Performance evaluation of containers for HPC

Cristian Ruiz, Emmanuel Jeanvoine, Lucas Nussbaum

Performance evaluation of containers for HPC

Cristian Ruiz, Emmanuel Jeanvoine and Lucas Nussbaum

Inria, Villers-lès-Nancy, F-54600, France
Université de Lorraine, LORIA, F-54500, France
CNRS, LORIA - UMR 7503, F-54500, France

Abstract. Container-based virtualization technologies such as LXC or Docker have gained a lot of interest recently, especially in the HPC context where they could help to address a number of long-running issues. Even if they have proven to perform better than full-fledged, hypervisor-based, virtualization solutions, there are still a lot of questions about the use of container solutions in the HPC context. This paper evaluates the performance of Linux-based container solutions that rely on *cgroups* and *namespaces* using the NAS parallel benchmarks, in various configurations. We show that containers technology has matured over the years, and that performance issues are being solved.

Keywords: HPC, virtualization, containers, NAS parallel benchmarks.

1 Introduction

The use of containers has been popularized during the last years as a lightweight virtualization solution. Briefly, this approach brings several benefits. First, using containers allows one to embed a full software stack in order to run an application in various contexts to enforce portability. Like classical virtualization, running applications inside containers enables isolation from the host system and from other containers. Administrator rights can therefore be assigned to users inside a container without impact on the host. On the top of these advantages, several projects surf on this trend, like the popular Docker project. Typically, containers are commonly used by software developers so that all members of a project can test the code within the same software environment. Containers are also used for demonstration purposes when applications need the deployment of a complex ecosystem, and more recently, for production purposes since it is a keystone for a clean deployment of several services.

Beyond classical uses of containers, we have investigated the interest of such an approach for experimental purposes. In a nutshell, and in addition of the benefits mentioned before, using containers to achieve experiments has other advantages. Embedding the full software stack required by an application allows to perform repeatable and reproducible experiments. Indeed, assuming that the source code of the application is available, as well as the software stack shipped in a container, anyone is able to reproduce an old experiment in conditions that are similar to the original ones (provided that it is possible to run it on the same,

or on similar hardware). In the field of distributed systems, using containers also has the advantage to oversubscribe resources with a minor overhead, compared to classical virtualization solutions. Thus, running dozens of virtual nodes on physical nodes is an easy task that allows to emulate platforms with an higher scale than they physically have.

If containers ease the way to perform experiments on distributed systems, we wanted to study the impact of using containers on the realism of experiments. Indeed, we will see in this paper that containers use several low level features of the system that might induce some modifications of code execution compared to an execution on real hardware. More specifically, we focus on a particular class of applications: the HPC applications. Such applications would really benefit from running inside containers since the software stack required is usually complex, involving a communication middleware and several solver libraries. They are likely to be influenced by the versions of their dependencies and by their compiler. Running different HPC applications on the same platform, with a single software stack is tricky and in some cases might not be possible. As far as they use very strict communication patterns and precise memory exchanges, the behavior of HPC applications is likely to be affected, and possibly in a bad way.

There are two important aspects that should be evaluated: isolation and performance as discussed in [18]. However, in this paper we focus on peformance evaluation. The goal is to study the impact of using containers when running HPC applications, and more precisely to bring an answer to the following question: *does it make sense to use containers in the context of HPC?*.

The rest of the paper is organized as follows. Section 2 briefly presents the various virtualization solutions and details the internals of container-based solutions. Then, Section 3 explores the related work that aim at evaluating virtualization solutions in the context of HPC. Section 4 presents an experimental evaluation of using containers for various HPC applications using NAS benchmarks [1]. Finally, Section 5 concludes the paper and presents the future work.

2 Context: virtualization and containers

Virtualization can trace its roots back to the mainframes of the 1960's and 1970's, but the idea has evolved a lot over time. The general purpose is to simulate the execution of several computers on a single one. The computer where the virtualization takes place is called *host* and the simulated computers are called *guests*. Basically, two families of virtualization can be distinguished: hardware-level virtualization and OS-level virtualization.

