Mongodb vs cassandra," in *Proceedings of the International C* Conference on Computer Science and Software Engineering*, Jul. 2013, pp. 14–22.

[22] R. Buyya, C. Vecchiola, and S. Selvi, *Mastering Cloud Computing, Foundations and Applications Programming*, Apr. 2013.

[23] M. Amini Salehi, B. Javadi, and R. Buyya, "Resource provisioning based on leases preemption in InterGrid," in *Proceeding of the 34th Australasian Computer Science Conference*, ser. ACSC '11, 2011, pp. 25–34.

[24] M. A. Salehi and R. Buyya, "Contention-aware resource management system in a virtualized grid federation," in *PhD Symposium of the 18th international conference on High performance computing*, ser. HiPC '11, Dec. 2011.

[25] Amazon, "AWS Nitro System," Accessed on 2020 Mar. 14. [Online]. Available: https://aws.amazon.com/ec2/nitro/

[26] "HCI: Hyper Converge Infrastructure," Accessed on 2020 Jan. 09. [Online]. Available: https://en.wikipedia.org/wiki/Hyper-converged_infrastructure

[27] "Nutanix: Hyper converge infrastructure," Accessed on 2020 Jan. 09. [Online]. Available: https://www.nutanix.com/

[28] "Maxta: Hyper converge infrastructure," Accessed on 2020 Jan. 09. [Online]. Available: https://www.maxta.com/

[29] "Cloudistics: Hyper converge infrastructure," Accessed on 2020 Jan. 09. [Online]. Available: https://www.cloudistics.com/

[30] Y. Goto, "Kernel-based virtual machine technology," *Fujitsu Scientific and Technical Journal*, vol. 47, no. 3, pp. 362–368, Jul. 2011.

[31] "Cgroup in kernel.org." [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[32] J. Bacik, "IO and cgroups, the current and future work." Boston, MA: USENIX Association, Feb. 2019.

[33] C. Wong, I. Tan, R. Kumari, J. Lam, and W. Fun, "Fairness and interactive performance of o (1) and cfs linux kernel schedulers," in *Proceedings of the International Symposium on Information Technology*, vol. 4, Aug. 2008, pp. 1–8.

[34] "BPF Compiler Collection (BCC), kernel tracing tool," Accessed on 2020 Jan. 09. [Online]. Available: https://github.com/iovisor/bcc

[35] W. Marshall, "Boot with grub," *Linux Journal*, vol. 2001, no. 85es, pp. 8–es, 2001.

[36] X. Li, M. A. Salehi, M. Bayoumi, N.-F. Tzeng, and R. Buyya, "Cost-Efficient and Robust On-Demand Video Stream Transcoding Using Heterogeneous Cloud Services," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, no. 3, pp. 556–571, Mar. 2018.

[37] X. Li, M. A. Salehi, Y. Joshi, M. K. Darwich, B. Landreneau, and M. Bayoumi, "Performance analysis and modeling of video transcoding using heterogeneous cloud services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, p. 910922, Apr. 2019.

[38] "SEARCH_MPI: Parallel integer search," Accessed on 2020 Jan. 09. [Online]. Available: https://people.sc.fsu.edu/~jburkardt/c_src/search_mpi/search_mpi.html

[39] "PRIME_MPI: Parallel prime search withing a specific range," Accessed on 2020 Jan. 09. [Online]. Available: https://people.sc.fsu.edu/~jburkardt/c_src/prime_mpi/prime_mpi.html

[40] "Apache Jmeter, load and performance test tool for many different applications/server/protocol types," Accessed on 2020 Jan. 09. [Online]. Available: https://jmeter.apache.org/

[41] Z. Guo and Q. Hao, "Optimization of KVM Network Based on CPU Affinity on Multi-Cores," in *Proceedings of the International Conference of Information Technology, Computer Engineering and Management Sciences - Volume 04*, ser. ICM 11, Sep. 2011, p. 347351.

[42] S. Shirinbab, L. Lundberg, and E. Casalicchio, "Performance evaluation of container and virtual machine running cassandra workload," in *Proceedings of the 3rd International Conference of Cloud Computing Technologies and Applications*, ser. CloudTech '17, Oct. 2017, pp. 1–8.

[43] M. A. Salehi, B. Javadi, and R. Buyya, "Resource provisioning based on preempting virtual machines in distributed systems," *Concurrency and Computation: Practice and Experience (CCPE)*, vol. 26, no. 2, pp. 412–433, Feb. 2014.

[44] M. A. Salehi, A. N. Toosi, and R. Buyya, "Contention management in federated virtualized distributed systems: implementation and evaluation," *Software: Practice and Experience (SPE)*, vol. 44, no. 3, pp. 353–368, Mar. 2014.

[45] "IBM Knowledge Center, Websphere Commerce, Docker Performance Tuning," Accessed on 2020 Jan. 09. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSZLC2_9.0.0/com.ibm.commerce.admin.doc/concepts/cpmdockertune.htm

[46] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proceedings of*

the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Feb. 2013, pp. 233–240.

[47] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," in *Proceedings of the Practice and Experience on Advanced Research Computing*, Jul. 2018, p. 11.

[48] O. Rudyy, M. Garcia-Gasulla, F. Mantovani, A. Santiago, R. Sirvent, and M. Vázquez, "Containers in hpc: A scalability and portability study in production biological simulations," in *Proceedings of the International Parallel and Distributed Processing Symposium*, ser. IPDPS '19, May 2019, pp. 567–577.

[49] I. Mavridis and H. Karatza, "Performance and overhead study of containers running on top of virtual machines," in *Proceedings of the 19th Conference on Business Informatics*, ser. CBI '17, vol. 2, Jul. 2017, pp. 32–38.