2022 IEEE International Conference on Edge Computing and Communications (EDGE) | 978-1-6654-8140-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/EDGE55608.2022.00020

A Functional and Performance Benchmark of Lightweight Virtualization Platforms for Edge Computing

Tom Goethals
Department of Information Technology
Ghent University - imec, IDLab
Gent, Belgium
ORCID 0000-0002-1332-2290

Merlijn Sebrechts
Department of Information Technology
Ghent University - imec, IDLab
Gent, Belgium
ORCID 0000-0002-4093-7338

Mays Al-Naday
School of CSEE
University of Essex
Colchester, England
ORCID 0000-0002-2439-5620

Bruno Volckaert
Department of Information Technology
Ghent University - imec, IDLab
Gent, Belgium
ORCID 0000-0003-0575-5894

Filip De Turck
Department of Information Technology
Ghent University - imec, IDLab
Gent, Belgium
ORCID 0000-0003-4824-1199

***Abstract*—The recent rise of edge computing and FaaS triggered a revolution in the field of software virtualization, improving performance and security. This paper benchmarks various lightweight virtualization platforms, such as microVMs and containers, in the context of edge microservices. Factors taken into account include toolchain maturity, networking capabilities, boot time, resource use, microservice performance, and ARM architecture readiness. We present a functional comparison and benchmarks on both a Raspberry Pi 4 and an x86-64 platform. The results indicate standard Docker containers offer reliable performance and low memory use, while microVM-based solutions such as Firecracker are more isolated. Moreover, OSv unikernels have extremely low boot times and significantly better performance than Docker containers. Finally, while gVisor offers improved security and excellent compatibility, its performance is only 10% of default Docker performance.**

***Index Terms*—containers, unikernels, microVM, microservices, virtualization, edge computing**

I. Introduction

The rise of edge computing and FaaS has triggered a revolution in the field of virtualization, in an attempt to offer improved performance and useful features for a variety of use cases. While virtual machines (VMs) and containers used to be the only available technologies, recent developments have resulted in various types of microVMs, isolation layers, and mixed technology that borrows features from both containers and VMs.

The work on microVMs [1] attempts to minimize the overhead of classical VMs, which contain an entire operating system merely to run a single application. While some approaches pre-build a minimal Linux kernel tailored to existing software, others offer their own API, often in C++, and attempt to integrate programs with the kernel itself (i.e. unikernels [2]).

Container technology has received criticism for not being sufficiently secure [3], resulting in initiatives to improve the isolation of containers from the host and other containers. Some solutions offer an alternate container runtime (the container equivalent of a hypervisor), which is easily integrated into existing engines (e.g. Docker [4], containerd [5]), while other solutions use a microVM as a security shell around containers.

This paper compares various lightweight virtualization platforms in terms of functionality and performance in the context of edge microservices. Factors taken into account include: toolchain maturity, networking capabilities, booting times, resource use, and microservice performance, as well as ARM architecture readiness. Docker containers, being a mature technology, are used as a baseline for performance and features. Concretely, the contributions of this paper are:

1) A cursory explanation of the differences between containers, microVM platforms and microVM kernels
2) A feature comparison of representative platforms of each class, including platform requirements, runtime dependencies, toolchain maturity, and high-level performance indicators.
3) Examining the ARM(64) readiness of each platform.
4) A benchmark for both ARM64 and x86-64, showing how each platform performs in the context of REST webservices, as well as service boot times and their impact on system resources.

The results indicate that while Docker containers offer reliable performance and compatibility with various architectures, microVMs generally boot faster, and OSv [6] unikernels outperform containers in single-core applications. The added security of the runsc runtime (gVisor [7]), however, results in significantly lower REST service performance and network throughput.

2767-9918/22/$31.00 ©2022 IEEE
DOI 10.1109/EDGE55608.2022.00020

The rest of this paper is organized as follows: Section II presents existing research related to various lightweight virtualization approaches. Section III describes the different types of virtualization considered for evaluation, while Section IV discusses concrete platforms and presents a functional comparison. In Section V, the performance benchmark setup and methodology are presented, while the results are presented in Section VI, and discussed in Section VII. Finally, Section VIII draws high level conclusions from the paper.

II. Related Work

Wang et al. [1] examine the performance of QEMU, Firecracker, and Docker in terms of boot times, disk I/O, network throughput and event processing. This provides raw performance numbers for basic indicators measured on the local machine. A similar study by Li et al. [8] explores MySQL and NGINX performance, showing that Kata Containers are generally on par with Docker, but gVisor is significantly slower.

Mavridis et al. [9] compare a number of container engines, including microVM- and unikernel-based runtimes, providing a taxonomy and categorization of the various technologies and layers involved. The result is a study of the interoperability of technologies, along with performance indications for cloud-oriented functionality. Wang et al. [10] provide a similar cloud-oriented study, focused on runC, gVisor and Kata Containers.

Another study [11] provides an in-depth examination of the continuum of kernel space versus user space functionality of various runtimes, focusing on the specific cases of gVisor and Firecracker compared to containers. Meanwhile, a comparison of unikernels and containers for industrial applications is presented by Chen et al. [12].