Hardware-level virtualization is what is usually meant when speaking about virtualization. An hypervisor, running either in the host operating system or in the hardware, is dedicated to executing and managing the guest virtual machines. Various ways to achieve virtualization, providing various trade-offs between performance and flexibility, have been designed over the years [2]. One important point is the interface offered to guests: in some cases, the virtualization solution will simulate real existing hardware, enabling guests running an

unmodified operating system; in others (with e.g. paravirtualization in Xen or KVM's *virtio* devices), the virtualization solution exposes abstract devices that can provide better performance thanks to a reduced overhead, but require the guest operating system to be aware of the virtualization solution.

OS-level virtualization is usually called container-based virtualization. Here, the host kernel allows the execution of several isolated userspace instances that share the same kernel but possibly run a different software stack (system libraries, services, applications). Linux-VServer and OpenVZ are two early attempts to provide such containers in Linux. Both are out-of-tree patches (not included in the *vanilla* Linux kernel). More recently, several efforts have been carried out to provide the required features in the standard Linux kernel. First, *cgroups* (control groups), introduced in 2006 (Linux 2.6.24) can be used to group processes and limit their resources usage. Second, *namespace isolation* can be used to isolate a group of processes at various levels: networking (*network namespace*, to allocate a specific network interface to a container), filesystem (*mount namespace*, similar to chroot), users (*user namespace*), process identifiers (*PID namespace*), etc.

Several containers solutions have been developed on top of *cgroups* and *namespaces*: LXC, Google's lmctfy, Docker, *systemd-nspawn*. It is worth noting that those solutions differentiate on how containers and images are managed (downloaded, created, etc.) but that they all use the same underlying kernel interfaces. Their efficiency and their interest in the context of HPC, which are the focus of the present paper, are thus independent of the management solution used. As will be explained in section 4, the experiments described in this paper were performed using Distem [12], our own emulation solution that leverages LXC. However, it was mainly used to facilitate the management of containers on hosts, and emulation features were disabled.

3 Related work

There has been a large number of attempts at evaluating the interest and the usability of all virtualization solutions for HPC. For example, In [19], Youssef et al. evaluated Xen using HPC Challenge benchmarks and LLNL ASC Purple benchmarks, with up to four nodes. In [9], Xen and KVM are compared in both paravirtualized and full virtualization modes using micro-benchmarks and the HPC Challenge benchmarks. Another study [10] focuses on the I/O performance of Xen, KVM and OpenVZ. Part of the evaluation is done using the NAS parallel benchmarks.

Public Cloud platforms have also been evaluated. Amazon EC2 has been evaluated [7] using Intel MPI benchmarks using clusters of up to 16 nodes, and Microsoft Azure has been evaluated for scientific applications [15].

The performance of container solutions have been the focus of less work. An early work [8] compared the performance of VMWare, Xen, Solaris containers and OpenVZ using custom benchmarks. In [5], the I/O performance of Docker is evaluated using MySQL. The extended version of that work [6] includes eval-

uations using Linpack, Stream, RandomAccess, nuttcp, netperf, fio, Redis and MySQL, and shows that Docker exceeds KVM performance in every tested case. However, all the tests were performed on a single machine. In [17], VMWare Server, Xen and OpenVZ are compared using NetPerf, IOZone, and the NAS Parallel Benchmarks. OpenVZ is shown to be the solution with performance close to the native one. A last study [18] includes evaluations with LXC. Linux VServer, OpenVZ, LXC and Xen are compared using the HPC Challenge benchmarks and the NAS Parallel Benchmarks on up to four nodes. That work outlines similar performance between all containers-based solutions, consistently better than Xen. This paper differentiates from the previous one in that we evaluated the following points: performance gains with different version of Linux kernel, overhead in the presence of oversubscription for executing HPC workloads and overhead of containers under a high HPC workload. Our experimental setup included up to 64 machines which aims at evaluating loads and configurations expected to be found in HPC environments.

4 Experimental Evaluation of Containers

In this section, we want to answer the following question: *what is the impact of using container-based virtualization to execute HPC workloads?* We split this question into three more specific ones:

- Q1. What is the overhead of oversubscription using different versions of Linux kernel?
- Q2. What is the performance of inter-container communication?
- Q3. What is the impact of moving an HPC workload with several MPI processes per machine, to containers?

