Microservice-Based Cloud Application Ported to Unikernels: Performance Comparison of Different Technologies

Janusz Jaworski, Waldemar Karwowski(✉), and Marian Rusek

Faculty of Applied Informatics and Mathematics, Warsaw University of Life Sciences, Nowoursynowska 166, 02–787 Warsaw, Poland
waldemar_karwowski@sggw.pl

Abstract. Microservice architecture is nowadays a popular design pattern for cloud applications. Usually microservices are launched on the servers of a cloud datacenter inside virtualization containers. This provides their isolation and performance comparable to virtual machines. However, security of virtualization containers remains a problem. They share the hosts machine operating kernel and thus their mutual isolations depends on proper implementation of kernel isolation features like cgroups or namespaces. Moreover the usage of kernel security modules like SELinux is hindered by different base Linux distributions used for containers creation. The concept of unikernels provides much better security for microservices. They are launched on a hypervisor instead of a full operating system kernel and contain only the base files needed to run the application. However their performance in realistic environments still remains a question. In this paper we port a microservice-based cloud application from Docker containers to Rumpkernel and OSv unikernels and analyse its performance. It is shown, that the performance of this unikernel-based port can match or exceed the more traditional approaches.

Keywords: Microservice architecture · Cloud computing · Unikernel

1 Introduction

Development of new internet technologies is connected with cloud services model. For information systems that are in the cloud and share public services publicly, security and safety are very important. In case of safety, the proper isolation of users using the application, guaranteeing the availability of the data only to the appropriate persons is crucial. At the same time, the protection of environment in which the application runs is very important. Moreover, a hundred or even million users use the software in parallel, which means that scaling is also necessary. To meet these requirements, new software architectures were created. The most prominent example is the microservice architecture. Nowadays, Docker virtualization containers are the most widely used technology in the cloud. Containers share a common operating system kernel with their host system and can use applications present on the host [1]. Traditional virtual machine image contains own copies of each application needed, which means that disk space used is bigger than for containers. Containers use the host’s processes, so the start of newly

L. Borzemski et al. (Eds.): ISAT 2019, AISC 1050, pp. 255–264, 2020.
https://doi.org/10.1007/978-3-030-30440-9_24

added container is almost immediate. Another solution is the unikernels technology designed to make applications be safer and more efficient than traditional, and highly scalable. Unikernels are lighter than containers, they are equally easy to scale, but they are better isolated from the environment and therefore safer. It is therefore important to investigate the practical performance of unikernel-based applications.

The history of the unikernels technology is connected with operating systems architecture research called libOS, which dates back to nineties of the last century. The concept of this architecture is based on the many small libraries that implement certain operating system mechanisms like file system, access control, device drivers etc. However, direct access to hardware and resources has allowed one application to use resources of other applications. To solve this problem, exokernel was designed and implemented [2], whose only task is to make access to specific resources only to eligible applications. Exokernel as opposed to a microkernel or a monolithic kernel does not use high-level abstraction but low-level interfaces. However, this solution has a major disadvantage, because the development of hardware libraries is not up to date and they must be adapted. Unikernels overcome this inconvenience using virtualization machine monitor (a.k.a. hypervisor) as the hardware abstraction layer. A unikernel is small, fast, secure virtual machines that lack operating systems [3] and can be booted up as a standalone virtual machine. Compared to virtual machines, unikernels eliminate unnecessary code, which results in memory savings and shorter boot time. Thanks to hypervisor there are no IPC (Inter Process Communication) and application components (microservices) are isolated. They communicate via Representational State Transfer (REST) interface only.