This overview indicates that cloud applications, along with the categorization and low-level technological mapping of microVMs and container engines, are well studied. As such, the focus of this paper is to benchmark a number of microVM and container platforms in terms of practical functionality specifically for edge computing. This includes a suitability comparison and performance indications for microservices, and an assessment of ARM architecture readiness.

III. Lightweight virtualization approaches

This section discusses various types of lightweight alternatives to regular virtual machines, visualized in Figure 1, and their suitability for microservices on edge devices.

A. Containers

Containers share the kernel of the host system and are isolated using Linux features such as namespaces and cgroups. Generally, “chroot” is used to alter the root filesystem of the process, while namespaces determine what types of hardware devices (e.g. network interfaces) are available to it. Soft and hard resource limits are imposed using cgroups.

This approach results in smaller image size and memory usage compared to VMs. Due to the shared kernel, containers can also directly use specialized host hardware such as GPUs, assuming that they contain the necessary drivers. Finally, containers have little to no overhead in terms of networking and processing, putting them on equal ground with standard Linux processes. Sharing a kernel, however, leaves both the host and containers vulnerable to exploits, and more recent initiatives seek to mitigate this risk [13].

The most popular container engine is Docker, although the underlying container engine containerd is a useful lower-level alternative.

B. MicroVM runtimes

MicroVM runtimes, such as Firecracker [14], are tiny hypervisors that aim to reduce the footprint and improve security of the virtualization layer. They lower the attack surface and overhead of a VM by implementing only minimal virtual devices and excluding all non-essential functionality. They typically offer performance comparable to native processes by using paravirtualized devices such as virtio [15] and hardware virtualization instruction sets such as Intel VT-x and AMD-v [16]. Guest operating systems often require small changes in order to run in such a minimal environment.

C. Specialized VM guests

To further reduce the overhead of virtualization, a number of specialized VM guests have emerged. These fall into two categories. The first category runs existing software on a highly minified Linux kernel such as the default Firecracker kernel [17]. While this ensures near perfect compatibility with most Linux software, there is a limit to how small they can be made. Custom kernels with reduced API support are usually supported, however, which can result in even smaller size and attack surface.

The second category, known as unikernels [2], integrate an executable directly into the kernel, resulting in a monolithic process that runs entirely in kernel mode. This allows improved performance by removing context switches, smaller images, and reduced attack surface [18]. The smallest, most optimized unikernel platforms, such as UniK [19], provide custom system interfaces and dedicated programming languages which produce unikernels of only hundreds of KiB for basic web functions such as a DNS server [20]. In contrast, POSIX-compatible unikernel platforms, such as OSv [6], produce significantly larger images, but can integrate existing POSIX executables such as Linux software [21]. Nevertheless, even these images are significantly smaller than minified Linux kernels.

D. Hybrid approaches

Kata Containers [22] runs each container inside a microVM, allowing it to fully isolate all containers from each other and from the host. This approach is rather resource intensive for edge devices, however, as it introduces the additional memory and processing overhead of a microVM [23].

gVisor [7] is an alternate container runtime compatible with Docker and containerd, which offers a custom implementation of the Linux system call interface. This improves isolation and

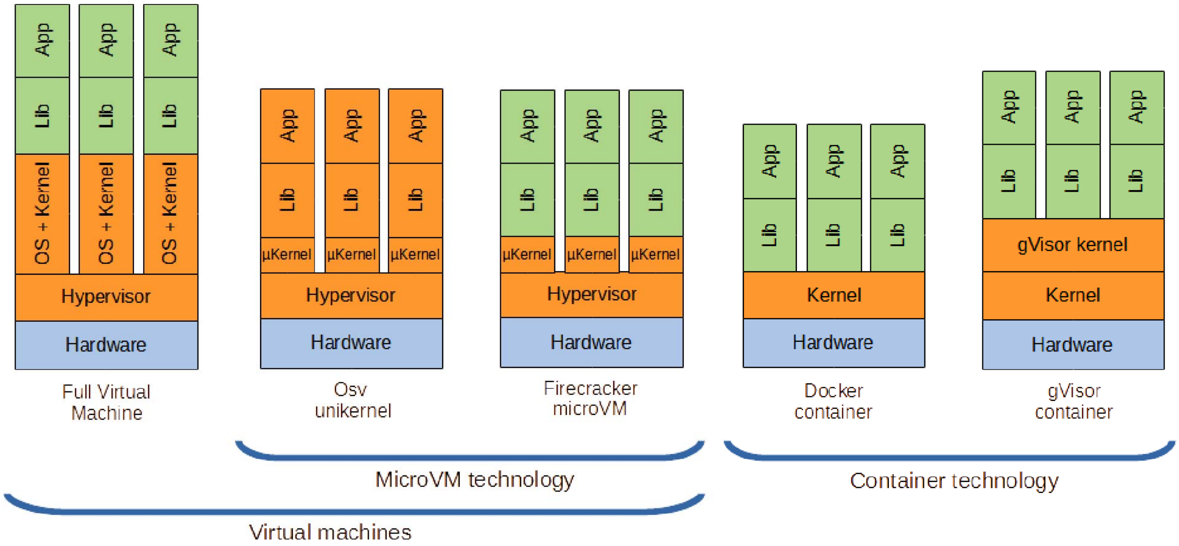

Fig. 1. Comparison of system architecture of various lightweight virtualization methods. Orange parts are kernel space, while green parts are user space.

security by emulating the host kernel in user space, rather than sharing it directly. Theoretically, this type of solution may add less processing and memory overhead than a microVM layer, although this depends on the implementation.

