A Performance Survey of Lightweight Virtualization Techniques

Max Plauth, Lena Feinbube, Andreas Polze

A Performance Survey of Lightweight Virtualization Techniques

Max Plauth, Lena Feinbube, and Andreas Polze

Operating Systems and Middleware Group
Hasso Plattner Institute for Software Systems Engineering
University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

Abstract. The increasing prevalence of the microservice paradigm creates a new demand for low-overhead virtualization techniques. Complementing containerization, unikernels are emerging as alternative approaches. With both techniques undergoing rapid improvements, the current landscape of lightweight virtualization approaches presents a confusing scenery, complicating the task of choosing a suited technology for an intended purpose. This work provides a comprehensive performance comparison covering containers, unikernels, whole-system virtualization, native hardware, and combinations thereof. Representing common workloads in microservice-based applications, we assess application performance using HTTP servers and a key-value store. With the microservice deployment paradigm in mind, we evaluate further characteristics such as startup time, image size, network latency, and memory footprint.

1 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 [9,21] and Fig. 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 [2,6,27,31] – not analyzing startup latency and other system metrics in depth – or focus on highlighting the strengths of one particular approach without comparing it to a broad range of alternative unikernels and container technologies [3, 6, 9, 16, 19, 27].

We close this gap by presenting 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

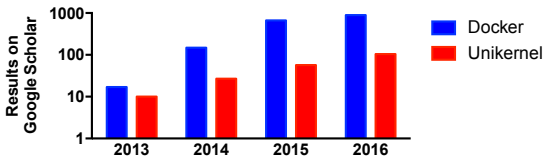


Fig. 1. The relevance of *Docker* and *unikernels* in the research community is indicated by the number of results on Google Scholar (as of May 15, 2017).

containers (*Docker*, *LXD*), unikernels (*Rumprun*, *OSv* and *MirageOS*), whole-system virtualization, native hardware, and certain combinations thereof. While previous work has laid a strong focus on *high performance computing* (HPC) applications (see Section 3), 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. Additionally, we provide further metrics, such as startup time, image size, network latency, and memory footprint. To facilitate full repeatability of our results, all test setups used throughout this paper have been made available online[1].

The remainder of the paper is organized as follows: Section 2 provides background about the employed virtualization approaches. Section 3 reviews related work that deals with quantifying the performance impact of lightweight virtualization approaches. Afterwards, Section 4 refines the scope of this work. Section 5 then documents the benchmark procedure yielding the results presented in Section 6. Finally, Section 7 concludes this work with final remarks.

2 Background

"Traditional", whole-system virtualization introduces performance and memory overhead, 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. [9] Even though techniques such as *kernel samepage merging* (KSM) [1] have managed to reduce memory demands, they do not provide an ultimate remedy as they dilute the level of isolation among virtual machines [12].

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

[1] `https://github.com/plauth/lightweight-vm-performance`

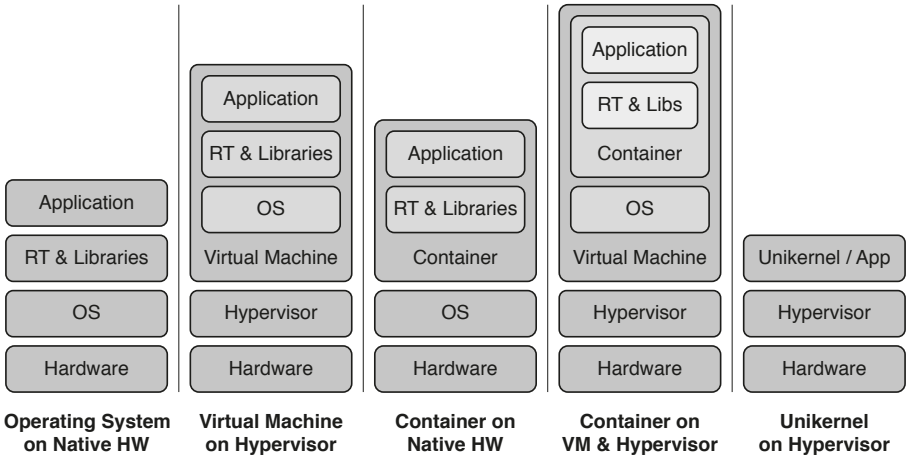


Fig. 2. Illustrated comparison of the software stack complexity of various deployment strategies, including native setups, virtual machines, containers, containers within virtual machines and unikernels.

