# A Performance Evaluation of Lightweight Approaches to Virtualization

Max Plauth, Lena Feinbube and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute for Software Systems Engineering,
University of Potsdam
Potsdam, Germany
e-mail: {firstname.lastname}@hpi.uni-potsdam.de

***Abstract*—The growing prevalence of the microservice paradigm has initiated a shift away from operating single image appliances that host many services, towards encapsulating each service within individual, smaller images. As a result thereof, the demand for low-overhead virtualization techniques is increasing. While containerization approaches already enjoy great popularity, unikernels are emerging as alternative approaches. With both approaches undergoing rapid improvements, the current landscape of lightweight approaches to virtualization presents a confusing scenery, impeding the task of picking an adequate technology for an intended purpose. While previous work has mostly dealt with comparing the performance of either approach with whole-system virtualization, this work provides an overarching performance evaluation covering containers, unikernels, whole-system virtualization, native hardware, and combinations thereof. Representing common workloads in cloud-based applications, we evaluate application performance by the example of HTTP servers and a key-value store.**

***Keywords*–Lightweight Virtualization; Performance; Unikernel; Container**

## I. Introduction

With the increasing pervasiveness of the cloud computing paradigm for all sorts of applications, low-overhead virtualization techniques are becoming indispensable. In particular, the microservice architectural paradigm, where small encapsulated services are developed, operated and maintained by separate teams, require easy-to-use and disposable machine images. Ideally, such infrastructure should allow for fast provisioning and efficient operation.

Approaches to lightweight virtualization roughly fall into the categories of *container virtualization* and *unikernels*. Both have been gaining notable momentum recently (see Figure 1). As more and more virtualization techniques are being introduced and discussed, making a choice between them is getting harder. Published performance measurements thus far either have a strong focus on throughput and execution time or focus on highlighting the strengths of one particular approach without comparing it to a broad range of alternative unikernels and container technologies.

We close this gap by presenting the results of an extensive performance analysis of lightweight virtualization strategies, which takes into account a broad spectrum both of investigated technologies and measured metrics. Our evaluation includes containers (*Docker*, *LXD*), unikernels (*Rumprun* and *OSv*), whole-system virtualization, native hardware, and certain combinations thereof. Our goal is to evaluate metrics that are applicable to cloud applications. For this purpose, we measure application throughput performance using HTTP servers and a key-value store.

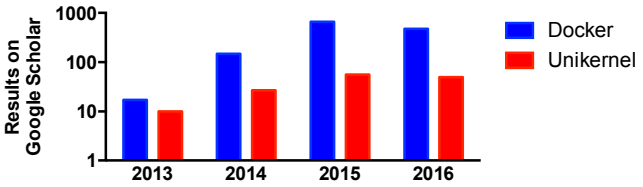



Figure 1. The increasing relevance of *Docker* and Unikernel in the research community is indicated by the number of search results on Google Scholar (as of October 19, 2016).

The remainder of the paper is organized as follows: Section II provides some background about the employed virtualization approaches. Section III reviews related work that deals with quantifying the performance impact of lightweight virtualization approaches. Afterwards, Section IV refines the scope of this work. Section V then documents the benchmark procedure yielding the results presented in Section VI. Finally, Section VII concludes this work with final remarks.

## II. Background

"Traditional", whole-system virtualization comes with performance and memory penalties, incurred by the hypervisor or *virtual machine manager* (VMM). This problem has been addressed by introducing *paravirtualization* (PV) and *hardware-assisted virtualization* (HVM). Still, the additional layer of indirection necessitates further context switches, which hurt I/O performance. [1] A further drawback of whole-system virtualization is the comparatively large memory footprint. Even though techniques such as *kernel samepage merging* (KSM) [2] have managed to reduce memory demands, they do not provide an ultimate remedy as they dilute the level of isolation among virtual machines [3].

