Will Serverless End the Dominance of Linux in the Cloud?

Ricardo Koller
IBM T.J. Watson Research Center
kollerr@us.ibm.com

Dan Williams
IBM T.J. Watson Research Center
djwillia@us.ibm.com

ABSTRACT

From the inception of the cloud, running multi-tenant workloads has put strain on the Linux kernel's abstractions. After years of having its abstractions bypassed via virtualization, the kernel has responded with a native container abstraction that is eagerly being applied in the cloud. In this paper, we point out that history is repeating itself: with the introduction of *serverless* computing, even the native container abstraction is ill-suited. We show that bypassing the kernel with unikernels can yield at least a factor of 6 better latency and throughput. Facing a more complex kernel than ever and a relatively undemanding computing model, we must revisit the question of whether the kernel should try to adapt, we should continue bypassing the kernel, or if it is finally time to try a new native OS for this important future cloud workload.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; *Software as a service orchestration system*;

ACM Reference format:
Ricardo Koller and Dan Williams. 2017. Will Serverless End the Dominance of Linux in the Cloud?. In *Proceedings of HotOS '17, Whistler , BC, Canada, May 08-10, 2017,* 5 pages.
https://doi.org/10.1145/3102980.3103008

1 INTRODUCTION

The Linux kernel has evolved into a sacred software component that underpins everything and that can run directly on almost any type of hardware. In particular, Linux now powers smartphones, tablets, cars, cloud data centers and high-performance supercomputers [13]. In the cloud domain, the kernel has once again seized the spotlight. For ten years, virtualization has relegated the kernel to be a simple and thin hypervisor. But now, as virtualization is being dismissed as too heavyweight, the relatively lightweight containers implemented by the kernel are slated as the new face of the cloud [23].

In this paper, we predict that this milestone is in fact the beginning of the end of the ubiquity of Linux in the cloud.[1] We base this prediction on the confluence of two fairly obvious trends: that the complexity of the kernel continues to grow, and that the unit of execution on the cloud continues to shrink.

First, the complexity of the kernel has significantly increased, especially in the past few years. To introduce support for containers, the Linux kernel underwent major changes to include new abstractions like `namespaces` and `cgroups`. This took years. The next significant changes will take even longer, because complexity affects implementation time, and the Linux kernel's complexity has been growing quadratically with time [12, 22, 24]. This is why bypassing the kernel makes sense: it is either to hard to implement something in the kernel, too hard to optimize the relevant code paths in the kernel, or too hard to secure the kernel. For example, fully bypassing the kernel is still commonplace for data plane network services using hardware-level virtualization and software frameworks like the Data Plane Development Kit (DPDK) [5].

Second, the units of software running as cloud workloads are evolving. The current trend in the cloud is to run a relatively small, lightweight application rather than a more heavyweight system [23]. The industry hype around so-called *serverless* architectures [2, 4, 6] is an indication that this trend will continue beyond applications to run even smaller, more lightweight lambdas or *actions*. A kernel abstraction designed for starting an application once every few hours is unlikely to perform well for an action starting once every hundred milliseconds or so. Indeed, we provide supporting evidence for this claim in Section 3.

Yet, if history is any indication, the need for an efficient multi-tenant implementation will inevitably cause the kernel to evolve to support serverless actions as a native abstraction. In this paper, we identify three approaches to adapt the kernel for these workloads. First, it is possible to continue to directly modify or extend the kernel, as was done with containers, despite the complexity required in doing so. Second, kernel-bypass architectures could become even more commonplace, relegating Linux once again to a sideline role. Finally, the kernel in the cloud could be completely replaced with something better suited to emerging cloud workloads.

The rest of the paper is organized as follows. In Section 2, we describe the characteristics and requirements of serverless and the tradeoffs present in each of the three approaches mentioned in the previous paragraph. Section 3 shows evidence of the current inability of Linux to efficiently implement new serverless abstractions, which we believe is a sign that the general purpose kernel's unchallenged use cloud is unsustainable. In Section 4, we discuss whether the kernel as we know it will linger along with bypass techniques or if it will be replaced by an entirely new design. Ultimately, in Section 5, we conclude, at least for serverless, that it is counterproductive to extend the kernel and that the efforts of the community would be better spent shrinking or completely replacing the kernel.