There are currently several implementations of unikernels. MirageOS uses the OCaml language, with libraries that provide networking, storage and concurrency support [2, 4], application code is compiled into a fully-standalone, specialised unikernel that runs under a Xen or KVM hypervisor. HalVM [5] is close to the MirageOS philosophy, but it is based on the Haskell language. IncludeOS [6] compiles C and C++ applications natively into a standalone application that boots without an operating system. RuntimeJS is an libOS for the cloud that runs JavaScript, it can be bundled up with an application and deployed as a lightweight and immutable VM image for KVM hypervisor. OSv and Rumprun provide compatibility for existing POSIX applications. OSv runs unmodified Linux applications under a Xen, KVM, and VMware. It supports many managed language runtimes including unmodified JVM, Python 2 and 3, Node.JS, Ruby, Erlang as well as languages compiling directly to native machine code. The Rumprun enables running POSIX applications as unikernels for hypervisors such as Xen and KVM. The research on HermiTux unikernel providing binary-compatibility with Linux applications is presented in [7].

There are not much performance tests of unikernel technologies. In [6] CPU-time required to execute 1000 DNS-requests, on IncludeOS and Ubuntu, running on Intel- and AMD systems was measured. On AMD, IncludeOS used 20% fewer CPU-ticks on average total than Ubuntu, on Intel it was only 5.5%. Performance evaluation for MirageOS, OSv and Linux for DNS server was investigated in [8]. Final benchmarks showed that OSv significantly exceeded the performance of Linux, but MirageOS failed. In [9] was performed an evaluation of KVM (Virtual Machines), Docker (Containers), and OSv (Unikernel), when provisioning multiple instances concurrently in an OpenStack cloud platform. Authors concluded that OSv outperforms the other options. The performance of unikernels versus Docker containers in the context of REST services and heavy processing workloads, written in Java, Go, and Python was investigated in [10]. The Go service was about 38% faster when running as a unikernel than as a container, while the Java version was about 16% faster as a unikernel. Python service was much slower according to Go and Java but with 15% improvement as a unikernel than as a container. In [11] performance evaluation of containers (Docker, LXD) and unikernels (Rumprun and OSv) is presented. Nginx HTTP server (version 1.8.0) was used to evaluate the HTTP performance for static content and whole-system virtualization. HTTP performance ranging from 0 to 1000 concurrent connections was measured. As a second benchmark application Redis (version 3.0.1) a key-value store, was used, performance of GET and SET operations ranging from 0 to 1000 concurrent connections was measured. Regarding application throughput, most unikernels performed at least equally well or even better than containers. Redis GET and SET benchmarks performed in [7] for Rumprun, OSv, Linux KVM, Docker and HermiTux confirmed above conclusion.

The tests mentioned above were made for isolated functionalities not for real applications. In this paper we tested performance of sample microservice application ported to OSv and Rumprun unikernels respectively. The rest of this paper is organized as follows: in Sect. 2 the architecture of different versions of a sample microservices-based application is introduced. In Sect. 3 performance experimental results for different technologies are presented. We finish with summary and brief conclusions in Sect. 4.

2 Sample Application

To make performance comparisons between different unikernels technologies a sample Docker example-voting-app[1] microservices-based application was chosen. It is an application that allows conducting a vote for a more popular household pet, the choice is between Dog and Cat. User can vote only for one candidate. It consists of two data containers and three microservices:

Vote—service written in Python using the Flask library2. It provides a Web page with two available options. User can give voice on one of them i.e. set environment variable. The getenv function from the standard Python os module returns the value of the chosen option. In addition, the application generates voters ID placed in the cookie to prevent user to give two valid votes. The HTML code is generated using the JINJA2 engine from the template file that is sent in response to the voter's browser. With this solution, it is possible to dynamically change the contents of the website code, it is not necessary to change the static HTML code. The voting ID and selection are saved in the JSON format and stored in the Redis database.

Worker—this service uses the Command Query Responsibility Separation (CQRS) design pattern. CQRS separates the logic of recording and reading data from each

[1] https://github.com/dockersamples/example-voting-app.

other. In example-voting-app, each unit has its own database respectively to read and write data. Worker is written in the C# language, it reads the data stored by the voting-app service in the Redis database and saved processed data in a relational database PostgreSQL in format readable for result-app. Moreover, worker verifies if the user with the specified identification number has already given a vote. If this happens, it does not add another vote but exchange the previous one. Very important advantage of worker is that if there is no connection to any database it does not stop its action, but periodically checks whether the connection can be created again. Therefore, if the connection to the PostgreSQL database is lost, the votes stored in the Redis database are not lost. After reconnection to the base, the votes may be recorded.

