KVM, Xen and Docker: a performance analysis for ARM based NFV and Cloud computing

Moritz Raho, Alexander Spyridakis, Michele Paolino, Daniel Raho
Virtual Open Systems, Grenoble, France
contact@virtualopensystems.com
http://www.virtualopensystems.com

***Abstract*—Virtualization is a mature technology which has shown to provide computing resource and cost optimization while enabling consolidation, isolation and hardware abstraction through the concept of virtual machine. Recently, by sharing the operating system resources and simplifying the deployment of applications, containers are getting a more and more popular alternative to virtualization for specific use cases. As a result, today these two technologies are competing to provide virtual instances for cloud computing, Network Functions Virtualization (NFV), High Performance Computing (HPC), avionic and automotive platforms. In this paper, the performance of the most important open source hypervisor (KVM and Xen) and container (Docker) solutions are compared on the ARM architecture, which is rapidly emerging in the server world. The extensive system and Input/Output (I/O) performance measurements included in this paper show a slightly better performance for containers in CPU bound workloads and request/response networking; conversely, thanks to their caching mechanisms, hypervisors perform better in most disk I/O operations and TCP streaming benchmark.**

I. Introduction

Usage of virtualization has been widely adopted over the last few decades in x86 architecture for the server market and more recently even desktop/laptop environments. Nowadays, virtualization is also gaining momentum in ARM based embedded systems, allowing to optimize hardware resources, to increase energy efficiency and to ensure application isolation [1]. Virtualization support in the latest ARM architectures has extended use cases beyond embedded systems, e.g. in cloud computing, Network Functions Virtualization (NFV), High Performance Computing (HPC) systems but also in avionic and automotive. Indeed, the introduction of hardware support for virtualization in ARMv7 and ARMv8 architectures has motivated the development of hypervisors for ARM processors e.g. Kernel-based Virtual Machine (KVM) and Xen. Another way to share hardware resources among different applications is by mean of containers. Unlike hypervisors which create virtual instances of the hardware, containers provide a more lightweight mechanism aiming to share a single Operating System (OS). An application running in a container thinks that it has an unshared access to a copy of the OS. In other words, containers virtualize the OS while hypervisors virtualize the hardware resources. Containers exists since decades but require custom kernels to be executed, which prevent their popularity in past days, until Linux-based containers fully supported mainstream kernels starting in 2008. Since containers don't require a full OS copy for each instance, they are reputed for both their low overhead and system resource consumption. However, containers are considered to be more vulnerable to security threats than hypervisors, although there is room to improve on this aspect. In addition, containers suffers from the limitation of not being able to run a different OS or kernel.

This paper presents a comparative performance study of two open source hypervisors and an open source container, for ARM platforms. The hardware target used in this study is the Arndale [2] development board, equipped with two ARMv7 Cortex-A15 processors based on Samsung's Exynos 5250 SoC, which includes virtualization and Large Physical Address (LPAE) extensions. The two selected hypervisors are KVM and Xen, which are well known and supported in the open source community; the chosen container solution is Docker, which is showing an increasing popularity in Linux systems.

The presented work is organized as follows: Section II provides a state-of-the-art description of the selected virtualization solutions, object of this benchmarking study, i.e., KVM on ARM, Xen on ARM and Docker. Section III provides description details of the different performed benchmarks. In Section IV, the experimental results are showcased and discussed. Finally, section V describes the related work, while in section VI conclusions are wrapped up and potential future directions are pointed.

II. Hypervisors and Container

A. Hypervisors

Hypervisors, also called Virtual Machine Monitors (VMM), share the hardware resources of a real machine between multiple Virtual Machines (VM) or guests. By virtualizing system resources such as CPUs, memory and interrupts, they provide a mean to run multiple OSs concurrently. Better performance and additional isolation in VM execution is delivered through hardware virtualization extensions that have been included in the x86 architecture since almost a decade [3]. In a similar manner, ARM has introduced its own hardware virtualization extensions a few years ago, allowing hypervisors such as KVM and Xen to provide efficient virtualization on ARM platforms.