WebAssembly (WASM), combined with WebAssembly System Interface (WASI) [24] forms another virtualization solution. WebAssembly itself originates from the necessity to accelerate JavaScript in web browsers, but evolved into a highly efficient form of bytecode, now often used in combination with Rust [25] as a programming language. WASI forms a system interface around WASM, much like gVisor does for containers. As WASI does not yet provide any networking capabilities[1], however, it is not yet useful in the context of edge microservices.

IV. Candidate Platforms

As the focus of this paper is microservices at the edge, all platforms must necessarily have fully functional networking capabilities, and ARM platform compatibility is required for many low-resource devices. Additionally, only platforms capable of running compiled Go code are considered, in order to compare the exact same service on the selected platforms. This requirement is introduced because some microVM and unikernel platforms only offer a highly modified C++ system API, often limited in capabilities, making a direct comparison to POSIX compatible alternatives impossible. Furthermore, some POSIX compatible platforms are not stable enough to run software written in various programming languages; as Golang is compiled into a standalone executable, being able to run it is used as an implicit minimum measure of platform maturity. Note that not all platforms are capable of both creating images and running them; while Docker can do both, OSv unikernel images are evaluated on KVM. Inversely, a default kernel is used to evaluate the compiled filesystem images on Firecracker.

Docker (v20.10.3) is a popular container engine which has contributed to several container standards. The basis of Docker containers consists of layered images, declaratively defined by Dockerfiles, which can be used recursively, adding additional files and libraries with each layer. While Docker itself manages images and some aspects of Linux namespaces and cgroups, it relies on containerd for lower-level container operations. Docker itself is fully ARM compatible, and is capable of running containers in the edge by itself or through orchestrators such as Kubernetes [26].

OSv (v0.56 branch) is a POSIX-compatible unikernel platform which aims for maximum compatibility with existing software, and can run on a variety of hypervisors. Its main focus is on speed and security through efficient drivers and kernel design, although low memory overhead has become an important secondary focus. It offers two methods of creating a unikernel; either by compiling a program directly into a custom kernel and selecting OSv libraries through configuration, or by using Capstanfiles, an analog of Dockerfiles which allow a layered compilation process. Both processes are relatively straightforward, but only the first one supports ARM cross-compilation. Although there have been efforts to integrate unikernels into orchestrators [27], there is no default OCI/CNI-compatible solution to orchestrate both unikernels and containers.

Firecracker (v0.25.2) is still in early development, and although v1.0.0 was recently released, it breaks compatibility with firectl [28], which is used by various image building applications for Firecracker. Based on KVM and originally developed for the cloud, Firecracker is essentially a specialized Virtual Machine Monitor for microVMs. It offers a default kernel, and third-party scripts can build Firecracker filesystems

[1] https://github.com/WebAssembly/WASI/issues/370

from compiled applications. Control of Firecracker happens either through the API socket, or through firectl, although networking capabilities are limited to using existing TAP devices on the host, and the integration of these devices into a larger scale network (i.e. what CNI plugins do for Kubernetes) is left as an exercise to the user. As Firecracker is mainly a microVM runtime which happens to provide a default kernel, it is possible to actually run OSv-created unikernels on Firecracker. Finally, Firecracker is fully operational on ARM devices.

gVisor (v20220103.0), which does not have an official release yet, is an alternative container runtime which replaces runC [29] for various container engines. Because it transparently replaces the low-level runC runtime, it inherits the higher-level advantages and disadvantages of whichever container engine it is used with. Although, it provides better security by emulating the host kernel rather than directly exposing it to containers. While it implements a large part of the Linux system call interface, the implementation is not yet complete and options such as using the host network namespace are not (yet) possible.

An overview of the features and properties of the presented virtualization platforms is shown in Table I.

V. Benchmarking Setup

This section discusses the physical test setup and methodology used to obtain the results, and the Go REST service used for the evaluations is described in detail.

A. Test Machine

For x86-64, the benchmarks are run on an Intel Core i5-3570k processor, with virtualization extensions enabled. The machine has a total of 8GiB RAM and a 120GB ADATA S510 SSD. Containers are run on Docker (v20.10.3) and Ubuntu 20.04, while Firecracker microVMs and OSv unikernels use the default version of KVM in Ubuntu. Docker containers are allowed to use the host network; while for gVisor, port forwarding is used instead, as it does not support using the namespace of the host network. For microVMs, iptables rules are set to forward host port 8080 to the VMs.

