This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

Virtualization on Internet of Things Edge Devices with Container Technologies: a Performance Evaluation

Roberto Morabito, *Ericsson Research*

Abstract— **Lightweight virtualization technologies have revolutionized the world of software development by introducing flexibility and innovation to this domain. Although the benefits introduced by these emerging solutions have been widely acknowledged in cloud computing, recent advances have led to the spread of such technologies in different contexts. As an example, the Internet of Things (IoT) and Mobile Edge Computing (MEC) benefit from container-virtualization by exploiting the possibility of using these technologies not only in data centers but also on devices, which are characterized by fewer computational resources such as single-board computers. This has led to a growing trend to more efficiently redesign the critical components of IoT/Edge scenarios (e.g., gateways) to enable the concept of device virtualization. The possibility for efficiently deploying virtualized instances on single-board computers has already been addressed in recent studies; however, these studies considered only a limited number of devices and omitted important performance metrics from their empirical assessments. This paper seeks to fill this gap and to provide insights for future deployments through a comprehensive performance evaluation that aims to show the strengths and weaknesses of several low-power devices when handling container-virtualized instances.**

Index Terms—Internet of Things; Edge Computing; Container Virtualization; Docker; Performance Evaluation.

I. Introduction

Over the last decade, the approach used to enhance the network efficiency and cope with the increasing number of connected devices to the Internet has been to move computation, control, and data storage into the cloud [11]. However, *cloud computing* now faces several challenges to meet the more stringent performance requirements of many application services, especially in terms of latency and bandwidth. *Edge computing* is an emerging paradigm that aims to increase infrastructure efficiency by delivering low-latency, bandwidth-efficient and resilient end-user services [12]. This *edge cloud* is not intended to replace centralized cloud-based infrastructure in its entirety, but rather to complement it by increasing the computing and storage resources available on the edge by adopting platforms that provide intermediate layers of computation, networking, and storage [13].

Especially in *Internet of Things* (IoT) scenarios, several edge processing tasks must be performed at the network edge on hardware characterized by processing, memory, and storage capabilities lower than is typical of server machines [7]. As an example, a *Single-Board Computer* (SBC) or comparable devices such as *Micro-Servers* can contain suitable hardware for performing such operations [28].

R. Morabito is with Ericsson Research, Jorvas, Finland, e-mail: roberto.morabito@ericsson.com

However, because of the limited computational capabilities of such devices, it is not always possible to deploy processing operations at the network edge. In this case, data centers are necessary to manage heavier computational requirements. Considering the heterogeneity of the entire scenario, there may be divergence (e.g., in terms of CPU architecture) between the various nodes involved. Simultaneously, however, the same software may need to be deployed at either the edge or in the data center. One way to ensure that data centers and constrained *edge entities* execute complementary software arises from the possibility of using lightweight virtualization technologies—in particular, containers. Container virtualization allows hardware resources to be decoupled from software, enabling packaged software to execute on multiple hardware architectures. Compared to alternative virtualization solutions such as hypervisors, container technologies provide several benefits such as rapid construction, instantiation, and initialization of virtualized instances. In addition, systems that rely on containers benefit from higher application/services allocations because of the smaller dimensions of the virtual images [14]. These container features are well-matched to the requirements of IoT/Edge scenarios.

Previous works have demonstrated the feasibility of using container technologies on IoT resource-constrained devices [6, 21]. However, the number of devices tested thus far has been extremely limited. For example, only the older version of the *Raspberry Pi* board has been considered as a suitable device. Furthermore, important performance metrics such as power consumption and energy efficiency have not been considered in previous analyses.

By investigating the performance of container virtualization on a wide set of low-power *edge devices* for the IoT, the objective of this paper is to provide insights for optimally using SBCs during the execution of virtualized instances. We adopt an empirical approach to quantify the overhead introduced by the virtualization layer under computing-intensive scenarios and networking-intensive traffic. Additionally, power consumption, energy efficiency, and device temperatures are considered in our analysis.

The remainder of this paper is organized as follows. Section II lists the related work. Section III provides background information about the hardware and software technologies employed in our study. Section IV gives a detailed description of the methodology and experimental setup used to carry out the empirical investigation. In Section V, we evaluate the impact of using container virtualization on top of different SBCs by taking various aspects into consideration. Section VI concludes the paper and provides final remarks.

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

II. RELATED WORK

The current literature includes several proposals for solutions in which virtualization technologies are employed at the network edge and/or on low-power nodes such as SBCs. However, Raspberry Pi is usually the only SBC considered for such deployments. Bukhary et al. in [4] evaluated Docker containers as an enabling technology for deploying an edge computing platform. The authors evaluated the technology in terms of (i) deployment and termination of services, (ii) resource and service management, (iii) fault tolerance, and (iv) caching capabilities, concluding that Docker represents a good solution for use in edge computing contexts. The authors of [5] and [6] included lightweight virtualization technologies in the design of an IoT gateway. Petrolo et al. [5] employed virtualized software to provide a dense deployment of services at the gateway level. Particularly interesting is their analysis of the possible interactions between IoT sensors and gateways. Such analysis suggests how the dynamic allocation of services, by means of containers, provides several benefits from a gateway performance perspective. In our previous work [6], we proposed a design for an IoT gateway that can also be efficiently employed in edge computing architectures. In that study, we showed how to efficiently and flexibly use Docker containers to customize an IoT platform that offers several virtualized services such as (i) *Device Management* capabilities, (ii) *Software Defined Networking* (SDN) support, and (iii) *Orchestration* and *Data Management* capabilities. In [7], container technologies were also used in a *Capillary Network* scenario. This study used Docker containers for packaging, deployment, and execution of software both in the cloud and in more constrained environments such as local *capillary gateways*. The dual purpose of these latter entities is to provide connectivity between short-range and cellular networks and to make different software components available for local device management and instantiation of *distributed cloud* processes. In [8] container-based virtualization and SBCs were identified as enabling technologies to enhance the provisioning of *IoT Cloud* services. Krylovskiy identified and analyzed the requirements for efficiently designing IoT gateways in [9]. The author considered the Docker containers to be suitable technology for meeting such requirements. In that study, several synthetic and application benchmarks were used to quantify the overhead introduced by the virtualization layer. However, the study's performance analysis was limited; it considered only the *Raspberry Pi* board. Furthermore, it lacked a comprehensive power consumption and energy efficiency evaluation. The *cloudlet* architecture proposed by the Carnegie-Mellon university [10], which represents an efficient approach for mobile-cloud computation can also be considered as linked to our work. Unlike the previous studies the granularity for virtualized instances was Virtual Machines (VMs). In [25], the authors proposed a *Container-based Edge Cloud PaaS Architecture* based on Raspberry Pi clusters, in which container technologies are used to favor migrating toward *edge cloud* architectures. The authors claim that the deployment of the Raspberry Pi clusters represents an enabling hardware technology to ensure important requirements such as cost-efficiency and low power consumption. However, this work lacked an empirical investigation to support its claims. Only the Raspberry Pi was considered for the edge cluster deployment; other hardware alternatives were not considered. Similarly, [26] used Docker containers and Raspberry Pi to leverage the deployment of a framework for *Fog Computing* networks. In addition, the evaluation of the overhead introduced by containers in SBC was limited when assessing the impact of different file-system configurations on disk performance. Specifically, Docker container performances were compared with native executions, and the study provided estimates of the overhead and delays introduced by different file systems (*AUFS*, *Device mapper*, and *OverlayFS*). Finally, in [27], Hajji et al. presented a detailed performance evaluation to understand the performance of a *Raspberry Pi cloud* when handling *big data* applications packaged through Docker containers.

Before moving on to the next section, we provide a brief overview on the real-world state of modern IoT applications. This overview can help readers understand—in conjunction with the empirical discussion that follows in this paper—what kinds of applications will perform better in the SBCs tested here; it is based on application performance requirements such as low energy consumption, high CPU and/or RAM performance, many simultaneous tasks, high-speed data connections, and so forth.

The IoT encompasses several domains including *E-health*, *Intelligent Transport Systems* (ITS), *Smart Cities*, *Smart Homes*, *Smart Industries*, etc. [42, 43, 44]. The applications in these categories give rise to a considerable number of use cases, and their performance requirements vary from case to case as do the ways such applications interact with IoT/Edge gateways. Indeed, the number of interactions between an IoT device and a gateway can vary significantly. Some types of sensors provide measured values to the gateway periodically (e.g., once per hour), as in, for example, *environment sensing*, *location sensing*, and *location info sharing* use cases [44]. The applications used in the gateway to handle such scenarios must be particularly efficient in terms of disk I/O responsiveness. In contrast, other types of devices such as sensors with cameras, may continuously generate significant amounts of data—e.g., for *remote control* use cases. For this second group of sensors, the gateway operations required are rather different compared to the first group. For example, video applications usually demand high CPU processing and network bandwidth. Furthermore, video data might be characterized by a high degree of redundancy. Data compression applications, which are particularly challenging in terms of CPU and memory, can be employed in the gateway to reduce the amount of data that must be transmitted over the uplink from the gateway to the cloud [6].

