Container-based Virtualization for HPC

Holger Gantikow[1], Sebastian Klingberg[1] and Christoph Reich[2]
[1]*Science & Computing AG, Tübingen, Germany*
[2]*Cloud Research Lab, Furtwangen University, Furtwangen, Germany*

Keywords: Container Virtualization, Docker, High Performance Computing, HPC.

Abstract: Experts argue that the resource demands of High Performance Computing (HPC) clusters request bare-metal installations. The performance loss of container virtualization is minimal and close to bare-metal, but in comparison has many advantages, like ease of provisioning.
This paper presents the use of the newly adopted container technology and its multiple conceptional advantages for HPC, compared to traditional bare-metal installations or the use of VMs. The setup based on Docker (Docker, 2015) shows a possible use in private HPC sites or public clouds as well. The paper ends with a performance comparison of a FEA job run both bare-metal and using Docker and a detailed risk analysis of Docker installations in a multi-tenant environment, as HPC sites usually are.

1 INTRODUCTION

Applications in the domain of High Performance Computing (HPC) have massive requirements when it comes to resources like CPU, memory, I/O throughput and interconnects. This is the reason why they are traditionally run in a bare-metal setup, directly on physical systems, which are interconnected to so-called clusters.

Such a cluster infrastructure offers the best performance, but of disadvantage is the time for setting up: a) The operating system, usually some Linux flavor, must be installed using automatic mechanisms like PXE and Kickstart to install a basic installation ready to log in and get customized. b) All the applications required for computation and general HPC related libraries, like MPI (MPI, 2015), have to be installed and fine tuned in the customization phase. This is usually done by configuration management tools like Chef (Chef, 2015) or Puppet (Puppet, 2015). c) Before the first computational jobs can be started, the installed systems have to be finally integrated in some job scheduler like GridEngine (Oracle, 2015), LSF (IBM, 2015), or TORQUE (Adaptive Computing, 2015) which ensures proper resource management and avoids over-usage of resources.

Even though these steps can be automated to the great extent, the whole process until being finally able to start a job is quite time consuming and leaves the system in a rather static setup difficult to adapt to different customer needs. Often different applications, or even different versions of the same one, have conflicting environmental requirements, like a specific Linux version or specific library version (e.g. *libc*). This leads to the risk of putting the consistency of the whole cluster at stake, when adding a new application, or a newer version. Libraries might have to be updated, which might imply an upgrade of the whole Linux operating system (OS). Which in return can lead to old versions which are usually required for the ability to re-analyze previous calculations not being functional.

Now given a scenario where applications need computing environment changes frequently, the setup might take several hours. Even when using disk images, this approach does not pay off for jobs only running a rather limited time. One would like to have high configuration flexibility, with low application interference on the same cluster and optimal resource utilization. Isolation is the key solution for this. The use of different virtual machines (VMs), as they offer a feasible solution for tailoring a suitable environment for each type of workload and even providing a portable entity for moving HPC jobs to the cloud, is a trend that is gaining more and more momentum. With such a setup compute servers are no longer used for bare-metal computing, but turned into host systems for virtualization instead, reducing the installation time and making the systems much more flexible for different workloads, as they only have to offer the

Gantikow H., Klingberg S. and Reich C..
Container-based Virtualization for HPC.
DOI: 10.5220/0005485605430550
In *Proceedings of the 5th International Conference on Cloud Computing and Services Science* (CLOSER-2015), pages 543-550
ISBN: 978-989-758-104-5
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

minimum environment to host a VM, whereas all the application specific environment is encapsulated inside the VM.

Even though the use of *hypervisor-based virtualization* using VMs is highly common, it comes with quite a few trade-offs performance-wise, which make them less suitable for demanding HPC workloads.

Our work makes the following contributions:

- We present *container-based virtualization* with *Docker* as a superior alternative to VMs in the field of HPC.
- We provide a comparison of the concepts of VMs and containers and their use for HPC.
- We evaluate its performance using a finite element analysis (FEA) job with ABAQUS (Abaqus FEA, 2015), a widely used application in the field of computer aided engineering (CAE).
- We discuss possible risks of container-based virtualization.

