Unikernels: The Next Stage of Linux's Dominance

Ali Raza
Boston University
aliraza@bu.edu

Parul Sohal
Boston University
psohal@bu.edu

James Cadden
Boston University
jmcadden@bu.edu

Jonathan Appavoo
Boston University
jappavoo@bu.edu

Ulrich Drepper
Red Hat
drepper@redhat.com

Richard Jones
Red Hat
rjones@redhat.com

Orran Krieger
Boston University
okrieg@bu.edu

Renato Mancuso
Boston University
rmancuso@bu.edu

Larry Woodman
Red Hat
lwoodman@redhat.com

Abstract

Unikernels have demonstrated enormous advantages over Linux in many important domains, causing some to propose that the days of Linux's dominance may be coming to an end. On the contrary, we believe that unikernels' advantages represent the next natural evolution for Linux, as it can adopt the best ideas from the unikernel approach and, along with its battle-tested codebase and large open source community, continue to dominate. In this paper, we posit that an upstreamable unikernel target is achievable from the Linux kernel, and, through an early Linux unikernel prototype, demonstrate that some simple changes can bring dramatic performance advantages.

CCS Concepts • **Software and its engineering → Virtual machines**; **Operating systems**; • **Security and privacy** → *Virtualization and security*; • **Computer systems organization** → *Real-time operating systems*;

Keywords Operating Systems, Unikernels, Linux, Library Operating Systems

ACM Reference Format:
Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321445

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321445

1 Introduction

Linux is the dominant OS across nearly every imaginable type of computer system today. Linux is running in billions of people's pockets, in millions of our homes, in cars, in planes, in spaceships [15], embedded throughout our networks, and on each of the top 500 supercomputers [2]. To accomplish this, the Linux kernel has been continuously expanded to support a wide range of functionality; it is a hypervisor, a real-time OS, an SDN router, a container runtime, a BPF virtual machine, and a support layer for Emacs. When considering this ever-growing set of requirements, and the complexity creep which ensues [26], it leads to the question: Can a single kernel *really* handle this massive range of conditions and use cases efficiently?

There is, in fact, evidence that the structure of the Linux kernel is problematic for a number of today's key use cases. For one, applications that require high-performance I/O use frameworks like DPDK [4] and SPDK [5] to bypass the kernel and gain unimpeded access to hardware devices [14, 28]. The most performance sensitive of these applications are often dedicated entire machines for their deployments, for example, infrastructure components like Ceph [32].

In the cloud, client workloads are run inside a dedicated virtual machine for security. Increasingly, these workloads are written to single-process language runtimes and deployed in parallel across VMs. Thus, a kernel designed to multiplex the resources of many users and processes is instead being replicated across many single-user, often single-process, environments [31].

In response, there has been a resurgence of research systems exploring the idea of the libraryOS, or *a unikernel*, a model where target application is linked with a specialized kernel and deployed directly on hardware, virtual or physical [12]. Compared with Linux, unikernels have demonstrated significant advantages in boot time [22], security [33], resource utilization [24], , and I/O performance [29]. In fact, unikernels have shown such promise for new models of *serverless computing* that some foresee the dominance of Linux may soon come to an end [18].

Linux has often lagged behind research systems in many key aspects such as multiprocessor scalability [20], virtualization [7], and process containment [25, 30]. However, in each of these aspects, Linux not only caught up, it soon became the standard. We believe that unikernels represent its next natural evolution; Linux can adopt some of the best ideas from research unikernel systems and, along with its large community and battle-tested codebase, continue to be the standard across current and future domains.

In this paper, we explore the idea of turning Linux itself into a unikernel, i.e., add support within the codebase to build a target application into an optimized unikernel binary, while avoiding or bypassing kernel features deemed unnecessary for the application's workload. Although preliminary, our initial results suggest that a unikernel target offers an immediate performance advantage for applications. Furthermore, we posit that the necessary changes made to Linux to support a unikernel target are few and self-contained, and therefore are likely to be accepted by the upstream community.