2.1 Container (OS-Level Virtualization)

Containers are based on the observation that the entire kernel induces overly much resource overhead for merely isolating and packaging small applications. Here, we distinguish two classes of container virtualization approaches: 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. Currently, with *LXD*, the latter approach is becoming more prominent again, as it allows for the creation of *virtual machine* (VM)-like behavior without the overhead of a hypervisor. In the following paragraphs, we discuss the the containerization technologies under investigation

Docker Among the application-oriented containers, the open source project *Docker* [7] currently is 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).

LXD The Linux-based container solution *LXD* [5] builds up upon the *LXC* (Linux container) [4] 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 [29].

2.2 Unikernel (Hypervisor Virtualization)

Unikernels are a new take on the library operating system concept, providing merely a thin layer of protection and multiplexing facilities for hardware resources whereas hardware support is left to employed libraries and the application itself. Whereas library operating systems (e.g., Exokernel [8]) had to struggle with having to support real hardware, unikernels avoid this burden by targeting only virtual hardware interfaces provided by *hypervisors* or VMMs [20]. 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 [19, 21].

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 [14]. 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 [15], as long as it does not require calls to `fork()` or `exec()`.

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 [16]. 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.

MirageOS Being developed from scratch, the *MirageOS* unikernel resembles a puristic, clean-slated approach. *MirageOS* builds up on top of the *Mini-OS* kernel from the *Xen* project and only supports software written in the *OCaml* programming language [21]. Denying any compatibility with existing POSIX-compatible software, the static type system and the strong runtime safety capabilities of *OCaml* lead to a high level of software robustness [20].

3 Related Work

An overview of publications about performance measurements of lightweight virtualization techniques from the last few years are presented in Table 1. Previous research has measured selected performance properties of lightweight virtualization techniques, mostly in comparison with a traditional whole-system virtualization approach. However, we are not aware of any comprehensive analysis of up-to-date container versus unikernel technologies.

Felter et al. [9] have presented a comprehensive performance comparison between *Docker* containers and the *KVM* hypervisor [17]. 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. Mao et al. [22] have studied the startup time of virtual machines for the major cloud providers Amazon EC2, Windows Azure, and Rackspace. Among different influencing factors, the image size was shown to have a significant impact on the startup performance. Kivity et al. [16] 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 1. Related work on performance measurements of lightweight virtualization approaches. Studies printed in gray indicate a HPC context.

	Docker	**LXC**	**Open VZ**	**Rump-run**	**OSv**	**Mirage OS**	**KVM**	**Xen**	**others**
Metrics	**Container**			**Unikernel**			**Virtualization**		
app. performance	[9, 25] [6]	[2, 31]	[27, 31]		[3, 16]	[3, 19]	[3, 9, 16] [2, 27]	[3] [27, 31]	[25] [31]
startup time	[25]					[19]	[13]	[22]	[25]
image size								[22]	
network latency	[9, 25]	[2]	[27]		[3, 16]	[3, 19]	[3, 9, 16] [2, 27]	[3] [2, 27]	[25]

4 Scope of this Work

Here, we present an extensive performance evaluation of containers (*Docker*, *LXD*), unikernels (*Rumprun*, *OSv* and *MirageOS*), 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.

This paper extends our work published in [26], providing commensurable network stack parameters for all tested approaches and measurements for additional properties such as *startup time*, *image size*, and *network latency*. Furthermore, *Xen* and *MirageOS* have been included as additional *hypervisor* and *unikernel* approaches. Startup time is a relevant metric in scenarios, where the infrastructure is booted on demand to process certain requests. Requirements regarding the infrastructure and runtime environment are getting more ad hoc, may change spontaneously, and call for rapid just-in-time deployment and reactive approaches. Such scenarios are becoming more common with the microservice development pattern.