The energy efficiency of key IoT/Edge entities is particularly crucial for battery-powered wireless sensor networks [41]. From this viewpoint, understanding which SBCs ensure the best performance/energy efficiency

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

TABLE I. RASPBERRY PI AND ODROID HARDWARE FEATURES

	Raspberry Pi 2 Model B	*Raspberry Pi 3 Model B*	*Odroid C1+*	*Odroid C2*	*Odroid XU4*
Chipset	Broadcom BCM2836	Broadcom BCM2837	Amlogic S805	Amlogic S905	Samsung Exynos5422
CPU	Quad Core @900MHz ARMv7 Cortex-A7	Quad Core @1.2GHz ARMv8 Cortex-A53	Quad Core @1.5GHz ARMv7 Cortex-A5	Quad Core @2GHz ARMv8 Cortex-A53	Quad Core @2GHz ARMv7 Cortex-A15 Quad Core @1.4GHz ARMv7 Cortex-A7
Memory	1GB LP-DDR2 400MHz	1GB LP-DDR2 900MHz	1GB DDR3 792MHz	2GB DDR3 912MHz	2GB DDR3 912MHz
GPU	Broadcom VideoCore IV	Broadcom VideoCore IV	2 x ARM Mali-450 MP2 600 MHz	3 x ARM Mali-450 MP2 700 MHz	ARM Mali-T628 MP6
Ethernet	10/100 Mb/s	10/100 Mb/s	10/100/1000 Mb/s	10/100/1000 Mb/s	10/100/1000 Mb/s
Flash Storage	MicroSD	MicroSD	MicroSD, eMMC5.0	MicroSD, eMMC5.0	MicroSD, eMMC5.0
Connectivity USB	4× USB 2.0 Host	4× USB 2.0 Host	4× USB 2.0 Host, 1× USB 2.0 OTG	4× USB 2.0 Host, 1× USB 2.0 OTG	1× USB 2.0 Host, 2× USB 3.0 Host
OS	Linux, Windows 10	Linux, Windows 10	Linux, Android	Linux, Android	Linux, Android
Price	$35	$35	$37	$40	$74

tradeoff facilitates the design of such constrained networks. By considering a wide set of workloads, knowledge of the boards' energy efficiencies can help in estimating the energy lifetime of battery-powered nodes, and help SBCs offer support for cases where they must be battery powered.

Finally, recent works have revealed a growing trend toward designing gateways shared between different tenants [40]. In many cases, the container is the technology used to ensure isolation between different users. This represents a scenario in which there is a need to understand how IoT/Edge entities behave when receiving simultaneous virtualized instances and heterogeneous workloads.

III. ENABLING TECHNOLOGIES

In this section, we provide an overview of the different technologies involved in our study.

A. Container Virtualization Technology

Compared to hypervisors, container-based virtualization provides different abstraction levels regarding virtualization and isolation. Hypervisors virtualize hardware and device drivers, generating a higher overhead. In contrast, containers avoid such overhead by implementing process isolation at the operating system level [14]. A single container instance combines the application with all its dependencies; it runs as an isolated process in *user space* on the host operating system (OS), while the OS kernel is shared among all the containers (Fig. 1). The lack of the need for hardware/driver virtualization, together with the *shared kernel* feature, provide the ability to achieve a higher virtualized instance density because the resulting disk images are smaller. In our performance evaluation, we used Docker[1] containers to package virtualized instances. Docker introduces an underlying container engine and a functional API that supports easy construction, management and removal of containerized applications. Within a Docker container, one or more processes/applications can run simultaneously. Alternatively, an application can be designed to work in multiple containers, which can interact with each other through a linking system. This also guarantees that no conflicts will occur with other application containers running on the same machine.

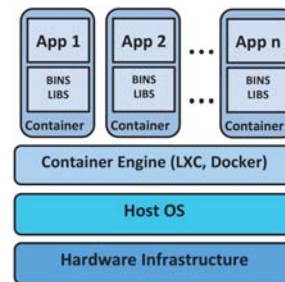

Fig. 1. Container-based virtualization architecture.

B. ARM-Based SBCs

The ARM architecture is becoming increasingly widespread due primarily to its low-power characteristics, low cost, and its use in smartphones, tablets, and other devices. [18]. Despite these characteristics, modern ARM multi-core processors can compete with general purpose CPUs [19]. Most current ARM processors are 32-bit, although the use of more powerful 64-bit devices is growing. These hardware enhancements are enabling the spread of solutions such as ultra-low power clusters [20] based on SBCs powered by ARM architectures.

IV. METHODOLOGY AND EXPERIMENTAL SETUP

This section provides information about the methodology and experimental setup used to perform this empirical investigation.

A. Tested Single-Board Computers

The wide range of SBCs used in our evaluation allows characterizing the performance and gaining a deep understanding of the potentialities of a wide set of devices. The devices selected for the analysis are described below.

The *Raspberry Pi 2*[2] *model B* (RPi2) is the *second generation* of the Raspberry Pi platform (Fig. 2a). Released in February 2015, RPi2 increased the performance and other hardware characteristics compared to previous versions. It includes a quad-core ARM Cortex-A7 CPU and 1 GB of RAM.

The *Raspberry Pi 3*[3] *model B* (RPi3) is the *third generation* of the Raspberry Pi platform (Fig. 2e) and was released in

[1]http://www.docker.io

[2]https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[3] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

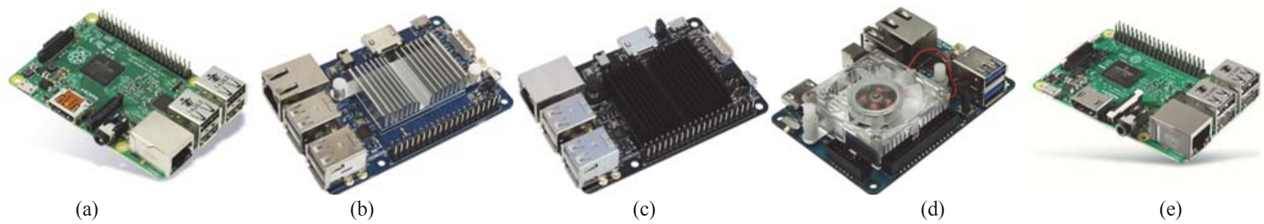

Fig. 2. Single-Board Computer under test: (a) RPi2. (b) OC1+. (c) OC2. (d) OXU4. (e) RPi3.

February 2016. It is the first model with a 64-bit CPU. The new model also integrates Bluetooth modules (4.1 and Low Energy) and Wi-Fi 802.11n (2.4 GHz).

Odroid is a series of SBCs manufactured by *Hardkernel Co., Ltd*. Most of the Odroid systems are capable of running both regular Linux and Android distributions.

Odroid C1+[4] (OC1+) was released in July 2015, replacing Odroid C1 with improved and enhanced hardware features (Fig. 2b). Although OC1+ represents the most similar board to the RPi2 in terms of its hardware characteristics, it includes a faster CPU and its Network Interface Card (NIC) has a faster line speed.

Odroid C2[5] (OC2) is the most recent *Odroid* SBC (released by Hardkernel in March 2016) and is the first Odroid model with a 64-bit CPU (Fig. 2c). Compared to OC1+, it includes additional RAM (2 GB).

Odroid XU4[6] (OXU4) features an ARM Octa-Core with big.LITTLE computing architecture (Fig. 2d). This chipset is characterized by a particularly heterogeneous CPU architecture that features two groups of cores: four ARM Cortex-A7 LITTLE cores (1.4 GHz), and four ARM Cortex-A15 big cores (2 GHz). The first group of cores reduces the power consumption at the expense of slower performance. In contrast, the second group consumes more power but features faster execution. The peculiarity of this architecture lies in the fact that if one core group is active, the other one is either powered down or used only if the first group saturates its resources. An application cannot work on both groups of cores at the same time.

Table I summarizes the hardware characteristics of the SBCs used in our study. The most relevant differences between the boards occur in terms of their CPU, flash storage, and Ethernet capabilities.

From a software perspective, for the Raspberry Pi we used an image provided by Hypriot running Raspbian Jessie with the Linux kernel 4.4.10 as a base OS. This image provides a lightweight environment optimized for executing Docker container technologies on top of Raspberry Pi devices. For the Odroid platforms, we used the following Hardkernel OS stable releases: Ubuntu version 14.04 for OC1+, Ubuntu version 16.04 for OC2, and Ubuntu 15.10 for OXU4.

All the SBCs were equipped with a 16 GB *Transcend Premium 400x Class 10 UHS-I microSDHC™* memory card for storage.

[4]http://odroid.com/dokuwiki/doku.php?id=en:odroid-c1
[5]http://odroid.com/dokuwiki/doku.php?id=en:odroid-c2
[6]http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu4

B. *Setup for Virtualized Environment*

The configuration used to customize the virtualized environment was similar for all the SBCs under evaluation. Docker version 1.12.0 was used as the container technology. The base Docker *images*—which represent the basic entity from which Docker containers are created—used to virtualize the software tools used for our benchmarking tests were as follows:

- ARMv7 *armhf/debian* image, with RPi2, OC1+, and OXU4.
- ARMv8 aarch64/debian image, with OC2 and RPi3.

In virtualization, *vCPU pinning* (or *processor affinity*) is a relevant aspect that influences the performance. vCPU pinning indicates the possibility of dedicating a physical CPU to a single instance or a set of virtual CPUs. Several vCPU pinning configurations exist. However, in the one selected for this analysis, each vCPU can run on any physical CPU core (Fig. 3). Such a configuration is typically termed a *random setup* and provides the advantage of higher CPU utilization [16].

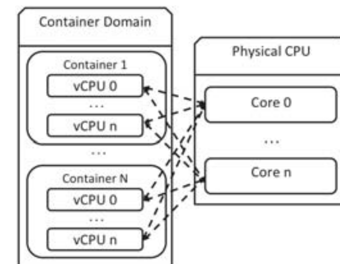

Fig. 3. CPU affinity setup.

C. *Testbed Setup for Power and Network Measurements*

The *power consumption* of the SBCs was measured using an external voltage meter (*USB-1608FS-Plus*), characterized by a 16-bit resolution and a setup similar to the one used in [15].

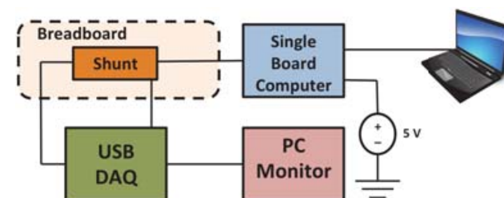

Fig. 4. Testbed setup.