The rest of the paper is organized as follows: Section 2 explores possible options for container-based virtualization. Section 3 discusses their advantages over VMs for HPC and describes the most viable Linux solution (*Docker*) for containers. Section 4 evaluates the performance overhead over native execution with a real life industrial computational job. Section 5 takes a look at possible security implications by using Docker. Section 6 concludes the paper.

2 RELATED WORK

The use of container-based virtualization for all sorts of workloads is not new, as the underlying concepts such as namespaces (Biederman, 2006) are mature. The core concept of isolating applications is seen in any Unix-like OS, with BSD Jails (Miller et al., 2010), Solaris Zones (Price and Tucker, 2004) and AIX Workload Partitions (Quintero et al., 2011) being available for years. Linux, the operating system that powers most HPC clusters and clouds, as opposed to Solaris and AIX, offers a similar solution called LinuX Containers (LXC), with its initial release back in 2008. Even though LXC offers good performance it never really caught on in the HPC community. Another option for Linux based containers is *systemd-nspawn*, which hasn't seen any widespread use so far. The most interesting option we are considering as a VM alternative for HPC in this paper is Docker, which recently became the industry standard for Linux containers, due to its ease of use, its features, like layered file system images and the ecosystem supporting it.

There have been several studies comparing VM to bare-metal performance (Matthews et al., 2007), (Padala et al., 2007) which have lead to much improvements in hardware support for virtualization and VM technology as such (McDougall and Anderson, 2010). Especially the two open-source Type-1 hypervisor solutions *Xen* (Barham et al., 2003) and the *Kernel Virtual Machine (KVM)* (Kivity et al., 2007), that turns the whole Linux kernel into a hypervisor, have seen lots of performance improvements, for example by combining hardware acceleration with paravirtualized I/O devices using virtio (Russell, 2008). This papers discusses the advantages of container-based virtualization for HPC, the amount of performance-overhead added by Docker when running a FEA compute job inside a container instead bare-metal and takes a closer look at the security-related implications when using Docker in a multi-tenant environment.

3 CONTAINERS FOR HPC

Whereas VMs still offer the most mature and reliable technology for isolating workloads, both in terms of security and stability for encapsulating applications and their dependencies in a portable entity, their performance loss still remains, as overhead is added by a hypervisor, also known as *Virtual Machine Manager (VMM)*, running on top of the hardware to control the VMs.

While hypervisor-based virtualization can still mean up to 25% reduction in turnaround time in certain scenarios (Stanfield and Dandapanthula, 2014), large compute clusters continue to still run bare-metal setups, even though this means a huge trade-off in flexibility. The same applies for many of the commercial HPC-on-demand offerings, for example addressing the CAE sector. Because they're based on bare-metal setups they frequently can't offer the elasticity customers are accustomed to from mainstream cloud offerings like Amazon EC2, as there are certain minimum quantities that have to be booked for making the re-installation pay off.

By using a container-based virtualization approach this might change to a certain extend.

3.1 Container Virtualization vs. Hypervisor

The technical concept of container-based virtualization differs quite a bit from hypervisor-based virtualization. The containers run in user space on top of an operating system's kernel, while the hypervisor is

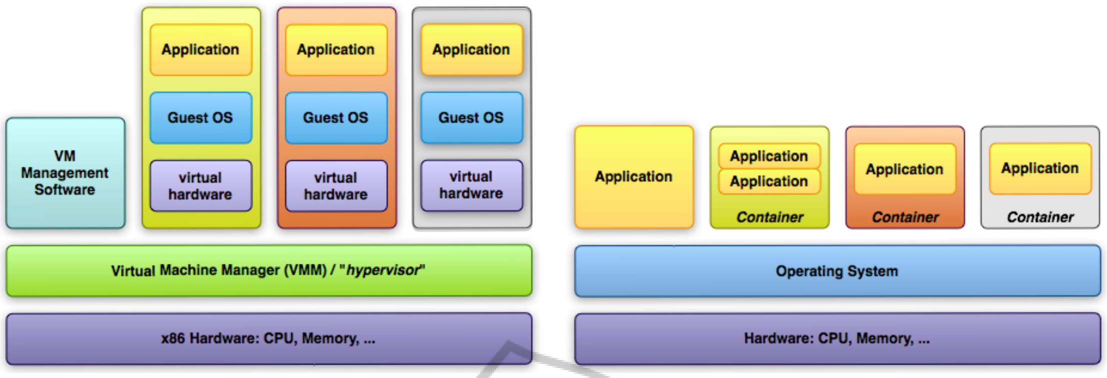

Figure 1: Hypervisor- (left) vs. Container-Based Virtualization (right).

scheduling system calls of the virtualized operating system as seen in Figure 1.