Our research questions are the following:

- How fast are containers, unikernels, and whole-system virtualization when running different workloads? Are the results from related work confirmed in our test cases?
- What is the most suitable virtualization technology for on-demand provisioning scenarios?
- What is the impact of the virtualization technology on general system properties such as *image size*, *network latency* and *memory footprint*?

5 Benchmark Procedure

This section provides a description of the benchmark methodologies applied within this work. All tests were performed on an HPE ProLiant m710p server cartridge [11] with the detailed specifications denoted in Table 2. Where applicable, all approaches were evaluated using *Xen*, *KVM* and native hardware to evaluate the performance impact of the employed virtualization approach. For container-based approaches, we also distinguish between native and virtualized hosts, where the latter represent the common practice for deploying containers on top of IaaS-based virtual machines. All configuration files, custom benchmarking utilities as well as modifications to existing utilities are provided online[1].

5.1 General Properties

Startup Time To avoid potential confounding variables, startup time is measured irrespectively from the application type. Referring to the test procedure suggested by Nickoloff [25], our test set-up is composed of a minimal application which sends a UDP packet containing a single character to a predefined host

Table 2. 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)
NIC	Mellanox Connect-X3 Pro (Dual 10GbE)
Operating system	Ubuntu Linux 16.04.1 LTS

and a counterpart application listening for said UDP packet. The listening application is executed on the virtualization host and issues the startup command for the corresponding *container* or *unikernel* VM and measures the time until the UDP packet is received.

Image Size In practice, image size strongly influences startup time [22], as images have to be transported over potentially slow networks. Hence, the eventual image sizes are reported for all examined technologies. To avoid skewed readouts caused by sparse image files, the actual disk utilization is retrieved using the `du` command line utility.

Network Latency Since network latency may be a decisive factor in latency-sensitive use cases such as *network function virtualization* (NFV) [23], the network round-trip time is measured between a dedicated host and the test object using the `ping` command line utility.

Memory Footprint Reducing the memory footprint is one of the main objectives of lightweight virtualization approaches. For native and *LXD*-based execution, memory consumption was measured using the `htop` command line utility. In the case of *Docker*, the `docker ps` command line facility was used to retrieve memory consumption measurements. As the memory footprint of VMs and *unikernels* is defined statically at the time of their instantiation, VM-sizing must be chosen carefully. Hence, we identified the least amount of memory that did not degrade performance by testing different values in steps of 8 MiB.

5.2 Application Performance

Representing common workloads of cloud-hosted applications, we picked HTTP servers and key-value stores as exemplary applications. As these I/O-intensive use cases involve a large number of both concurrent clients and requests, the network stack considerably contributes to the overall application performance. 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 [30], resulting in the configuration denoted in Table 3.

Table 3. 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 [28]. The resulting configuration for *Rumprun* is specified in Table 4, and the corresponding configuration for *OSv* is documented in Table 5. 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.

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

Table 4. Optimized settings for the *Rumprun* network stack.

Path	Parameter	Value
./sys/conf/param.c	MAXFILES	20000
./sys/netinet/in.h	IPPORT_ANONMIN	1024
./sys/netinet/in.h	IPPORT_ANONMAX	65535
./sys/netinet/tcp_timer.h	TCPTV_KEEP_INIT	30*PR_SLOWHZ
./sys/netinet/tcp_timer.h	TCPTV_KEEPINTV	30*PR_SLOWHZ
./sys/sys/socket.h	SOMAXCONN	1024

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 with the exception of *OSv* and *MirageOS*. As no port of *Nginx* exists for *MirageOS*, we had to trade in the aspect of full commensurability with *Nginx* and use the *conduit server* code example [24] in order not to exclude *MirageOS*

Table 5. Optimized settings for the *OSv* network stack.