The measurements involving network communications were performed using an Intel Core 2 Duo PC running Linux 3.13.0 with an Intel 82567LM Gigabit Ethernet card. The PC was directly connected to the NIC of the SBC under test. Fig. 4 shows the entire testbed environment setup.

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

D. Workloads for the Evaluation

As mentioned earlier, the main goal of our study was to understand how different SBCs react to specific workloads generated by applications running within Docker containers. Specifically, we want to characterize the performance from two different aspects. First, we want to define an upper bound for the performance of such devices when handling virtualized applications that challenge a particular segment of the underlying hardware. This kind of evaluation represents a key aspect because it allows us to verify that the virtualization layer does not generate excessive overhead that affects overall board performance. To achieve this, we used different benchmark tools to generate intensive *CPU*, *Disk I/O*, *Memory*, and *Network I/O* workloads. For the CPU and Network tests, we performed all the measurements using up to eight/sixteen guest domains. Second, we also considered heterogeneous virtualized instances to quantify any possible overhead introduced by containers; these measurements were made using applications that are closer to real-world workloads.

We adopted native performance (i.e., running the benchmark tool without including any virtualization layer) as a base case to quantify the overhead. We also repeated each measurement target with different tools, to analyze the consistency between different results. The results were averaged over 20 runs. The tables and graphs in this paper show the averages of such measurements.

V. Measurement Results and Analysis

In this section, we present the results of our performance analysis. The paragraph is organized into different subsections according to the specific workload being considered.

Before beginning the different subsections, Table II shows the power consumption of the five SBCs under test when in an *idle* state, without any virtualized entity running on them. The *sleep* Unix command was used to set the devices to *idle*, and the experiment lasted 300 seconds. OXU4 consumes the greatest amount of power—more than double the idle energy consumption of the Raspberry Pi.

TABLE II. Power Consumption in Idle

Device	*Power Consumption (Watts)*
RPi2	1.32
RPi3	1.42
OC1+	2.31
OC2	2.45
OXU4	3.99

A. CPU Performance

We tested CPU performance using *sysbench*[7]. This stress test is designed to challenge the CPU by calculating prime numbers. The computation is made by dividing each candidate number with sequentially increasing numbers and verifying whether the remainder (modulo calculation) is zero. Fig. 5 shows the *execution time* (measured in seconds) with up to sixteen concurrent running instances–both native and virtualized.

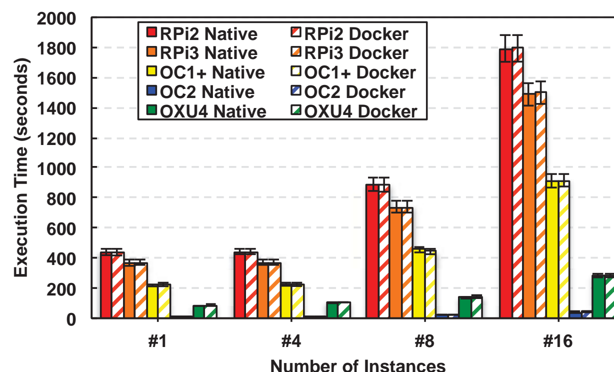

Fig. 5. Sysbench CPU stress test.

From Fig. 5, three main insights can be disclosed: i) the container engine introduces a negligible impact on CPU performance, with an approximately 2% percentage difference in the worst case; ii) OC2 significantly outperforms the rest of the tested SBCs; iii) for all devices, performance degradation can be observed when the number of concurrent instances exceeds four; however, the observed performance degradation when the number of instances to be managed exceeds four units is strictly related to the CPU architecture of the tested devices. For all the devices featuring a 4-core CPU, for four instances the CPU shares its resources among the different instances in a fair and effective manner. As the number of instances increases, the CPU must schedule and share its resources differently between the running instances, as four concurrent instances already saturate the maximum CPU capacity. Consequently, increasing the number of instances produces a gradual performance degradation. As Fig. 5 shows, the execution time doubles when the number of instances rises from four to eight and doubles again from eight to sixteen instances, revealing a linear increase. However, a different trend can be observed for the OXU4 SBC, which embeds an 8-core CPU. First, it must be clarified that if the OXU4 had an embedded CPU with 8-cores characterized by the same CPU clock, we would not have observed a performance degradation for fewer than eight instances because the CPU would have had enough resources to fairly manage all the virtualized instances. However, as discussed earlier, the OXU4 features a heterogeneous 8-core CPU in which the device begins using the lower-speed cores only when the faster cores are fully saturated. This CPU's architectural peculiarity produces the effect of a higher execution time as soon as the group of lower-speed cores begins working. In any case, in contrast to the 4-core CPU devices, the execution time grows linearly only when OXU4 is managing more than eight containers, corresponding to the point at which the OXU4's CPU resources are fully saturated.

From a power consumption perspective (Fig. 6), the consumption of the Raspberry Pi boards, OC1+, and OC2 vary, ranging from 1.88W (RPi2) to 3.26W (OC2). This variation range highlights the high energy efficiency of these devices.

[7]http://manpages.ubuntu.com/manpages/wily/en/man1/sysbench.1.html

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

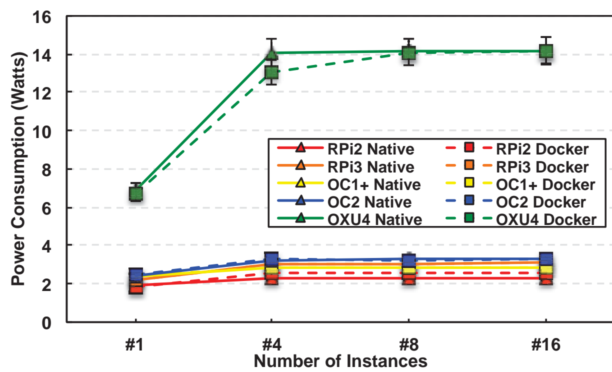

Fig. 6. Power consumption of the SBCs under evaluation while performing the sysbench test.

However, there is clearly a performance/power consumption tradeoff that varies from device to device. This analysis will be discussed at a later stage. OXU4 consumes more power consumption compared to the other tested devices. Specifically, its power consumption increases linearly from 7W to 14W as the number of containers varies from one to four. This significant power consumption variation is due to its use of the higher-speed core group. In fact, it can be observed that the increase diminishes as soon as the lower-speed cores start handling newly allocated instances: this change occurs when the fifth container is allocated. Another interesting aspect is that power consumption becomes constant as soon as the number of concurrent instances exceeds four for all the devices (with the exception of OXU4, which exhibits such behavior at more than eight instances). As discussed above, this trend depends strictly on the CPU architecture. When the number of running containers is greater than or equal to four (eight for OXU4) the CPU works at its maximum capacity and its resources are already saturated. Consequently, allocating additional instances does not increase power consumption; instead, such increases occur at the expense of performance as shown in Figure 5.

Linpack[8] tests system performance using a simple linear algebra problem. Specifically, this algorithm uses a random matrix **A** (of size N), and a right-hand side vector **B** defined as follows: **A** * **X** = **B**. Linpack provides the output result in *MegaFLOPS* (Millions of Floating Point Operations Per Second):

$$mflops = ops / (cpu * 1000000)$$

where *ops* denotes the number of operations per second performed, and *cpu* denotes the number of CPU cycles. In our evaluation, N was set to 1000.

Fig. 7 depicts the outcomes from the Linpack test. Similar to the previous case, the Docker virtualization layer introduces no relevant overhead.

Analysis of the results shows that the MFLOPS oscillate around the same values regardless of the number of running instances. A partial exception can be seen for OXU4, which shows this trend only when the number of running containers is larger than eight. Furthermore, in contrast with the sysbench test results, no performance degradation can be observed as the number of concurrent instances increases. This is because a rating based on MFLOPS is strongly dependent on and limited to the program being executed—Linpack in this case. This specific test scenario shows that even with a high number of concurrent instances, all the devices can execute the Linpack test at the same efficiency because they produce, on average, the same number of MFLOPS. The reason for the initial OXU4 performance deterioration is again attributable to its heterogeneous CPU architecture. Indeed, on OXU4, the execution of the benchmark test is initially allocated to the faster cores. When the resources of the faster cores are saturated, the lower-speed cores are activated to handle additional instances but with a reduced MFLOP capacity. This degrades the average performance. However, when the number of running containers is greater than eight, the MFLOPS oscillate around a constant value, as observed for the rest of devices. It can also be observed that OXU4 outperforms all the other SBCs.

With regard to power consumption, similar to the sysbench test, power consumption remains constant once four or more containers are in execution. However, the tested devices differ in how the power consumption increases from one to four instances. In particular, the devices featuring an ARMv7 CPU (RPi2 and OC1+) produce an increase of approximately 35%, while OC2 and RPi3, which both feature ARMv8 CPUs, generate an increase of roughly 50%. The CPU clock rate also influences SBC power consumption. In fact, among the ARMv8 devices, OC2 consumes an average of 1W more than RPi3; while among the ARMv7 devices, OC1+ results in a higher power consumption on the order of 0.9W. Therefore, these results reveal ways in which CPU architecture, CPU clock rate, and power consumption are related.

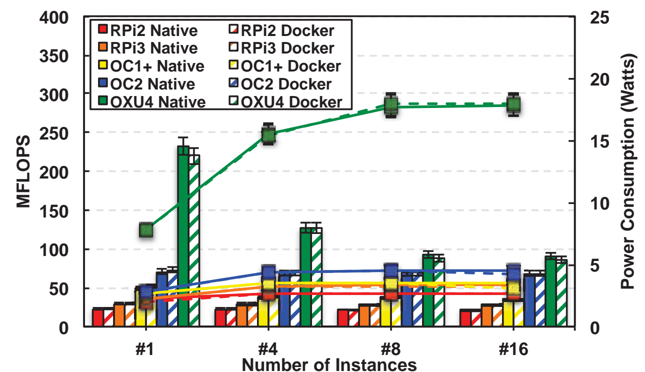

Fig. 7. Linpack test results. Line chart represents the power consumption.