This work focuses on lightweight virtualization approaches, which, addressing both issues, have gained notable momentum both in the research community and in industry (see Figure 1). Figure 2 illustrates how these approaches aim at supporting the deployment of applications or operating system images

 ISBN: 978-1-61208-529-6 

while eluding the overhead incurred by running a full-blown operating system on top of a hypervisor. With *containers* and *unikernels* constituting the two major families of lightweight virtualization approaches, the main characteristics and two representatives of each family are introduced hereafter.

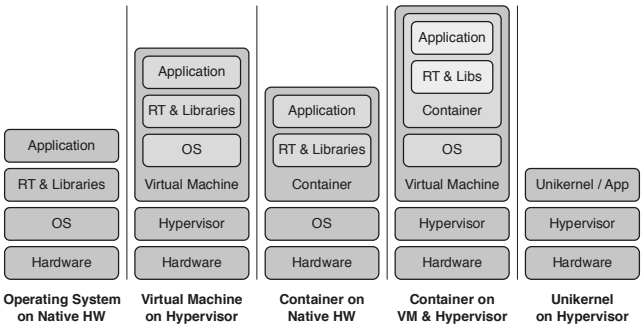



Figure 2. Illustrated comparison of the software stack complexity of various deployment strategies, including native setups, virtual machines, containers, containers within virtual machines and unikernels.

### A. Container (OS-Level Virtualization)

Containers were introduced as an oppositional approach to whole-system virtualization. They were based on the observation that the entire kernel induces overly much resource overhead for merely isolating and packaging small applications. Two classes of container virtualization approaches exist: application- and OS-oriented containers. For application-oriented containers, single applications constitute the units of deployment. For OS-oriented containers, the entire user space of the operating system is reproduced. This approach was particularly popular with *virtual private server* (VPS) solutions, where resource savings were essential. Currently, with *LXD*, this approach is becoming more prominent again, because it allows for the creation of *virtual machine* (VM)-like behavior without the overhead of a hypervisor.

In the following paragraphs, we discuss some common containerization technologies available. We do not consider orchestration-oriented tools such as Kubernetes [4], its predecessor Borg, or CloudFoundry's PaaS solution Warden [5] here.

*1) Docker:* Among the application-oriented containers, the open source project *Docker*[6] is currently the most popular approach. It relies on Linux kernel features, such as namespaces and control groups, to isolate independent containers running on the same instance of the operating system. A *Docker* container encapsulates an application as well as its software dependencies; it can be run on different Linux machines with the *Docker engine*.

Apart from providing basic isolation and closer-to-native performance than whole-system virtualization, *Docker* containerization has the advantages that pre-built *Docker* containers can be shared easily, and that the technology can be integrated into various popular *Infrastructure as a Service* (IaaS) solutions such as *Amazon web services* (AWS).

*2) LXD:* The Linux-based container solution *LXD* [7] builds up upon the *LXC* (Linux container) [8] interface to Linux containerization features. *LXD* uses the *LXC* library for providing low-overhead operating system containers. In addition to advanced container creation and management features, *LXD* offers integration into the OpenStack Nova compute component [9].

*3) lmctfy: lmctfy* (Let Me Contain That For You) [10] is an open source Google project which provides Linux application containers. It internally relies on Linux cgroups, and provides further user-mode monitoring and management features. Intended as an alternative to *LXD*, the status of *lmctfy* has been declared as stalled [11] on May 28, 2015, which is why we do not include *lmctfy* in our evaluation.

### B. Unikernel (Hypervisor Virtualization)

*Unikernels* are a reappearance of the library operating system concept, which only provides a thin layer of protection and multiplexing facilities for hardware resources whereas hardware support is left to employed libraries and the application itself. While library operating systems (e.g., Exokernel [12]) had to struggle with having to support real hardware, unikernels avoid this burden by supporting only virtual hardware interfaces provided by *hypervisors* or VMMs. [13] With the absence of many abstraction mechanisms present in traditional operating systems, the unikernel community claims to achieve a higher degree of whole-system optimization while reducing startup times and the VM footprint [14], [15].