[1] We are not the first to take issue with a current OS: indeed, 22 years ago Engler and Kaashoek asserted that "[T]hroughout the history of computer science there has been a fairly constant opinion that current operating systems are inadequate" [8].

HotOS '17, May 08-10, 2017, Whistler , BC, Canada

ACM ISBN 978-1-4503-5068-6/17/05…$15.00
https://doi.org/10.1145/3102980.3103008

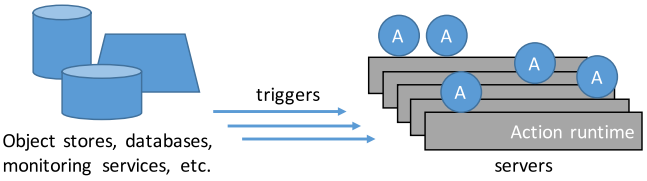

Figure 1: A serverless architecture. User-supplied actions (denoted by blue circle A's) are invoked in response to events from services in the cloud data center.

2 RETHINKING THE KERNEL

It should not be a surprise that as time has gone on, the complexity of the Linux kernel has increased quadratically with time [12, 24]. Beyond this observation, one could ask what percentage of that complexity is necessary to implement various cloud abstractions. In this section, we briefly describe an emerging cloud model: the *serverless* computing model. We assert that, for serverless, very little of the complexity of the Linux kernel is necessary, further tipping the scales away from Linux as the best kernel for the job.

2.1 The serverless model

Figure 1 depicts an overview of a serverless architecture. The basic idea behind serverless is that users upload code that is associated with an event in the cloud (e.g., from a data store or other service). The code, called a lambda, or an *action*, is only run when the event happens and billed on a 100 ms granularity; the user does not need to pay for a server to wait idly for events (hence "serverless"). Existing implementations limit how long an action can run [2, 6]. The canonical example action computes an image thumbnail, triggered every time an image is uploaded into cloud storage. Applications that trivially fit the serverless model include periodic antivirus scans or spam classifiers where each action operates on a single email or file. The boundaries of what types of workloads can be mapped into serverless are still being explored: for example, mapreduce has been implemented in a serverless model where each mapper or reducer is started as an action [11].

As serverless systems in the wild are still nascent, the precise definition of an action varies from system to system. Conceptually, actions comply to a restricted computing model. They implement functions, not general-purpose servers or applications. In the extreme, restricting actions to be non-preemptable, run-to-completion threads of execution with a single input and a single output would not violate the conceptual model. For the purposes of this paper, we primarily consider two relatively general-purpose implementations of actions: libraryOS approaches [9, 18], specifically unikernels [14–16, 26], and Linux containers.

Regardless of the specifics of the programming model and its implementation, there are some clear technical requirements for the actions in a serverless platform. First, as actions run arbitrary code from multiple tenants, they must remain well-isolated from other tenants and the host platform. This excludes implementations based solely on native Linux process, for example. Second, actions must perform well. The specific types of performance we are interested in are:

- **Low latency:** A user expects actions to launch instantaneously on an event. From a provider's perspective low latency also reduces complexity: it removes the need to manage caches that hide high startup latency. To give a ballpark number, we would the serverless platform to be able to fetch and start any action, without any previous cached state, under 100 ms, regardless of system load.
- **High throughput:** From a provider perspective, the number of actions run on a machine must be high enough to cover the cost of running the machine, even if the actions are very short-lived. We did some back of the envelope calculations using Amazon Lambda [3] and EC2 pricing [1]. A 4-core EC2 instance (to match our experimental setup in Section 3), costs $0.188 per hour (`t2.xlarge`). In order for Lambda to be cost effective, and taking into account that users pay $0.000000417 per action (256 MB per action), we need to have at least $(0.188/0.000000417)/(60 * 60)$, or 125.23, actions/second.