B. *Memory Performance*

To test RAM memory performance, we used the Unix command *mbw*[9], which determines the available memory bandwidth by copying large arrays of data into memory. We also performed three other tests (*memcpy*, *dumb*, and *mcblock*).

Native and container performance can be considered comparable for each tested device (Fig. 8) with the exception of OXU4, which introduced an overhead of around 16% during the *memcpy* and *mcblock* tests. Comparing only the boards with 1 GB RAM, OC1+ always outperformed RPi2. This probably occurred due to the different RAM *I/O Bus Clock frequencies* and *data transfer*

[8]http://www.netlib.org/linpack/

[9]http://manpages.ubuntu.com/manpages/wily/en/man1/mbw.1.html

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

rates of the two devices (RPi2 uses LPDDR2 RAM, while OC1+ uses LPDDR3 RAM). RPi3 performed better than OC1+ on the *memcpy* and *mcblock* tests, but not on the *dumb* test. When evaluating this latter result, it must be considered that OC1+ has a faster data transfer rate (DDR) than RPi3, which uses RAM memory with a higher *Bus Clock frequency*. OC2 and OXU4 clearly produce higher *Average Speed* results compared to the other boards. These results can easily be explained by the larger RAM capacity of both boards. Nevertheless, OXU4 outperforms OC2 during the *memcpy* and *mcblock* tests despite having the same RAM hardware features. This result may be explained by the fact that these two operations require data to move over the *system bus*. The CPU may regulate data migration over the system bus. Consequently, the greater CPU computational resources of OXU4 compared to OC2 affect this result. Another interesting aspect of the RAM performance analysis comes from comparing OC1+ and OC2. Although a significant performance difference exists between the two boards (OC2 achieved roughly double the average speed of OC1+) the power consumption of both SBCs was approximately 3W.

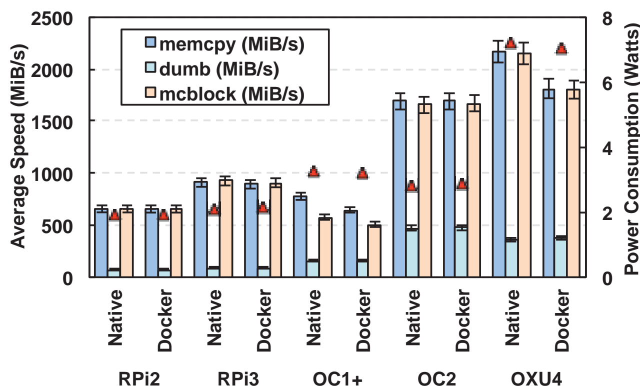

Fig. 8. Memory RAM performance comparison. The red markers represent power consumption.

Fig. 9 shows the sequential read and write performance averaged over 60 seconds, using a typical 1 MB I/O block size. Docker introduce negligible overhead in both tests for all the tested SBCs. with the exception of the Raspberry Pi boards, where the only significant overhead was introduced during the sequential write test, amounting to nearly 50% for RPi2 and approximately 37% for RPi3. We used the disk performance analysis tool *iostat*[11] to investigate the reason behind these Raspberry Pi results. The *iostat* tool can be used to monitor and report CPU statistics and system I/O device loading. From an analysis of the *iostat* logs, we noticed that the overhead may be caused by a high percentage of *iowait* periods. According to its definition, *iowait* indicates the percentage of time that the CPU is idle while the system services an outstanding disk I/O request. As explained in [45], a high *iowait* value indicates that the system has an application problem, an inefficient I/O subsystem configuration, or a memory shortage. The latter is possibly the reason the aforementioned overhead occurs. This disk stress test was performed with a high and intensive workload; therefore, devices with lower resources can experience issues in optimally scheduling disk-writing operations. When managing smaller files, it is reasonable to expect that such overhead would decrease, as was shown in [6, 21].

As discussed earlier, Disk I/O evaluation was performed using a *MicroSD* card as a storage device. However, unlike the Raspberry Pi family, all the Odroid boards provide integrated support for eMMC cards. This alternative storage solution offers superior performance in terms of read/write speed. We executed a random read/write benchmark test to explore the higher capabilities of eMMC storage solutions. The results are shown in Fig. 10, which shows that the eMMC cards can reach disk speeds on the order of hundreds of MBs per second—while for the MicroSD cards, memory speed oscillates around a range of hundreds of Mbs per second.

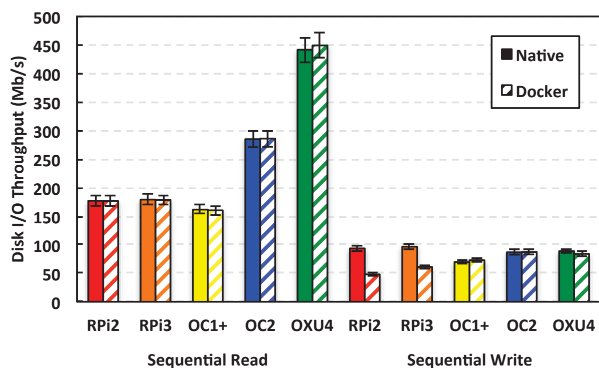

Fig. 9. Disk I/O performance for sequential read/write tasks.

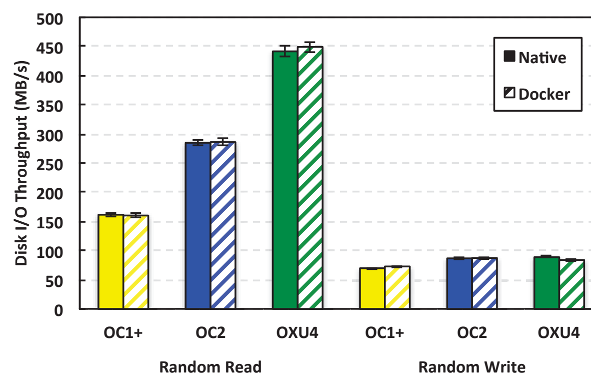

Fig. 10. Odroid boards eMMC performance test.

C. Disk I/O Performance

We used *fio*[10] *2.0.8* to run sequential read/write instances for a 6 GB file stored on the *MicroSD* card. *Sysbench* was used to test random read/write disk operations with the *embedded MultiMediaCard* (eMMC).

However, we note that disk performance is highly dependent on the type of MicroSD card used. As demonstrated in [46], above-average disk performance can be achieved based on the type of MicroSD used.

[10]http://linux.die.net/man/1/fio

[11]http://sebastien.godard.pagesperso-orange.fr/man_iostat.html

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

D. Network Performance

The network configuration used for our test is shown in Fig. 11. The *Virtual NIC* for all the running containers shares the same network bridge, which in turn is mapped to the physical Ethernet card. Each hardware platform performs network operations (e.g., packet forwarding, packet buffering, scheduling, etc.) dependent upon the design and implementation of their different software components (operating system, drivers, etc.), that could have different performance impacts.

We used the tool *iperf3*[12] for the network performance analysis. *Iperf* measures network performance between hosts, generating bidirectional data transfers of both TCP and UDP traffic.

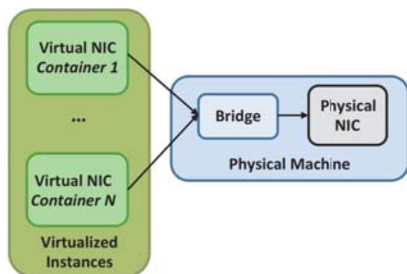

Fig. 11. Network configuration setup.

Taking into account that the NIC uses different code paths when sending and receiving TCP traffic, we performed bidirectional tests to quantify the overhead produced by the virtualization layer when the SBC is both receiving and sending network traffic. On the tested SBCs, both *iperf server* and *iperf client* inside one or multiple Docker containers were executed.

Docker uses NAT as its default network configuration. With the alternative configuration `--net=host`, Docker uses the host interface directly, avoiding NAT. This alternative configuration improves performance at the expense of security.

In our previous work [6], we tested the `--net=host` option for each experiment and found that this setup eliminated any overhead, allowing the systems to achieve nearly native performance. However, in this study we report the results only with the NAT setup because that configuration is most commonly used in real-world environments.

For the TCP traffic analysis, to improve the readability of the graphs, we discuss the performance of Raspberry Pi and Odroid separately because they are characterized by a different NIC speed.

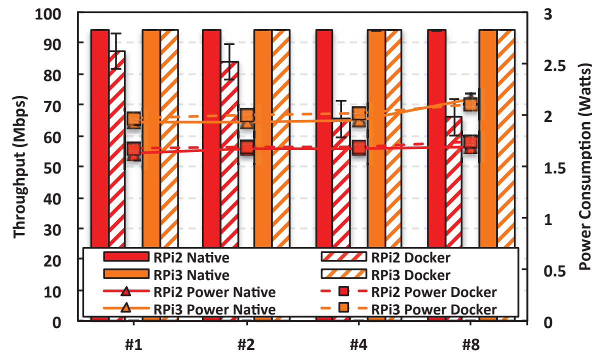

Fig. 12. Raspberry Pi 2 and Raspberry Pi 3 network performance while receiving TCP traffic. The line charts represent power consumption.

In the case of a TCP server (Fig. 12), the container engine does not impact the throughput performance of RPi2 and RPi3 when executing a single network instance. However, the overhead increases for the RPi2 according to the number of concurrent running instances (for eight simultaneous TCP flows, the overhead is approximately 30%).