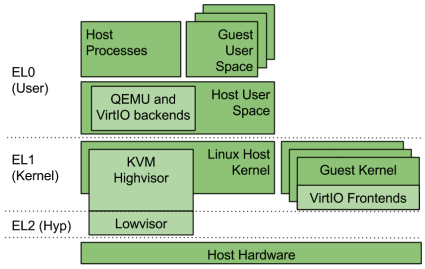
978-1-5090-1201-5/15/$31.00

1) KVM/ARM: KVM is an open source hypervisor originally built for x86 since 2007, then ported to the ARM architecture in 2012. As its counterpart in x86, the ARM version of KVM is integrated in the Linux kernel, thus benefiting from reusing many Linux functionalities such as memory management and CPU scheduling. While KVM for x86 was merged into the Linux kernel mainline in kernel version 2.6.20, the KVM on ARM port was upstreamed in Linux kernel version 3.9 [4] by Virtual Open Systems and ARM with the support of the open source community. As described in [5], due to architectural differences between the x86 and ARM virtualization extensions, while KVM on x86 could entirely reside in the kernel, KVM on ARM had to be split in two parts. In fact, a KVM on ARM solution entirely built in the host kernel would have required significant and intrusive modification to the kernel for its execution in ARMs Hyp mode. This was not considered viable in terms of portability and performance, and it would have been too invasive for the kernel itself. Therefore the KVM on ARM implementation has been split in the so-called Highvisor and Lowvisor. The Highvisor lies in ARMs kernel space and handles most of the hypervisor functionalities. The Lowvisor resides in Hyp mode and is responsible for enforcing isolation, handling hypervisor traps and performing the world switches (context execution switches between VMs and host). Although this mechanism implies that traps between the highvisor and the VMs requires two additional traps from and to the lowvisor in Hyp mode, it is shown [5], that no significant performance overhead is produced. An essential benefit of this implementation, compared to a bare-metal hypervisor such as Xen, is that porting KVM on ARM to a new ARM platform becomes seamless, provided the platform is able to run a Linux version higher than 3.9. Since ARM platforms are often built in a non-standard way and Linux is almost always supported across all recent ARM platforms, this is a key gain for KVM in terms of portability and ease of use.

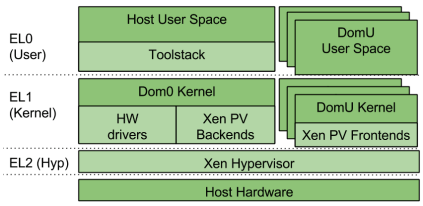
Similar to x86, KVM on ARM doesn't offer by itself machine or device models. Machine and device abstractions are too specific for the KVM virtualization model and would introduce unnecessary complexity in the hypervisor. Instead, KVM relies in user space tools such as the Quick Emulator (QEMU) [6] and kvmtool, which are meant for emulating guest hardware devices and instantiating virtual machines. In the KVM paradigm guests are seen by the host as normal POSIX processes, with QEMU (user space emulation) residing in the host userspace and utilizing KVM to take advantage of the hardware virtualization extensions. QEMU and KVM are able to run fully-virtualized unmodified guests without any mandatory need of a para-virtualization layer. This is an important benefit when running legacy operating systems in a virtual environment. However, since emulation is reputed to add significant overhead, KVM also supports I/0 para-virtualization through Virtio [7]. Virtio provides drivers for network, block and balloon devices which need to be enabled in the guest kernel, the corresponding backend drivers run in the host user space (e.g. QEMU or kmvtool).

Fig. 1a shows the KVM on ARM system architecture.

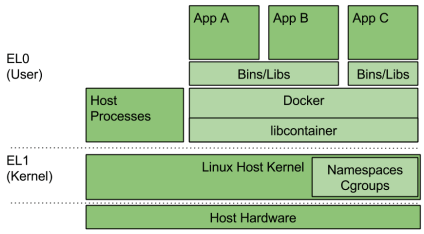
2) Xen on ARM: Originally developed in 2003, the x86 version of Xen, is a bare-metal hypervisor that supports both fully virtualized and para-virtualized guests [8]. Xen made



(a) KVM/ARM Architecture



(b) Xen on ARM Architecture



(c) Docker Architecture

Fig. 1: System Architectures

a lot of changes in its code and architecture when ported to ARM. Xen on ARM is much smaller in terms of code size while providing similar performance. On ARM, the code for Xen is reduced to one type of guest which uses para-virtualized drivers and the ARM virtualization extensions. Most of the QEMU software stack has been removed, making full-virtualization impossible. This implies that porting operating systems in Xen, becomes a problem when trying to run proprietary solutions. However, Linux and FreeBSD distributions are supported by Xen and the relevant para-virtualized frontends and backends can be easily activated in the respective kernel.

The Xen hypervisor resides in Hyp mode, providing CPU, memory, interrupt and timer virtualization features [9]. On top of the Xen hypervisor layer everything is executed as a guest placed in different domains. The most privileged domain is called Dom0, a guest which is responsible for managing other virtual machines placed in DomU. Xen discovers the hardware via a given device tree and assigns all unused devices to Dom0 via a generated flatten device tree, so that Dom0 can boot in a native way. Xen declares its presence to Dom0 and limits Dom0 resources also via the generated device tree. Dom0 runs native drivers for its attached devices and starts a set of para-virtualized backend drivers. On the DomUs, corresponding frontend drivers are running in order to give them access to the virtualized devices. Also, for the moment, a few para-virtualized backends from user space are provided

by Qemu. In order to increase security and isolation, Xen gives the possibility to run unprivileged guests with the only scope of running the native driver and corresponding backends of a specific device. Finally, A Xen toolstack running as a normal user-space process in the control domain (Dom0) can be used to manage the unprivileged domains.

ARM hardware virtualization extensions added the Hyp mode, which separates the kernel from the hypervisor. By building a standalone hypervisor that resides in the Hyp mode, Xen on ARM can take full advantage of the virtualization extensions without additional overhead. Xen has the potential for better guest performance than KVM, however, such an implementation involves an extra work for porting Xen on every new ARM SoC [9].

Fig. 1b shows the Xen on ARM system architecture.