For ARM, a Raspberry Pi 4 with 4GiB RAM is used, with a different operating system per platform for compatibility reasons. Ubuntu Desktop 21.10 with KVM is used for OSv, Ubuntu Server 20.04 for gVisor, and Raspberry Pi OS (kernel version 5.10) for the other platforms. Other aspects of the test setup are identical to the x86-64 evaluations.

For single-core benchmarks, virtual machines and containers are limited to one CPU core by passing the appropriate flags to their runtimes. For multi-core benchmarks, all instances are given four cores. The test machine is only used as a server; all performance monitoring apart from memory status and boot times is run on a separate machine (the "client"). The client is connected to the test machine in a LAN, to avoid result collection from interfering with the performance of the benchmarked platforms, and it performs service requests using Apache JMeter [30].

The code of the REST service and walkthroughs for building and executing Go web services on the various platforms are made available on GitHub[2].

B. REST service

For the benchmarks, a simple REST service is created that contains an in-memory array of to-do items. The service supports the following methods:

- GET /todos: lists all to-do items
- GET /todos/get/{id}: gets the to-do with the specified id and returns it as JSON
- POST /todos/create/: creates a new to-do item from the posted JSON string
- POST /sort: sorts a posted array of numbers using Bubble Sort
- GET /largeData: simulates a larger response by returning a pre-built JSON string of 100KiB

While these methods are straightforward, they are intended to simulate requests that are typical for edge devices, e.g. sensor values or short status messages. The largeData method is intended to simulate communication with nodes further away from the edge, which generally consists of larger chunks of data.

The web service is implemented in Go 1.17.4, and uses the Gorilla Toolkit mux (v1.6.2) [31] for its simplicity. Multithreading code is not required; Go automatically creates a number of threads befitting its hardware environment.

C. REST Request stress-test

This test uses the get and create to-do methods to study the ability of the chosen platforms to handle a large number of small requests. Both single- and multi-core benchmarks are run to determine how well each platform scales. To ensure a constant maximum load, the client machine uses 40 threads in single-core benchmarks, and 100 threads in multi-core benchmarks. Although exact scaling requires 160 threads, 100 is sufficient for the fastest platform, and a higher number of threads merely results in higher latency for waiting requests. The number of requests per thread is 20000 for get requests, and 5000 (ARM) or 10000 (x86-64) for create requests.

D. Processing jobs

The sort method is used to determine the ability of each platform to perform tasks involving a high CPU load and intensive memory use. An array of 20000 sequential numbers in descending order is posted to the web service, sorted using Bubble Sort, and returned in ascending order. The conversion to and from JSON strings represents a non-trivial overhead, but as these operations are CPU- and memory-oriented, they are an acceptable part of the test. This type of task can be compared to receiving updates for small AI models and integrating them, or processing small batches of sensor data.

[2] https://github.com/togoetha/lw-edge-virtualization

TABLE I
OVERVIEW OF FEATURES AND PROPERTIES BY VIRTUALIZATION PLATFORM. *RATING DERIVED FROM BENCHMARKS IN SECTION VI, + AND - TO BE INTERPRETED AS STRONG AND WEAK POINTS

	Docker (containers)	OSv	Firecracker	gVisor
Main focus	Flexibility	Speed	Speed	Security
Features	Compatibility, widely adopted	Security, flexibility	Low resources, compatibility	Compatibility, transparency
Requirements	Linux (kernel)	Hypervisor	Linux + KVM	Linux + container engine
Image creation	Docker	OSv	default/third party	Docker
Runtime(s)	Docker, containerd	Xen, KVM, Firecracker	Firecracker (KVM)	Docker, containerd
Toolchain	++	+/-	+/-	++
Security	+/-	++	++	+
Networking	++	+	- -	+
Resource use*	++	-	-	+/-
Speed*	+	++	+/-	- -
Start time*	-	++	+/-	-
ARM support	Yes	Functional	Yes	Yes

E. Network throughput

The largeData method is used to determine the ability of each platform to saturate a 1Gbps LAN connection. Practical situations with this kind of behavior involve short periods of peak data transfer, e.g. sending aggregate data to a server for further processing or sending batches of (meta)data to other nodes.

F. Gathered metrics

All network metrics are gathered using Apache JMeter, including requests/s, request latency, and throughput. Memory use is determined by measuring the reserved memory for each process directly involved in running a VM or container. For example, for Docker containers both the container process and containerd-shim are considered, but not Docker or containerd themselves, while for Firecracker only the firecracker process running the VM is taken into account. The measurements are gathered after the service is started, but before any service calls are performed. Booting (or starting) times are determined differently depending on the platform used. For containers, the difference in system time is measured between starting the container and entering the main method of the containerized process. For microVMs, booting times are printed by the runtime during the booting process.

VI. RESULTS

In this section the benchmarks results are presented. In addition to request throughput capacity, multi-core scaling, memory footprint, response latency, and boot times will be discussed. As both x86-64 and ARM results are presented, the results are color coded, with a blue/red scheme for ARM and beige/gray for x86-64.

Because the subject of this paper is lightweight virtualization for edge devices, the rest of this section focuses on ARM results; comparing them to x86-64 where useful.