Path	Parameter	Value
./include/osv/file.h	FDMAX	0x30D40
./libc/libc.cc	RLIMIT_NOFILE	20000
./bsd/sys/netinet/in.h	IPPORT_EPHEMERALFIRST	1024
./bsd/sys/netinet/in.h	IPPORT_EPHEMERALLAST	65535
./bsd/sys/netinet/in.h	IPPORT_HIFIRSTAUTO	1024
./bsd/sys/netinet/in.h	IPPORT_HILASTAUTO	65535
./bsd/sys/netinet/tcp_timer.h	TCPTV_KEEP_INIT	60*hz
./bsd/sys/netinet/tcp_timer.h	TCPTV_KEEPINTV	60*hz
./bsd/sys/sys/socket.h	SOMAXCONN	1024
./include/api/sys/socket.h	SOMAXCONN	1024

from the HTTP server discipline. Regarding *OSv* however, we refrain from running HTTP benchmarks due to the lacking availability of an adequate HTTP server implementation.

Our measurement procedure employs the benchmarking tool *weighttp* [18] and the *abc* wrapper utility [10] for automated benchmark runs and varying connection count parameters. The *abc* utility has been slightly modified to report standard 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. As static content, we use our institute website's *favicon*[2]. 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.

Key-Value Store In our second application benchmark discipline, we use *Redis* (version 3.0.1) as a key-value store. Except for *MirageOS*, *Redis* 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.

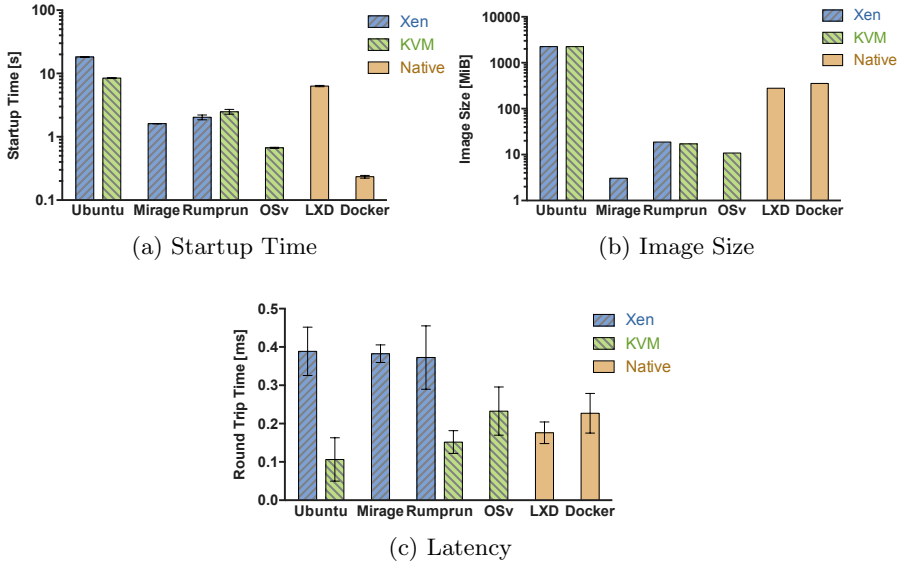
[2] `http://hpi.de/favicon.ico`

6 Results & Discussion

Here, we provide and discuss the results obtained from the benchmark procedure elaborated in Section 5. All values are expressed as mean ± SD (n = 30).

6.1 General Properties

Startup Time The measurements presented in Fig. 3a illustrate that both unikernels and containers can achieve much faster startup times compared to whole-system virtualization using *Ubuntu* Linux. The distinct differences between *LXD* and *Docker* demonstrate, that a large portion of the startup time of a Linux system is not caused by the kernel itself, but that it can be traced back to the services launched upon startup.



(a) Startup Time

(b) Image Size

(c) Latency

Fig. 3. A logarithmic scale is used to accommodate a wide range of values. (a) Startup time in seconds as measured using the procedure documented in [25]. (b) Image size in MiB as reported by the *du* utility. (c) Round-trip time in milliseconds as measured from a dedicated host.