We first discuss in more detail the advantages of the unikernel model in Section 2. We then discuss a few design goals and explore some approaches that we could take to adapt Linux to a unikernel in Section 3, and describe the approach for our initial prototype in Section 4. We then discuss in Section 5 optimization opportunities, and the research challenges in Section 6.

2 Unikernels

The unikernel is a cloud-era handle for the classic systems technique of linking an application with a library of operating system components (including memory management, scheduler, network stack and device drivers) into a single flat address space, creating standalone binary image that is bootable directly on (virtual) hardware [22]. The advantage of this approach is that kernel functionality can be specialized to fit the needs of the target application to increase the performance of the application or to support it within a highly restricted execution domain.

In recent years there has been an increase in the number of new unikernel systems, most of which target cloud and Internet workloads. Network performance is a common driving influence for unikernels as simplified IO paths and removal of domain crossings are common techniques for improving the latency and throughput of network-driven workloads. Memcached running on a unikernel TCP/IP stack, compared to that of Linux, demonstrates a throughput improvement of over 200% [29]. Similarly, the small memory footprints and short boot times of unikernels are beneficial to cloud providers who deploy client workloads in dedicated environments. Unikernels deployed within a microVM have shown 6×-10× improvement in boot times over containers [18]. Similarly, a *micropython* unikernel had an image sizes of 1MB and required only 8MB of memory to run [24].

Protection and isolation provide more motivation for unikernel research, because an application equipped with a library OS can be made to run in a highly-restricted execution domains, such as an SGX enclave [8], or behind a set of software-defined interfaces [13, 33].

Despite the security and performance benefits of unikernels, they have yet to be widely adopted outside of the domain of research systems and experimental platforms. We attribute this to the increased engineering burden for developers that comes with porting applications to a runtime with only partial support for legacy software interfaces.

The root of the problem lies in the way that unikernels have been developed. As of today, the creation of a new unikernel followed one of two approaches: a *clean slate* approach where the kernel is largely built from scratch, or a *strip down* approach where an existing kernel codebase is stripped of functionality deemed unnecessary for the unikernel. With a clean slate approach, unikernel designers have full control over the language and methodology used to construct the kernel. With such freedom, the resulting implementation can be extremely specialized and limited to particular class of application (for example, MirageOS only supports applications written in OCaml [21]). Implementations in clean-slate unikernels can also be finely-tuned for performance and provide efficient, low-level interfaces that applications can be directly written for. Unikernels such as OSv [17], IncludeOS [9], and EbbRT [29] attempt to balance high-performing components together with a C-standard runtime and partial support for common POSIX-like interfaces. The problem is that clean-slate unikernels cannot (and should not) hope to support the same myriad of interfaces and options provided by a general purpose kernel, at least not without abandoning or obfuscating the efficient pathways and finely-tuned implementation that make a clean slate approach attractive to begin with. With limited support for legacy software, porting and supporting existing applications on a clean slate unikernel becomes a non-trivial endeavour, and may be quickly deemed "not worth the effort."

Alternatively, strip-down unikernels attempt to make porting software easier by preserving the general-purpose libraries and interfaces of a legacy kernel codebase. Also known as *rump kernels*—a name inspired by the infamous purge of royalists from Parliament following the English Civil War—this process involves creating a fork of an existing kernel codebase and manually purging it of the components deemed unnecessary to the target unikernel. For example, the RumpRun unikernel contains a heavily-reduced version of NetBSD [16]. But like in any governing body, the removal of a key set of components, while continuing to support a wide range of interests, can become problematic. In this

case, creation of an out-of-tree fork abandons a fundamental asset of the original kernel codebase, its global community of contributors. The fixes and updates made to the evolving source kernel do not come free to a rump kernel. Instead, a rump kernel must be updated manually at regular intervals to continue to provide an up-to-date platform that supports existing software.

3 Key Goals and Possible Approaches

Based on our and others' research on unikernels and our experience with the Linux community, we believe a Linux-based unikernel is achievable and can exist as part of the of the kernel source tree. A Linux unikernel with retain many of the advantages of Linux, i.e., battle tested code base, large open-source community, huge support for legacy software, etc., while introducing some key properties that have given unikernels their advantage: single address space, small memory footprint, customizable kernel pathways. Overtime, the community can extend the Linux unikernel to incorporate many of the techniques and lessons learned from unikernel research systems, and move the performance and security of Linux closer to that of a lightweight research unikernel.