A. Single-threaded REST Performance

Fig. 2 shows the to-do get and create performance of each virtualization platform on x86-64, using a single core (and single thread) for all processing. OSv is 50% faster than default Docker, while Firecracker with its default kernel is about 20% slower. These numbers indicate that microVM performance can vary greatly depending on kernel design and virtual drivers. Containers using gVisor are only 25% as fast as those using runC, showing that the emulated software kernel of gVisor presents a significant overhead for the benefit of added security, by isolation.

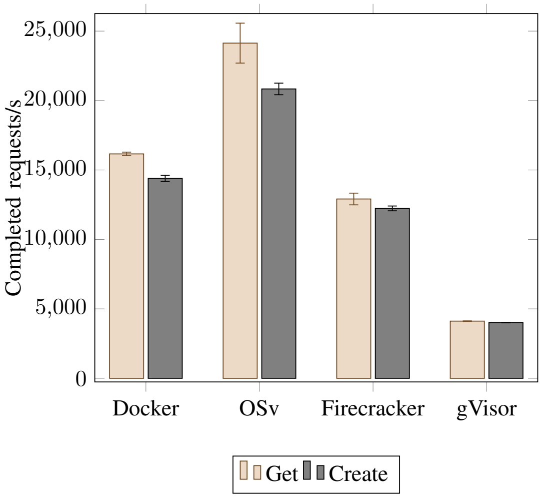

Fig. 2. Request processing performance in different virtualization solutions on **x86-64**. *Higher is better.* Unikernels show considerable performance improvements while gVisor incurs a significant performance penalty.

Fig. 3 shows the results of to-do get and create requests on ARM. OSv unikernels offer the fastest processing, being 16% faster than Docker containers. Despite running a similar microVM kernel, Firecracker only manages 72% of the performance of OSv unikernels on KVM. Considering the added protection of running in a VM, the 15% performance hit of Firecracker compared to Docker containers may be acceptable in some scenarios. gVisor is the slowest alternative, offering only around 10% of the performance of Docker containers. The comparison with the x86-64 results is striking; the gap between Docker and OSv is significantly smaller due to the KVM vhost using a relatively high amount of CPU, thus starving the unikernels. However, this is not a common

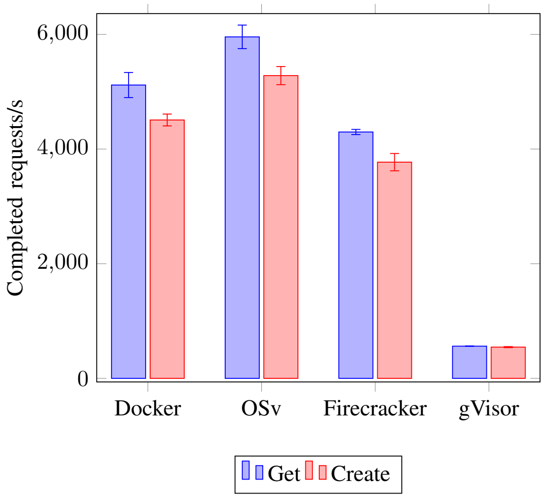

Fig. 3. Request processing performance in different virtualization solutions on **ARM**. *Higher is better.* The performance penalty of gVisor is even more pronounced on ARM systems. Unikernels still show a performance improvement over other guest approaches.

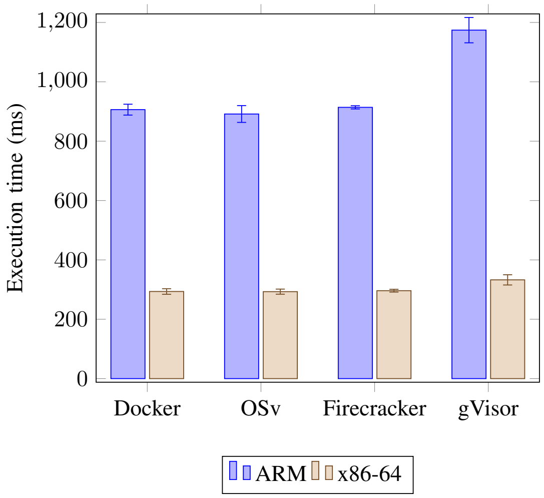

Fig. 4. Bubble sort execution time in different virtualization solutions on ARM and X86-64. *Lower is better.* The results are similar, showing all virtualisation solutions handle CPU and memory instructions with native efficiency.

property of microVMs, as Firecracker performs slightly better compared to Docker, relative to x86-64. gVisor takes the biggest performance hit, as it is entirely software-based.

Fig. 4 shows the results for processing job requests. All platforms offer similar performance, indicating that all platforms handle CPU and memory instructions with native efficiency. However, there are some minute differences; Firecracker is around 1% slower than Docker, likely due to the combination of its microkernel and KVM. OSv, on the other hand, is 1% faster than Docker, showing a high level of kernel optimization. gVisor however, is nearly 20% slower than the alternatives, indicating significantly slower memory access due to the runsc runtime. Compared to x86-64, most platforms are around 32% as fast, although gVisor only manages 28%.