*1) Rumprun:* The *Rumprun* unikernel is based on the *rump kernel* project, which is a strongly modularized version of the *NetBSD* kernel that was built to demonstrate the *anykernel* concept [16]. With the goal of simplified driver development in mind, the *anykernel* concept boils down to enabling a combination of monolithic kernels, where drivers are executed in the kernel, and microkernel-oriented user space drivers that can be executed on top of a rump kernel. One of the major features of the *Rumprun* unikernel is that it supports running existing and unmodified POSIX software[17], as long as it does not require calls to `fork()` or `exec()`.

*2) OSv:* The *OSv* unikernel has been designed specifically to replace general-purpose operating systems such as Linux in cloud-based VMs. Similarly to *Rumprun*, *OSv* supports running existing and unmodified POSIX software, as long as certain limitations are considered [18]. However, *OSv* provides additional APIs for exploiting capabilities of the underlying hypervisor, such as a zero copy API intended to replace the socket API to provide more efficient means of communication among *OSv*-based VMs.

## III. Related Work

Previous research has measured selected performance properties of lightweight virtualization techniques, mostly in comparison with a traditional whole-system virtualization approach.

 ISBN: 978-1-61208-529-6 

TABLE I. OVERVIEW OF RELATED WORK ON PERFORMANCE MEASUREMENTS OF LIGHTWEIGHT VIRTUALIZATION APPROACHES.

| | Docker? | LXC? | OSv? | OpenVZ? | Virtualization |
|---|---|---|---|---|---|
| [20] | ✓ | ✓ | | ✓ | N/A |
| [18] | | | ✓ | | KVM |
| [1] | ✓ | | | | KVM |
| [15] | ✓ | | | | KVM |
| [21] | | | ✓ | | Xen, KVM |
| [22] | ✓ | | | | AWS |

Felter et al.[1] have presented a comprehensive performance comparison between *Docker* containers and the *KVM* hypervisor [19]. Their results from various compute-intensive as well as I/O-intensive programs indicate that "*Docker* equals or exceeds *KVM* performance in every case tested". For I/O-intensive workloads, both technologies introduce significant overhead, while the CPU and memory performance is hardly affected.

Kivity et al.[18] focus on the performance of *OSv* in comparison to whole-system virtualization with *KVM*. Both micro- and macro-benchmarks indicate that *OSv* offers better throughput, especially for memory-intensive workloads.

Table I further summarizes the most recent publications of performance measurements of lightweight virtualization techniques.

## IV. SCOPE OF THIS WORK

Here, we present an extensive performance evaluation of containers (*Docker*, *LXD*), unikernels (*Rumprun* and *OSv*), and whole-system virtualization. Related work has focused on subsets of the approaches we consider, but we are not aware of any comprehensive analysis of up-to-date container versus unikernel technologies. Our goal is to present data which is applicable to cloud-based applications.

Our research questions are the following:

- How fast are containers, unikernels, and whole-system virtualization when running different workloads? Do the results from related work hold in our test case?
- What is the most suitable virtualization technology for on demand provisioning scenarios?

## V. BENCHMARK PROCEDURE

This section provides a description of the benchmark methodologies applied within this work. All tests were performed on the test system specified in Table II. Where applicable, all approaches were evaluated using *KVM* and native hardware. For container-based approaches, we also distinguish between native and virtualized hosts, where the latter represents the common practice for deploying containers on top of IaaS-based virtual machines.

Representing common workloads of cloud-hosted applications, we picked HTTP servers and key-value stores as exemplary applications for application performance measurements.