B. Linux-Based Container

Containers provide isolated user space instances while sharing a common kernel. Linux based container management tools, such as LXC (Linux Container) and Docker, use Linux control groups (cgroups) and namespacing technologies to provide respectively resource management [10] and isolation [11] between the container instances.

Similar to Linux processes, cgroups are organized hierarchically and child cgroups inherit their parent attributes. The main difference resides in the fact that, unlike processes, cgroups are not part of the same tree, many of them can exist simultaneously. Each cgroup can be used to manage the resources of a group of processes. For example CPU time or memory can be limited inside a cgroup. Thus, cgroups can be used to resize and limit the resources of a container. Additionally, cgroups are also used to terminate all processes inside a container. However, as Linux containers share only one kernel, a problem is that processes running inside a container, are not aware of their resource limitations and can still view all the resources of the system [12].

There are 6 Linux namespaces that can be accessed by the clone system call. Linux provides namespaces for network, mount points, user and groups, process ids and hostname. Each of these namespaces abstract a system resource that can be used to provide isolation between processes that are not part of the same namespace. Linux based-containers use namespaces to provide isolation between themselves, for example using user namespaces, privileges for a user inside a container are not extended outside of the container [13].

Two types of containers exist, operating system containers, and application containers. OS containers are containers that run an init system, and application containers run a single application. The advantage of running a single application is to reduce resources allocated by unnecessary management processes. However, depending on the use case it might be preferable to run a full OS container.

1) Docker: Docker emerged in 2013 and has become very popular in a short amount of time due to its completeness and ease of deployment. Docker is a container management tool. It originally builds upon LXC containers providing an easy way to package an application in a container. Unlike LXC, which creates OS containers, Docker mainly manages single application containers. Even if LXC and other management tools are still supported by Docker, since version 0.9, Docker introduced libcontainer, which, like LXC, is a wrapper of Linux cgroups and namespaces.

Docker makes use of union file-systems such as devicemapper (after Docker 0.7) or Advanced Multi Layered Unification Filesystem (AUFS). A Union file system provides a layered stack of file systems. It allows to use a file system as a basis for containers. This file system basis is read-only and a write layer is added on top of it so that containers can commit their changes from the original file system. Commits of the modified file system are done from the host, while containers are running. However, if there is no commit, the changes to the original file system are not persisted. Union file systems allow multiple containers to use the same OS image basis while allowing containers to have their own files. Such an implementation result in a significant gain of storage and allows faster deployment of containers. Each new version of the container file system is a difference of changes from the previous one. This makes it possible to keep track of all changes between the commits, exactly as version control systems operate. Finally, Docker provides a public registry and supports private registries that enables to push and pull images from. These registries provide various Linux base images but also ready-to-use software such as databases or Web servers [14]. The top level Docker architecture can be seen in fig. 1c.

III. Benchmark applications

This section illustrates the different benchmark applications which are used to measure the performance overhead of KVM, Xen and Docker compared to a non-virtualized Linux host.

A. Hackbench

Hackbench creates a given number of pairs of schedulable entities, which can communicate either via sockets or pipes. The output of hackbench corresponds to the time it took for the pairs to send data between them. The system scheduler and CPU performance is measured and reported [15]. Hackbench from the version 0.87 of rt-tests was used in this paper.

B. Byte-Unixbench

Byte-Unixbench is a benchmark suit with the purpose of testing various aspects of the system's performance. In addition to provide the raw score results for each of the below mentioned benchmark applications, it produces also an overall index value. Byte-Unixbench is also capable of handling SMP systems and includes graphic performance tests, these are however omitted for this paper. For this study byte-unixbench5.1.3 is used.

1) Dhrystone: Developed in 1984 by Rheinhold P. Weicker, dhrystone is representative of CPU performance. The test focuses on measuring the number of instructions executed per second in integer calculations. Even if Dhrystone has known weaknesses [16] it is included in the study for its popularity.

2) Whetstone: Whetstone measures the speed of floating-point operations. Similarly to Dhrystone, the Whetstone benchmark is a synthetic benchmark, meaning that it is designed to imitate the processor usage of some common set of programs.

3) Execl Throughput: This test provides a measurement of the maximum number of execl calls that are performed per second. The exec() family of functions replaces the current process image with a new process image.

4) Pipe Throughput and Pipe-based Context Switching: In the pipe throughput test the number of times per second a process can write 512 bytes to a pipe and read them back is counted. The pipe-based context switching test is measuring the number of times two processes can exchange an increasing integer through a pipe.

5) Process Creation: This test is related to memory bandwidth performance. It counts how many time a process can fork and reap a child that immediately exits. Each time a process is created, memory has to be allocated.

6) Shell Scripts: For this test, the number of times per minute a process can start and reap a set of concurrent copies of a shell script is counted. These shell scripts apply a series of transformations to a data file

7) System Call Overhead: Here the cost of entering and exiting the kernel is measured. These context switches are triggered by making a series of getpid system calls.

C. IOzone

IOzone is used to perform file-system benchmarks. It tests file I/O performance for read, write, re-read, re-write, read backwards, fread and fwrite, fread and re-fread, random read and write, read strided and record re-write. [17].