It has already been demonstrated that in terms of computation time, OS-level virtualization techniques have almost zero overhead [18]. However, the use of virtual network device will certainly introduce an overhead into the computation time and network performance, due to the additional processing required by the Linux kernel. Therefore, the goal is to measure this overhead when executing HPC workloads. The overhead was measured performing three different experiments:

- The first experiment shows factors that affect the overhead introduced to the execution of HPC workloads: Linux kernel version, virtual network device and number of containers run on the machine (oversubscription).
- The second experiment measures the overhead caused by inter-container communication.
- The third experiment measures the overhead caused by the virtual network interconnection.

4.1 Experimental setup

Experiments were conducted using the Grid'5000 Testbed[3], on the *paravance* cluster located in the *rennes* site. Each node of this cluster is equipped with two Intel Xeon E5-2630v3 processors (with 8 cores each), 128 GB of RAM and a 10 Gigabit Ethernet adapter. Regarding the software stack, we used: Debian Jessie, Linux kernel versions: 3.2, 3.16 and 4.0, TAU version 2.23.1, OpenMPI version 1.6.5 and NPB version 3.3. We wrote recipes[1] to install the necessary software using Kameleon[11], which automates and ensures the installation of the same software stack on the containers and on the real machines.

We instrumented the benchmarks: LU, EP, CG, MG, FT, IS from NPB benchmark suite[1] using TAU [13] in order to carry out the evaluation. Each benchmark exhibits different communication patterns: EP communicates few times using `MPI_Reduce`, IS and FT use all-to-all communication, CG uses a one dimensional chain pattern, LU and MG a ring pattern. These behaviors were characterized in [14]. Table 1 shows the percentage of CPU and communication time observed for all the benchmarks used. In all the experiments, resources are not over-provisioned which means that the number of containers deployed are less or equal to the number of cores present on the physical machines where containers are deployed on. Each experiment is run 20 times and mean values are plotted together with a 95% confidence interval.

Linux provides many different ways to bring networking to containers. First, physical interfaces can be dedicated to a specific container (with LXC's *phys* mode). But there are also several ways to share a physical network interface between several containers. A single NIC can emulate several different NICs using either Linux's *macvlan* support, or hardware SR-IOV support in the NIC. Or the single NIC can be connected to a software bridge, to which containers will be connected using Linux *veth* devices pairs. In this paper we only use this latter configuration, as it is both the default and the most flexible way to achieve networking with containers solutions such as LXC and Docker. However, we plan to compare this *veth + bridge* setup with other options in our future work.

4.2 Linux kernel version and oversubscription

In this section, we show the impact of the three following factors: oversubscription using containers, virtual network interface and Linux kernel version. The performance of the virtual network interface is evaluated under a mix of communication that takes place between containers hosted on the same machine (*inter-container communication*) and containers hosted in different machines. Research question *Q1* is addressed in this section.

Setup: we used 8, 16, 32 and 64 physical machines where we deploy from 1 to 8 containers per physical machine. We run the chosen benchmarks both inside the containers and natively on the machines using several versions of Linux kernel: 3.2, 3.16 and 4.0. Here, an under-provisioned HPC workload is

[1] https://github.com/camilo1729/distem-recipes

used which means that the machines are not fully used, just one MPI process per machine or container (depending on the experiment).

Results: Figure 1a and Figure 1b show the execution time of the benchmarks executed on: a) *native* using 32 physical machines and b) 32 containers. The 32 containers were hosted using different number of physical machines (from 8 to 32). Figure 1a shows the behavior observed with the CG benchmark, which is representative of what happens with almost all benchmarks; the kernel 3.2 introduced a prohibitive performance degradation in the execution time of the benchmarks which appears when more than two containers are deployed per physical machine. Figure 1b shows an unexpected result for the benchmark EP where the Linux kernel 3.2 shows a better behavior. Deep kernel inspection has to be performed here to understand exactly what happens. Overall, we observed a maximum performance gain of around 1577 % when passing from 3.2 to 3.16 and 22 % when passing form 3.16 to 4.0. These values were calculated after removing the performance gains observed running the benchmarks natively.