TABLE II. SPECIFICATIONS OF THE TEST SYSTEMS.

| | |
|---|---|
| Server model | HPE ProLiant m710p Server Cartridge |
| Processor | Intel Xeon E3-1284L v4 (Broadwell) |
| Memory | 4 × 8GB PC3L-12800 (SODIMM) |
| Disk | 120GB HP 765479-B21 SSD (M.2 2280) |
| NIC | Mellanox Connect-X3 Pro (Dual 10GbE) |
| Operating system | Ubuntu Linux 16.04.1 LTS |

As these I/O-intensive use cases involve a large number of both concurrent clients and requests, the network stack contributes to the overall application performance considerably. Hence, in order to eliminate an unfavorable default configuration of the network stack as a confounding variable, we modified the configuration on Linux, *Rumprun* and *OSv*. Since many best practices guides cover the subject of tuning network performance on Linux, we employed the recommendations from [23], resulting in the configuration denoted in Table III.

TABLE III. OPTIMIZED SETTINGS FOR THE *Linux* NETWORK STACK.

| Path | Parameter | Value |
|---|---|---|
| /etc/sysctl.conf | fs.file-max | 20000 |
| /etc/sysctl.conf | net.core.somaxconn | 1024 |
| /etc/sysctl.conf | net.ipv4.ip_local_port_range | 1024 65535 |
| /etc/sysctl.conf | net.ipv4.tcp_tw_reuse | 1 |
| /etc/sysctl.conf | net.ipv4.tcp_keepalive_time | 60 |
| /etc/sysctl.conf | net.ipv4.tcp_keepalive_intvl | 60 |
| /etc/security/limits.conf | nofile (soft/hard) | 20000 |

Based on this model, we modified the configuration parameters of both *Rumprun* and *OSv* to correspond to the Linux-based settings [24]. Currently, there is no mechanism in *Rumprun* to permanently modify the values of the *ulimit* parameter. As a workaround, the *Rumprun* sysproxy facility has be activated by passing the parameter `-e RUMPRUN_SYSPROXY=tcp://0:12345` to the `rumprun` command-line utility upon start. Using the `rumpctrl` utility, the configuration values of the *ulimit* parameter have to be changed remotely, as exemplified in Listing 1.

```
1 export RUMP_SERVER=tcp://[IP]:12345
2 . rumpctrl.sh
3 sysctl -w proc.0.rlimit.descriptors.soft=200000
4 sysctl -w proc.0.rlimit.descriptors.hard=200000
5 sysctl -w proc.1.rlimit.descriptors.soft=200000
6 sysctl -w proc.1.rlimit.descriptors.hard=200000
7 sysctl -w proc.2.rlimit.descriptors.hard=200000
8 sysctl -w proc.2.rlimit.descriptors.soft=200000
9 rumpctrl_unload
```

Listing 1. The *ulimit* values of *Rumprun* have to be changed remotely using the *sysproxy* facility and the associated `rumpctrl` utility.

### A. Static HTTP Server

We use the *Nginx* HTTP server (version 1.8.0) to evaluate the HTTP performance for static content, as it is available on all tested platforms. Regarding *OSv* however, we refrain from running HTTP benchmarks due to the lacking availability of an adequate HTTP server implementation.

To be able to deal with a high number of concurrent requests, we apply optimized configuration files for *Nginx*. Our measurement procedure employs the benchmarking tool *weighttp* [25] and the *abc* wrapper utility [26] for automated benchmark runs and varying connection count parameters. The *abc* utility has been slightly modified to report standard

 ISBN: 978-1-61208-529-6

deviation values in addition to average throughput values for repeated measurements. The benchmark utility is executed on a dedicated host to avoid unsolicited interactions between the HTTP server and the benchmark utility. While HTTP server benchmark guidelines usually recommend executing both HTTP server and benchmark utility on the same machine [23], we intentionally included the traversal of an actual network in the setup to represent real-world conditions more accurately. As static content, we use our institute website's *favicon* [27]. We measured the HTTP performance ranging from 0 to 1000 concurrent connections, with range steps of 100 and *TCP keepalive* being enabled throughout all measurements.