Towards this vision we outline the following goals for realizing a Linux-based unikernel:

1. Most applications and user libraries should be able to be integrated into a unikernel without modification; building the unikernel should just mean choosing a different GCC target.
2. Avoid any ring transition overheads; overhead experienced by any application requesting kernel functionality should be equivalent to a simple procedure call.
3. Allow cross-layer optimization; the compiler and/or developer should be able to co-optimize the application and kernel code.
4. The changes in Linux source code should be minimal so that they can be accepted upstream and the unikernel can be an integral part of Linux going forward. This will ensure unikernels are not an outsider but a build target anyone can choose to compile their applications for.

Projects that share the goal of intermixing application and kernel code include 1) User Mode Linux (UML) [10] which allows the kernel to run in userspace as a process; 2) Linux kernel library (LKL), [27] which packages the kernel as a library and creates a virtual machine in which the kernel executes; and 3) LibOS [1], which builds the kernel network stack as a shared library and runs in userspace. All these approaches try to reuse the kernel code one way or the other but do not address our first two goals.

Two approaches to avoid ring transitions are 1) integrating applications into the kernel as a Linux kernel module, and 2) allowing unmodified applications to run in ring zero along with the kernel [3, 23]. Both of these approaches preserve the full functionality of Linux, while allowing one or more applications to be optimized. We eventually rejected these approach because they do not really meet our third goal of cross-layer optimization. Also, for kernel modules, the application needs to be rewritten which violates the first goal.

We have chosen a pure unikernel approach where the kernel is statically linked to run a single application. Only with this approach can we enable configure time and link time optimizations that are not possible if arbitrary user-level applications can be run alongside the application we are optimizing for.

4 Unikernel Linux (UKL)

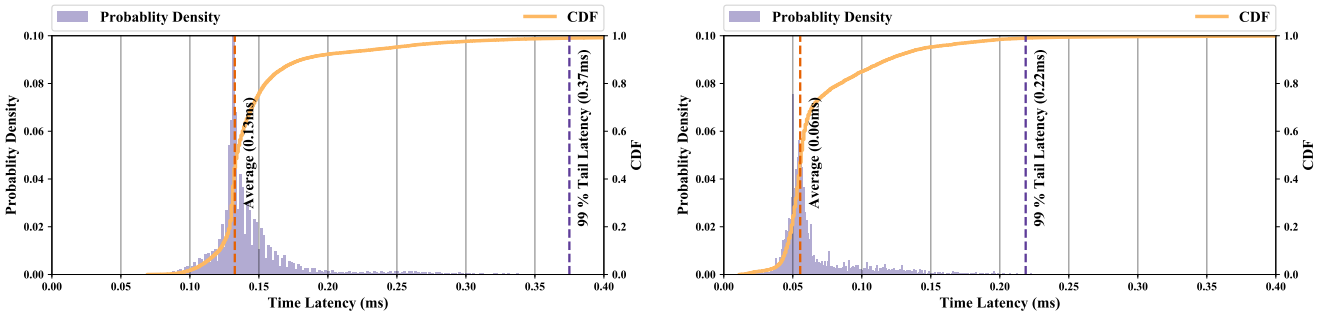
We have created a working prototype of Unikernel Linux (UKL). Below we describe the implementation steps involved, the build process, some early challenges we hit, and an initial set of performance results.

4.1 Implementation Overview

To create the UKL prototype, we:

- Added a new kernel configuration option to allow the user to select if he/she wants to compile the Linux kernel as UKL.
- Added a call to an undefined symbol (protected by an `#ifdef`) that can be used to invoke application code rather than creating the first userspace process.
- Created a small UKL library which has stubs for syscalls. These stubs hide the details of invoking the required kernel functionality now that the regular interface (i.e., the `syscall` instruction) is no longer used.
- Changed glibc so that instead of making syscalls into the kernel, it makes function calls into UKL library.
- Changed the kernel linker script to define new segments such as thread local storage (TLS) segments which are present in application ELF binaries.
- Added a small amount of initialization code before invoking the application to replace initialization normally done by user level code, e.g., for network interface initialization.
- Modified the kernel linking stage to include the application code, glibc and UKL library to create a single binary.