Image Size The results presented in Fig. 3b indicate that container approaches undercut the image size of whole-system virtualization roughly by an order of magnitude, whereas unikernels reduce image sizes by one (*Rumprun* and *OSv*) or two (*MirageOS*) additional orders of magnitudes compared to containers. The substantial reduction of image sizes can lead to a considerable advantage in IaaS scenarios, where image size often correlates with instantiation time [22].

Network Latency The measurements presented in Fig. 3c indicate similar response times for *Rumprun*, *OSv*, and the container-based approaches. However, the choice of the hypervisor strongly affects the round-trip time performance. Even though para-virtualized network devices were used for both *Xen* and *KVM*, the latter yields much faster round-trip times for all tested guest systems.

6.2 Application Performance

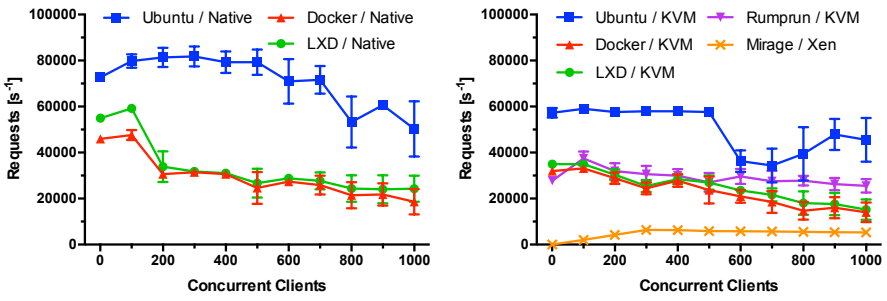
For a statistically meaningful evaluation, an ANOVA and a post-hoc comparison using the Tukey method were applied. For the hypervisor-based approaches using both *Xen* and *KVM*, the choice of the hypervisor had no statistically significant effect on application performance. Hence, only the results for *KVM* are plotted to avoid visual clutter.

Static HTTP Server The ANOVA test revealed a significant impact of the lightweight virtualization technique on the HTTP server performance ($p < 0.0001$, $F(9, 2970) = 3921$). Containers introduce a significant amount of overhead compared to native execution ($p < 0.0001$), both in native (see Fig. 4a) and virtualized environments (see Fig. 4b). A likely cause for this overhead is that all traffic has to go through a NAT in common configurations for both container-based approaches.

On the side of unikernels, *MirageOS* is running out of competition, as the employed *conduit server* can not be compared with a heavily optimized HTTP-server such as *Nginx*. For *Rumprun* however, it is surprising to see a similar performance compared to containers. Only for 600 concurrent clients and more, slight but statistically significant performance improvements can be observed for *Rumprun* compared to containers ($p < 0.0001$). With HTTP-servers heavily relying on the performance of the operating systems network stack, it can be assumed that the Linux networking stack has undergone massive optimization efforts that the *NetBSD* network stack can hardly compete with. To verify this hypothesis, we performed the same HTTP benchmark procedure using *NetBSD 7.0.1* in a virtual machine. Here, *Rumprun* performed distinctly better than *NetBSD* (data not shown), which indicates the potential of the unikernel-concept. With further optimizations of the network stack, *Rumprun* might achieve similar or even better performance than a regular Linux-based virtual machine.

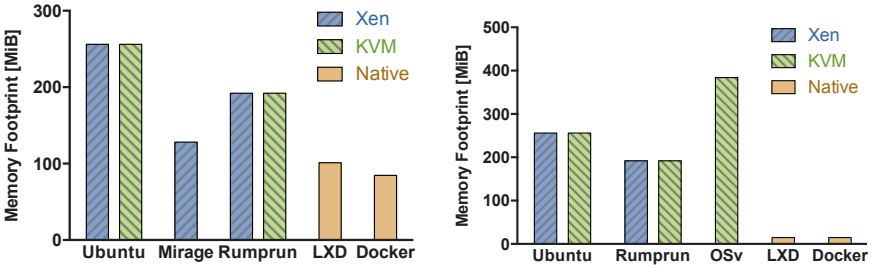
In terms of memory footprint, unikernels manage to undercut the demands of a full-blown Linux instance (see Fig. 5a). However, containers still can get by with the least amount of memory. The major advantage of containers remains the possibility of dynamic memory allocation, whereas virtual machines are restricted to predefining the amount of allocated memory at the time of instantiation. Still, *MirageOS* demonstrates that a puristic approach can yield distinctly reduced memory footprints, even though sophisticated porting efforts are necessary.