In Section 3, we show experimentally that the Linux kernel is currently ill-suited to reach these performance objectives.

2.2 How to adapt for serverless

Assuming that the kernel will eventually evolve to natively support multi-tenant actions as it did to support multi-tenant containers, we identify three possible approaches, shown in Figure 2. The first one, denoted (a), is to enhance the Linux kernel and its container-related capabilities for actions. This appears to be the approach favored by industry, perhaps by default, as containers are the most popular configuration in existing serverless systems [2, 6]. The second option, denoted (b), is to bypass the complexity of the kernel and add another layer from which actions can be started. Examples of this approach could include a hypervisor (e.g., QEMU) or unikernel monitor [26] (e.g, `ukvm`) to run the action as a unikernel. In this way, only basic device drivers are used out of the Linux kernel, and functionality like TCP/IP is implemented in the unikernel's library OS. The final option, denoted (c), is to write a more appropriate host OS from scratch that is more suitable to run serverless actions. This does not mean configuring a stripped-down Linux; this option implies a completely new set of user abstractions that may not even look like our traditional and familiar processes.

As shown in Figure 2, choosing between these 3 options involves tradeoffs in two dimensions. Specifically, the different approaches trade impediments to efficiently supporting actions that stem from 1) dealing with a complex legacy code base or 2) dealing with a lack of familiar/useful abstractions.

- **Impediments due to complexity** (the bottom axis on Figure 2). Although an old project like the Linux kernel has lots of useful abstractions, it is a complex code base. Software engineering studies have shown that maintenance cost increases polynomially with the number of lines in a software project [12, 22]: $Cost = constant * Lines^{1.3}$. To worsen the outlook, the number of lines in the Linux kernel is increasing at a quadratic rate with time [12, 24]. This also has a direct impact on how hard it is to extend the

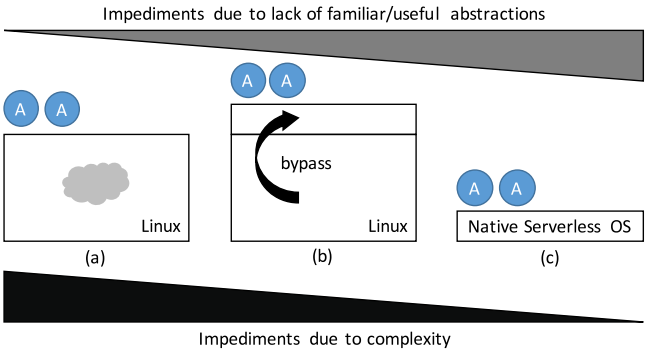

Figure 2: Three options for designing native support for serverless actions (denoted by blue circle A's). In option (a), the kernel is extended similarly to how container support was recently introduced. In option (b), the kernel (and its complexity) is largely bypassed. In option (c), a new, tailor-made kernel is created.

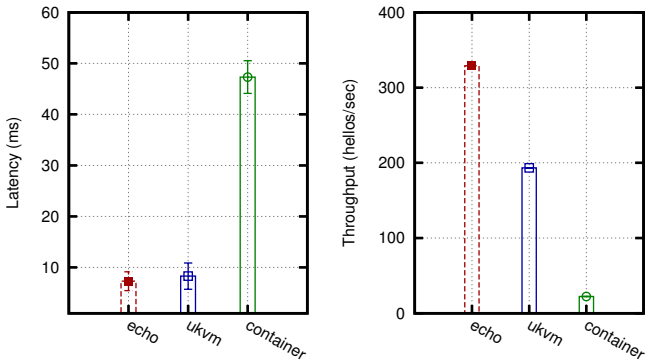

Figure 3: Latency and throughput of actions implemented as a raw Linux process (*echo*), as a unikernel on `ukvm` and as a `runc` container.

code base. Extending the kernel to support a new serverless abstraction (e.g., option (a)) sounds like daunting task.