Our prototype uses the latest versions of Linux (v5.0.5) and glibc (v2.28). Any source file in glibc making syscalls is copied into a separate subdirectory and edited so that it makes procedure calls to the UKL library instead. Obviously, the number of lines changed here is same as the number of times syscalls are made. The changes in glibc, contained in a separate sub-directory, ensure that the normal build process does not break. Once the code becomes stable, we will not need to have a separate sub-directory; UKL function calls



(a) Userspace Echo server

(b) UKL Echo server

Figure 1. Probability density and CDF of latencies for the userspace and UKL echo server. The average latency of UKL echo server (0.06 ms) is less than half of that of the userspace case (0.13 ms), and the 99% tail latency for UKL (0.22 ms) is 41% faster than the userspace case (0.37 ms).

can live alongside normal glibc code. The total size of glibc archive is 45MB.

We changed 11 lines and added 20 new lines in total to the Linux kernel to turn it into a unikernel. These changes do not disrupt the normal Linux kernel: turning off the UKL config option creates a normal Linux binary and turning it on creates the UKL binary. These modest changes are more likely to be accepted upstream.

4.2 Build Process

The UKL build process is straightforward.

1. Compile glibc into an archive of object files without any linking.
2. Compile the application code into object files without any linking, i.e., with the `-c` option.
3. Compile the UKL library into an object file.
4. Build the Linux kernel with UKL config option turned on. As mentioned above, the linking stage in the kernel build process is slightly modified to link together all the object files created earlier.

The first three steps can be performed in any order. The first step (compiling glibc) takes just under two minutes on a modest 4 core Intel i7 laptop with 16GB RAM, but this only needs to be done once. As expected, the fourth step takes the longest if being done for the first time. This is simply because it normally takes time for the entire Linux kernel to build, depending on the machine being used. If the Linux kernel is already built, building a simple TCP echo server into a unikernel takes 1 minute 52 seconds on the 4 core Intel i7 laptop. This is extremely helpful for debugging purposes because changing the application code, re-building everything, and deploying UKL on QEMU/KVM is fast. One can simply build an application into a unikernel with less than two minute overhead, comparable to building a normal executable ELF in userspace.

4.3 Early Challenges

A number of challenges came up for which we report some preliminary solutions.

Namespace Issues: Some routines, e.g., memset, memmove, etc., exist both in the Linux kernel and glibc, and raise build time errors of multiple definitions. We fixed that by suppressing the glibc versions, but a more intelligent solution will have to be devised. For instance, we might do partial linking of glibc with application code and partial linking of the kernel separately, followed by a final linking step.

Malloc: We have mapped malloc in the UKL library through the vmalloc routine in the kernel. Going forward we might need to rethink this memory management design because we want to enable the rich, general purpose functionality of glibc malloc for application code. We also want to allow kernel level code to allocate buffers which can later be freed by an application using those buffers directly.

Primordial Thread Setup: We had to set up TLS memory in kernel space, the way glibc does [11], because that is what glibc expects for the primordial thread.

Major design questions like these will need to be addressed going forward.

4.4 Initial Evaluation

Figure 1 compares the performance of TCP echo server, written in C, running as a userspace process on Linux with the same TCP echo server linked into a unikernel. In both cases the server is deployed as a VM on QEMU/KVM on a host machine and the client is running on that host machine. While this is obviously a toy example, it's encouraging that the unikernel version of the application achieves an average latency half of that of the userspace application and a tail latency that is 41% faster.

Our prototype has given us strong confidence that it is feasible to convert Linux into a unikernel and, even without

any optimizations, obtain some performance advantages. We now have evidence that the required changes to Linux kernel and glibc code bases are limited and we do not foresee significant challenges getting them upstreamed. The (large number) of shortcuts to create this early prototype can be addressed without any major changes to either project.