B. Multi-threaded REST Performance

When handling large numbers of requests, it may be useful to spawn multiple threads in a single process to avoid the overhead of starting multiple processes. Fig. 5 shows the relative performance of the evaluated platforms when running a REST service provided with four cores (and four threads). Docker performance scales perfectly with the number of cores, but the VM-based platforms are significantly less suitable for multi-core scaling. Although Firecracker manages 240% to 250% of the performance of a single core, it fails to make use of all the processing power provided. OSv unikernels suffer from a significant overhead, scaling at around 160% to 185% on KVM, although some of this can be attributed to the KVM vhost. Paradoxically, gVisor, and Docker to a lesser extent, scale superlinearly with the number of available cores, indicating that some static CPU overhead is amortized over all available cores.

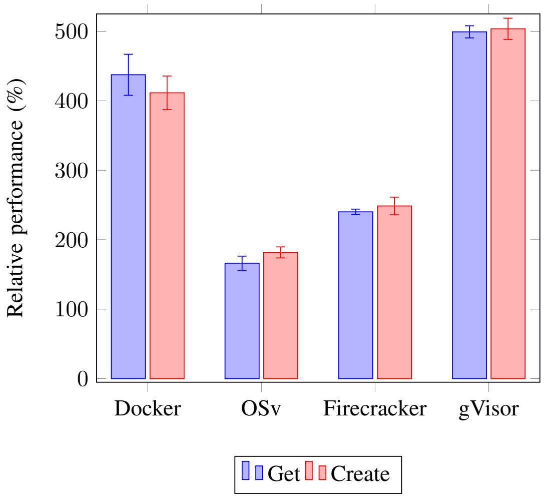

Fig. 5. Relative performance of multi-threaded vs single-threaded request processing on ARM. *Higher is better.* While Docker and gVisor scale superlinearly, implying amortized overhead, OSv and Firecracker are not able to fully utilise additional cores.

C. Latency and Throughput

Because edge microservices are often used to provide a smoother user experience than with cloud-based solutions, a reliably low service response time is important. Fig. 6 shows the range of response times for to-do create (POST) requests for the benchmarked platforms, from the 2nd to 98th percentiles. Note that these response times are influenced by the throughput from Fig. 3, so inflated numbers should be expected for slower platforms. While VM-based platforms have significantly higher median response times than Docker containers, their maximum response times are significantly more stable, staying below 20ms. gVisor exhibits the same

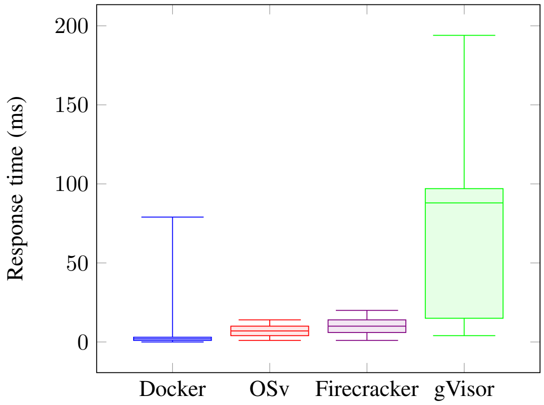

Fig. 6. Response latency variation of different virtualization solutions on ARM. *Lower is better.* While Docker has significantly lower average latency, its maximum latency varies significantly. Virtualization-based platforms have significantly more stable maximum response latency, even though their average latency is higher.

high maximum response times as Docker, making containers less suitable for applications that need guaranteed real-time responsiveness. Throughput-corrected latency L_C can be calculated, using $L_C = L_P(R_P/R_D)$, where L_P is median platform latency, and R_P and R_D are platform requests/s and Docker requests/s, indicating median VMs latencies 3 times higher than those of Docker, and a median gVisor latency 4 times higher.

Fig. 7 shows the traffic generated by largeData requests (100KB) as the service runs on a single core. Docker and Firecracker almost saturate the gigabit connection, but OSv is limited by the KVM vhost and only manages around 720Mbps. gVisor performance is in line with the other benchmarks, at around 10% of Docker performance. Interestingly, OSv has the lowest CPU use while fully saturating a gigabit connection on x86-64, indicating that the virtualization capabilities of the ARM platform, or at least the Raspberry Pi 4, are not quite suitable for large amounts of virtual network traffic.

D. Memory Footprint

To accurately compare memory use, support processes directly linked to running the service instance are included in the measurements. For Docker, the containerd-shim process is added, while for gVisor, several spawned “exe” processes are included. VM-based platforms do not spawn additional processes, but the full memory requirement of the VM processes is considered, measured using the lowest possible memory allocation that allowed them to boot and idle.

Figure 8 shows the memory requirements for each platform on both ARM and x86-64 are similar, with Docker requiring slighty more memory on ARM and the other platforms using slightly more memory on x86-64.

While Docker containers have a small overhead of around 7MiB over a native process, gVisor requires significantly more memory for the runsc runtime, and the memory cost of kernel isolation and added security is around 50MiB.

