2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN) | 978-1-6654-4579-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/LANMAN52105.2021.9478808

Fast Function Instantiation with Alternate Virtualization Approaches

Vivek Jain, Shixiong Qi, K. K. Ramakrishnan
University of California, Riverside

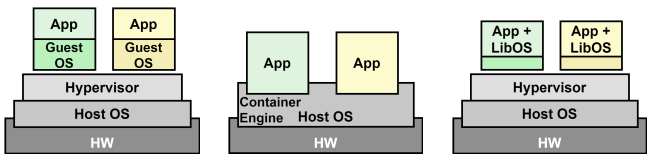
***Abstract*—This paper focuses on the need for emerging domains such as serverless and in-network computing, where applications are often hosted on virtualized compute instances (*e.g.,* containers and unikernels), to have applications startup as quickly as possible. We provide a qualitative and quantitative analysis of containers and unikernels with regard to the startup time. We analyze these in-depth and identify the key components and their impact under scale on the startup latency. We study how startup time scales as we launch multiple instances concurrently. We study the contribution of popular Container Networking Interfaces (CNIs), to the startup time.**

***Index Terms*—Unikernel, Container, cold start, NFV, startup**

I. Introduction

On-demand services require agile compute and good scalability to support a wide variety of applications with low latency and efficient resource utilization. Early-stage cloud deployments depended on Virtual Machines (VMs) as the main virtualization framework for having monolithic applications running on bare-metal using the infrastructure-as-a-service paradigm. VMs (Fig. 1(a)) rely on hardware-level virtualization and contain a full-fledged Operating System (OS) with a large memory footprint and may take multiple seconds to initiate. We have evolved to container (Fig. 1(b)) technology for process-level virtualization, leveraging the host OS. Due to it being lightweight with much smaller startup times, containers are now the industry de-facto option for running applications, often breaking them up into micro-services. Unikernels (Fig. 1(c)) [1] are another virtualization technology that uses the concept of Library OS [2], where applications are built as a VM with minimal OS functionality. Since most OS/Kernel functionalities are omitted, unikernels are very lightweight, and their bootup time is significantly shorter than traditional VMs.

An application's startup time includes initializing the computing platform. While containers and unikernels have a lower startup time over traditional VMs, further improvements are desirable. A number of areas need fast initialization of compute instances: ranging from deploying per-flow network functions [3] for in-network compute and network function virtualization (NFV) to support serverless computing [4], and rapid application migration across edge cloud instances. For example, when a user invokes an inactive serverless function, the cloud orchestration system provisions a new compute instance (*e.g.,* container) to execute the function. The provisioning of compute instances includes downloading the code, setting up the container environment, and starting the container. The time for provisioning a compute instance (referred as *coldstart time*) depends on various factors such as code size, memory size, program runtime. To mitigate this issue, cloud providers usually keep the function container running for some time, *i.e., warm* containers. In addition to these factors, computing platforms like containers are also vulnerable due to their dependency on shared underlying kernel capabilities with the use of locks for resource access, impacting the startup time when multiple containers concurrent instantiated.



(a) Virtual Machines (b) Containers (c) Unikernels

Fig. 1. Virtualization technologies

Several works have investigated the fast startup of container instances for emerging domains, *e.g.,* NFV and serverless. Zhang *et al.* [3] have demonstrated the ability to launch 80K NFs instances during a second interval. Similarly, the authors [5] have shown a reduction of up to 80% in execution time in launching container instances. The fast function initialization is the most sought-after compute requirement for supporting modern applications. This paper analyzes the startup time of different computing platforms and finds the bottleneck components contributing most to the startup time. For instance, our experiments show that network initialization and mounting play a significant part in starting up multiple containers. These insights will help us design a low startup latency computing platform to fulfill the agile computing requirements. We make the following contributions:

- We provide the qualitative analysis of computing platforms, mainly containers and unikernel. We choose Linux containers, the Kubernetes orchestrator, along with the *runc* container runtime. This will provide more detailed system-level insights (*e.g.,* setting namespaces) and cluster-level factors (*e.g.,* network setup overheads). For the unikernel, we choose the OSv because it has been shown to have a short startup time, as low as 5*ms* [6].
- We provide an in-depth quantitative analysis of components such as a hypervisor, network setup, and mounting techniques that affect startup latency and highlight their effect during scale. We evaluate pod network startup time of vari-

978-1-6654-4579-5/21/$31.00 ©2021 IEEE

ous Container Network Interface (CNI) plugins by analyzing the latency breakdown associated with different networking components affecting startup time. An extensive analysis of CNI is presented by considering different pod deployment patterns and scales to help understand the scalability in terms of startup latency.