### B. Key-Value Store

In our second application benchmark discipline, we use *Redis* (version 3.0.1) as a key-value store, which is available on all tested platforms. In order to rule out disk performance as a potential bottleneck, we disabled any persistence mechanisms in the configuration files and operate *Redis* in a cache-only mode of operation. For executing performance benchmarks, we use the *redis-benchmark* utility, which is included in the *Redis* distribution. The benchmark utility is executed on a separate host to represent real-world client-server conditions more accurately and to avoid unsolicited interactions between the benchmark utility and the *Redis* server. We measured the performance of GET and SET operations ranging from 0 to 1000 concurrent connections, with range steps of 100 and both *TCP keepalive* and pipelining being enabled throughout all measurements. The CSV-formatted output of *redis-benchmark* was aggregated to yield average values and standard deviation using a simple python script.

## VI. Results & Discussion

Here, we provide and discuss the results obtained from the benchmark procedure elaborated in Section V in an analogous structure. In order to retrieve a sufficiently meaningful dataset, each condition was benchmarked 30 times [28]. Furthermore, each benchmark was preceded by a warm-up procedure. For a statistically meaningful evaluation of the collected data, an ANOVA test and a post-hoc comparison using the Tukey method were applied. All values are expressed as mean $\pm$ SD (n = 30).

### A. Static HTTP Server

Container-based approaches are generally expected to introduce little overhead compared to the native operating system performance. While this appears to be true for disk I/O, memory bandwidth, and compute performance [1], networking introduces a significant amount of overhead ($p < 0.0001$) as illustrated in Figure 3a. A likely cause for this overhead is that all traffic has to go through a NAT in common configurations for both container-based approaches. Comparing containers with whole-system virtualization, it does not come as a surprise to see significant performance advantages on the side of containers for 200 concurrent clients and above ($p < 0.0001$).

We also considered the condition where containers are executed on top of whole-system virtualization images. This setup reflects the common practice in IaaS scenarios where containers are usually deployed on top of a virtual machine instead of a native operating system instance. When deployed above a hypervisor, containers evince a similar behavior as in the native use case: Containers add significant overhead on top of a virtualized Linux instance ($p < 0.0001$).

Proceeding with the evaluation of unikernel performance, it is surprising to see a similar performance of *Rumprun* compared to containers. Even though *Rumprun* can achieve slim performance enhancements over containers, significant improvements start to join in merely for 600 concurrent clients and more ($p < 0.0001$). At first sight, these results may appear disappointing given the fact that unikernels should offer better performance in I/O intensive workloads due to the absence of context switches. However, we suspect that the mediocre performance originates from comparing apples with oranges, as HTTP-servers heavily rely on the performance of the operating systems network stack, where the Linux networking stack has undergone massive optimization efforts that the *NetBSD* network stack can hardly compete with. To verify this hypothesis, we performed a brief evaluation where we executed the same benchmark setup using *NetBSD* 7.0.1 instead of *Ubuntu 16.04*. For that purpose, we used a *KVM*-based virtual machine and the same network configuration parameters as in the other setups. Here, we obtained performance measurements much slower than *Rumprun* (data not shown), which demonstrates the potential of the unikernel concept with *Rumprun* outperforming a virtualized instance of its full-grown relative *NetBSD*. With further optimizations of the network stack, *Rumprun* might achieve similar or even better performance than a regular Linux-based virtual machine.

Regarding the memory footprint, unikernels manage to undercut the demands of a full-blown Linux instance (see Figure 4a). However, containers still can get by with the least amount of memory. The major advantage of containers remains that memory can be allocated dynamically, whereas virtual machines are restricted to predefining the amount of allocated memory at the time of instantiation.