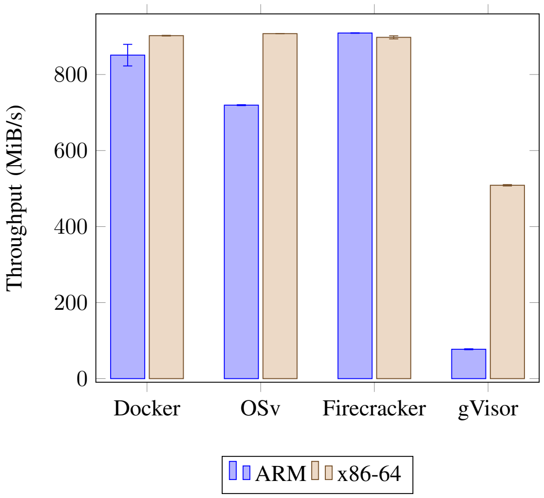

Fig. 7. Sustained throughput for 100KiB response payload (ARM/x86-64). *Higher is better.* gVisor performs considerably worse, showing the overhead of user-space syscall emulation.

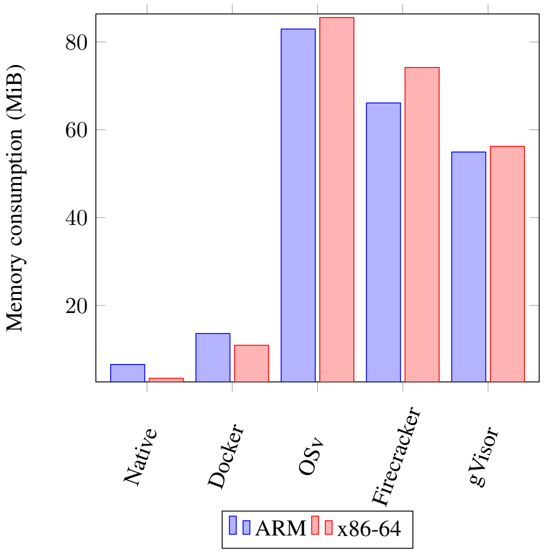

Fig. 8. RAM usage of a basic service on ARM and x86-64. *Lower is better.* Docker shows incredible memory savings compared to both gVisor and VM-based platforms.

VM-based platforms are expected to use more memory for their kernels and drivers, and the difference between VMs and Docker containers is 55-75MiB, with OSv at the high end. While these measurements merely represent the idle state of the web service, such differences are significant on edge devices where free memory after boot is sometimes limited to hundreds of MiB.

E. Boot Times

The boot times (or start times) of the various platforms on ARM are shown in Fig. 9. All alternatives can be made to boot in around a second or less, although the start times

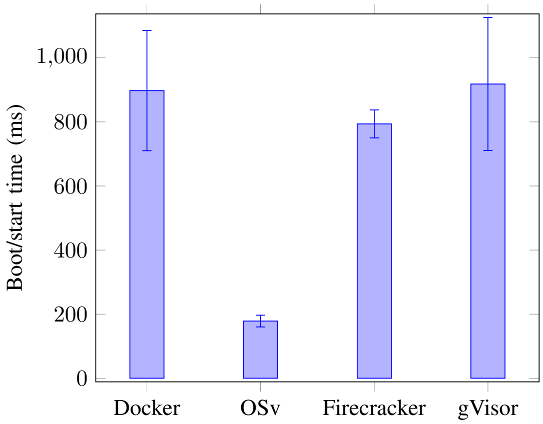

Fig. 9. Boot and start times of the service for the evaluated virtualization platforms on ARM. *Lower is better.* OSv boots in just 200ms, while Firecracker performs slightly better than Docker.

of container-based platforms vary significantly. Firecracker microVMs boot slightly faster than container-based alternatives, and OSv unikernels generally boot extremely fast in just 200ms, making them a highly suitable option for real-time demand-driven deployment of microservices.

VII. Discussion

Each platform has its specific strengths and weaknesses, leading to suitable use cases. Docker containers require the least resources, while offering good performance and excellent multi-core scaling, but they exhibit unreliable latency and have potential security issues. As such, they are useful for any services that rely on host system compatibility, and which process data without user interaction, for example containerized AI models that run on specialized host hardware. OSv unikernels can run on various hypervisors, and offer better performance, better security, faster booting times, and more stable latency than Docker containers. On the other hand, they have the highest memory overhead and do not scale well on multiple cores. While the memory overhead does not encourage vertical scaling to handle load spikes, single-core unikernels can be deployed strategically and dynamically on edge gateways to handle spikes of QoS sensitive requests. Firecracker, with its default kernel, is slightly less performant than Docker containers, scales significantly better than OSv, and may justify its memory overhead through compatibility with existing software and ease of use. As such, it is suitable for services that require strong Linux compatibility and isolation, making it a prime platform to virtualize existing, privacy-sensitive IoT edge services. gVisor has the latency disadvantage of Docker combined with a high memory overhead and low performance, but may be a feasible choice if a service architecture demands the use of containers and good security. For example, a containerized, privacy-sensitive AI model which requires host hardware. Finally, the sub-optimal multi-core scaling of VM-based platforms is not limiting, as most edge services are designed for lightweight operation, and are not generally assigned, much less guaranteed, all of the CPU cores on a device.