- **Impediments due to lack of familiar/useful abstractions** (the top axis on Figure 2). Well-established projects like the Linux kernel have several APIs that make it easy to write an initial version of a serverless platform (i.e. containers as actions). On the other hand, writing a new OS from scratch (e.g., option (c)) would require the creation of completely new (although more appropriate) abstractions for serverless actions. As noted above, and discussed in Section 4, there is an opportunity to define the action abstraction in such a way as to dramatically simplify the implementation of such a "native serverless" OS.

In the middle, option (b) is a compromise. It involves a somewhat convoluted architecture in terms of bypassing the kernel, but is relatively easy to implement and high performing. The rise of interest in library OS's through unikernels [15] has decreased many of the impediments caused by the lack of useful abstractions. We next adopt this strategy to provide a counterpoint while demonstrating that a state-of-the-art Linux abstraction is not appropriate for serverless actions.

3 THE KERNEL IS NOT READY FOR SERVERLESS

In this section, we perform some experiments to demonstrate that Linux containers are unable to achieve the performance goals we put forth in Section 2. At the same time, we show that a kernel-bypass approach using unikernels can achieve these goals and further assert that such an approach is in fact easier to implement.

3.1 Containers do not perform well enough

We are interested in the raw performance of the underlying action abstraction in the two metrics described in Section 2: latency and throughput. Ideally, we would like to see latency under 100 ms and throughput above 125.23 *actions/second*. We do not consider caching strategies, as they inevitably affect user experience as a performance wall [10]. In the Linux case, we evaluate containers. We compare the container performance to a kernel-bypass approach that uses ukvm-based unikernels [26] and also report performance for normal Linux processes as a baseline.

Experimental setup. We evaluate the performance of running the simplest action possible: a hello-world function that takes no input and outputs a "Hello" on the console. The baseline native Linux process used is `echo Hello`.

To give the Linux container the best performance possible, we create an optimized container by reducing it to a microcontainer [25] with no `libc`. In other words, rather than including an entire Linux distribution, the container only includes a statically-linked binary that does not even link `libc`. The binary is a small application written in inline assembly that directly invokes a system call to output "Hello" on `stdout`. The container is invoked with `runc` using a standard `config.json` and a precreated layered `rootfs` using `overlayfs`. We did not set up network interfaces by default.

The comparison unikernel, referred to as *ukvm* in the experiments, runs a Solo5/ukvm unikernel that prints "Hello" onto the serial console then exits [7]. It is invoked with `ukvm-bin`, a specialized unikernel monitor generated to run the unikernel [26]. `ukvm-bin` is modular, and a "hello world" unikernel only needs a serial console to output a string, so that is the only device (i.e. module) set up by the monitor. `ukvm-bin` runs as a Linux process and loads the unikernel in a new VCPU context via the Linux KVM system call interface.

We ran all experiments on a 4-core Intel i7 2.6 GHz X1 Carbon ThinkPad, with an SSD and RHEL 7.3, Linux 3.10 kernel.

Examining latency. For latency, we measure the time spent setting up and initializing the container or unikernel for it to be able to output the string ("Hello") over `stdout`. For each run, the latency is calculated as the difference between the timestamp (host time) at which the message was received and the timestamp at which the action was invoked.

Figure 3 (on the left) shows the results (averaging 5 runs): In our setup, the baseline, *echo*, a regular (non-namespaced) Linux process takes only 5.8 ms. Starting a unikernel, depicted *ukvm*, takes 7.8 ms (only 34% more than a regular Linux process).

Although `runc` is a minimal container runtime (when compared to e.g., `Docker`), it still does much more than just start the process. It sets the isolation mechanisms for the process by invoking namespace and cgroup APIs. This takes 23 milliseconds on a pre-allocated `rootfs` mount. Using the overlay file system, the average startup time was 47 ms (about 22 ms are spent setting up an overlay FS). Both the microcontainer and unikernel binaries are small enough that fetch time is under 2 ms.