As the operating system kernel is the exact same for all *containers* running on one host and can not be changed, containers are limited to the same host operating system. VMs can run other operating systems e.g. Windows on top of Linux. This is a valid point for many scenarios, but the majority of HPC clusters and IaaS clouds tend to homogeneously run Unix (e.g. Linux). Container virtualization still offers the required flexibility to run e.g. an Ubuntu Linux container on top of a Red Hat Enterprise Linux host. Which also means a containerized process appears to be running on top of a regular Linux system.

Container-based virtualization relies on two Linux kernel features of the host system to provide the required isolation:

- *kernel namespaces* (Biederman, 2006): enabling individual views of the system for different processes, which includes namespaces for processes, file systems, user ids, etc. and enables the creation of isolated containers and limiting access of containerized processes to resources inside the container.
- *kernel control groups (cgroups)*: this subsystem puts boundaries on resource consumption of a single process or a process group and limits CPU, memory, disk and network I/O used by a container.

As mentioned before there are several options for using containers with Linux, with Docker having gained most attention recently due to its features and ease of use. Docker is released by the team at Docker, Inc. under the Apache 2.0 license.

3.2 Docker and HPC

Docker offers a good solution containerizing an application and its dependencies. As seen in Figure 2

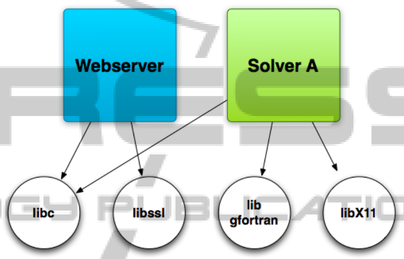

Figure 2: Applications and library dependencies.

applications usually share several libraries.

With container virtualization it is possible to isolate libraries as seen in Figure 3 to allow coexistence of special or incompatible library versions or even an *outdated* Linux distribution in a shippable entity easily. This solves the problem of running legacy code (might be needed for verifying old results of a computation) on a modern system, without the risk of breaking the system. As long as the code is not dependent on a special version of the kernel, as the kernel can not be changed inside a container.

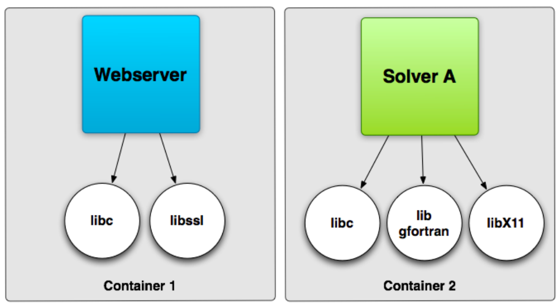

Figure 3: Applications and library dependencies encapsulated in containers.

Containers equipped with all tools and libraries for a certain tasks can be easily deployed on own clusters, systems of a related party for evaluation without having to rebuild the whole computational work-flow or at a complete third party if additional resources are needed. All that is required is a suitable runtime for starting the container.

This also cuts down the complete re-purposing of compute resources to a simple start a new container image, as compute nodes only have to offer the Docker runtime and have access to the container files.

Compared to virtual machines this significantly reduces the amount of resources required. Both in memory footprint, as containers share a lot of resources with the host system as opposed to VMs starting a complete OS and in terms of storage required. Startup time is reduced from the time booting a full OS to the few seconds it takes till the container is ready to use.

For reducing storage requirements Docker makes use of *layered file system images*, usually *UnionFS* (Unionfs, 2015) as a space conserving mechanism, which is lacking in several other container solutions. This allows file systems stacked as layers on top of each other (see Figure 4), which enables sharing and reusing of base layers. For example the base installation of a certain distribution, with individually modified overlays stacked on top, can provide the required application and configuration for several different tasks.