D. Netperf

Netperf is used to benchmark various different types of networking activity. It can measure unidirectional throughput and end-to-end latency. Netperf uses BSD Sockets to perform measurements for TCP and UDP [18].

IV. Performance analysis and Results

This section discusses the results obtained from system and I/O benchmarks for a non virtualized host, Xen Dom0, Xen DomU, KVM guest and a Docker container on the Insignal Arndale board, which has a dual core 1.7GHz Cortex A15 CPU on a Samsung Exynos 5250 SoC and 2GB of memory. The board boots by running U-Boot through the MMC interface and is connected to an external SSD disk for the file-system. As for network benchmarking, the device under test is running the Netperf client connected to a Netperf server in the local network of 100Mbps bandwidth.

The virtual machines, container and host are running on top of an identically configured upstream Linux kernel of version 4.1.2 and a Debian 8.1 file-system. The version used for the Xen hypervisor and for the toolstack is 4.5.1, and while KVM is built in the kernel and relies on QEMU 2.3.5, the installed version of Docker is 1.6.2. The host, virtual machines and container are limited to 1GB of RAM and use one CPU when running the benchmarks. The KVM virtual machine is provided with an extra configuration to make use of 2MB huge pages (which is default on Xen domains) and VirtIO drivers for network and disk devices. Vhost-kernel backend drivers for KVM are not upstream for ARM and are not activated, however they could enhance the I/O performance of the KVM guest. Regarding the network configuration, Xen, Docker and KVM are configured to make use of bridges and while Docker uses Virtual Ethernet (veth) interfaces and NAT for its containers, Xen and KVM guests interface is configured through TAP devices. Additionally, in order to keep with real world results, caching mechanisms were not disabled neither in the benchmark applications (this applies especially for IOzone), nor in the guest and host operating systems or the container.

By using the same kernel configuration for all tested systems, test results can be compared and the various observed performance overheads rely on the different types of virtualization implementations. This also means that the host has KVM activated, and, while not providing a native execution in the test results, it has been shown in literature [19] that KVM and virtualization kernel configurations induce no overhead for system and I/O performance on ARMv7.

All test results presented in this section are obtained by averaging ten consecutive runs and are shown with the corresponding standard deviation.

A. System performance

1) Hackbench: Fig 2 shows the results for the hackbench tests. In each of these tests, 1000 messages of 100 bytes are sent between a varying number of receiver/sender pairs. The number of tasks, varying from 40 to 640, is given by the number of pairs multiplied by 40, which correspond to the Hackbench file descriptor size. Compared to the host, the Xen privileged domain and Docker induce almost no performance overhead. The KVM guest and Xen DomU have a small comparable overhead, which is slightly increasing with the number of tasks, showing up to 5% overhead for 640 tasks.

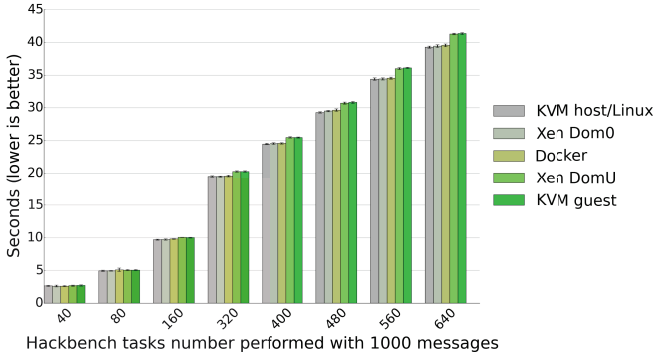


Fig. 2: Hackbench performance results

2) UnixBench: Fig 3 shows UnixBench test results. All tested systems score equally for Dhrystone2 and Whetstone benchmarks and the execl and system call overhead tests. These tests are related to CPU performance, which are reputed to induce very low performance overhead for virtual machines and containers. The same applies to the one and eight concurrent shell scripts, where almost no overhead in regards to the host is observed for Docker, KVM and Xen Dom0 and only a small overhead of respectively 2% and 3% can be remarked

for the unprivileged Xen domain. For the process creation tests, related to memory bandwidth performance, a small performance decay of 2.5% for Xen and KVM guests and 3.5% for Docker is observed. Similarly, the pipe throughput and pipe-based context switch benchmarks show performance overheads for Docker of respectively 3.5% and 12.5% and a standard deviation of 5% for the pipe throughput. On context switches, the virtual machines score better than the host, this is mostly due to caching mechanisms.

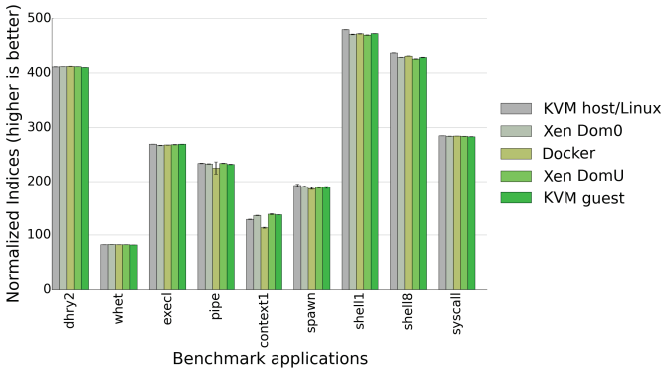


Fig. 3: UnixBench performance results

For all UnixBench tests and the Hackbench results, the overall performance overhead for the virtual machines and the container compared to the host is minimal.