II. Qualitative Analysis of Virtualization Platforms

This section provides an overview of different virtualized computing platforms, *i.e.,* containers and unikernels. Given the critical need for fast initialization (*e.g.,* function as a service, NFV), we focus on providing a detailed analysis of components involved during startup and highlight their effects, including when they have to be deployed on a large scale.

A cloud service provider (CSP) typically uses orchestration software such as Kubernetes [7]. Most of these orchestration engines have a similar two-stage pipeline: i) selecting a node to run a compute instance; ii) starting the compute instance on a selected node by setting up the required functionalities (*e.g.,* network and storage). We believe the first stage is independent of virtualized technology. The second stage is specific to computing technology and is discussed next. Apart from this, we also study few orchestration features, *e.g.,* CNI plugins used for setting up container networking. Understanding the impact of these on the startup in detail will help in devising strategies for their improvement.

A. *Unikernels*

Unikernels are specialized, single-address-space virtual machines that are based on Library OSs [2] that bundle required functionality. Unikernels are designed to run a single application process, and the customization makes unikernels lightweight with a faster startup than traditional VMs. The overall application startup time for unikernels comprises three phases: i) Hypervisor context setup time, ii) unikernel initialization time, and iii) application initialization time. We discuss the former two initialization phases (the third phase depends on the application itself).

i) Hypervisor context setup time: Hypervisors have a significant impact on the startup time of the unikernel. For example, QEMU follows a 16-bit real mode boot model to start any x86 VM, in which a bootloader code is executed first that starts the kernel process. On the other hand, hypervisors like Firecracker directly execute the kernel binary, reducing overall OS boot time. Similarly, the initialization of the VM process also depends on the hypervisor. For instance, QEMU's startup time is primarily spent in the dynamic linker, which grows higher as the number of features increases.

ii) Unikernel startup time: The startup procedure for a unikernel is similar to a traditional VM bootup process and depends on the underlying hypervisor. We listed the breakdown of steps involved in the OSv kernel startup in Table I. These involve initializing various console capabilities (serial vs. VGA), mounting file system (read-only vs. write-enabled), *etc.* In our experiments, we observe that these components significantly influence the startup time, especially when a number of unikernels are initiated. But these components are not always required by an application. For instance, serverless functions are often stateless, and some also do not involve disk write operations. So the startup time can be improved by using a read-only file system.

TABLE I
Steps involved in OSv startup

Task	Description
Disk read (real mode)	read compressed kernel and app image
Uncompress kernel image	uncompress kernel and run it
Network initialization	initialize network stack, setting IP address
Drivers loading	loading device drivers e.g., block, network
File system mounting	mounting image file system
Other	thread initialization, .init functions, etc

B. *Containers*

In Kubernetes, a pod is an atomic unit that comprises one or more containers. We follow the "one-container-per-pod" model [7], using the terms container and pod interchangeably. Containers encapsulate an application and all of its dependencies into one bundle, utilizing underlying kernel capabilities for features like namespaces and cgroups. Namespaces provide an abstraction to the process of owning an isolated instance of the global kernel resources (*e.g.,* network ports). There are different types of namespaces available in the Linux kernel: *mount, process ID, network, IPC, User, UTS*. Cgroup functionality limits the use of resources such as the number of CPU cores and memory. The container engine is responsible for creating namespaces and setting up cgroup policies for a container. Container startup involves two phases: container process creation & network setup discussed next.

i) Container process creation: Fig. 2 shows the sequence of operations involved in creating a container sandbox. *i)* First, the downloaded container image is extracted. *ii)* The low-level container runtime creates a new container process using a `fork()` system call. *iii)* The container process sets the extracted images as its root file system using `chroot()` system call. *iv)* It also creates new namespaces using `unshare()` and `setns()` system calls. *v)* The next step is to perform the mount operation. *vi)* The CPU and memory limits are configured using a control group utility. *vii)* Finally, security level and root capabilities are set.