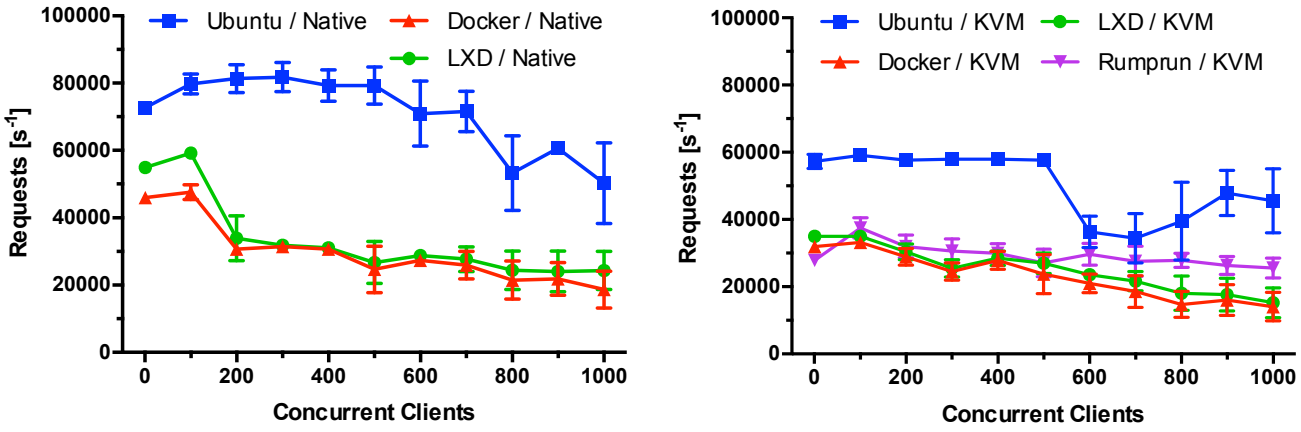
### B. Key-Value Store

As illustrated in Figure 5, the key-value store exhibits similar results regarding container-based approaches and whole-system virtualization: Regardless of native or virtualized deployments, containers come with a significant amount of overhead ($p < 0.0001$). In contrast, *Rumprun* and *OSv* offer slight but nevertheless significant performance improvements compared to Linux under many conditions. With *Redis* being less sensitive to the performance of the network stack, this use case demonstrates the potential of unikernels. Regarding memory consumption (see 4b), containers still offer the highest degree of flexibility. While *Rumprun* still undercuts the memory footprint of Linux, *OSv* required distinctly more memory in order to withstand the benchmark. However, this increased memory demand appears to be caused by a memory leak or a similar bug in the *OSv*-port of *Redis*.

## VII. Conclusion

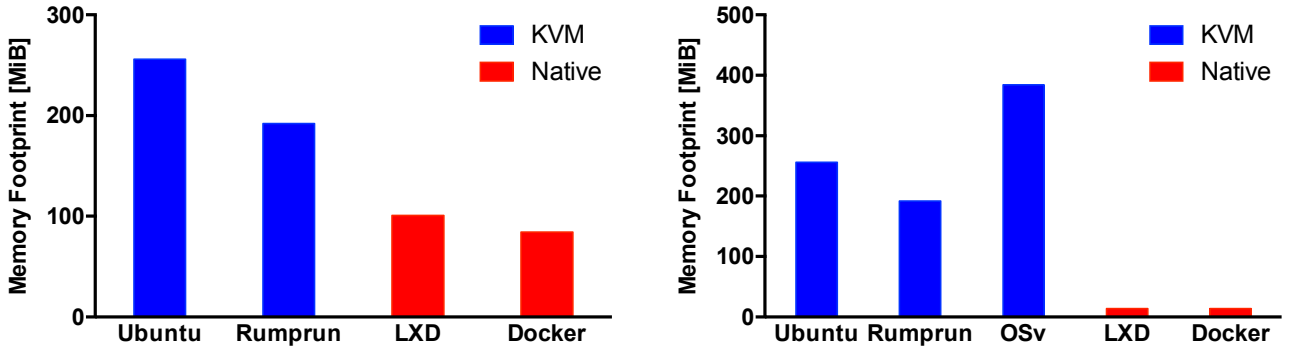
With both containers and unikernels undergoing rapid improvements, the current landscape of lightweight approaches to virtualization presents a confusing scenery. Comparative publications thus far have mostly highlighted the strengths of