B. IO performance

1) IOzone: IOzone disk tests have been performed with 4KB, 1MB and 10MB file sizes, 4KB being equal to the block size of the EXT4 file-system and 1MB and 10MB represent small to medium file sizes. Tests for 100KB and 2MB file sizes were also run, but provided very similar results to the one presented, and consequently are omitted for readability reasons. For the 1MB and 10MB tests, multiple results, corresponding to various record lengths (varying from 4KB to the closest power of two of the file size), were reported by IOzone and averaged for each of the presented file-system operations.

Fig 4 shows the test results for 4KB file size. We see that Xen DomU and KVM are scoring better for random write, backward read, stride read, record rewrite and all fwrite and fread operations. While, in the write operation the host is leading followed by Xen Dom0 and Docker, read, random read and re-read operations are showing too high standard deviations to determine which system is leading. Due to the small file size, caching mechanisms are heavily influencing the results.

In fig 5, were 1MB file size is used, the overall performance is increased for all operations and the standard deviations tend to be more contained. While Xen DomU and KVM are leading on the read related operations, the host and the privileged Xen domain perform better on write related operations.

For the 10MB file size, presented in fig 6, the host and Xen Dom0 are equally leading on all write related operation, however Xen Dom0 domain is performing better than the

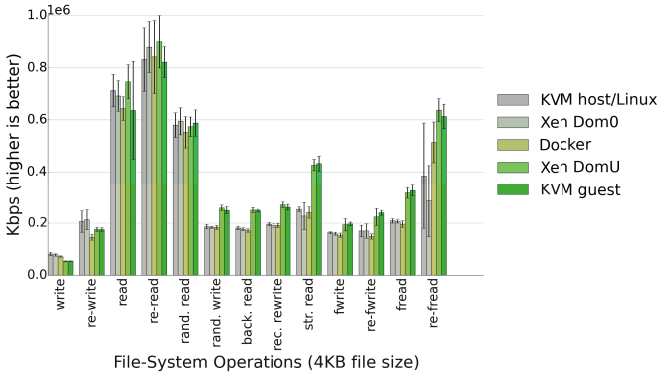


Fig. 4: IOzone performance results for 4KB file size

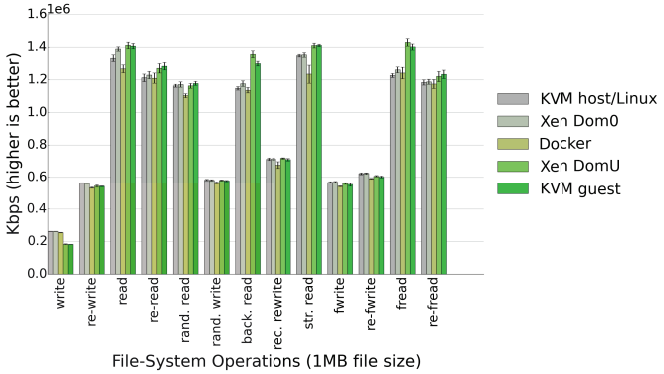


Fig. 5: IOzone performance results for 1MB file size

host on all read operations. KVM and XEN DomU show very similar performance and outperform Docker on every file-system operation aside from the standard write operation. Docker almost equalize the host and Xen Dom0 for each of the three file sizes in standard write operation.

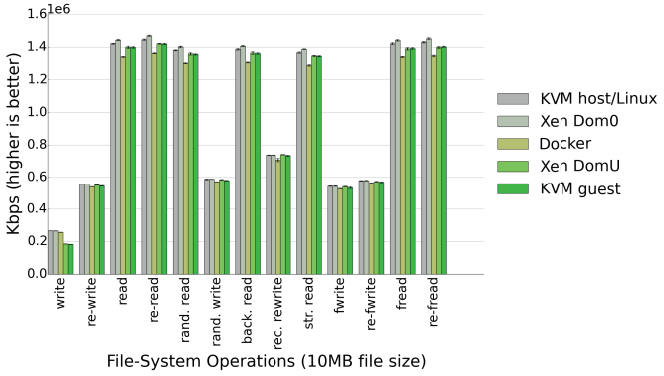


Fig. 6: IOzone performance results for 10MB file size

2) Netperf: On fig 7 the results of Netperf TCP bulk data transfer benchmark (TCP_STREAM), with different packet size, can be seen. From 64 bytes packet size on, all tested systems reached the maximum throughput and equalize the host performance, meaning, that in real world applications, bulk data transfer is efficiently handled by OS and hardware