Result—the last element of the example-voting-app application is a service that displays the results of voting in a website. The service is written in the JavaScript language, script is running in the Node.js environment. When started, it connects with PostgreSQL database in which the worker service has recorded results. Service periodically retrieves results from PostgreSQL using polling. Results are displayed in the user’s browser both in text and graphic form. Thanks to polling and the Node.js environment, the service updates the customer’s website automatically. For the result-app service, the time window is 1000 ms, exactly every one second it executes the query to the database of the latest voting results. Naturally, if the connection will be lost, service displays the results that were available to it before the crash and tries to connect the next call.

Because the application consists of five services implemented in different programming languages, Rumprun and OSv were selected as a possible tools for preparing the unikernel version. Other tools would require code to be rewritten to other programming languages, for example OCaml (MirageOS unikernel).

3 Experimental Results

3.1 Setup

The performance tests described below were conducted on a commodity PC with an ASRock Z77 Extreme4 mainboard; Intel Core i5-3470 (3.2 GHz) processor; 16 GB DDR3 (666 MHz) memory; NetLink BCM57781 Gigabit Ethernet PCIe card; OCZ-VECTOR150 hard drive running Ubuntu 16.04.03 LTS operating system with Linux kernel 4.4.0-92-generic. KVM hypervisor was also installed and all microservices packaged as unikernels were launched on it. Also NetBSD was used inside a KVM virtual machine. It’s kernel settings from the sys/conf/param.c file were modified to bring it on pair with Linux configuration. For example the MAXFILES parameter was set to the value of 200000 which is the same as used by the authors of [11] in their Linux kernel configuration (fs.file-max setting).

3.2 Redis

Redis is in memory key-value store, popular in cloud applications. It is also a part of the example-voting-app introduced in Sect. 2. In our Rumprun unikernel tests Redis version 3.0.6 from rumprun-packages repository was used[2]. For OSv unikernel tests Capstana porting tool was used[3], but we slightly modified the source code of OSv Redis package, so the same Redis server and redis.conf versions as in the Ruprun tests were used. Next both unikernels were launched on the KVM hypervisor.

The same Redis source code was used to build service on Ubuntu and NetBSD operating systems. Note that it is different from a regular Redis which utilizes the fork system call for background savings of the database into a /data/dump.rdb file. Unikernels run only a single process and do not allow for fork system call, so this periodic save feature is missing from the Rumpkernel port used in our experiments.

Redis package includes redis-benchmark utility which was used in our tests. It was started for 10 parallel connections and 10 pipeline requests. The results from this and similar measurements for OSv unikernel, Ubuntu running on the bare metal, and Ubuntu and NetBSD running inside virtual machines are expressed in requests per second and presented in Table 1.

Table 1. Results of redis-benchmark testing of the Redis microservice launched on bare metal, Rumprun and OSv unikernels and Ubuntu and NetBSD virtual machines. All numbers are expressed in requests per second.

	Ubuntu	Rumprun	OSv	Ubuntu (KVM)	NetBSD (KVM)
PING_INLINE	632911.38	389105.06	518134.72	483091.78	89126.56
PING_BULK	1098901.12	578034.69	806451.62	714285.69	97943.19
SET	8223.01	3750.52	6579.81	3648.84	2729.33
GET	746268.62	406504.06	649350.62	495049.50	88652.48
INCR	8140.01	3914.81	7007.71	3659.38	2856.82
LPUSH	8293.94	4036.98	6844.63	3638.22	2908.41
LPOP	8152.62	4318.35	7150.01	3702.74	2911.63
SADD	657894.75	387596.91	458715.59	512820.53	81967.21
SPOP	980392.19	483091.78	675675.69	613496.94	87873.46
LRANGE_100	94161.95	73421.44	53418.80	91157.70	37893.14
LRANGE_300	29291.15	21199.92	15401.20	25940.34	14457.13
LRANGE_500	16806.72	14496.96	9737.10	14176.35	10117.36
LRANGE_600	12253.40	10563.01	7275.37	9441.09	7376.26
MSET	7833.31	2917.41	5296.33	3503.73	1903.96

To ease comparison rows were normalized to the largest value in a row and plotted as colour bars in Fig. 1.