Both are under the 100 ms bar we proposed in Section 2, although containers are almost 6 times slower to boot. We should reiterate that this is a best case for containers, as we measured container startup times for Alpine Docker containers to be in the 100s of milliseconds. Additionally these more traditional (non-micro) containers can also have fetch times in the 100s of milliseconds.

Examining throughput. As described in Section 2, throughput is of crucial importance to providers. We measured the system throughput using a simple workload generator. We used an open system model, where the generator spawns actions at a given rate, regardless of feedback from the system under load [20]. To obtain the max throughput of the system, load is increased until a saturation point. We note that this method can be skewed by contention from the load generation.

Figure 3 (on the right) shows the results. The max throughput numbers (in hellos/sec) are 329 for echo, 193 for ukvm, and 24 for runc. ThatâĂŹs an 8x performance improvement for unikernels over containers. Looking at the back-of-the-envelope calculation in Section 2, we see that this performance could actually have real-world implications: given those numbers, a container throughput (24 hellos/sec) would not be enough to be cost-effective (125 actions/sec) for the serverless provider. On the other hand, bypassing the kernel with ukvm does enter the desired range at 193 hellos/sec.

3.2 Why didn't we just fix containers?

The fact that we chose a bypass implementation isn't by any means proof that Linux has reached a point where it is more cost effective to implement serverless actions using the bypass model. However, we did not choose to modify the Linux kernel because we suspected that it would need new features and performance improvements, which may have impacted many subsystems (e.g., like the changes to support containers).

A less personal indication of the complexity of Linux becoming a burden has come from industry. Despite great attention from the industry and the popular use of containers for serverless platform implementations [2, 6, 10] the Linux kernel has still not yet been adapted or fixed to provide the necessary throughput for serverless.

4 WHAT ROLE WILL LINUX HAVE IN THE CLOUD?

Based on these observations, it might be time to rethink the Linux kernel as the default OS for the cloud, or at least the parts of the cloud (like serverless) where the unit of execution doesn't look like a system or a traditional process. As we saw in the previous sections, there are two clear trends: the Linux kernel complexity is rapidly growing and the units of the cloud are getting more lightweight and restricted. The consequence of the units of the cloud being more restricted is that they need less from the layer underneath (i.e. the OS). As a result, bypassing the kernel, or even building a serverless platform from scratch looks relatively more attractive compared to modifying the kernel than it did 10 years ago.

Bypassing the kernel. What will happen to the Linux kernel if it is routinely bypassed for serverless workloads? It is possible that the kernel will evolve to make bypass easier and easier, reducing the complexity of taking such an approach. However, the goals of a Linux kernel for a traditional system differ wildly from the goals of a Linux kernel primarily for bypass. One could argue that changes that affect large parts of the kernel (e.g., `namespaces` or `cgroups`) are not desirable as they affect the stability and security of the kernel. Would the kernel community split between those that want a thin, bypass-ready kernel for serverless (and other performance-focused domains) and those that want a more traditional system?

Replacing the kernel. If serverless actions are defined in a restricted manner, as described in Section 2, a âĂIJnative serverlessâĂİ OS could be implemented without many features of a traditional OS:

- **Non-preemptable scheduling**. If actions are short and non-interruptible, then we could use a non-preemptive scheduler for the actions.
- **Limited set of I/O related calls**. If the only necessary I/O is an input at startup and output at completion, then actions do not need access to files, networking, nor many of the +200 syscalls implemented by Linux.
- **No inter-process communication (IPC)**. IPC is mainly used for messages in a client-server model or shared memory for multiple processes. It is not needed in serverless actions, as they can only interact with one another via events generated at completion and used as input at startup by another action.
- **No process synchronization**. Semaphores and locks might make sense for synchronizing access to shared data, or more generally to let multiple processes cooperate in an orderly fashion. The only cooperation allowed between actions is done via queuing of events.

Such an OS may share many features with a multi-kernel like Barrelfish [21] or Arrakis [17], which have an explicit goal of not sharing data between processes running in different cores. The main difference would be that instead of providing a POSIX-like abstraction of processes, perhaps actions would have a limited library OS designed to only support the required interactions: one input and one output. The benefit of this would be simplicity, potentially better performance, and a smaller attack surface.