 ISBN: 978-1-61208-529-6 



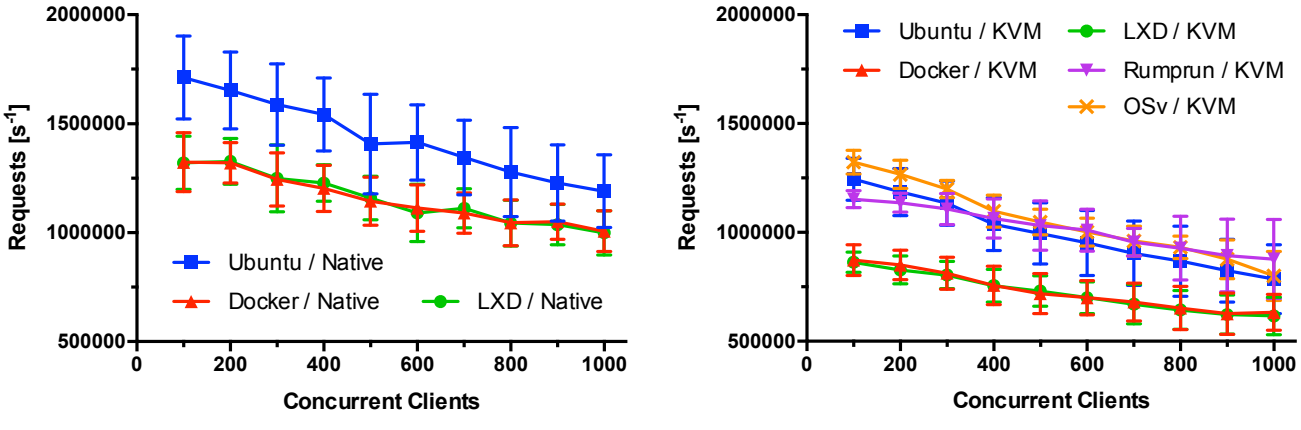

(a) (b)

Figure 3. Throughput performance of *Nginx* (version 1.8.0) was evaluated on native hardware (a) and in virtualized environments (b). Throughput was measured using the weighttp benchmark and the modified abc wrapper utility.




(a) (b)

Figure 4. The memory footprints of the static HTTP server scenario (a) and the Key-Value Store scenario (b) were measured for each each virtualization technique. Due to the variety among the tested approaches, different tools were used to obtain memory consumption readings.




(a) (b)

Figure 5. Throughput performance of *Redis* (version 3.0.1) was evaluated on native hardware (a) and in virtualized environments (b). The plotted values show the throughput for GET requests as retrieved through the *redis-benchmark* utility.

 ISBN: 978-1-61208-529-6 

one particular approach without comparing it to a broad range of alternative technologies. To take remedial action, we present an extensive performance evaluation of containers, unikernels, and whole-system virtualization.

Regarding application throughput, most unikernels performed at least equally well or even better than containers. We also demonstrated that containers are not spared from overhead regarding network performance, which is why virtual machines or unikernels may be preferable in cases where raw throughput matters. These are just some aspects demonstrating that while containers have already reached a sound level of maturity, unikernels are on the verge of becoming a viable alternative. Even though we did not see unikernels outperforming a virtualized Linux instance, our brief comparison between *NetBSD* and *Rumprun* also suggested that unikernels have the potential of outperforming their full-grown operating system relatives.

## ACKNOWLEDGMENT

We would like to thank Vincent Schwarzer for engaging our interest in *unikernels* in the course of his master's thesis [24]. Furthermore, we thank the HPI Future SOC Lab for granting us access to the hardware resources used in this paper.