[2] https://github.com/rumpkernel/rumprun-packages/tree/master/redis.

[3] https://github.com/cloudius-systems/osv-apps/tree/master/redis-memonly.

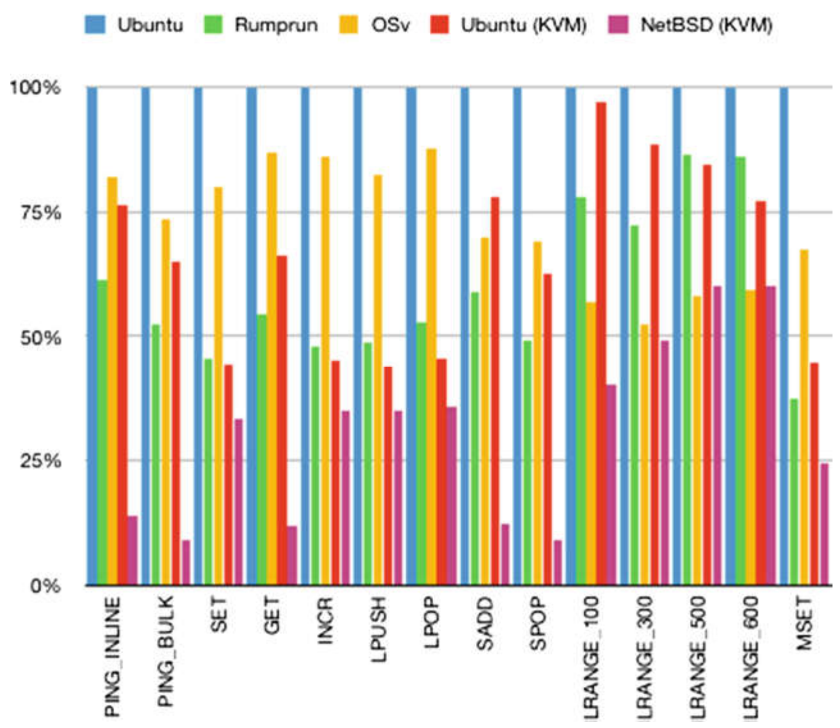

Fig. 1. Bar plot of the results of redis-benchmark normalized to the highest value (100%).

It is seen from inspection of this plot that Ubuntu running on bare metal is always the fastest one (blue bar). However OSv unikernel (orange bar) catches up and in the first 7 tests (up to LPOP) it offers at least 80% of the bare metal performance. Surprisingly it is also faster than Ubuntu running in a KVM virtual machine. The situation changes for the LRANGE tests where Ubuntu on KVM takes the second place but also Rumprun (green bar) starts to offer acceptable performance. In the SADD and SPOP tests OSv performs almost as good as Ubuntu on KVM and outperforms it again in the last MSET test. Note that the Rumprun unikernel is always faster than NetBSD (violet bar) whose drivers it utilizes. The large difference between Linux and NetBSD may come from the fact, that the Linux network stack is heavily optimized and Linux runs better in a KVM environment (NetBSD weren't tested on bare metal). In [11] only GET requests were studied for 100 concurrent users and similar results to the corresponding row from Table 1 were obtained—the performance of Rumprun, OSv and Ubuntu/KVM was at about 70% of Ubuntu performance (these authors did not study NetBSD).