4.5 Required work

Work to make the approach more widely usable includes:

1. Make UKL a GCC target so that existing makefiles can be used for arbitrary applications.
2. Develop a simple model for initializing devices and interfaces required by an application.
3. Add `crt0` support for C++ constructors.
4. Integrate the full glibc initialization code into the kernel startup.
5. Add customizable initialization of devices and kernel functionality.

5 Where We Are Going

After this prototype, we aim to improve it and answer the open design and research questions discussed earlier in section 4. Once we have a stable unikernel which can run any Linux application, there are a series of optimizations we can integrate that will have minimal impact on Linux or the application. First, and most obviously, link time optimizations can be explored given that the compiler/linker can see at link time the entire object code, enabling removal of unused code to reduce size, inlining code, especially short functions, for better latency, and exploiting value range propagation even across the application/kernel boundary.

Second, we can enable profile driven optimizations, which work nicely on statically linked binaries, where the entire unikernel is profiled to enable subsequent compiler/linker optimizations to further optimize code layout and cache footprint.

Third, the implementation of user-level synchronization mechanisms can be simplified and therefore sped up since the runtime environment can control scheduling and make assumptions about a thread being rescheduled or not.

Fourth, we can exploit information available in the application and/or library to bypass system call wrappers and directly invoke internal kernel functions; e.g., if we know that a descriptor is being used for a network socket, jump directly to the respective routine and avoid demultiplexing code.

Right now the entire Linux kernel is linked into the unikernel. One longer term research goal would be to strip down the unikernel to the bare requirements to support an application. We envision developing tools that can analyze a program and automatically strip out unneeded kernel functionality with fine-grained compile-time configurations for the kernel with the help of compiler extensions and source code annotations. The programmer, or tools analyzing the userlevel code, can express additional knowledge not represented in the source code for use of the compiler. For instance, it can provide assumptions on non-constant parameters (some file descriptor is always for a file). Also, we can simplify case handling in the kernel e.g., by expressing that network traffic is limited to TCP, various places in the kernel can be annotated to remove tests or indirect calls to handle other cases without any changes to source code.

As the unikernel model becomes an accepted part of Linux, our goal would be to enable the same kind of rich optimization that has been achieved in existing research unikernels. For example, exposing an alternative interface to read which does not dictate the location of the buffer but lets the kernel dictate the location (e.g., in a ring buffer or DMA buffer) [19]. As another example, exposing a flattened version of the network stack which does not implement the information security aspect of the full stack; if there is only one process there is no need for privacy and a single ring buffer is sufficient, enabling true zero-copy networking. As a more extreme example, complex runtimes like that for OpenMP can exploit true system-global information to dynamically create threads based on that information (number, affinity) rather than the heuristics they are forced to use today.

6 Concluding Remarks

There is clear evidence from the research on unikernels that they have substantial advantages for a wide class of applications in use today where a virtual or physical computer can be dedicated to running a single application. Linux has been enormously successful in adapting innovation and integrating it. Rather than unikernels being a threat to the dominance of Linux, we believe that the next natural evolutionary phase of Linux is to enable a unikernel model.

In this paper we have demonstrated that Linux can be turned into a unikernel, a fundamental question we were not sure of at the start of this project. Moreover, we have shown that the changes required are modest, and gathered initial evidence that they offer at least some performance advantages, two conditions that make such changes likely to be accepted upstream. We also have some evidence that there is substantial community interest in this work; our blog post [6] describing the vision in mid November was in the words of the editor "by far the most read post for 2018 on next.redhat.com."

We are at this point quite confident that Linux can and will adopt unikernels as a viable deployment model. Once this is accomplished, we have identified a whole series of natural optimizations that can, and likely will, be pursued by the broader Linux community. We believe that this will open up fundamental new research opportunities; the existence of a commercial grade unikernel will enable a much

broader research community to explore optimizations across the hardware, system software and application spaces, opportunities that until now have been only available to unikernel researchers.

While it is quite possible that we are wrong, our expectation is that, over time, the unikernel target will become the most important target for Linux, offering advantages for real-time environments, HPC, cloud applications, infrastructure components, etc. If this occurs, it will be a fundamental transformation, where the dominant use of an operating system changes away from the kernel/user model we have all grown up with. This may well be the beginning of the end of multi-user general purpose operating systems. Perhaps we are finally putting the "U" back into Unix.