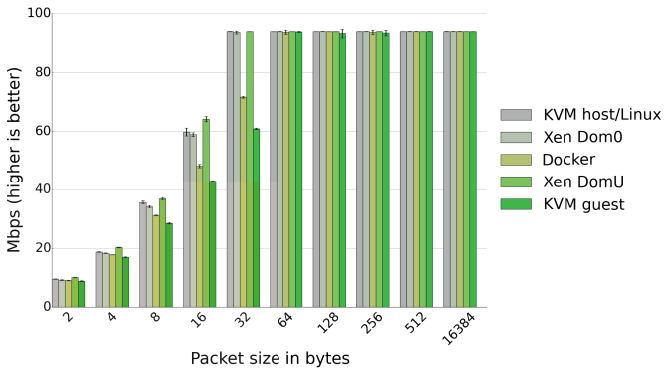


Fig. 7: TCP streaming performance with Netperf

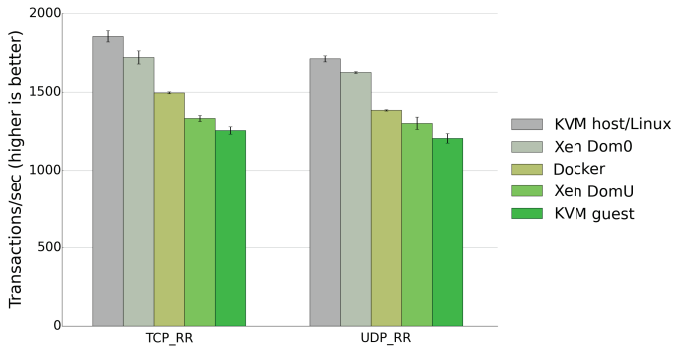


Fig. 8: TCP and UDP Request/Response performance with Netperf

virtualization and produce no overhead for 100Mbps networks for the container and virtual machines. However, smaller packet sizes require more CPU power and the maximum bandwidth of the network is not reached for any of the systems at 2, 4, 8 and 16 bytes packet sizes. The maximum throughput is reached by the host and the two Xen domains at 32 bytes packet size, leaving Docker and KVM with a respective maximum performance overhead of 23.5% and 35.5%. While KVM and Docker are always behind before reaching the maximum throughput of the network, their overhead is minimal for 2 and 4 bytes packet sizes. A small overhead can be seen for Xen Dom0, which reaches 4% at 8 bytes. Xen DomU has 6.5% at 16 bytes packet size, which can be seen as experimental anomalies.

Fig 8 shows the results obtained from Netperf request-response (RR) benchmark for the TCP and UDP protocol. For both TCP and UDP, KVM performs the smallest number of transactions per second and Xen Dom0 induces the smallest overhead compared to the host. Docker achieve more transactions per second than Xen DomU and KVM. For TCP, the performance overhead is of 7%, 19.5%, 28% and 32.5% for respectively Xen Dom0, Docker, Xen DomU and KVM. Relatively high standard deviation can be seen for Xen Dom0 (2.5%), the host (2%) and KVM (almost 2%). For UDP, while the overall performance for all systems is slightly worst as for TCP, the respective performance overheads for Xen Dom0, Docker, Xen DomU and KVM are 5%, 19%, 24% and 29%, which is rather better. Standard deviation for Xen DomU and KVM is of 3% and 2.5%. Here, OS virtualization performs better than hardware virtualization, but both of them produce a significant overhead.

V. Related Work

Virtual Machines were introduced in the 70s through IBM mainframes [20] and formal system requirement for virtualization were specified [21] within the same decade. The rise of hardware virtualization on x86 architectures began in the late 1990s with VMware, allowing full virtualization by using binary translation [22]. While para-virtualization was introduced on x86 platforms by Denali [23] in 2002, Xen developed its open-source solution one year later [24]. Xen supports full virtualization using the x86 hardware virtualization extensions since 2005 such as KVM, which was introduced on x86 in 2007 and merged in kernel version 2.6.20. Support for virtualization on ARM before the hardware virtualization extensions has been tried in various projects using para-virtualization solutions such as discussed in [25], [26] for Xen and [27], [28] for KVM. However, because, these projects required modifications in the guest kernel and were mostly slow, they are now replaced by the well supported Xen on ARM and KVM/ARM projects described in section 2.

The ancestor of nowadays containers is the chroot system call developed in the end of the 70s, providing rudimentary jails. FreeBSD Jails [29] extended this concept and appeared in the early 2000s, offering an isolation technology for FreeBSD. A first operating system virtualization implementation was designed by Linux-VServer [30] in 2001, which required to patch the linux kernel. In 2004, Oracle Solaris 10 [31] introduced an enhanced concept of jails called zones. Open Virtuzzo [32] (OpenVZ) provided an open source container technology with resource managements and live migration, requiring however an unofficial Linux kernel. Cgroups where merged into kernel version 2.6.24 in 2007 and allowed Linux Containers [33] (LXC) to provide the first container technology supported by Linux mainstream kernels, by taking advantage of cgroups and namespacing. Docker finally appeared in 2013 providing application containers and extra features to build and retrieve images as explained in section 2.