3.3 NodeJS

In that paper [11] performance of a Nginx web server with a static web page running on Rumprun and Ubuntu VM was also studied. However, according to Netcraft May 2019 Web Server Survey[4] Nginx accounts only for 21% of active sites. The competing monolithic web server Apache has 30%, but the most rapidly growing category of web

[4] https://news.netcraft.com/archives/2019/05/10/may-2019-web-server-survey.html.

servers is Other at 35%. These are custom web servers probably run as microservices in the cloud. In this section we analyse the performance of the result microservice from the example-voting-app described in Sect. 2. This microservice is written in NodeJS and coupled to PostgreSQL database running as Docker container on the host. We ported it to Rumprun[5] and OSv[6] unikernels using the procedure outlined in Subsect. 3.2. To test their performance the Apache HTTP server benchmarking tool[7] was used. The results are presented in Table 2 and Fig. 2.

Table 2. Results of Apache HTTP benchmark (ab) testing of the result micoservice launched on bare metal, Rumprun and OSv unikernels as well as Ubuntu and NetBSD virtual machines. Latencies are expressed in miliseconds.

	Ubuntu	Ubuntu (KVM)	Rumprun	OSv	NetBSD (KVM)
REQUESTS/SEC	2238.73	2567.29	4083.43	2494.76	2056.50
KBYTE/SEC	4234.79	4856.28	7724.22	4719.08	3890.07
LATENCY 50%	44.0	38.0	23.0	39.0	46.0
LATENCY 90%	48.0	45.0	28.0	47.0	64.0
LATENCY 100%	152.0	78.0	74.0	83.0	127.0

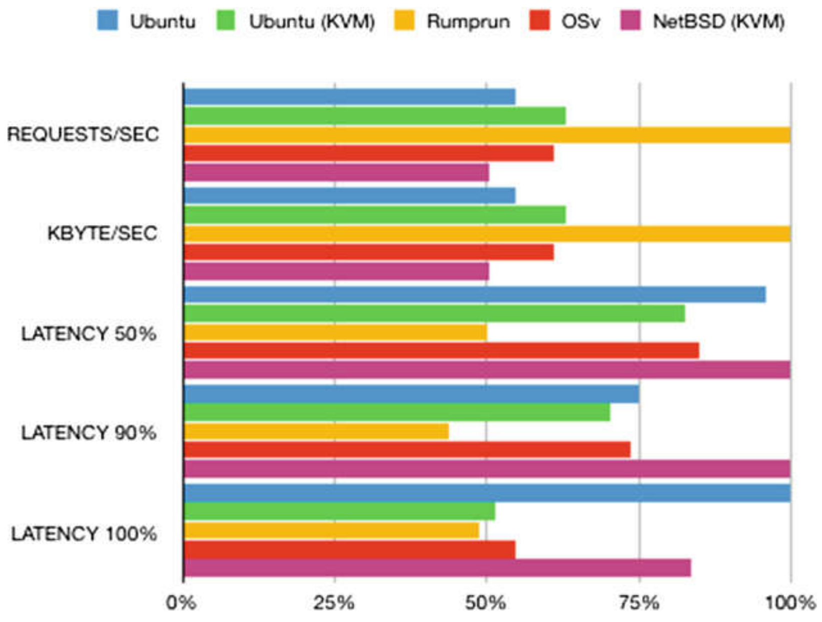

Fig. 2. Bar plot of the ab benchmark of the result micoservice normalized to the highest value.

Surprisingly Rumprun is the clear winner now offering the highest throughput and lowest response times. OSv offers only mediocre performance at the level of 60% but

[5] https://github.com/rumpkernel/rumprun-packages/tree/master/nodejs.

[6] https://github.com/cloudius-systems/osv-apps/tree/master/node.

[7] https://httpd.apache.org/docs/2.4/programs/ab.html.

does not fall behind other solutions (this may be due to the fact that the Rumprun repository is at version 4.3 whereas in OSv version 4.1 is used). This is stark contrast with results of the static web server testing from [11] where Rumprun performed only at 38% of the bare metal server, and Ubuntu on KVM at 75% of bare metal Ubuntu (at 100 concurrent users). We double checked our results using weighttp but the conclusions are similar: Rumprun achieved 4092 requests per second and OSv—2482. The ratio is again 60%.