VIII. Conclusion

Novel virtualization methods in the edge are introduced, and various applicable types of virtualization are discussed, mainly focusing on containers, microVMs, and hybrid solutions. Candidate platforms for a benchmark of virtualization in the edge are presented, with Docker for containers, OSv for unikernels, Firecracker for microVMs in general, and gVisor as a hybrid solution for containers. The features and functional advantages of each platform are discussed and compared.

A test setup is provided, together with a basic REST service and a number of evaluation scenarios comparable to typical edge communication and workloads. Performance evaluations and metric gathering are performed to supplement the functional comparison.

The results show that each platform has its own strengths and weaknesses, which are discussed along with example scenarios and applications.

References

[1] Z. Wang, “Can “micro vm” become the next generation computing platform?: Performance comparison between light weight virtual machine, container, and traditional virtual machine,” in *2021 IEEE International Conference on Computer Science, Artificial Intelligence and Electronic Engineering (CSAIEE)*, 2021, pp. 29–34.

[2] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, “Unikernels: Library operating systems for the cloud,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013.

[3] K. Brady, S. Moon, T. Nguyen, and J. Coffman, “Docker container security in cloud computing,” in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2020, pp. 0975–0980.

[4] “Docker: Empowering app development for developers,” Feb. 2022. [Online]. Available: https://www.docker.com/

[5] “containerd - an industry-standard container runtime with an emphasis on simplicity, robustness and portability,” Feb. 2022. [Online]. Available: https://containerd.io/

[6] “Osv, a new operating system for the cloud,” Feb. 2022. [Online]. Available: https://github.com/cloudius-systems/osv

[7] “gvisor application kernel for containers,” Feb. 2022. [Online]. Available: https://gvisor.dev/

[8] G. Li, K. Takahashi, K. Ichikawa, H. Iida, P. Thiengburanathum, and P. Phannachitta, “Comparative performance study of lightweight hypervisors used in container environment.” in *CLOSER*, 2021, pp. 215–223.

[9] I. Mavridis and H. Karatza, “Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud,” *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a, p. e6365. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6365

[10] X. Wang, J. Du, and H. Liu, “Performance and isolation analysis of RunC, gVisor and kata containers runtimes,” *Cluster Computing*, jan 2022.

[11] Anjali, T. Caraza-Harter, and M. M. Swift, “Blending containers and virtual machines: A study of firecracker and gvisor,” in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 101–113. [Online]. Available: https://doi.org/10.1145/3381052.3381315

[12] S. Chen and M. Zhou, “Evolving container to unikernel for edge computing and applications in process industry,” *Processes*, vol. 9, no. 2, p. 351, 2021.

[13] S. Sultan, I. Ahmad, and T. Dimitriou, “Container security: Issues, challenges, and the road ahead,” *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[14] "Secure and fast microvms for serverless computing," Feb. 2022. [Online]. Available: https://firecracker-microvm.github.io/

[15] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[16] H. Lee, "Virtualization basics: Understanding techniques and fundamentals," *School of Informatics and Computing Indiana University*, vol. 815, 2014.

[17] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[18] J. Talbot, P. Pikula, C. Sweetmore, S. Rowe, H. Hindy, C. Tachtatzis, R. Atkinson, and X. Bellekens, "A security perspective on unikernels," in *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. IEEE, 2020, pp. 1–7.

[19] "Unik, a platform for automating unikernel & microvm compilation and deployment," Feb. 2022. [Online]. Available: https://github.com/solo-io/unik

[20] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*. IEEE, 2015, pp. 250–257.

[21] B. Xavier, T. Ferreto, and L. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 277–280.

[22] "Kata containers - the speed of containers, the security of vms," Feb. 2022. [Online]. Available: https://katacontainers.io/

[23] R. Kumar and B. Thangaraju, "Performance analysis between runc and kata container runtime," in *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2020, pp. 1–4.

[24] "Webassembly/wasi: Webassembly system interface," Feb. 2022. [Online]. Available: https://github.com/WebAssembly/WASI

[25] "Rust - a language empowering everyone to build reliable and efficient software." Feb. 2022. [Online]. Available: https://www.rust-lang.org/

[26] "kubernetes - production-grade container orchestration," Feb. 2022. [Online]. Available: https://kubernetes.io/

[27] H. Lefeuvre, G. Gain, D. Dinca, A. Jung, S. Kuenzer, V.-A. Badoiu, R. Deaconescu, L. Mathy, C. Raiciu, P. Olivier *et al.*, "Unikraft and the coming of age of unikernels," *login; The Usenix Magazine*, 2021.

[28] "firectl is a command-line tool to run firecracker microvms," Feb. 2022. [Online]. Available: https://github.com/firecracker-microvm/firectl

[29] "runc," Feb. 2022. [Online]. Available: https://github.com/opencontainers/runc

[30] "Apache jmeter," Feb. 2022. [Online]. Available: https://jmeter.apache.org/

[31] "Gorillamux - a powerful http router and url matcher for building go web servers with," Feb. 2022. [Online]. Available: https://github.com/gorilla/mux