Recent comparisons between Xen and Kvm have been tried for x86 in multiple studies. In [34], which, compares both hypervisors to VMware ESXi for performance and energy efficiency, is highlighted the high overhead induced by virtualization on large scale HPC workloads. In another study [35] based on HPC Cloud environments, it is claimed that KVM is a better choice over Xen and Virtual Box on x86, because of its near-native performance and feature-rich experience. Older studies on x86, such as [36] and [37], compare Xen with KVM without virtIO. The former suggests to use Xen over KVM, because of KVM not being enough mature at the time, while in the latter KVM shows better isolation but at the same time, a worst scalability. Profiling methodologies for KVM on ARM were discussed in [38], which showed how to use the perf events subsystem of the Linux kernel and the ARM PMU counters to provide a performance analysis of the Cortex-A15. Performance overhead of KVM on ARMv7 was measured in [39] with Linux kernel 3.9, in which KVM just merged and showed satisfying results for the hypervisor, however with

some unexplained irregular behaviors in terms of performance. In [5], KVM/ARM on the Arndale Board and KVM on x86 performance overhead were compared for Linux 3.10, showing low overhead and similar performance for KVM on ARM. In [19], low performance overhead are measured for KVM on the TI OMAP 5432 board with Linux 3.12.4 and showing also the benefit of using VirtIO over QEMU. Low virtualization performance impact for Xen on ARM Cortex-A7 is shown in [40] for Linux 3.13. A recent comparison of Xen, KVM and Xvisor on Cortex-A7 regarding CPU overhead, memory bandwidth and lock synchronization is provided by [41] with Linux-3.16-rc3, claiming lower virtualization overhead for Xvisor and similar performance between Xen and KVM. However, in this study no I/O tests were executed.

While, according to [42], containers, and more especially Docker provide many advantages in the cloud over virtual machines (e.g., rapid deployment and portability across machines), [43] argues that VMs are more secure. In [44], Docker is presented as a tool to support reproducible research, thus adding an interesting use case to the container management tool. An early performance comparison of containers and Virtual Machines is tried in [45], showing VServer outperforming Xen on x86. Similarly, in a more recent study on x86 [46], Xen is outperformed by LXC, OpenVZ and VServer in HPC environments. A newer study compares LXC integrated in Docker and the CoreOS Linux distribution with Xen on x86, showing again better performance for the container, also pointing out the different use cases for both solutions. In [12] Docker is compared to KVM, showing slightly better results for Docker and arguing that some features in Docker present a trade off between ease of management and performance.

No performance comparison study between Docker and KVM or Xen on ARM has been tried, and there are only a few, above mentioned, papers presenting Xen and KVM performance overheads on ARM.

VI. Conclusion and Future Work

In this paper, an introduction to KVM, Xen and Docker system architectures for ARM has been provided. It has been discussed that KVM is easier to port on ARM than Xen, and that it can run any operating systems thanks to QEMU. Xen on ARM is limited to host operating systems that implement Xen's para-virtualization frontends, while Docker can only run Linux-based containers. Virtual machines share the hardware resources, containers share the operating system, which make Docker easier to use and faster to deploy but less secure than KVM and Xen.

Experimental test results for KVM, Xen and Docker on ARMv7 were presented, comparing the three solutions in terms of system and I/O performance overhead. Hackbench results showed negligible overhead for Docker and Xen privileged domain and a small comparable performance overhead for Xen and KVM guests. On the UnixBench test suite, comparable results for most of the tests have been observed. However, thanks to their caching mechanisms, hypervisors performed better on pipe-based context switches. For the same reason, IOzone reported high scores in most file-system operations for Xen and KVM, except for the write operation in which lower overhead was observed for Docker. Network performance analysis has been done using Netperf and showed near native performance for TCP throughput with packets of 64 bytes or higher for all solutions. For smaller packets, Xen has shown great capabilities and outperformed Docker and KVM. Request/response performance results have shown Docker scoring better than Xen and KVM guests, but here the performance overhead caused by each solution was high. Additionally, it has been seen that Xen privileged domain caused a small overhead compared to a non-virtualized host on most of the tests.

Overall, the benchmark results showed very low performance overhead for all three solutions, with some variations depending on the performed test. While hypervisors provide high isolation given by hardware extensions, containers are used nowadays for their ease to deploy and use. As a consequence, an alternative to explore is to combine the two technologies, by running containers inside virtual machines. In this way, cloud, HPC, NFV environments, avionic and automotive platforms can benefit from the best of each technology, taking into account the low overhead observed for both types of technology.

In addition, this study can be extented to consider multi core performance results and to include scalability tests, measuring performance isolation for multiple virtual machines and containers. Solutions involving combinations of containers and Virtual machines such as Hyper [47], using KVM or Xen with Docker, could be studied in terms of performance, deployment and security. It would also be interesting to measure KVM I/O performance on ARM with vhost-kernel networking and disk backend drivers enabled. Finally, a comparison of hypervisors and/or containers has to be done on ARMv8 64bits architecture.

Acknowledgment

This research work has been supported by the Seventh Framework Programme (FP7/2007-2013) of the European Community under the grant agreement no. 610640 for the DREAMS project.