3.4 Python

The last microservice studied was vote coupled to the Redis database. It is written in Python and utilizes the Flask library. We managed to successfully build Rumprun image[8] but OSv creation using Capstana[9] failed. Probably because the python3x app available in the repository is tweaked to filter out static libraries to lower image size. The results of the performance tests with ab are presented in Table 3 and Fig. 3.

Table 3. Results of Apache HTTP benchmark (ab) testing of the vote microservice launched on bare metal, Rumprun unikernel as well as Ubuntu and NetBSD virtual machines. Latencies are expressed in miliseconds.

	Ubuntu	Ubuntu (KVM)	Rumprun	NetBSD (KVM)
REQUESTS/SEC	1040.19	564.52	106.75	139.121
KBYTE/SEC	1505.36	816.42	155.22	1041.65
LATENCY 50%	93.0	171.0	934.0	134.0
LATENCY 90%	103.0	245.0	951.0	152.0
LATENCY 100%	148.0	341.0	1035.0	242.0

This time the Rumprun version performs very poorly, even worse than the NetBSD VM. Poor performance of Python unikernels was noticed by the authors of [10]. They studied a REST microservice written in different programing languages (Go, Java and Python) using an OSv unikernel. For Python Flask RESTfull[10] was used but their application was not coupled to Redis—this may be another reason why our image failed to build properly.

3.5 Scalability

Scalability of microservices is very important for cloud applications. Their launch time should be as low as possible. In Table 4 we have the times measured from image startup to the first result obtained from a microservice it contains. We see that unikernel-based images start up to 3 times faster than corresponding Ubuntu virtual

[8] https://github.com/rumpkernel/rumprun-packages/tree/master/python3.

[9] https://github.com/cloudius-systems/osv-apps/tree/master/python3x.

[10] https://flask-restful.readthedocs.io/en/latest/.

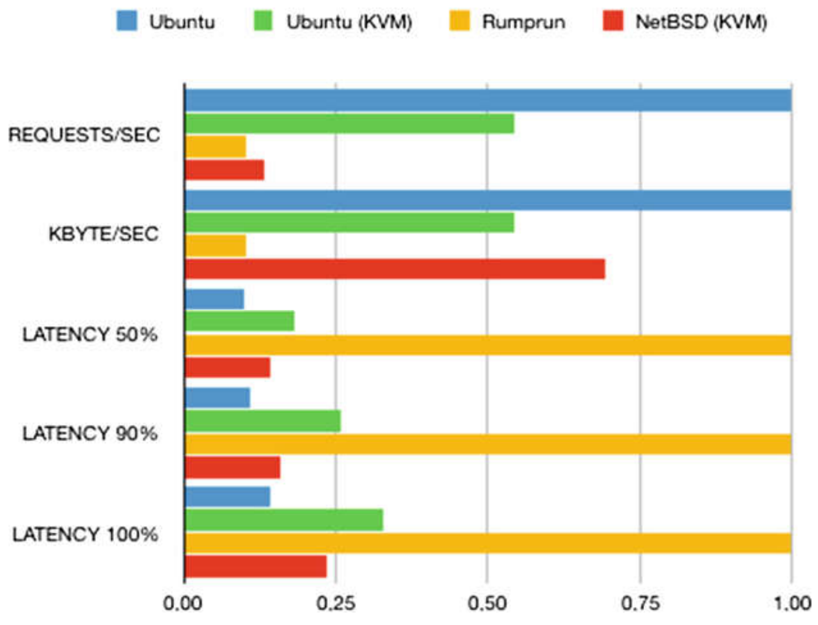

Fig. 3. Bar plot of the results of Apache HTTP benchmark for vote microservice normalized to the highest value (100%).

machine. The NetBSD VM is again an exceptionally poor performer on KVM Linux system. It is interesting to note, that OSv version of result starts faster than a Rumprun version. This is contrary to the common believe that OSv is a "fat" unikernel. For example the authors [11] reported that images generated by this tool are twice as large as Rumprun images.