The overhead of changing the number of physical machines (from 8 to 64) that host the containers is shown in Figure 1c for Linux kernel 4.0. The most affected benchmarks are MG, LU and FT. Regarding the benchmark FT the overhead comes from the fact that it uses blocking all to all communication which generates a lot of traffic. The other two benchmarks are affected because their memory access pattern provokes cache misses when several processes are allocated on the same physical machine. The highest overhead observed ranges from 15 % to 67 % and corresponds when 8 containers are deployed per physical machine.

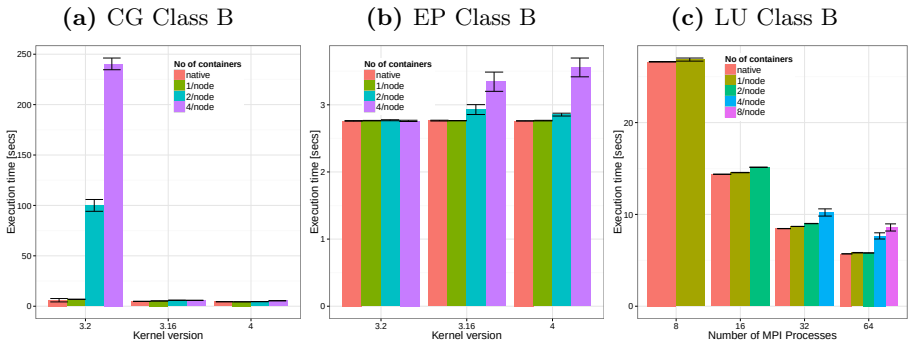


Fig. 1: Kernel version overhead using up to 64 nodes.

Conclusions: Despite the remaining overhead in network, considerable improvements have been made in the Linux kernel over the versions[2] for the execution of intensive network application in large setups. The way containers are maped into the infrastructure have an important impact on performance. Two

[2] See commit 8093315a91340bca52549044975d8c7f673b28a1 introduced in Linux 3.11

factors should be taken into account: a) the memory access pattern of applications and b) the fact that there is a considerable difference in communication time between containers located in the same physical machine and containers located in different physical machines.

4.3 Inter-container communication

		CG.B		EP.B		FT.B		IS.C		LU.B		MG.C	
Multinode		%	time	%	time	%	time	%	time	%	time	%	time
	cpu	47	3285.85	79	4342	58	4276	60	3161	78	11221.429	70	4822.81
	comm	39	2721.28	3	142	28	2019	21	1097	15	2106.848	15	1024.00
	init	15	1044.65	19	1044	14	1044	20	1044	7	1049.755	15	1044.55
Container		%	time	%	time	%	time	%	time	%	time	%	time
	cpu	71	4831.80	80	4682	75	5415	67	3313	83	14621.023	84	6451.96
	comm	14	934.52	2	141	10	722	11	560	11	2014.924	3	206.01
	init	15	1052.53	18	1051	15	1052	21	1053	6	1056.211	14	1057.48
SM		%	time	%	time	%	time	%	time	%	time	%	time
	cpu	70	4714.92	78	4595	76	5440	66	3311	81	14989.101	80	6456.2
	comm	14	937.96	4	258	10	725	13	640	13	2349.951	7	601.9
	init	16	1039.72	18	1038	14	1039	21	1040	6	1039.657	13	1038.3

Table 1: Profiles of the different NAS benchmarks obtained when executed with 16 MPI processes. Time is given in milliseconds. For the case *multinode* 8 physical machines were used

The goal of this test is to measure specifically the performance of inter-container communication by comparing it against communication among physical machines and communication among MPI processes hosted in the same physical machine. This section addresses the research question *Q2*.