The implementation of these system calls is often optimized (*e.g.,* `fork()` leverages copy-on-write strategy to avoid a copy operation). But they still suffer the penalty of a mode and context switch, which can take several microseconds [8]. Accumulating all the system call penalties can grow to the order of 100s of microseconds for single container creation. When multiple containers are created in parallel, the penalty increases due to frequent context switches, causing frequent cache and TLB flushes. Moreover, the creation of a network

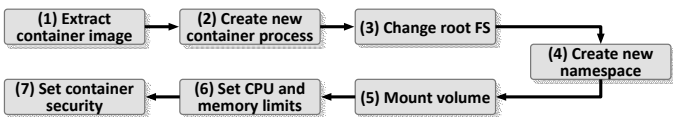


Fig. 2. Steps for container process creation

namespace does not set up virtual interfaces and assign IP addresses, which requires additional steps discussed next and are often done with external tools.

ii) Network setup: After a container sandbox is created, the network configuration is performed to enable communication between pods. As containers leverage the virtual Ethernet interface (veth) for networking, the network setup is started by creating a veth-pair in the default network namespace. One veth is moved to the container's network namespace. The veths are both then brought up and the veth in the container's network namespace is assigned with an IP address. Routes are configured in the default network namespace to ensure connectivity. Each of these operations involves updating the network namespace context inside the kernel. Network namespaces are organized in the form of a linked list inside the Linux kernel. Any update operation (*e.g.,* attaching veth) needs to acquire a global lock first and then perform a linear lookup for the namespace to be updated. Two issues arise: i) As the number of namespaces grows, the search time also increases. ii) In case of multiple network creation requests, the parallel update operation will be ordered due to the need to acquire a global lock, allowing one update at a time [9]. This serialized access significantly affects the startup time.

Furthermore, orchestration engines like Kubernetes rely on CNIs to automate network setup operations. A variety of available CNIs uses different approaches (*e.g.,* eBPF vs. iptables) to set up the pod network [10]. These CNIs introduce additional overhead based on the functionality they provide. For instance, Cilium leverages eBPF for packet forwarding/filtering, which requires an eBPF program to be loaded into and then attached to the kernel hooks inside the network devices (*e.g.,* veth). This process can take time and impacts the overall setup time.

III. Evaluation and Analysis

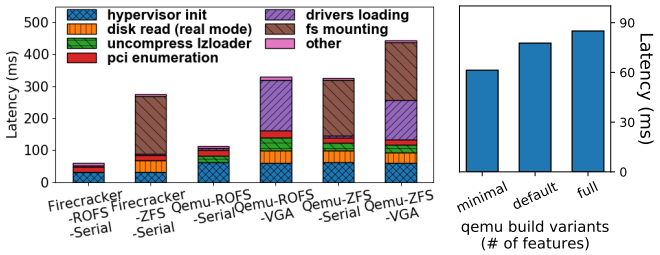
A. Experimental Setup

We evaluate the startup time of unikernels and containers on CloudLab [11]. The CloudLab node has an Intel Xeon CPU E5-2640v4@2.4GHz with 20 Cores, 64 GB of RAM, and running Ubuntu 18.04.1 LTS (kernel 4.15.0-137-generic). All the experiments are repeated more than ten times.

B. Unikernels

We selected the OSv unikernel because of its known fast startup times (as low as 5*ms* excluding hypervisor initialization time). In our experiments, each unikernel instance needs 1 CPU, 1 GB of memory, and contains one virtual NIC with a static IP assignment as part of OSv unikernel initialization.

1) ***Detailed Startup breakdown: single instance****:* Fig. 3(a) shows the effect of different components used in a unikernel, and the alternatives for each, *e.g.,* type of console drivers, file system type, and the hypervisor. We present the contribution of the hypervisor initialization, file system mounting, drivers loading, *etc*, on the overall unikernel startup latency. We evaluated the startup time of OSv on two hypervisors (QEMU and Firecracker both using KVM acceleration). Similarly, we explored different file systems (read-only FS vs. ZFS), and console drivers (serial vs. VGA). However, we only evaluate



(a) Unikernel boot time breakdown with 1 instance (b) QEMU init. time

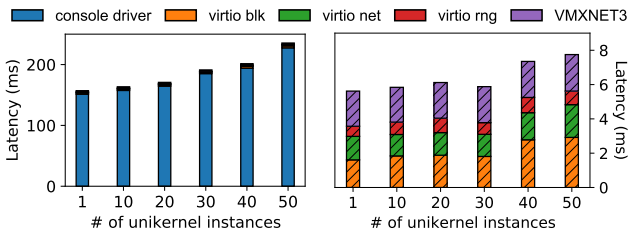
Fig. 3. Unikernel boot time breakdown & QEMU init. time with different QEMU builds

performance with serial devices on Firecracker because of its limited support for emulated devices.