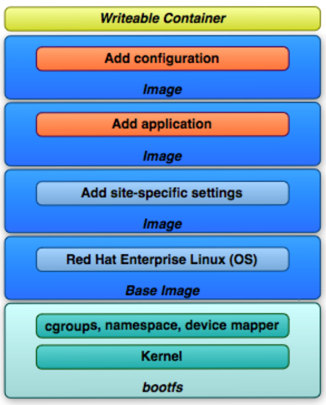
Writeable Container
Add configuration
Image
Add application
Image
Add site-specific settings
Image
Red Hat Enterprise Linux (OS)
Base Image
cgroups, namespace, device mapper
Kernel
bootfs

Figure 4: The layers of the Docker file system.

If a weakening of complete isolation is acceptable, it is also possible to pass directories, for example containing job input data into a container, so not all the required files for a compute job have to be included in the container image. One has to consider that strong isolation is desired, if providing a multi-tenant environment.

One popular VM feature it currently lacking. Compared to hypervisor-based setups Docker can not live-migrate running workloads to another host (Clark et al., 2005), which might be desirable for planned system downtime. Even though this might not be required for most HPC environments, as killing and recreating a container might be faster, the *Checkpoint/Restore In Userspace* (CRIU) project is currently working on at least providing checkpoint and restore functionality (CRIU-Project, 2015). This feature would be much more required for high availability (HA) clusters than for HPC clusters.

4 EXPERIMENTAL EVALUATION

Performance-wise, without all the overhead added by hypervisor and VMs, containers as a light-weight virtualization mechanism can achieve almost the same performance as native execution on a bare-metal system does, as other benchmarks (Felter et al., 2014), (Xavier et al., 2013) underline.

As we were interested in the amount of overhead generated by containerizing a HPC workload, we decided to benchmark a real-world ABAQUS example in two different storage scenarios, comparing the containerized execution time to the native execution. ABAQUS (Abaqus FEA, 2015) is frequently used for finite element analysis (FEA) and computer aided engineering (CAE) for example in the aerospace and automotive industries, as it provides wide material modeling capability and multi-physics capabilities. ABAQUS jobs tend to be CPU and memory intense, requiring lots of scratch space too.

The application was installed to local disks on a CentOS 7 SunFire X2200 server with the following hardware configuration:

- CPU: 4x Dual-Core AMD Opteron (tm) 2220
- RAM: 16GB
- HDD: local conventional disks without RAID
- Infiniband: Mellanox MT25204

The job used for evaluation is the freely available *s4b* from the *samples.zip* package included in the ABAQUS 6.12-3 installation. It was installed onto local disks and the software licensing was done using a local license file.

As the server provided access to a Lustre parallel distributed file system, which is frequently used for large-scale cluster computing, we decided to execute the job one time with the folder for temporary data located on local disks and the other time on the Lustre file system. Both storage configurations were used with bare-metal execution and inside a Docker container. Docker was installed using packages shipped with CentOS 7.

There were no limits imposed on Docker using cgroups to ensure maximum performance. The resources available to the container were only limited by the actual resources on the host. When using one host for multiple containers simultaneously usage should be limited as expected. We used only one container per host for not distorting the outcome.

To rule out side effects, the job was run twenty times in each configuration and the *Total CPU Time* (in seconds), which is the sum across all involved CPUs of the time spent executing ABAQUS (user time) and the time spent by the OS doing work on behalf of ABAQUS (system time), from the job output file was taken to ascertain the total runtime of the simulation (see Figure 5).

The results, diagrammed in Figure 5, show the following:

- Docker offers near native execution speed
- there is constant but minimal overhead added
- average runtime (native vs Docker)

local disk 114s vs 116,5s - overhead: 2,21%
Lustre 117,3s vs 118,5s - overhead: 1,04%

To be clear: this means only two up to five seconds longer total execution time to complete the s4b example job, which is a fraction of the time a VM would even need to boot.