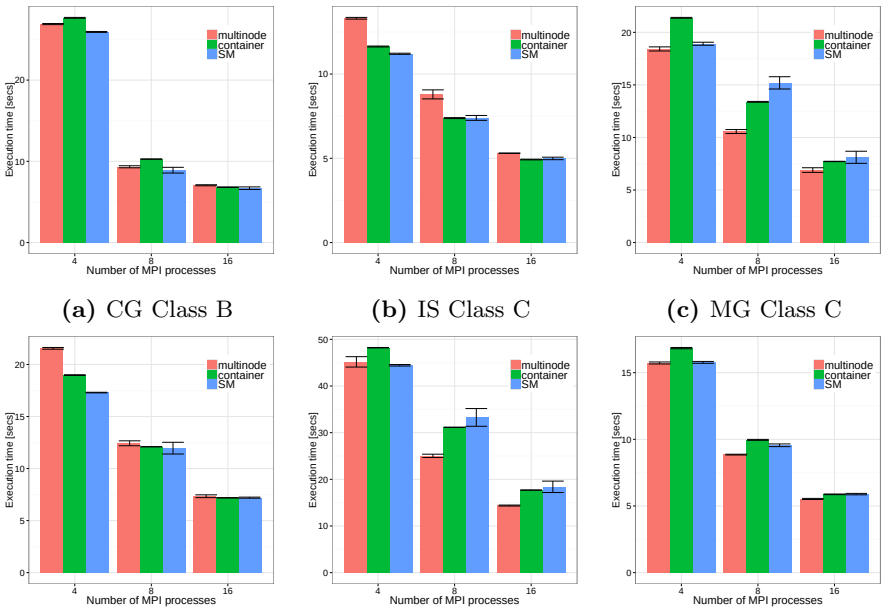
Setup: We run the chosen benchmarks from NPB using 4, 8 and 16 MPI processes in different configurations: a) *container*: using 2, 4 and 8 containers configured with two cores and deployed on 1 physical machine, b) *SM*: using just one physical machine (processes communicating via shared memory) but running the equivalent number of MPI processes, c) *multinode*: using 2, 4 and 8 physical machines.

Results: Figure 2 shows the impact on the execution time for the different benchmarks. Inter-container communication through the virtual network device is compared against communication between real machines and communication between MPI processes that use the shared memory module *sm* provided by OpenMPI.

We can observe that for some benchmarks (MG, LU, and EP) the execution time is lower in the configuration *multinode* (using different physical machines) than in the configuration *SM* (execution within the same machine). This seems counterintuitive as MPI processes over different machines use the network, however, it could be due to cache misses and memory bandwidth saturation. Under these conditions, we observe that *containers* have a behavior similar to *SM* with a maximum overhead of 13.04% for MG.C. This overhead gets smaller as more

MPI processes are used, making containers slightly better regarding communication time as it is shown in Table 1. The table also shows that the communication time in most of the cases is always better using the virtual network device than using the real network. Additionally, we can observe that the time spent in the `MPI_Init` method given by *init* in the table is roughly the same. This reduces the possibility that the run of the benchmarks were impacted by some network issue. The presence of variability in the configuration *SM* is due to the fact that the MPI processes are distributed differently each run. The version of MPI used does not bind automatically MPI processes to cores in the machine.

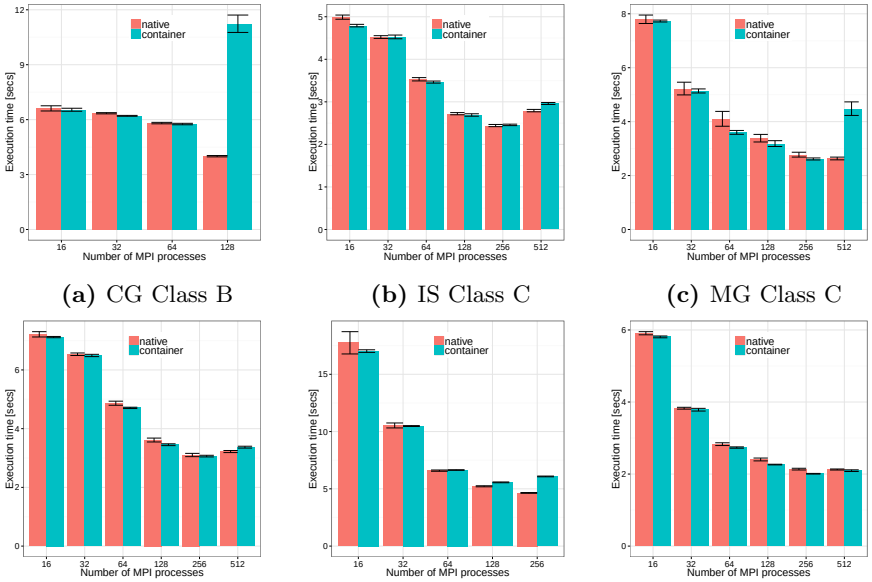
Conclusions: Although inter-container communication is faster than communication among physical machines, there is an important degradation of the CPU performance for applications that are memory bound. However, it should be remarked that MPI shared memory communication suffer from the same limitation. In this scenario communicating over the virtual network device does not add an extra cost.