. This discussion is also related to the idea that "the OS is the control plane" [17]. Linux may likely remain active in the cloud, even in a serverless context, in a control-plane role. This could happen at different levels. Linux could be a per-machine controller dictating how actions are queued and passed around between actions running in different cores, fitting well with a bypass design.

Or, Linux may be a per-cluster controller, passing actions to servers running a specialized native serverless OS, similar to in EbbRT [19].

5 CONCLUSIONS?

It should be surprising that taking an approach that uses relatively heavyweight hardware in a VCPU context (e.g., performing `vmexit` instead of `sysenter` for a "system call") is not only acceptable but can perform an order of magnitude faster than using a native OS abstraction. Yet, today, in a serverless setting, this is actually the case. Will we struggle on in this way or have we finally reached the point, precipitated by the serverless cloud trend, where it is reasonable to rethink the kernel?

REFERENCES

[1] 2016. Amazon Elastic Compute Cloud Pricing. https://aws.amazon.com/lambda/pricing/. (2016). (Accessed on 2017-01-25).

[2] 2016. AWS Lambda. https://aws.amazon.com/lambda/. (2016). (Accessed on 2016-03-04).

[3] 2016. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/. (2016). (Accessed on 2017-01-25).

[4] 2016. Azure Functions. http://azure.microsoft.com. (2016). (Accessed on 2017-01-25).

[5] 2016. Data Plane Development Kit (DPDK). http://www.dpdk.org/. (2016).

[6] 2016. IBM OpenWhisk. https://developer.ibm.com/open/openwhisk/. (2016). (Accessed on 2016-03-04).

[7] 2016. The Solo5 Unikernel. https://github.com/djwillia/solo5. (2016). (Accessed on 2017-01-25).

[8] D. R. Engler and M. F. Kaashoek. 1995. Exterminate all operating system abstractions. In *Proc. of USENIX HotOS*. Orcas Island, WA.

[9] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of ACM SOSP*. Copper Mountain, CO.

[10] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with OpenLambda. *Elastic* 60 (2016), 80.

[11] Ho Ming Li. 2017. lambda-refarch-mapreduce. https://github.com/awslabs/lambda-refarch-mapreduce. (Jan. 2017). (Accessed on 2017-05-25).

[12] Ayelet Israeli and Dror G Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* 83, 3 (2010), 485–501.

[13] Jim Zemlin. 2013. The Year of Linux on the...Everything. https://www.linux.com/blog/2013-year-linux-theeverything. (2013). (Accessed on 2017-05-26).

[14] Antti Kantee. 2015. The Rise and Fall of the Operating System. *USENIX ;login:* 40, 5 (2015), 6–9.

[15] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proc. of ACM ASPLOS*. Houston, TX.

[16] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proc. of USENIX NSDI*. Seattle, WA.

[17] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2016. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)* 33, 4 (2016), 11.

[18] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. 2011. Rethinking the library OS from the top down. *ACM SIGPLAN Notices* 46, 3 (2011), 291–304.

[19] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 671–688.

[20] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open Versus Closed: A Cautionary Tale. In *Proc. of USENIX NSDI*. San Jose, CA.

[21] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. 2008. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*. 27.

[22] Ian Sommerville. 2010. *Software engineering*. Pearson. 623 pages.

[23] David Strauss. 2013. The future cloud is container, not virtual machines. *Linux Journal* 2013, 228, Article 5 (April 2013).

[24] Dominik Strzałka. 2012. Fractal properties of Linux kernel maps. *Computer Science and Engineering* 2, 6 (2012), 112–117.

[25] Travis Reeder. 2017. Microcontainers - Tiny, Portable Docker Containers. https://www.iron.io/microcontainers-tiny-portable-containers/. (Jan. 2017). (Accessed on 2017-05-25).

[26] Dan Williams and Ricardo Koller. 2016. Unikernel monitors: extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association.