A point worth commenting on is the reason for the lesser overhead when accessing Lustre. The lower difference in overhead when using Lustre can be explained by the fact, that the container uses the RDMA stack for IBoIP as directly as the host does, while accessing a local disk obviously needs to be passed through a UnionFS technology which affects the I/O flow here, at a small but mentionable minimum.

5 CONTAINER VIRTUALIZATION RISKS

As mentioned before the cornerstones of the performance and security isolation offered by Docker are cgroups and namespaces, both very mature technologies, which do a good job of avoiding Denial of Service (DoS) attacks against the host and limiting the view of what a container can see and has access to. Recent analysis on Docker (Bui, 2015) shows that the internal concepts of containers, even when using default configuration, are reasonable secure.

When deploying Docker in a multi-tenant environment, certain security aspects have to be considered:

Container vs VM. When it comes to security aspects, isolation of filesystem, devices, IPCs, network and management as described in Reshetova's paper (Reshetova et al., 2014) important. Generally it can be said, that containers have been seen as less secure than the full isolation offered by hypervisor virtualization, which is still true.

Vulnerabilities. Recent research by Ian Jackson and his colleague George Dunlap (Jackson, 2015) compared the number of Common Vulnerabilities and Exposures (CVE) in 2014 for paravirtualized Xen, KVM + QEMU, and Linux containers, that could lead to either privilege escalation (guest to host), denial of service attacks (by guest of host) or information leak (from host to guest) and showed that the risk for any of the three is higher when using containers.

One reason is that every additional interface available is a possibility for additional vulnerabilities. A hypervisor provides an interface similar to hardware and so-called hyper-calls (Xen offering 40). These are only very few calls a VM can interact with, compared to the much wider Linux system call interface used by containers. Making use of hardware virtualization for hypervisor-based setups may even add additional risks, as a detailed study (Pék et al., 2013) shows.

Docker Daemon. Since running Docker containers requires the *Docker Daemon* to run as root on the host, special attention should be given to its control, what leads to certain good practices for using Docker from a security point of view.

Misuse. Because it is possible to pass through file systems available on the host, only trusted users should be allowed access to start images or pass arguments to the Docker commandline-tool `docker`. Great harm can be done, if for example the hosts' complete file system is granted access to from inside the container. Membership of the *docker*-group is usually enough to invoke a command like `docker run -v /:/tmp myimage rm -rf /tmp/*` which would start a container, pass the hosts' filesystem to */tmp* inside the container and directly delete it.

NFS. This risk intensifies in a NFS-environment, where someone with unrestricted access to Docker can bind to an existing share, for example containing user homes, and circumvent access control based on numeric user IDs (UID) by creating a container for spoofing his UID. This can be mitigated by offering only NFSv4 shares to Docker hosts, where Kerberos as an additional authorization layer is available.

Application Container vs System Container. When using Docker for HPC applications the best thing to do is utilizing the container as an *application container* that directly starts the computing job after the container starts running. This eliminates the possibility for a user to pass parameter to the `docker-`