a) Effect of hypervisor: The hypervisor initialization time is the time between when the hypervisor process is forked, and when the guest kernel begins its initialization. Fig. 3(a) shows that the hypervisor initialization, which depends on the type of hypervisor, heavily influences the overall unikernel startup time. The hypervisor initialization time is independent of the other OSv components installed. Further, the hypervisors' boot method has a direct impact on the startup time of a unikernel. QEMU uses the real mode booting model that requires a BIOS to load and boot a unikernel. This process increases the startup time because it has to read and uncompress the kernel image file. On the other hand, Firecracker follows the 64-bit Linux Boot Protocol Standard, which avoids the additional penalty incurred by the BIOS and bootloader and instead directly executes the unikernel binary.

The number of features supported by hypervisor also affects its initialization time. For example, QEMU's startup time is spent in detecting shared libraries and dynamically linking them. As shown in Fig. 3(b), the overall QEMU initiation time further increases with the number of features. The x-axis represents three different variants of the QEMU build. The rest of our experiments use minimal QEMU features required to run OSv, so as to get a fair estimate of the startup time.

b) Effect of device drivers: Fig. 4 shows the detailed breakdown of time spent during the driver loading phase. To simulate the realistic application requirement, our OSv instance uses a console device, a block device for disk access, a single NIC for performing network communication, and a random number generation device (rng-device). With our experiments, we found that the driver for the console device heavily dominates the driver loading phase. As shown in Fig. 4, the overall driver loading time is $\sim 157ms$ when the



(a) latency with VGA driver (b) latency with serial driver

Fig. 4. Driver loading latency with VGA and serial driver

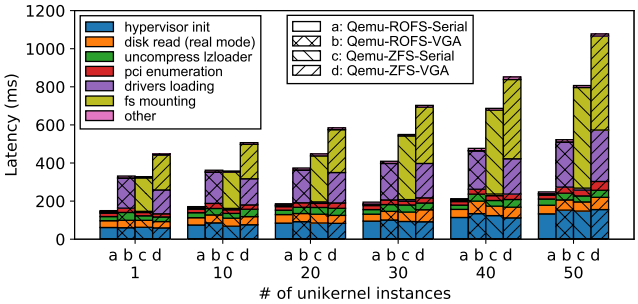


Fig. 5. Unikernel startup latency, multiple instances launched simultaneously.

VGA driver is used, but can be reduced to just $\sim 6ms$ by only using a serial console. The penalty of the VGA initialization comes from the library that copies the current screen during the initialization phase, resulting in frequent write operations, causing KVM to perform memory-mapped IO (MMIO). Furthermore, some features such as PCI enumeration (used for detecting devices on PCI Bus) and rng-device can be disabled to reduce the startup time further, as needed.

c) Effect of file systems: The file system has a drastic impact on the startup time. OSv supports an efficient read-write file system, ZFS, that takes $\sim 175ms$ for mounting (Fig. 3(a)). This order of startup penalty is not desirable for latency-sensitive stateless applications. To mitigate this issue, a read-only file system can be used, and that takes only $\sim 0.7ms$ for mounting.

*2) **Detailed Startup Breakdown: multiple instances**:* In this experiment, we analyze the behavior of multiple unikernels launched in parallel. Fig. 5 shows the breakdown of time spent during startup in the different components, for this scaling scenario where we increase the number of unikernels launched together by varying the OSv configuration. The latency in milliseconds is the $P_{95\%}$ for last instance to finish its startup operation. We breakdown this startup time across the distinct components as before, for four different file system/driver combinations.

These results are based on the QEMU hypervisor with the minimum number of QEMU features. As we can observe from Fig. 5, the hypervisor initialization time increases as the number of unikernel instances increase because of contention in process scheduling and KVM performing frequent MMIO operation. Similarly, Fig. 4 shows the driver loading time as the number of instances increases. The left y-axis represents the time taken by the driver loading phase with VGA console device. The right y-axis is the driver initialization time with a serial console device. The time taken by the VGA driver is much higher and increases with the number of OSv instances due to multiple write operations handled by KVM. On the other hand, the driver loading time with only the serial device driver is a lot less and is impacted much less as the number of instances scales up. In contrast, the other components (*e.g.,* network initialization, block device loading) do not see a significant impact as the number of unikernels scale up.