Key-Value Store The ANOVA test revealed a significant performance impact of the lightweight virtualization technique ($p < 0.0001$, $F(7, 7920) = 4099$). As illustrated in Fig. 6, 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. Regarding memory consumption (see 5b), 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.



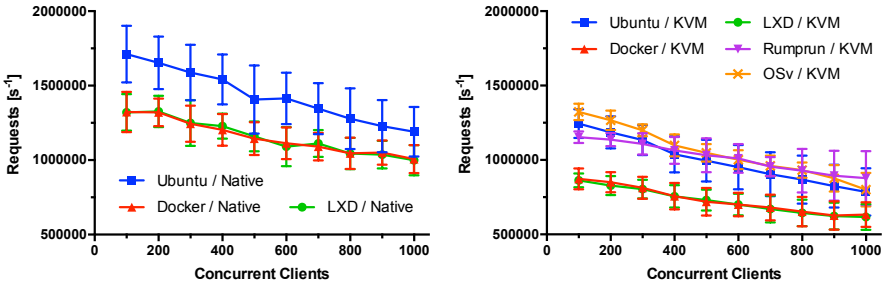
(a) Native (b) Virtualized

Fig. 4. Throughput of *Nginx* (1.8.0) was evaluated on native hardware (a) and in virtualized environments (b). For MirageOS, the *conduit server* was used. Throughput was measured using *weighttp* and the modified *abc wrapper* utility.



(a) Static HTTP Server (b) Key-Value Store

Fig. 5. The memory footprints of the static HTTP server scenario (a) and the Key-Value Store scenario (b) were measured for each virtualization technique.



(a) Native (b) Virtualized

Fig. 6. Throughput 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.

7 Conclusion

Performance evaluations of lightweight virtualization techniques thus far have mostly dealt with application performance and neglected relevant system properties such as startup latency, image size, network latency and memory footprint. Furthermore, many of these studies focused on highlighting the strengths of 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, focusing on metrics that are applicable to cloud applications.

Regarding application throughput, most unikernels performed at least equally well as 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. Even though *Docker* can achieve the shortest startup times considering the raw numbers, unikernels are competitive due to tiny image sizes and much shorter startup times than full virtual machines, especially in cases where the image has to be transferred to the compute host first. 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 [28]. 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. Arcangeli, A., Eidus, I., Wright, C.: Increasing memory density by using KSM. In: Proceedings of the Linux Symposium. pp. 19–28. Citeseer (2009)
2. Beserra, D., Moreno, E.D., Endo, P.T., Barreto, J., Sadok, D., Fernandes, S.: Performance Analysis of LXC for HPC Environments. In: 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS). pp. 358–363 (Jul 2015)
3. Briggs, I., Day, M., Guo, Y., Marheine, P., Eide, E.: A Performance Evaluation of Unikernels (2015)
4. Canonical Ltd.: LXC, `https://linuxcontainers.org/lxc/introduction/`, visited on 2017-07-15
5. Canonical Ltd.: LXD, `https://linuxcontainers.org/lxd/introduction/`, visited on 2017-07-15
6. Di Tommaso, P., Palumbo, E., Chatzou, M., Prieto, P., Heuer, M.L., Notredame, C.: The impact of Docker containers on the performance of genomic pipelines. PeerJ 3, e1273 (2015)
7. Docker Inc.: Docker, `https://www.docker.com/`, visited on 2017-07-15
8. Engler, D.R., Kaashoek, M.F., O'Toole, Jr., J.: Exokernel: An Operating System Architecture for Application-level Resource Management. In: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. pp. 251–266. SOSP '95, ACM, New York, NY, USA (1995)
9. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE Intl. Symposium on Performance Analysis of Systems and Software. pp. 171–172 (Mar 2015)
10. G-WAN ApacheBench: abc, `http://gwan.com/source/ab.c`, visited on 2017-07-15
11. Hewlett Packard Enterprise: HPE ProLiant m710p Server Cartridge QuickSpecs. `https://goo.gl/0dV579` (2015)
12. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! A fast, Cross-VM attack on AES. In: International Workshop on Recent Advances in Intrusion Detection. pp. 299–319. Springer (2014)
13. Jones, M., Arcand, B., Bergeron, B., Bestor, D., Byun, C., Milechin, L., Gadepally, V., Hubbell, M., Kepner, J., Michaleas, P., et al.: Scalability of VM Provisioning Systems. arXiv preprint arXiv:1606.05794 (2016)