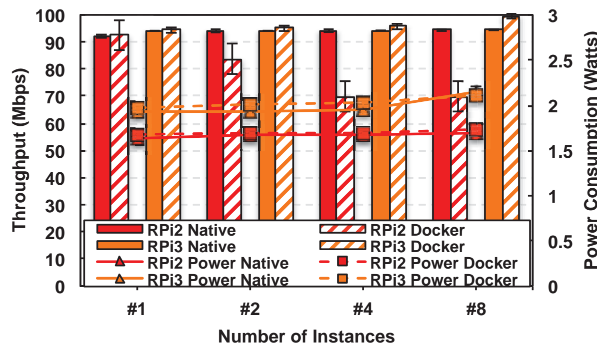

Fig. 13. Raspberry Pi 2 and Raspberry Pi 3 network performance while sending TCP traffic. The line charts represent power consumption.

A similar trend can be observed in Fig. 13, which depicts the case when RPi2/RPi3 are acting as clients. A throughput degradation of approximately 26% can be observed when sending even four TCP flows.

[12] https://iperf.fr/

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

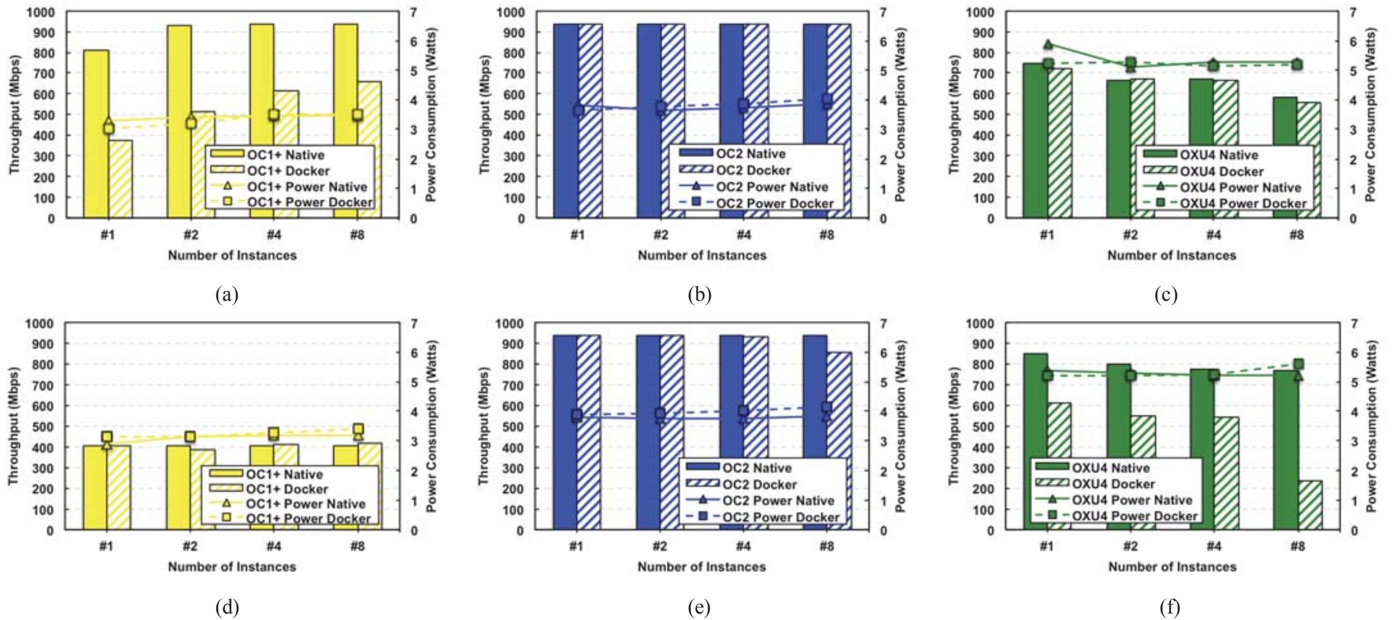

Fig. 14. Odroid boards TCP traffic results. TCP Server: (a) OC1+, (b) OC2, (c) OXU4, TCP Client: (d) OC1+, (e) OC2, (f) OXU4. The line charts represent power consumption.

The main findings of the TCP traffic analysis for the Odroid boards (Fig. 14) are summarized below.

OC1+. For native execution, a substantial throughput difference exists between the server (Fig. 14a) and client (Fig. 14d) cases: the client throughput is approximately half of the server throughput. This result occurs because of the different send and receive code paths that the OS uses for TCP traffic [29]. In terms of virtualization overhead, for the TCP client case the containers do not introduce any relevant impact; however, when OC1+ is receiving TCP traffic, Docker introduces an overhead of approximately 50% compared to native execution—for a single virtual instance. However, the performance of OC1+ improves as the number of instances increases. Such behavior is attributable to the fact that a single TCP flow is unable to saturate a 1 Gb/s link, while a combination of multiple streams overcomes this limitation [30].

OC2. The network performance of OC2 can be considered the desired outcome: both native and Docker performances are essentially the same (Fig. 14b–e). The only outlier can be identified in Fig. 14e, which depicts the overhead introduced by Docker when eight containers are simultaneously sending TCP traffic. The measured overhead is on the order of 10%. Such performance degradation is attributable to the CPU overload generated by managing eight concurrent virtualized streams. Fig. 14e shows a substantial increase in CPU context switching and cycles consumed compared to the native case.

OXU4. On average, the TCP server test shows that Docker negligibly impacts performance compared with native execution (Fig. 14c). Neverthless, the overall throughput decreases (by up to 23%) as the number of simultaneous connections increases. In contrast with OC1+, the simultaneous presence of parallel streams does not saturate the 1 Gb/s link; instead, it generates a performance degradation due to the CPU overload produced by the multi-stream flow—which is similar to the OC2 client case. However, different from the OC2 case, we can observe such results for both the native and virtualized cases. In the TCP client test, the virtualization layer produces a significant overhead ranging from 27% to 70% (Fig. 14f). Unlike the server case, in this test, the performance degradation affects only the virtualized instances.

For UDP traffic, we want to quantify the power consumption of the different SBCs when handling the same amount of sent/received traffic—90 Mb/s in our example. As Fig. 15 shows, the power consumption differs slightly from the *idle* power consumption reported in Table II, which implies that the NIC generates extremely low overhead when handling UDP traffic. Furthermore, no tangible differences can be observed between the native and Docker cases.

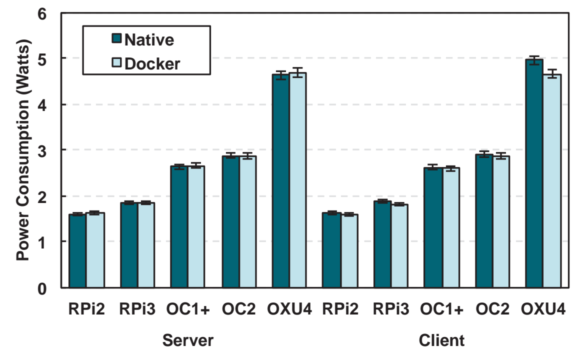

Fig. 15. Power consumption of the different SBC, when 90 Mb/s of UDP traffic is sent/received.

The TCP network performance analysis showed the existence of a non-negligible overhead introduced by Docker for a subset of cases in some of the tested boards (RPi2, OC1+, OXU4, and sin some cases, for OC2). Furthermore, we observed how the performance differs between servers and clients in such cases. To understand

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

the reasons behind these results, we used the Linux hardware performance analysis tool *Perf*[13] to collect system-level statistics. This tool reveals how hard the CPU works to deliver network traffic. As introduced in the result discussion, by analyzing the Perf logs, we discovered that the higher overhead generated in such SBCs is generated by an increasing number of CPU context switches and consumed cycles. This occurs because—in contrast to the native case—network packets must be processed by extra layers in a containerized environment. Another important insight of this analysis is related to the better network performance of both 64-bit CPU boards (RPi3 and OC2) compared to the 32-bit CPU boards (RPi2, OC1+, and OXU4). These results reflect the fact that Docker officially supports only 64-bit CPU systems; consequently, it appears to be better optimized for devices featuring 64-bit ARM architectures than those with 32-bit architectures. Finally, it must be pointed out that the inability to use the full capacity of the NIC interface in the Odroid boards has also been acknowledged by the manufacturer through similar benchmark tests [31, 32].

From the power consumption perspective, we can draw some general conclusions from the above results and discussion that apply to all the tested devices. The devices' power consumption follows the trend of the native network throughput in every single case. An increase/decrease in network throughput produces a consequent power consumption increase/decrease; although the network throughput variation is not as noticeable, this behavior can be observed in Figs. 14a and 14f. Although this result is expected, it is interesting to observe how the devices' power consumption for the Docker case follows the same trend as the native case. This occurs even when relevant overhead introduced by the virtualization layer exists—again, this trend is particularly noticeable in Figs. 14a and 14f. This outcome represents a bottleneck: the devices cannot be as efficient as in the native case, but at the same time they use identical rates of power consumption. Therefore, this result does not reflect a favorable tradeoff between performance and power consumption. As explained previously, the causes behind such bottlenecks are that the OS and/or a lack of software optimization require too many CPU clock cycles and overtax your system unexpectedly, which also impacts the power consumption.

E. Mixed Load Performance

The performance evaluation presented in the previous subsections was conducted using benchmark software tools that stress a specific hardware segment of the device. This represents a reasonable approach because it allows us to define an upper bound for the performance of each portion of the system hardware. However, real-world applications challenge the hardware in a more distributed manner. Therefore, we performed further tests to evaluate the impact introduced by container technologies when the SBCs are handling heterogeneous workloads. This evaluation was conducted using the *stress*[14] benchmark tool, which is a workload generator that allocates a configurable amount of load in the system in terms of CPU, memory, generic I/O, and disk operations. We defined three different workloads characterized by an increasing computational cost: (i) *Low Load*, (ii) *Average Load*, and (iii) *High Load*. Table III describes workload characteristics in more detail.