C. Containers

We evaluate Linux containers by using the *runc* container runtime engine due to its broad adoption. Similar to unikernel, each container instance requires 1 CPU and 1 GB of memory. The container process does not set up veths and networking rules while creating the network namespace and depends on tools like *ip* and CNI plugins. We used *perf* tool and added trace functionality in *runc* to measure the latency caused by each component.

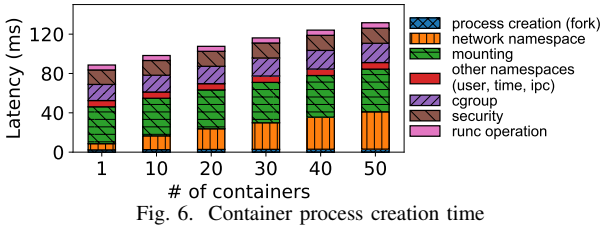
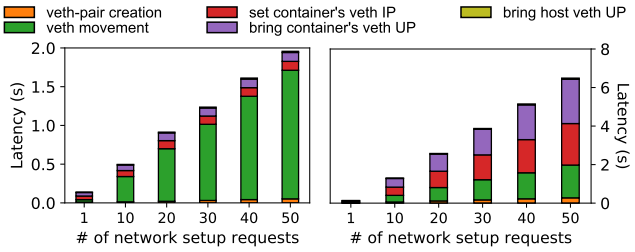


Fig. 6. Container process creation time

1) Container Process Creation: Fig. 6 shows the breakdown for container process creation. The y-axis represents the latency of different components such as creating a new process (`fork`), creating a different namespace, setting cgroup rules. As observed in Fig. 6, setting the cgroup and security rules take considerable time while creating a new container process. They do not see a significant impact as the number of containers scales up in parallel. It also shows latency contributed by different operations performed by *runc*, such as verification of config file. We separate network and mount namespace from other namespace creation to highlight their contribution. Like cgroup and security components, mounting contributes to startup time significantly, but it does not scale with the number of instances. While network namespace creation takes a comparatively shorter time for a single instance, the latency increases unusually with the number of parallel creation.

2) Container Network Setup: To understand the performance impact and evaluate the scalability of container network setup, we use the *ip* command to perform a 'network setup request', which represents a set of network setup operations as illustrated in §II-B. We measure the latency of each operation with increasing number of requests. The operations in a single request are executed in order and multiple requests can be handled in parallel to amortize latency. However, the global lock in the network setup operation will limit the parallelism of multiple requests and affect the amortization. To evaluate the impact of global lock in the case of multiple network creation requests, we set up a comparison experiment, in which all the requests are processed in sequence and no amortization can be achieved to reduce the latency.

Fig. 7(a) and 7(b) show the latency of each network setup operation under different number of network creation requests. With multiple network creation requests processed in parallel, the dominant factor in the network setup latency is the veth movement across network namespaces, which accounts for more than 90% of overall latency. Compared to handle multiple network setup requests in order, having multiple requests processed in parallel can reduce significant latency. For instance, with 50 requests, processing requests in sequence requires $6.5s$ but processing requests in parallel requires only $1.95s$. The latency is amortized by the operations except veth movement. When handling multiple veth movement operations in parallel, the latency increases linearly with the number of network setup requests, which keeps the same with the case that handling multiple veth movement operations in sequence.



(a) requests processed in parallel (b) requests processed in sequence

Fig. 7. Container network startup latency breakdown

The observation in veth movement operation demonstrates the impact of the global lock, which only allows one operation at a time and significantly affects the latency.

3) Comparison between different CNIs: Flannel, Weave, Calico, Cilium, and Kube-router are popular open-source CNI plugins and have been widely adopted in the Kubernetes ecosystem to setup pod networking [10]. These CNIs follow the general paradigm in §II-B to start up the network of a pod. However, the CNIs have different design considerations (*e.g.,* eBPF datapath) which results in differences in their network startup latency. To examine the dominant factors in the pod network setup latency, we break up the steps that CNIs follow to setup the pod network and measure the latencies generated by different steps accordingly (see Fig. 8).