14. Kantee, A.: Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels. Ph.D. thesis, Aalto University, Finland (2012)
15. Kantee, A.: The Rise and Fall of the Operating System. ;login:, the USENIX magazine pp. 6–9 (2015)
16. Kivity, A., Laor, D., Costa, G., Enberg, P., HarEl, N., Marti, D., Zolotarov, V.: OSv—Optimizing the Operating System for Virtual Machines. In: 2014 USENIX Annual Technical Conference. pp. 61–72 (2014)
17. KVM project: KVM, `http://www.linux-kvm.org/page/Main`, visited on 2017-07-15
18. lighty labs: weighttp, `https://redmine.lighttpd.net/projects/weighttp/`, visited on 2017-07-15
19. Madhavapeddy, A., Leonard, T., Skjegstad, M., Gazagnaire, T., Sheets, D., Scott, D., Mortier, R., Chaudhry, A., Singh, B., Ludlam, J., et al.: Jitsu: Just-In-Time Summoning of Unikernels. In: 12th USENIX Symposium on Networked Systems Design and Implementation. pp. 559–573 (2015)
20. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J.: Unikernels: Library operating systems for the cloud. In: Proceedings of the 18th Intl. Conference on Architectural Support for Programming Languages and Operating Systems. pp. 461–472. ACM, New York, NY, USA (2013)
21. Madhavapeddy, A., Scott, D.J.: Unikernels: The Rise of the Virtual Library Operating System. Communications of the ACM 57(1), 61–69 (2014)
22. Mao, M., Humphrey, M.: A Performance Study on the VM Startup Time in the Cloud. In: 2012 IEEE Fifth International Conference on Cloud Computing. pp. 423–430. IEEE (Jun 2012)
23. Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., Huici, F.: ClickOS and the Art of Network Function Virtualization. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. pp. 459–473. USENIX Association (2014)
24. MirageOS: Conduit Server Code Example. `https://github.com/mirage/mirage-skeleton/tree/master/conduit_server`
25. Nickoloff, J.: Evaluating Container Platforms at Scale. `https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c`
26. Plauth, M., Feinbube, L., Polze, A.: A Performance Evaluation of Lightweight Approaches to Virtualization. In: 8th Intl. Conference on Cloud Computing, GRIDs, and Virtualization. IARIA (Feb 2017)
27. Regola, N., Ducom, J.C.: Recommendations for Virtualization Technologies in High Performance Computing. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science. pp. 409–416. IEEE (2010)
28. Schwarzer, V.: Evaluierung von Unikernel-Betriebssystemen für Cloud-Computing. Masters thesis (in german), Hasso Plattner Institute for Software Systems Engineering, University of Potsdam (Jun 2016)
29. The OpenStack project: Nova, `https://github.com/openstack/nova`, visited on 2017-07-15
30. Veal, B., Foong, A.: Performance Scalability of a Multi-Core Web Server. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems. pp. 57–66. ACM (2007)
31. Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., Rose, C.A.F.D.: Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 233–240 (Feb 2013)