TABLE III. MIXED WORKLOAD CHARACTERIZATION

Workload	***Description***
Low	A load average of two is imposed on the system by specifying one CPU-bound process and one memory allocation process.
Average	A load average of three is imposed on the system by specifying one CPU-bound process, one memory allocation process, and one disk-bound process (50 MB).
High	A load average of four is imposed on the system by specifying one CPU-bound process, one memory allocation process, one disk-bound process (50 MB), and one generic I/O process.

For this set of tests, the metric that we want to monitor is *system load*, which indicates the overall amount of computational work that a system performs and includes all the processes or threads waiting on I/O, networking, database, etc. [22]. The *average load* represents the *average system load* over a period of time. In the evaluation, the time interval was set to 300 seconds. The aim of this evaluation was twofold. First, it allowed us further assess the impact of virtualization when executing applications characterized by heterogeneous characteristics. Furthermore, it allowed us to quantify the difference between the *average system load* of the different SBCs when assigned the same workload. The aforementioned metric can be collected by means of Unix tools such as *dstat*[15]. Fig. 16 shows the *1-min average system load* for the Odroid boards and RPi3. Various insights can be drawn from these results. First, for all the different workloads, native and Docker executions behave comparably. These results represent an important outcome: they confirm the lightweight characteristics of container technologies even when the SBC is handling mixed workloads. This result was confirmed when the system was assigned a *High* workload, which was defined to heavily challenge the system. A comparison of the different devices reveals that RPi3 produces the highest system load when compared to the Odroid boards.

[13]https://perf.wiki.kernel.org/index.php/Main_Page/
[14]http://people.seas.harvard.edu/~apw/stress/
[15]http://dag.wiee.rs/home-made/dstat/

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

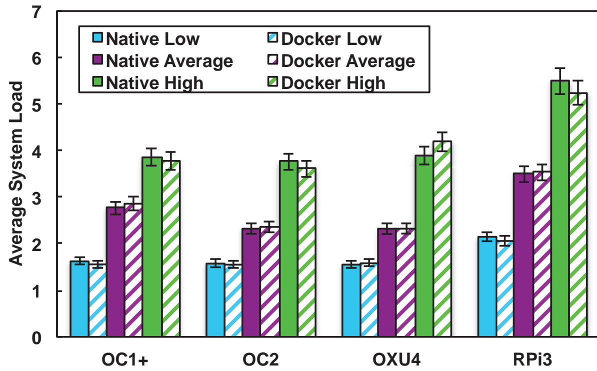

Fig. 16. Average system load (1-min) for three different heterogenous workloads.

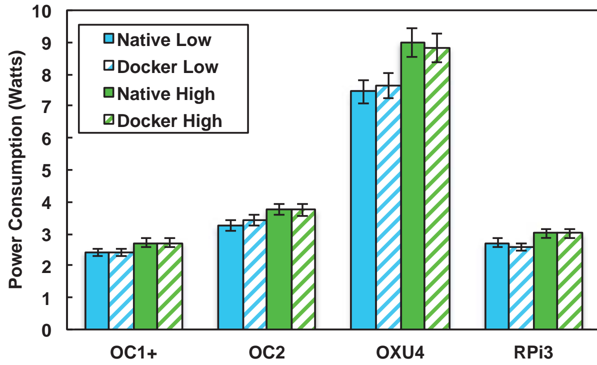

Fig. 17. SBC power consumption when executing the Low and High mixed workloads.

The difference increases with the workload complexity. Compared to the Odroid boards, RPi3 introduces a higher *average system load*. This increase is approximately 40% for the *low* and *average* workloads, and approximately 45% for a *high* workload. Fig. 17 shows the power consumption increase for the *low* and *high* workloads. The highest rise is produced by OXU4, on the order of 18%. For OC1+ and OC2, the increase is approximately 14%, although the base power consumption is higher in OC2 (which consumes roughly 1 W more than OC1+). In addition, the RPi3 generates a power consumption increase on the order of 17% when handling extremely different workloads.

F. Energy Efficiency Evaluation

In the preceding subsections, we used the term "*power consumption*" to indicate the device's average power consumption while handling a specific workload. In this section, we want to evaluate the *energy efficiency* of the tested hardware to assess which SBC is the most energy efficient.

In this context, energy consumption can be defined as the power consumption of a device over time:

$$energy = \int power\ dt\ [Joule] \tag{1}$$

To determine the energy of an SBC, we must consider the amount of computational work the SBC performs when executing a particular task. The computational work varies according to the hardware segment analyzed and the benchmark tool used to characterize its performance.

Table IV summarizes all the performance metrics used in our empirical investigation.

TABLE IV. BENCHMARK METRICS SUMMARY

Hardware	*Benchmark Tool*	*Performance Metric*
CPU	sysbench	# of events/seconds
	linpack	MFLOPS
Memory	mbw	MiB/s
	stream	MB/s
Disk I/O	fio, sysbench	Mb/s
Network	iperf3	Mb/s

Similar to [17], regardless of the considered metric and the particular test, we can conventionally and indistinctly define *transactions per time*

unit as the number of transactions per second (*tps*):

$$n\ \text{tps} = \frac{p}{m}\frac{transaction}{second} \tag{2}$$

Also, as stated in [17], *"Because of the transactions' dependency on the specific application scenario, only results from the same benchmark are comparable. Such performance figures must always be qualified by the respective benchmark."*

In our context, the *energy efficiency* expresses how efficiently an SBC completes a specific task using a certain amount of energy. *Energy efficiency* can be defined as follows:

$$energy\ \text{efficiency} = \frac{number\ \text{of transactions}}{energy\ consumption} \tag{3}$$

It can also be defined as the amount of work done per time unit given a certain amount of power:

$$energy\ \text{efficiency} = \frac{tps}{Watts} \tag{4}$$

Based on the above definition, the higher the energy efficiency is, the better an SBC transforms electricity into effective computation.

Fig. 18 depicts the energy efficiency of the different SBCs. The graphs show the percentage difference with respect to the SBC that performs most efficiently on each test, from an energy perspective.

To improve the readability of the graph, we consider only native performance because we have already empirically demonstrated that containers introduce no significant overhead for most tests.

The following insights were revealed from this analysis.

(i) *CPU*. SBC efficiency changes as the number of concurrent instances increases. The behavior of the OXU4 board is particularly interesting; it the most efficient SBC when a single instance is running but the least efficient when processing eight simultaneous instances (Fig. 18a).

(ii) *Memory*. The results of the memory stress (Fig. 18b) show that OC2+ is the most energy efficient board regardless of the benchmark tool used (*stress* or *mbw*);

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

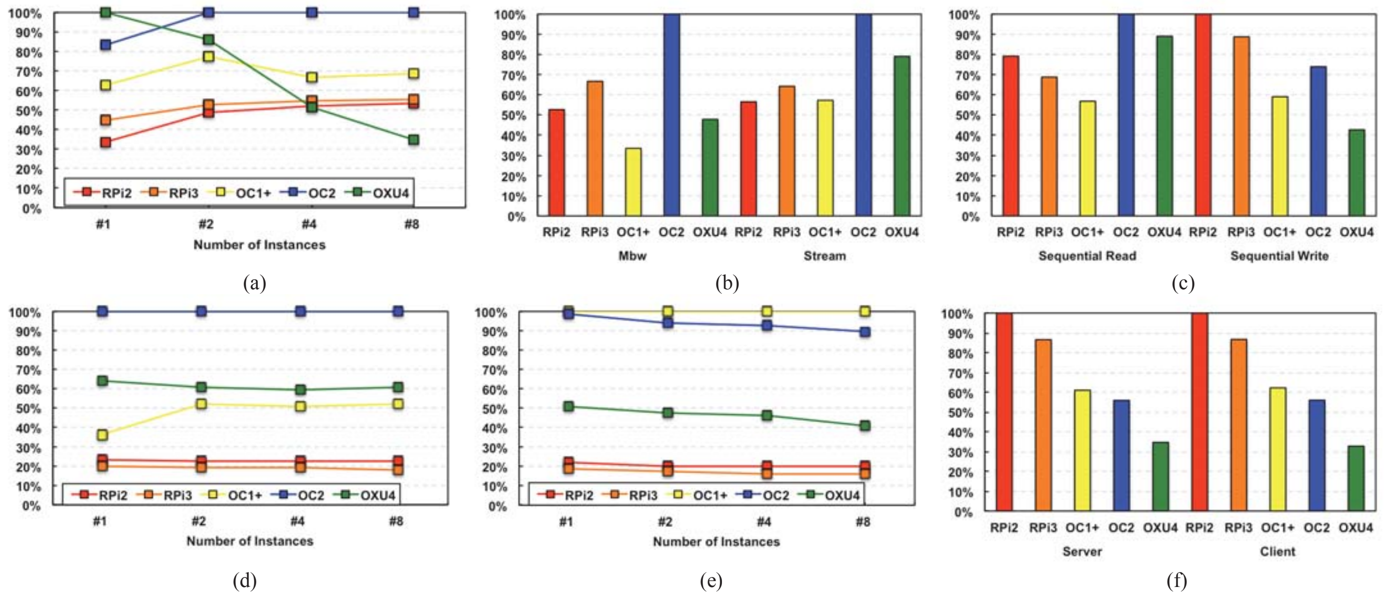

Fig. 18. Energy Efficiency Results: (a) CPU, (b) Memory, (c) Disk I/O. Network I/O: (d) TCP Client, (e) TCP Server, (f) UDP.

the other SBCs behave differently depending on the software used to perform the test.

(iii) *Disk I/O*. Similar to the memory case, the energy efficiency also varies in the disk I/O analysis based on the type of operation performed by the device (Fig. 18c). OC2 is the most efficient for *sequential read* operations, while RPi2 is the most efficient for *sequential write* operations.