References

[1] R. Mijat and A. Nightingale, "Virtualization is coming to a platform near you," *ARM White Paper*, 2011.

[2] "Arndale manual," October 2012. [Online]. Available: http://www.arndaleboard.org/wiki/index.php/WiKi

[3] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM Sigplan Notices*, 2006.

[4] "Kvm on arm upstreamed port." [Online]. Available: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=749cf76c5a363e1383108a914ea09530bfa0bd43

[5] C. Dall and J. Nieh, "Kvm/arm: the design and implementation of the linux arm hypervisor," *ACM SIGARCH Computer Architecture News*, 2014.

[6] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[7] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, 2008.

[8] "Xen project software overview." [Online]. Available: http://wiki.xen.org/wiki/Xen_Project_Software_Overview

[9] "Xen arm with virtualization extensions whitepaper," 2014. [Online]. Available: http://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper

[10] "Linux cgroup manual." [Online]. Available: http://man7.org/linux/man-pages/man5/systemd.cgroup.5.html

[11] "Linux namespaces manual." [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html

[12] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," 2014.

[13] E. W. Biederman and L. Networx, "Multiple instances of the global linux namespaces," in *Proceedings of the Linux Symposium*, 2006.

[14] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, 2014.

[15] "Netperf network benchmark." [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man8/hackbench.8.html

[16] A. R. Weiss, "Dhrystone benchmark: History, analysis, scores and recommendations," 2002.

[17] D. N. William and D. Capps, "Iozone file system benchmark," 2006.

[18] "Netperf network benchmark." [Online]. Available: www.netperf.org/

[19] G. Papaux, D. Gachet, and W. Luithardt, "Processor virtualization on embedded linux systems," in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*. IEEE, 2014, pp. 65–69.

[20] R. P. Goldberg, "Survey of virtual machine research," *Computer*, 1974.

[21] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, 1974.

[22] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, 2005.

[23] A. Whitaker, M. Shaw, S. D. Gribble *et al.*, "Denali: Lightweight virtual machines for distributed and networked applications," Citeseer, Tech. Rep., 2002.

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, 2003.

[25] D. Rossier, "Embeddedxen: A revisited architecture of the xen hypervisor to support arm-based embedded virtualization," *White paper, Switzerland*, 2012.

[26] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones," in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, 2008.

[27] C. Dall and J. Nieh, "Kvm for arm," *Proceedings of the Ottawa Linux Symposium*, 2010.

[28] J.-H. Ding, C.-J. Lin, P.-H. Chang, C.-H. Tsang, W.-C. Hsu, and Y.-C. Chung, "Armvisor: System virtualization for arm," in *Proceedings of the Ottawa Linux Symposium (OLS)*, 2012.

[29] P.-H. Kamp and R. N. Watson, "Jails: Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, 2000.

[30] B. Des Ligneris, "Virtualization of linux based computers: the linux-vserver project," in *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, 2005.

[31] D. Price and A. Tucker, "Solaris zones: Operating system support for consolidating commercial workloads." in *LISA*, 2004.

[32] K. Kolyshkin, "Virtualization in linux," *White paper, OpenVZ*, 2006.

[33] R. Rosen, "Linux containers and the future cloud," *Linux J*, 2014.

[34] S. Varrette, M. Guzek, V. Plugaru, X. Besseron, and P. Bouvry, "Hpc performance and energy-efficiency of xen, kvm and vmware hypervisors," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*, 2013.

[35] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox, "Analysis of virtualization technologies for high performance computing environments," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, 2011.

[36] A. Chierici and R. Veraldi, "A quantitative comparison between xen and kvm," in *Journal of Physics: Conference Series*, 2010.

[37] T. Deshane, Z. Shepherd, J. Matthews, M. Ben-Yehuda, A. Shah, and B. Rao, "Quantitative comparison of xen and kvm," *Xen Summit, Boston, MA, USA*, pp. 1–2, 2008.

[38] A. Motakis, A. Spyridakis, and D. Raho, "Introduction on performance analysis and profiling methodologies for kvm on arm virtualization," in *SPIE Microtechnologies*, 2013.

[39] L. Rasmusson and D. Corcoran, "Performance overhead of kvm on linux 3.9 on arm cortex-a15," *ACM SIGBED Review*, 2014.

[40] X. Gong, Q. Du, X. Li, J. Zhang, and Y. Lu, "Performance overhead of xen on linux 3.13 on arm cortex-a7," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on*, 2014.

[41] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded hypervisor xvisor: A comparative analysis."

[42] D. Liu and L. Zhao, "The research and implementation of cloud computing platform based on docker," in *Wavelet Active Media Technology and Information Processing (ICCWAMTIP), 2014 11th International Computer Conference on*, 2014.

[43] U. Gupta, "Comparison between security majors in virtual machine and linux containers," *arXiv preprint arXiv:1507.07816*, 2015.

[44] R. Chamberlain and J. Schommer, "Using docker to support reproducible research," 2014.

[45] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," *ACM SIGOPS Operating Systems Review*, 2007.

[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 *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, 2013.

[47] "Hypervisor-agnostic docker engine." [Online]. Available: http://hyper.sh/