The network startup latency of Flannel and Kube-router ($\sim 70ms$ in total) is the lowest among all the CNIs. Their latencies are dominated by setting up the veth-pair and a gratuitous ARP operation. The gratuitous ARP operation has to be triggered in the pod network namespace, which introduces an extra network namespace switch and $\sim 35ms$ latency. Weave suffers from a higher latency ($\sim 200ms$ in total) due to more network namespace switches involved in the pod network creation. Except the network namespace switch in veth-pair setup, extra switches are introduced by serveral operations in the pod network namespace: gratuitous ARP, renaming the veth, and appending new iptables rules to support multicast communication. When Calico starts the pod network creation, it utilizes the REST API to inspect the readiness of the backend data store (*i.e.,* kubernetes api server or etcd), which introduces extra latency. Calico relies on backend data store to maintain the network configurations, *e.g.,* IP pool, network policies. Significant latency ($\sim 80ms$) is introduced by the IP assignment operation, as Calico needs to interact with the backend data store to query for available IP resources and returns the assigned IP addresses through REST API. Cilium consumes totally $\sim 150ms$ for setting up the network. An extra network namespace switch is introduced due to rename the pod's veth. In addition, Cilium heavily relies on eBPF to construct the dataplane. During the pod network startup, it generates the eBPF code and links it into the kernel, which leads to the high latency.

4) Network startup latency with different deployment patterns: To examine the network startup latency with a large amount of pods deployed, we measure the network startup latency with two deployment patterns: **(A)** starting up one pod for measurement on a single node, with different numbers

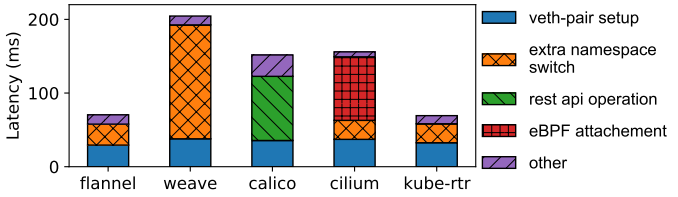


Fig. 8. Latency breakdown for a single pod network startup

of background pods already up on that node; **(B)** starting up multiple pods for measurement on a single node, with no background pods up on that node. With pattern **A** applied (Fig. 9 (a)), the network startup latency of the measurement pods does not have a significant increase even with the increasing number of background pods, which means the background pods have almost no effect on the network startup latency of newly deployed pods. With pattern **B** applied (Fig. 9 (b)), the network startup latency of Cilium increases rapidly as the number of new pods grows, which indicates worse scalability in terms of network startup latency. Cilium has conflicts in loading and attaching the eBPF programs to designated kernel hooks, which incurs more delays for a large scale deployment. Flannel and Kube-router incur lowest startup latency amongst all CNIs, which are the ideal options when users demand fast container startup. However, there does not seem to be a 'best' CNI plugin that is universally applicable. The choice on the CNI depends on a number of different factors, *e.g.,* datapath performance, network policy support [12], *etc.*

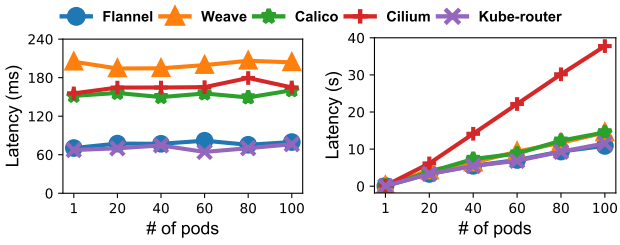
IV. Designing an Agile Compute

Our experiments highlight the components that hinder compute technologies from achieving fast startup. These components can be managed by the application developer and CSPs. This section summarizes latency introduced by such components and highlights the design choice to help achieve fast compute instantiation.

A. Unikernels

Unikernels' startup performance is obstructed by the underlying hypervisor and driver. Different features can help improve the startup performance of unikernels.

1) Hypervisors: Using generic hypervisors, *e.g.,* QEMU, that are built to host generic VMs, they often do not yield the performance expected for latency-critical compute instances. The booting protocol plays a vital role in contributing to latency. QEMU introduces significant latency because of the need to load an OSv image and decompress it. The image size directly influences disk read operation and is inversely proportional to compression speed. This trade-off leads to



(a) Pattern A (b) Pattern B

Fig. 9. Network startup latency with different deployment patterns

similar overhead and does not mitigate the core issue. This issue is resolved in hypervisors like firecracker that use the latest booting protocol that directly executes the kernel binary.