(a) CG Class B **(b)** IS Class C **(c)** MG Class C

(d) FT Class B **(e)** LU Class B **(f)** EP Class B

Fig. 2: Execution time using different NAS benchmarks. In the experiment a single MPI process is run per core. Containers were deployed on a single physical machine. In the multinode case up to 8 real physical machines were used.



(a) CG Class B **(b)** IS Class C **(c)** MG Class C

(d) FT Class B **(e)** LU Class B **(f)** EP Class B

Fig. 3: Evaluating `veth` overhead using a single container per machine. The container was configured with all cores available (16) on the host. The applications were executed inside the containers and natively on the physical machine using one MPI process per core.

4.4 Multinode inter-container communication

We performed a similar experiment as the one shown in the previous section. However, in this section the goal is to measure the overhead of communicating containers located in different physical machines where several MPI processes are run per container. This experiment illustrates the use of containers as a mechanism to improve the portability of complex software stacks and addresses the research question *Q3*.

Setup: We deployed a single container per physical machine. We run the different benchmarks natively and using containers for an increasing number of physical machines (1, 2, 4, 8, 16 and 32). Each container is configured with all cores available (16 cores). We run 16 MPI processes per physical machine or container (depending on the experiment) which makes a total of 512 MPI processes (32 physical machines).

Results: Figure 3 shows the execution time of the applications running inside containers and running directly on the machine. The figure show the results just before the speed up starts to drop down. We can classify the results into two groups: 1) a group composed of the benchmarks FT, EP and IS which send a few number of MPI messages (around 20 messages per execution), 2) a group composed of the benchmarks LU, CG and MG which send a large number of MPI messages (around a 100 times more than the first group of benchmarks).

In the first group, we observed a maximum overhead of 5.97% (with 512 MPI processes). In the second group, we observed a higher overhead starting from 30% for the benchmark LU. Such a high overhead is due to network congestion present in the virtual interface. Additionally, we can observe that suddenly the overhead obtained using containers reaches 180% for the CG benchmark when 128 MPI processes are used. This can be explained by the highly number of MPI messages sent by this benchmark, around a 1000 times more than the first group of benchmarks which increase network congestion and leads to TCP timeouts. This behavior has been already observed in [4] and it is probably related to the TCP *incast problem* [16]. We could only observe the presence of TCP timeouts and retransmissions by monitoring network traffic and observing execution traces. We have not been able to identify if or where packets were dropped. The high overhead comes from the fact that the kernel Linux sets by default the *TCP minimum retransmission timeout* (RTO) to 0.2 seconds. Therefore, the application has to wait a minimum of 0.2 seconds before continuing to receive messages. This was observed around 20 times during the execution of the CG benchmark which added 4 seconds to its execution time. We were able to tweak the RTO value, setting it to $2ms$ which reduced the overhead from 180% to 24.7%.

Conclusions: This section showed how network bound applications can be severely affected by the default container network interconnection. We found a way to alleviate the overhead by tweaking parameters of the Linux network stack. The overhead observed could be diminished by integrating more advance network interconnection such as Linux's *macvlan*, SR-IOV or OpenvSwitch [3].