(iv) *Network*. The results of the TCP network performance analysis were similar for most of the SBCs (RPi2, RPi3, and OXU4). Particularly interesting is that, in contrast to the TCP client test (Fig. 18d), OC1+ outperforms all the other boards in the TCP server evaluation (Fig. 18e). Finally, the UDP evaluation shows that the Raspberry boards are more efficient when compared to the Odroid boards (Fig. 18f). This result is expected based on the analysis discussed in the previous subsection.

Fig. 19 shows the energy efficiency results of the *mixed workload performance* analysis. OC1+ performs best because it considers the tradeoff between managing heterogeneous workloads and power consumption. However, the other boards also guarantee a medium-high level of efficiency.

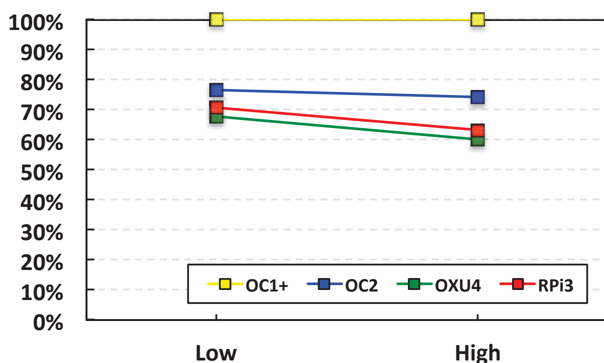

Fig. 19. SBC energy efficiency when executing the Mixed Load Test (Low and High workload configurations).

G. Container Activation Time Analysis

We also evaluated the variation in a *container's activation time* when the SBC is managing a workload that gradually becomes more complex. Activation time represents an issue for heavier types of system virtualization (i.e., Virtual Machines) on a server because booting a VM can require minutes depending on the server load. Container activation time was evaluated for server machines in [23]; however, a full characterization of container activation time on low-power nodes is lacking.

For this evaluation, we used *stress* to allocate increasing loads to the different systems. We tested the activation time in four different cases: when the SBCs are in idle state and then when two, four, or eight CPU-bound processes are imposed on the system. Fig. 20 shows the results of this evaluation. The RPi3 is the device that maintains the activation time within a very short range—on average between 1300 ms and 1400 ms—even when handling heavier workloads. The Odroid boards showed no relevant difference between *idle* and the `stress -c 2` case.

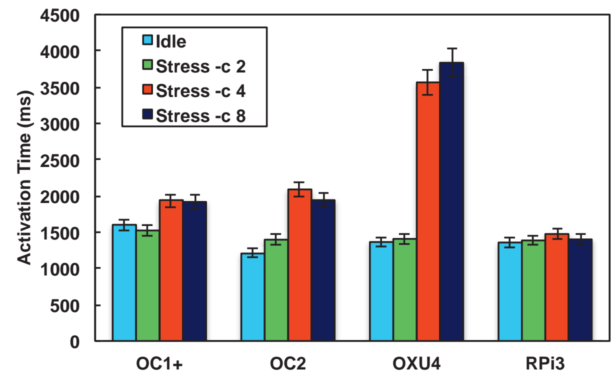

Fig. 20. Container activation time for OC1+, OC2, OXU4, and RPi3 under different workloads.

However, heavier workloads impact the activation time of the different Odroids differently. The percentage increase

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

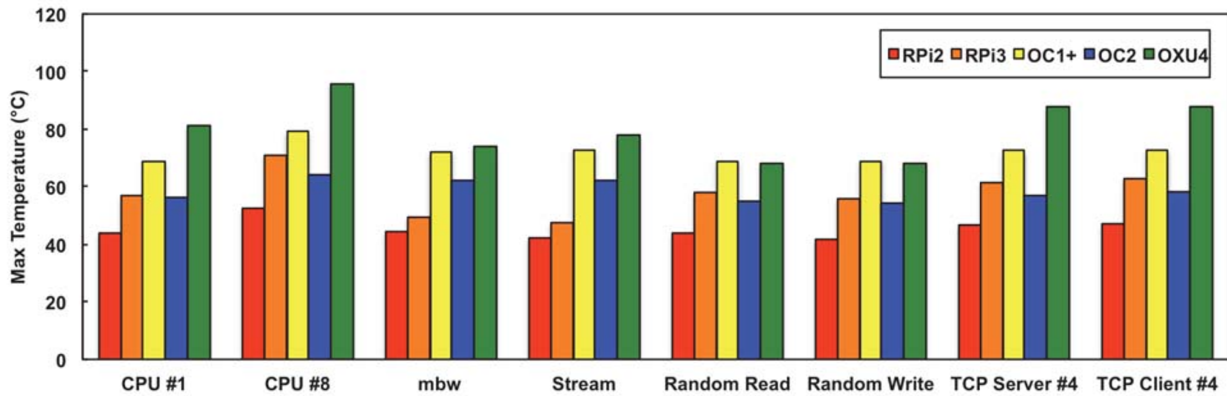

Fig. 21. Maximum temperatures reached by the SBCs when executing different computing tasks.

between the *idle* and the `stress -c 8` case is 20% for the OC1+, 61.57% for the OC2, and 181% for the OXU4. The OXU4 SBC substantially increases the activation time as the complexity of its workload increases. This result is consistent with the energy efficiency analysis, which showed the lower efficiency of OXU4 when managing heavy workloads and/or several concurrent instances. However, considering all the available results, container activation time remains below 2000 ms in most cases. This represents a significant result if we consider the reduced hardware capabilities of SBCs. Moreover that activation time can be further reduced through alternative container-engine setups [23].

H. Board temperature measurements

The temperatures reached by each SBC during the execution of the different benchmark tests is another interesting parameter that deserves to be analyzed. Temperature can be relevant in scenarios where SBCs are used on a large scale such as replacing server machines with SBC clusters, which can provide a better energy efficiency/monetary cost tradeoff as demonstrated in [24, 25]. Here, we want to estimate the maximum temperature reached by the different devices, which can help when designing efficient cooling systems for clusters of SBCs. The `vcgencmd measure_temp` command returns the CPU temperature of the Raspberry Pi, while the CPU temperature of the Odroid boards can be accessed using the command line `/sys/devices/virtual/thermal/temp`.

It is worth mentioning that, by default, RPi2 and RPi3 are passively cooled boards that do not include any heat sink or fan, while the Odroid boards require auxiliary components to ensure an effective cooling system. The processors on the OC1+ and OC2 boards have a relatively small area to dissipate heat. Therefore, both boards use a heat sink to improve heat dissipation. The OXU4 uses a software-controlled fan in addition to the heat sink. . Table V shows the temperature of each SBC when in an idle state.

TABLE V. BOARD TEMPERATURE AT IDLE

Device	*Temperature (°C)*
RPi2	36
RPi3	51
OC1+	70
OC2	54
OXU4	60

As in the previous subsection, we show only a subset of the full results (Fig. 21). The CPU analysis allows us to understand how the temperature increases as the number of concurrent virtualized instances (denoted by CPU #1 and CPU #8) increases. The RPi3 exhibits the highest temperature increase—approximately 25%. The temperature increase for the Odroid boards varies between 15% for OC1+ and OC2 to 19% for OXU4, and the RPi2 exhibits the same behavior. In the *Memory I/O* test, we included the results of both *mbw* and *stream* tests to determine whether any connection exists between temperature increases and the use of different benchmark tools. Regardless of the memory benchmark tool used, the boards' temperatures are roughly equivalent. The same logic applies to both the disk I/O and network analysis. For the former, we wanted to detect any potential difference between read and write disk operations and for the latter, we wanted to detect any potential differences when the devices were sending or receiving TCP traffic.

VI. CONCLUSIONS

The main goal of this paper was to conduct an extensive performance evaluation to assess the feasibility of running virtualized instances on a broad range of low-power nodes such as SBCs. The motivation behind this study lies in the increasing employment of such devices in specific *Edge-IoT* scenarios.

Our in-depth analysis of the empirical characteristics of SBCs generated fundamental insights about the performance of such devices, including the following:

- Employing container-virtualization technologies on SBCs produces an almost negligible impact in terms of performance when compared to native executions. This result remains valid even when running several virtualized instances simultaneously.
- By considering the tradeoff between performance and power consumption (energy efficiency) under a wide set

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

of workloads, we empirically demonstrated the energy efficiency of the SBCs. Energy efficiency represents a crucial aspect in scenarios in which the devices are battery powered. Indeed, from our study is possible to estimate the battery duration of a device based on expected workload characteristics.

- Overall, the Odroid C2 outperforms all the other tested devices in most of the performance tests.
- The Odroid boards can efficiently manage data-intensive applications (e.g., *Big Data* applications) thanks to their support for eMMC cards, which improve performance considerably compared to MicroSD.
- The Odroid C2 and the Odroid XU4 are the most suitable boards for executing memory-intensive applications, thanks to their higher RAM capacities.
- In general, the network performance analysis showed that 64-bit CPU devices do not introduce any tangible overhead compared to 32-bit CPU devices.
- Raspberry Pi boards are highly efficient when handling low volumes of network traffic—especially UDP traffic. This result can be useful in creating efficient IoT gateway designs that are specifically intended for executing lightweight IoT applications, e.g., the Constrained Application Protocol (*CoAP*) and Message Queuing Telemetry Transport (*MQTT*) protocols.
- By considering the limited resources of SBCs compared to server machines, we showed that the *container activation time* required by SBCs remains relatively small even when the SBCs are overloaded.
- The maximum temperatures reached by the various tested boards varies depending on the applied workload.

The choice of one device rather than another may vary based on the different requirements of service providers and applications. Therefore, the empirical insights achieved by this study can aid in efficiently designing integrations of the analyzed devices in different scenarios according to specific requirements of different Edge-IoT applications.

For the sake of completeness, it is worth highlighting that although our study has shown how container-based virtualization can represent an efficient and promising way to enhance the features of IoT architectures, several aspects still deserve further investigation, especially studies that improve our understanding of where this technology can be applied most efficiently. As an example, referring to the works mentioned in Section II, the advantages that accrue from executing containerized applications on IoT/Edge gateways are clear. However, there is still a lack of research to evaluate the interactions among multiple gateways while considering that Docker does not currently fully provide support to perform live container migrations between different entities. Furthermore, in such more complex scenarios, whether the strict requirements of many IoT applications can still be preserved should also be investigated.