2) Device Driver: Device drivers also introduce delays in startup time. An unawareness of underlying devices can degrade the overall startup performance of the unikernel. An example is highlighted in our experiment where the VGA driver is used for console output, resulting in a high penalty. Short-lived serverless applications are expected to incoming requests and do not log information on the console. These applications do not fully require the VGA driver, making the serial console a good candidate for such scenarios.

B. Containers

The network setup phase significantly affects containers' startup performance. The problem persists among different CNIs and further increases with scaling-up due to their complex offerings. The network setup latency can be relieved by maintaining the pool of prepared network namespaces and attach to a container rather than reactively creating it when the container is requested.

V. Related Work

Several works have investigated the startup performance of unikernels and containers across different parameters such as application throughput, memory footprint, TCP throughput, overall latency. Goethals *et al.* [13] compares the application-layer performance of OSv unikernel and containers for microservices applications on parameters such as response latency, request processing rate, and memory footprint. Xavier *et al.* [14] assessed the performance of containers and unikernels in terms of startup latency. Kurek [15] also compared the performance of Docker with IncludeOS unikernel on attributes such as TCP throughput and ping latency. However, none of them focused on the detailed breakdown of startup time which is crucial for further optimization. Oakes *et al.*, Thomas *et al.*, and Mohan *et al.* [4], [5], [9] identified network setup as a critical bottleneck in containers but they do not take into account CNI plugins which are often coupled with container orchestration software and introduce additional overheads. Gadepalli *et al.* [16] have studied WebAssembly-based serverless functions which have a more lightweight runtime compared to containers, highly desirable at the edge. They optimize the Wasm-based function instantiation by avoiding heavyweight linking and loading during startup, which can achieve μ-second level startup latency. They are currently limited to single serverless function worker nodes.

VI. Conclusions

This paper provides a qualitative and quantitative analysis between containers and unikernels. A unikernel's startup time is strongly influenced by the underlying hypervisor, file system, and drivers. With a careful choice of components, especially device drivers, a significant part of that latency can be reduced. On the other hand, a container's startup time is significantly affected by file system mounting. Further, the CNI plugins introduce additional latency. Additionally, we find that network initialization drastically increases startup latency when the number of containers initiated increases. In contrast, scaling up the number of unikernels initiated is better, as long as the devices are carefully selected. Based on our experimental observations, unikernels that utilize a suitable hypervisor (*e.g.,* one that directly executes the kernel binary) with carefully selected support for device drivers, or containers that can mitigate the network setup latency through pre-configured network namespace pools may be ideal for building an agile computing platform with fast startup.

Acknowledgment

We thank the US NSF for their generous support of this work through grant CNS-1763929. This research was also sponsored by the OUSD(R&E)/RT&L and was accomplished under Cooperative Agreement Number W911NF-20-2-0267. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL and OUSD(R&E)/RT&L or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. We also thank the anonymous reviewers for their valuable feedback.

References

[1] Madhavapeddy et. al., "Unikernels: Library operating systems for the cloud," *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 461–472, 2013.

[2] Engler et. al, "Exokernel: An operating system architecture for application-level resource management," *ACM SIGOPS Operating Systems Review*, vol. 29, pp. 251–266, 1995.

[3] Zhang et. al., "Flurries: Countless fine-grained nfs for flexible per-flow customization," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016, pp. 3–17.

[4] Oakes et. al, "SOCK: Rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 57–70.

[5] Mohan et. al., "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[6] Kivity et. al., "OSv—optimizing the operating system for virtual machines," in *USENIX Annual Technical Conference*, 2014.

[7] "Kubernetes." [Online]. Available: https://kubernetes.io/

[8] Baumann et. al., "A fork () in the road," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 14–22.

[9] T. et. al., "Particle: ephemeral endpoints for serverless networking," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[10] Qi et. al., "Understanding container network interface plugins: design considerations and performance," in *IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2020.

[11] Ricci et. al, "Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications," *the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.

[12] Q. et. al., "Assessing container network interface plugins: Functionality, performance, and scalability," *IEEE Transactions on Network and Service Management*, 2020.

[13] Goethals et. al., "Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications," in *IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. IEEE, 2018.

[14] Xavier et. al., "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016.

[15] T. Kurek, "Unikernel network functions: A journey beyond the containers," *IEEE Communications Magazine*, vol. 57, pp. 15–19, 2019.

[16] G. et. al., "Sledge: a serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 265–279.