Table 4. Startup times of different microservice versions measured in seconds.

	Vote	Result
Ubuntu (KVM)	9.12	9.17
Rumprun	2.29	2.39
OSv	–	1.65
NetBSD (KVM)	29.71	31.36

4 Summary

Performance of a relatively simple cloud application ported to unikernel architecture is studied. The application was originally written for Docker virtualization containers and consists of five containers: vote, results, worker, Redis and database. Worker is a trivial one, it only copies votes from Redis to database. The database used is PostgreSQL, there was no reason to port it to unikernels because it is not scalable and runs on a dedicated server. Therefore only vote, results and Redis were ported and studied.

Two unikernel technologies were chosen for our study. Rumprun is an implementation of a rump kernel and it can be used to transform just about any POSIX-compliant program into a working unikernel. The rump kernel project provides the modular drivers from NetBSD in a form that can be used to construct lightweight, special-purpose virtual machines. The second choice is OSv which was originally created by Cloudius Systems as a means of converting almost any application into a functioning unikernel. It can accept just about any program that can run in a single process.

They both provide performance better than monolithic virtual machines and sometimes better than a bare-metal implementation. Thus, they make a good basis for modern cloud-based applications. The only problem we encountered was poor performance of Python, which uses a disproportionate amount of operations that makes it run slower in a Rumpkernel environment. We were not able to port the Python-based vote microservice onto the OSv unikernel. This is because Cloudius Systems has been transformed into a new company called ScyllaDB and they are no centred on servicing OSv. The OSv open source community is now maintaining the software, which is fully operational, but some packages like Python do not receive sufficient support.

References

1. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D.J., Singh, B., Gazagnaire, T., Smith, S. A., Hand, S., Crowcroft, J.A.: Unikernels: library operating systems for the cloud. ACM SIGPLAN Not. **48**(4), 461–472 (2013). ASPLOS'13
2. Engler, D.R., Kaashoek, M., O'Toole, J.: Exokernel: an operating system architecture for application-level resource management. ACM SIGOPS Oper. Syst. Rev. **29**(5), 251–266 (1995)
3. Pavlicek, R.C.: Unikernels: Beyond Containers to the Next Generation of Cloud. O'Reilly Media (2017)
4. Madhavapeddy, A., Scott, D.J.: Unikernels: rise of the virtual library operating system. Commun. ACM **51**(1), 61–69 (2014)
5. Galois Inc.: The Haskell Lightweight Virtual Machine (HaLVM) source archive. https://github.com/GaloisInc/HaLVM. Accessed 24 May 2019
6. Bratterud, A., Walla, A., Haugerud H., Engelstad, P.E., Begnum, K.M.: IncludeOS: a minimal, resource efficient unikernel for cloud services. In: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science, pp. 250–257 (2015)
7. Olivier, P., Chiba, D., Lankes, S., Min, C., Ravindran, B.: A binary-compatible unikernel. In: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 59–73 (2019)
8. Briggs, I., Day, M.J., Guo, Y., Marheine, P., Eide, E.: A Performance Evaluation of Unikernels (2014). http://media.taricorp.net/performance-evaluation-unikernels.pdf. Accessed 24 May 2019
9. Xavier, B.G., Ferreto, T., Jersak, L.C.: Time provisioning evaluation of KVM, docker and unikernels in a cloud platform. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 277–280 (2016)
10. Goethals, T., Sebrechts, M., Atrey, A., Volckaert, B., Turck, F.D.: Unikernels vs containers: an in-depth benchmarking study in the context of microservice applications. In: 2018 IEEE 8th International Symposium on Cloud and Service Computing, pp. 1–8 (2018).
11. Plauth, M., Feinbube, L., Polze, A.: A performance evaluation of lightweight approaches to virtualization. In: CLOUD COMPUTING 2017: The Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, pp. 34–48 (2017)