The mobility of edge entities in several IoT/Edge use cases is another aspect that should be considered. Such scenarios introduce even more strict requirements in terms of latency. In [33], Farris et al. proposed an approach for provisioning ultra-short latency applications in MEC environments by exploiting the potential of container technologies. The proposed framework supports proactive service migrations only for stateless applications. The authors included a set of challenges that future research needs to address. Examples are support for *stateful* applications, a problem closely related to the lack of full support for live container migration, and defining specific policies aimed at optimizing container management.

Many concerns have been expressed about the level of security guaranteed by applications developed within containers [34]. One of the main concerns was due to the lack of *namespace* isolation, which made *dockerized* applications more vulnerable. The latest released versions of Docker include several security enhancements to cope with these issues [36]. Nonetheless, Docker continuously provides detailed guidelines for developing safer Docker ecosystems [37, 38]. A further effort to ensure better security in *dockerized* systems is represented by the collaboration between Docker and the *Center for Internet Security*, which has led to the release of the *Docker Security Benchmark*, a developer's tool that can check for a wide variety of known security issues within virtualized applications [39].

Referring specifically to IoT contexts, it is crucial to encourage the development of more specific security mechanisms that consider the strict requirements of IoT applications/scenarios but do not impair the lightweight features of container-based technologies. From this point of view, several Linux-Docker-based frameworks have already been proposed as solutions that can enhance IoT security [35].

ACKNOWLEDGMENTS

This work is partially funded by the FP7 Marie Curie Initial Training Network (ITN) METRICS project (grant agreement No. 607728). The author would like to thank Nicklas Beijar for his helpful feedback, and the reviewers for their insightful comments on the paper, as these comments led to an improvement of the work.

REFERENCES

[1] Patel, M., et al. "Mobile-Edge Computing Introductory Technical White Paper."*White Paper, Mobile-edge Computing (MEC) industry initiative* (2014).

[2] Bonomi, Flavio, et al. "Fog computing and its role in the internet of things."Proceedings of the first edition of the MCC workshop on Mobile cloud computing. ACM, 2012.

[3] Coady, Yvonne, et al. "Distributed cloud computing: Applications, status quo, and challenges." *ACM SIGCOMM Computer Communication Review* 45.2 (2015): 38-43.

[4] B. I. Ismail *et al.*, "Evaluation of Docker as Edge computing platform," *Open Systems (ICOS), 2015 IEEE Confernece on*, Melaka, 2015, pp. 130-135.

[5] Petrolo, Riccardo, et al. "The design of the gateway for the Cloud of Things."*Annals of Telecommunications* (2016): 1-10.

[6] R. Morabito, N. Beijar " Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies," To appear in Sensing, Communication, and Networking - Workshops (SECON Workshops), 2016 13th Annual IEEE International Conference on, London, 2016.

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2017.2704444, IEEE Access

[7] Novo Oscar, et al. "Capillary Networks – Bridging the Cellular and IoT Worlds". *Internet of Things (WF-IoT), 2015 IEEE World Forum on*. IEEE, 2015.

[8] Celesti, Antonio, et al. "Exploring Container Virtualization in IoT Clouds." *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2016.

[9] Krylovskiy, Alexandr. "Internet of Things gateways meet linux containers: Performance evaluation and discussion." *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*. IEEE, 2015.

[10] Satyanarayanan, Mahadev, et al. "The case for vm-based cloudlets in mobile computing." *IEEE pervasive Computing* 8.4 (2009): 14-23.

[11] H. Freeman and T. Zhang, "The emerging era of fog computing and networking [The President's Page]," in *IEEE Communications Magazine*, vol. 54, no. 6, pp. 4-5, June 2016.

[12] Chang, Hyunseok, et al. "Bringing the cloud to the edge." Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on. IEEE, 2014.

[13] Byers, Charles C., and Patrick Wetterwald. "Fog Computing Distributing Data and Intelligence for Resiliency and Scale Necessary for IoT: The Internet of Things (Ubiquity symposium)." Ubiquity 2015.November (2015): 4.

[14] Roberto Morabito, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. Lightweight Virtualization: a Performance Comparison." *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015.

[15] Kaup Fabian,et al. "PowerPi: Measuring and modeling the power consumption of the Raspberry Pi." *Local Computer Networks (LCN), 2014 IEEE 39th Conference on*. IEEE, 2014.

[16] C. Xu; Z. Zhao; H. Wang; R. Shea; J. Liu, "Energy Efficiency of Cloud Virtual Machines: From Traffic Pattern and CPU Affinity Perspectives," in *IEEE Systems Journal* , vol.PP, no.99, pp.1-11

[17] Schall, Daniel, Volker Hoefner, and Manuel Kern. "Towards an enhanced benchmark advocating energy-efficient systems." *Technology Conference on Performance Evaluation and Benchmarking*. Springer Berlin Heidelberg, 2011.

[18] Smith, Brad. "ARM and Intel battle over the mobile chip's future." *Computer*41.5 (2008): 15-18.

[19] Rajovic, Nikola, et al. "Tibidabo: Making the case for an ARM-based HPC system." *Future Generation Computer Systems* 36 (2014): 322-334.

[20] PicoCluster is Big Data in A Tiny Cube. [Online]. Available at: https://www.picocluster.com/, last accessed 09/July/2016

[21] R. Morabito, "A performance evaluation of container technologies on Internet of Things devices," *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, San Francisco, CA, USA, 2016, pp. 999-1000.

[22] R. Walker, "Examining load average," Linux Journal, vol. 2006, no. 152, p. 5, 2006.

[23] Harter, Tyler, et al. "Slacker: fast distribution with lazy docker containers." *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 2016.

[24] Tso, Fung Po, et al. "The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures." *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*. IEEE, 2013.

[25] Pahl, Claus, et al. "A container-based edge cloud PaaS architecture based on Raspberry Pi clusters." *Future Internet of Things and Cloud Workshops (FiCloudW), IEEE International Conference on*. IEEE, 2016.

[26] Bellavista, Paolo, and Alessandro Zanni. "Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi." *Proceedings of the 18th International Conference on Distributed Computing and Networking*. ACM, 2017.

[27] Hajji, Wajdi, and Fung Po Tso. "Understanding the Performance of Low Power Raspberry Pi Cloud for Big Data." *Electronics* 5.2 (2016): 29.

[28] R. Morabito, "Inspecting the Performance of Low-Power Nodes during the Execution of Edge Computing Tasks," 2017 IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, NV, 2017, pp. 148-153.

[29] Felter, Wes, et al. "An updated performance comparison of virtual machines and linux containers." *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 2015.

[30] Leitao, Breno Henrique. "Tuning 10Gb network cards on Linux." *Proceedings of the 2009 Linux Symposium*. 2009.

[31] Odroid C1. [Online]. Available at: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G141578608433, last accessed 11/Mar/2017.

[32] Odroid XU4. [Online]. Available at: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825, last accessed 11/Mar/2017.

[33] Farris, I., et al. "Providing ultra short latency to user centric 5G applications at the mobile network edge." *Transactions on Emerging Telecommunications Technologies* (2017).

[34] Security Risks and Benefits of Docker Applications Containers. [Online]. Available at: https://zeltser.com/security-risks-and-benefits-of-docker-application/, last accessed 17/Mar/2017.

[35] The Future of IoT: Containers Aim to Solve Security Crisis. [Online]. Available at: https://www.linux.com/news/future-iot-containers-aim-solve-security-crisis, last accessed 17/Mar/2017.

[36] DOCKER ENGINE 1.10 SECURITY IMPROVEMENTS. [Online]. Available at: https://blog.docker.com/2016/02/docker-engine-1-10-security/, last accessed 21/Mar/2017.

[37] Introduction to Container Security – Understanding the isolation properties of Docker. [Online]. Available at: https://www.docker.com/sites/default/files/WP_IntrotoContainerSecurity_08.19.2016.pdf, last accessed 21/Mar/2017.

[38] Docker security. [Online]. Available at: https://docs.docker.com/engine/security/security/, last accessed 21/Mar/2017.

[39] CIS Docker 1.12.0 Benchmark. [Online]. Available at: https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.12.0_Benchmark_v1.0.0.pdf, last accessed 21/Mar/2017.

[40] Morabito, Roberto, et al. "Enabling a lightweight Edge Gateway-as-a-Service for the Internet of Things." *Network of the Future (NOF), 2016 7th International Conference on the*. IEEE, 2016.

[41] Aziz, Azrina Abd, et al. "A survey on distributed topology control techniques for extending the lifetime of battery powered wireless sensor networks." *IEEE communications surveys & tutorials* 15.1 (2013): 121-144.

[42] ETSI, "Technical report 103 055," Technical report, 2011. [Online]. Available at: http://www.etsi.org/deliver/etsi_tr/103000_103099/103055/01.01.01_60/tr_103055v010101p.pdf, last accessed 24/Mar/2017.

[43] Al-Fuqaha, Ala, et al. "Internet of things: A survey on enabling technologies, protocols, and applications." *IEEE Communications Surveys & Tutorials* 17.4 (2015): 2347-2376.

[44] Chen, Shanzhi, et al. "A vision of IoT: Applications, challenges, and opportunities with china perspective." *IEEE Internet of Things journal* 1.4 (2014): 349-359.

[45] Summary for moitoring disk I/O. [Online]. Available at: https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.performance/summ_mon_disk_io.htm, last accessed 18/Feb/2017.

[46] microSD Card Benchmarks. [Online]. Available at: https://www.pidramble.com/wiki/benchmarks/microsd-cards, last accessed 21/Feb/2017

2169-3536 (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.