References

[1] LibOS. https://github.com/libos-nuse/linux-libos-tools. (Accessed on 04/08/2019).

[2] November 2018 | TOP500 Supercomputer Sites. https://www.top500.org/lists/2018/11/. (Accessed on 04/08/2019).

[3] Kernel mode linux | linux journal. https://www.linuxjournal.com/article/6516, May 2003. (Accessed on 04/08/2019).

[4] Data Plane Development Kit. https://www.dpdk.org/, 2018. (Accessed on 04/06/2019).

[5] Storage Performance Development Kit. https://spdk.io/, 2018. (Accessed on 01/16/2019).

[6] UKL: A Unikernel Based on Linux. https://next.redhat.com/2018/11/14/ukl-a-unikernel-based-on-linux/, November 2018. (Accessed on 04/08/2019).

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, 2014. USENIX Association.

[9] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 250–257. IEEE, 2015.

[10] Jeff Dike. A user-mode port of the Linux kernel. In *Annual Linux Showcase & Conference*, 2000.

[11] Ulrich Drepper. ELF Handling For Thread-Local Storage. December 21, 2005.

[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*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[13] Jon Howell, Bryan Parno, and John R. Douceur. Embassies: Radically refactoring the web. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 529–545, 2013.

[14] Intel. https://www.dpdk.org/, 2010. [Online; accessed 17-January-2019].

[15] Jake Edge. ELC: SpaceX lessons learned. https://lwn.net/Articles/540368/. [Online; accessed 7-April-2019].

[16] Antti Kantee. The Rise and fall of the Operating System. http://www.fixup.fi/misc/usenix-login-2015/login_oct15_02_kantee.pdf. (Accessed on 04/08/2019).

[17] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. OSv - Optimizing the Operating System for Virtual Machines. In *Proceedings of USENIX ATC 2014: 2014 USENIX Annual Technical Conference*, page 61, 2014.

[18] Ricardo Koller and Dan Williams. Will Serverless End the Dominance of Linux in the Cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 169–173. ACM, 2017.

[19] Orran Krieger, Michael Stumm, and Ron Unrau. The Alloc Stream Facility: A Redesign of Application-level Stream I/O. *Computer*, 27(3):75–82, March 1994.

[20] Krieger, Orran and Auslander, Marc and Rosenburg, Bryan and Wisniewski, Robert W. and Xenidis, Jimi and Da Silva, Dilma and Ostrowski, Michal and Appavoo, Jonathan and Butrico, Maria and Mergen, Mark and Waterland, Amos and Uhlig, Volkmar. K42: Building a Complete Operating System. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM.

[21] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[22] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the Virtual Library Operating System. *Queue*, 11(11):30, 2013.

[23] Toshiyuki Maeda and Akinori Yonezawa. Kernel Mode Linux: Toward an Operating System Protected by a Type Theory. In *Annual Asian Computing Science Conference*, pages 3–17. Springer, 2003.

[24] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233. ACM, 2017.

[25] Kirk McKusick. The jail facility in FreeBSD 5.2. https://www.usenix.org/publications/login/august-2004-volume-29-number-4/jail-facility-freebsd-52, 2005. (Accessed on 04/05/2019).

[26] Michael Larabel. The Linux Kernel Has Grown By 225k Lines of Code So Far This Year From 3.3k Developers. https://www.phoronix.com/scan.php?page=news_item&px=Linux-September-2018-Stats, 2018. (Accessed on 01/16/2019).

[27] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. LKL: The Linux Kernel Library. In *Roedunet International Conference (RoEduNet), 2010 9th*, pages 328–333. IEEE, 2010.

[28] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[29] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *Operating Systems Design and Implementation*, volume 16, pages 671–688, 2016.

[30] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[31] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 133–145, Berkeley, CA, USA, 2018. USENIX Association.

[32] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D.E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed

File System. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[33] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels As Processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 199–211, New York, NY, USA, 2018. ACM.