5 Conclusions and Future Work

In this paper, we study the impact of using containers in the context of HPC research. To this end, we conduct different sets of experiments to evaluate two interesting uses of containers in the context of HPC research: portability of complex software stacks and oversubscription. The evaluation was carried out using several benchmarks with different profiles of execution and a significant number of machines which is a configuration expected to be found in an HPC context. The evaluation shows the limits of using containers, the type of application that suffer the most and until which level of oversubscription containers can deal with without impacting considerably the application performance. While considerable overhead using containers were obtained, it was shown that the technology is getting mature and performance issues are being solved with each new release of the Linux kernel. Future work will be dedicated to complete this study by measuring the impact of using containers on disk I/O and other containers features like memory limitation.

[3] http://openvswitch.org/

Acknowledgement

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

1. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html
2. Agesen, O., Garthwaite, A., Sheldon, J., Subrahmanyam, P.: The evolution of an x86 virtual machine monitor. SIGOPS Oper. Syst. Rev. 44(4), 3–18 (Dec 2010), http://doi.acm.org/10.1145/1899928.1899930
3. Balouek, D., et al.: Adding virtualization capabilities to the Grid'5000 testbed. In: Cloud Computing and Services Science, Communications in Computer and Information Science, vol. 367, pp. 3–20. Springer (2013)
4. Bedaride, P., Degomme, A., Genaud, S., Legrand, A., Markomanolis, G., Quinson, M., Stillwell, Mark, L., Suter, F., Videau, B.: Toward Better Simulation of MPI Applications on Ethernet/TCP Networks. In: PMBS13 - 4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems. Denver, United States (Nov 2013), https://hal.inria.fr/hal-00919507
5. Felter, W., other: An updated performance comparison of virtual machines and linux containers. In: ISPASS 2015. pp. 171–172 (March 2015)
6. Felter, W., et al.: An updated performance comparison of virtual machines and linux containers. Tech. rep., IBM (2015)
7. Hill, Z., Humphrey, M.: A quantitative analysis of high performance computing with amazon's ec2 infrastructure: The death of the local cluster? In: Grid 2009. pp. 26–33 (Oct 2009)
8. Matthews, J.N., et al.: Quantifying the performance isolation properties of virtualization systems. In: Experimental Computer Science. p. 6 (2007)
9. Nussbaum, L., Anhalt, F., Mornard, O., Gelas, J.P.: Linux-based virtualization for hpc clusters. In: Linux Symposium. Montreal, Canada (2009)
10. Regola, N., Ducom, J.C.: Recommendations for virtualization technologies in high performance computing. In: CloudCom 2010. pp. 409–416 (2010)
11. Ruiz, C., Harrache, S., Mercier, M., Richard, O.: Reconstructable software appliances with kameleon. SIGOPS Oper. Syst. Rev. 49(1), 80–89 (Jan 2015)
12. Sarzyniec, L., et al.: Design and Evaluation of a Virtual Experimental Environment for Distributed Systems. In: PDP 2013. pp. 172 – 179. Belfast, UK (Feb 2013)
13. Shende, S.S., Malony, A.D.: The tau parallel performance system. The International Journal of High Performance Computing Applications 20, 287–331 (2006)
14. Subhlok, J., other: Characterizing nas benchmark performance on shared heterogeneous networks. In: HCW 2002. pp. 91–. Washington, DC, USA (2002)
15. Tudoran, R., et al.: A performance evaluation of azure and nimbus clouds for scientific applications. In: CloudCP 2012. pp. 4:1–4:6. New York, NY, USA (2012)
16. Vasudevan, V., Phanishayee, A., Shah, H., Krevat, E., Andersen, D.G., Ganger, G.R., Gibson, G.A., Mueller, B.: Safe and effective fine-grained tcp retransmissions for datacenter communication. In: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication. pp. 303–314. SIGCOMM '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1592568.1592604

17. Walter, J., Chaudhary, V., Cha, M., Guercio, S., Gallo, S.: A comparison of virtualization technologies for hpc. In: AINA 2008. pp. 861–868 (March 2008)
18. Xavier, M., et al.: Performance evaluation of container-based virtualization for high performance computing environments. In: PDP 2013. pp. 233–240. Belfast, UK (Feb 2013)
19. Youseff, L., et al.: Evaluating the performance impact of xen on mpi and process execution for hpc systems. In: VTDC 2006. pp. 1–. Washington, DC, USA (2006)