This paper has received funding from the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866.

## DISCLAIMER

This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

## REFERENCES

[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Mar. 2015, pp. 171–172.

[2] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in Proceedings of the Linux Symposium. Citeseer, 2009, pp. 19–28.

[3] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, Cross-VM attack on AES," in International Workshop on Recent Advances in Intrusion Detection. Springer, 2014, pp. 299–319.

[4] T. K. Authors, "Kubernetes," visited on 2017-02-13. [Online]. Available: http://kubernetes.io/

[5] CloudFoundry, "Warden," visited on 2017-02-13. [Online]. Available: https://github.com/cloudfoundry/warden

[6] Docker Inc., "Docker," visited on 2017-02-13. [Online]. Available: https://www.docker.com/

[7] Canonical Ltd., "LXD," visited on 2017-02-13. [Online]. Available: https://linuxcontainers.org/lxd/introduction/

[8] ——, "LXC," visited on 2017-02-13. [Online]. Available: https://linuxcontainers.org/lxc/introduction/

[9] The OpenStack project, "Nova," visited on 2017-02-13. [Online]. Available: https://github.com/openstack/nova

[10] Google, "lmctfy," visited on 2017-02-13. [Online]. Available: https://github.com/google/lmctfy

[11] R. Jnagal, "Commit: update project status," visited on 2017-02-13. [Online]. Available: https://github.com/google/lmctfy/commit/0b317d7

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An Operating System Architecture for Application-level Resource Management," in Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 251–266.

[13] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 461–472.

[14] A. Madhavapeddy and D. J. Scott, "Unikernels: The Rise of the Virtual Library Operating System," Communications of the ACM, vol. 57, no. 1, 2014, pp. 61–69.

[15] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam et al., "Jitsu: Just-in-time summoning of unikernels," in 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015, pp. 559–573.

[16] A. Kantee, "Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels," Ph.D. dissertation, Aalto University, Finland, 2012.

[17] ——, "The Rise and Fall of the Operating System," ;login:, the USENIX magazine, 2015, pp. 6–9.

[18] A. Kivity, D. Laor, G. Costa, P. Enberg, N. HarEl, D. Marti, and V. Zolotarov, "OSv—Optimizing the Operating System for Virtual Machines," in 2014 USENIX Annual Technical Conference (USENIX ATC '14), 2014, pp. 61–72.

[19] KVM project, "KVM," visited on 2017-02-13. [Online]. Available: http://www.linux-kvm.org/page/Main

[20] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in Cloud Engineering (IC2E), 2014 IEEE International Conference on, Mar. 2014, pp. 610–614.

[21] I. Briggs, M. Day, Y. Guo, P. Marheine, and E. Eide, "A performance evaluation of unikernels," 2015.

[22] J. Nickoloff, "Evaluating Container Platforms at Scale," Mar. 2016. [Online]. Available: https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c

[23] B. Veal and A. Foong, "Performance Scalability of a Multi-Core Web Server," in Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems. ACM, 2007, pp. 57–66.

[24] V. Schwarzer, "Evaluierung von Unikernel-Betriebssystemen für Cloud-Computing," Masters Thesis (in German), Hasso Plattner Institute for Software Systems Engineering, University of Potsdam, Jun. 2016.

[25] lighty labs, "weighttp," visited on 2017-02-13. [Online]. Available: https://redmine.lighttpd.net/projects/weighttp/

[26] G-WAN ApacheBench, "abc," visited on 2017-02-13. [Online]. Available: http://gwan.com/source/ab.c

[27] Hasso Plattner Institute, "HPI Favicon," visited on 2017-02-13. [Online]. Available: http://hpi.de/favicon.ico

[28] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically Rigorous Java Performance Evaluation," vol. 42, no. 10. New York, New York, USA: ACM Press, oct 2007, p. 57.

 ISBN: 978-1-61208-529-6 