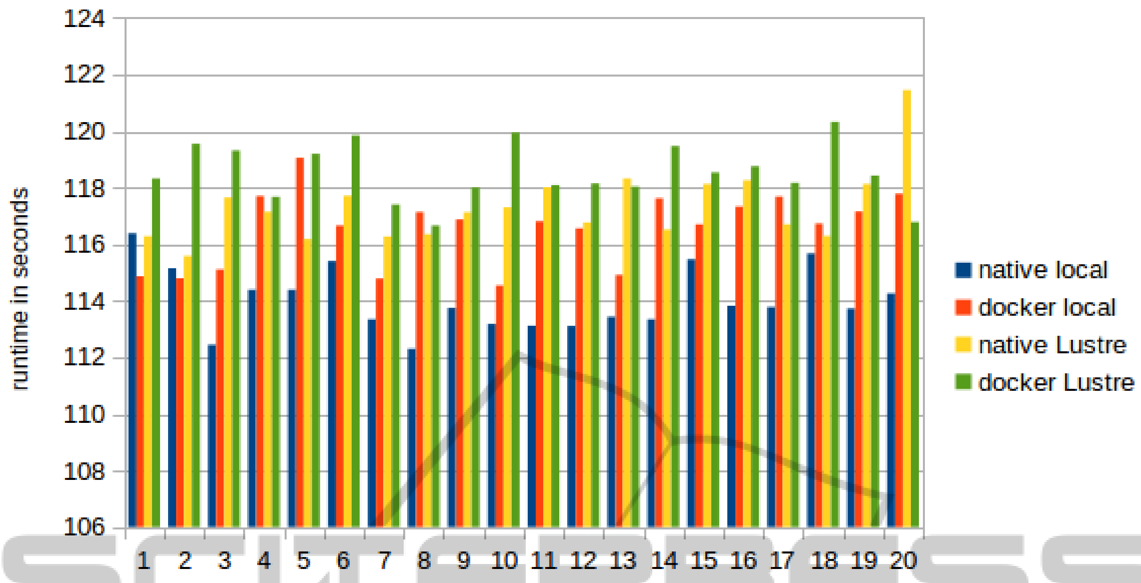

Figure 5: Total CPU Time (seconds).

command-line and to utilize the container as a more VM-style *system container*. As this disables the possibility to interactively use and explore the system, offering only pre-defined, parameter-controlled containers greatly reduces the risk of wanted or accidental misuse.

Docker Security. According to (Jérôme Petazzoni, 2013) development focusing on security is on the way, which will limit the Docker daemons' root privileges to certain sub-processes like virtual network setup. Further improvements aim at a possibility to map the root user of a container to a non-root user on the host system, reducing the impact of a possible privilege escalation. These new *user namespaces* will also optimize sharing file systems, as users within a container can be mapped to users outside the container. LXC already uses this feature (**?**), so it should be only a matter of time until Docker implements *user namespaces*.

Image Security. Attention should be paid to the source the container images are being pulled from. Docker Inc. offers a convenient way (called *Docker Hub*) to access thousands of preconfigured images which are ready to deploy. These images are for users who quickly want to set up an Apache web server or development environment for testing and do not offer any HPC related applications. But these images might be used as base for creating own HPC images. As security researchers state (Rudenberg, 2014) Docker includes a mechanism to *verify* images, which does imply that the checksum of the downloaded image has been validate. But this is actually not the case and offers possibilities for attacks. Docker solely checks for a signed manifest and never actually verifies the image checksum from the manifest. Another potential attack arises from a vulnerability when dealing with malformed packages (Jay, 2014), as malformed packages can compromise a system. The advice in this case is to only download and run images from trusted sources, at best an internal image registry, which might be best-practice after all, not just for HPC clusters behind a corporate firewall.

6 CONCLUSION

Container-based virtualization using Docker solves many problems of bare-metal HPC, when flexibility to rapidly change the installed software, deploy new versions or use applications with conflicting dependencies on the same cluster is key.

Environmental details for a job, like a certain Linux distribution with a special compiler version could be included in a field in the job description, like required CPU and memory are nowadays. The workload manager then would pick a host fulfilling the hardware requirements, pulls the workload-specific image and starts the container that runs the job.

As our evaluation with an ABAQUS test job has shown, Docker offers near native execution speed, generating a mean loss in performance of 2,21% in our scenario with local disk I/O and 1,04% when accessing a Lustre clustered file system.

The point that Docker performs almost on the same level as the bare-metal execution shows that the Docker engine has almost trivial overhead and thus offers performance traditional hypervisor-based VMs can not offer at the moment.

Since our research only focused on conceptual advantages and single host performance and many HPC applications rely on MPI for distributed computing future testing should be done in this area, taking a look at the performance of the Docker engine in distributed multi-host, multi-container scenarios and with other applications from the HPC field.

From a security standpoint VMs offer a more secure solution at the moment, but whether containers offer enough security depends on the overall HPC work-flow and the security requirements. A cloud provider offering a multi-tenant self-service solution with several customers on one cluster or even one host might want to implement an additional layer of security. In a regular HPC environment this might not be needed, as long as the necessary precautions are taken and users are not allowed to directly interact with Docker to provision potentially malicious containers but through a middle-ware like a job scheduler or parameter-controlled *sudo* scripts, that do careful parameter checking.

When it comes to patch management Docker could even provide an advantage over VMs, as the kernel is out of the focus of a container and shared among all hosts, meaning that if a kernel vulnerability is found only the Docker host has to be patched, which might be even done on the fly using tools like Ksplice.

Security of container-based solutions will further increase over time, with lot's of development being already underway. Linux containers have gotten a lot of attention over recent time and more people utilizing it will lead to closer examination and continuous improvements.

REFERENCES

Abaqus FEA, S. (2015). ABAQUS. http://www.simulia.com.

Adaptive Computing (2015). TORQUE. http://www.adaptivecomputing.com/products/open-source/torque/.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA. ACM.

Biederman, E. W. (2006). Multiple instances of the global linux namespaces. In *Proceedings of the 2006 Ottawa Linux Symposium*, Ottawa Linux Symposium, pages 101–112.

Bui, T. (2015). Analysis of docker security. *CoRR*, abs/1501.02967.

Chef (2015). Chef: Automation for Web-Scale IT. https://www.chef.io/.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. (2005). Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA. USENIX Association.

CRIU-Project (2015). Checkpoint/Restore In Userspace (CRIU). http://www.criu.org/.

Docker (2015). Docker. https://www.docker.com/.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2014). An updated performance comparison of virtual machines and linux containers. *technology*, page 28:32.

IBM (2015). LSF. http://www-03.ibm.com/systems/platformcomputing/products/lsf/.

Jackson, I. (2015). Surviving the zombie apocalypse – security in the cloud containers, kvm and xen. http://xenbits.xen.org/people/iwj/2015/fosdem-security/.

Jay, T. (2014). Before you initiate a docker pull. https://securityblog.redhat.com/2014/12/18/before-you-initiate-a-docker-pull/.

Jérôme Petazzoni (2013). Containers & Docker: How Secure Are They? https://blog.docker.com/2013/08/containers-docker-how-secure-are-they/.

Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. (2007). kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada.

Matthews, J. N., Hu, W., Hapuarachchi, M., Deshane, T., Dimatos, D., Hamilton, G., McCabe, M., and Owens, J. (2007). Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA. ACM.

McDougall, R. and Anderson, J. (2010). Virtualization performance: Perspectives and challenges ahead. *SIGOPS Oper. Syst. Rev.*, 44(4):40–56.

Miller, F., Vandome, A., and John, M. (2010). *FreeBSD Jail*. VDM Publishing.

MPI (2015). Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/.

Oracle (2015). Grid Engine. http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html.

Padala, P., Zhu, X., Wang, Z., Singhal, S., Shin, K. G., Padala, P., Zhu, X., Wang, Z., Singhal, S., and Shin, K. G. (2007). Performance evaluation of virtualization technologies for server consolidation. Technical report.

Pék, G., Buttyán, L., and Bencsáth, B. (2013). A survey of security issues in hardware virtualization. *ACM Comput. Surv.*, 45(3):40:1–40:34.

Price, D. and Tucker, A. (2004). Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th Conference on Systems Administration (LISA 2004), Atlanta, USA, November 14-19, 2004*, pages 241–254.

Puppet (2015). puppet: Automate IT. http://puppetlabs.com/.

Quintero, D., Brandon, S., Buehler, B., Fauck, T., Felix, G., Gibson, C., Maher, B., Mithaiwala, M., Moha, K., Mueller, M., et al. (2011). *Exploiting IBM AIX Workload Partitions*. IBM redbooks. IBM Redbooks.

Reshetova, E., Karhunen, J., Nyman, T., and Asokan, N. (2014). Security of os-level virtualization technologies: Technical report. *CoRR*, abs/1407.4245.

Rudenberg, J. (2014). Docker image insecurity. https://titanous.com/posts/docker-insecurity.

Russell, R. (2008). Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103.

Stanfield, J. and Dandapanthula, N. (2014). HPC in an OpenStack Environment.

Unionfs (2015). Unionfs: A Stackable Unification File System. http://unionfs.filesystems.org.

Xavier, M., Neves, M., Rossi, F., Ferreto, T., Lange, T., and De Rose, C. (2013). Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240.