Performance analysis of KVM-based microVMs orchestrated by Firecracker and QEMU

Security and Network Engineering, University of Amsterdam, The Netherlands

Sunday 22nd December, 2019

Philipp Mieden
philipp.mieden@os3.nl

Philippe Partarrieu
philippe.partarrieu@os3.nl

***Abstract*—Virtualization has become a driving technology behind hosting services, allowing providers to offer isolated virtual machines to customers, while executing hundreds of them on the same physical hardware. In order to support serverless applications and provide startup times close to containers, a small virtual machine type called microVM is becoming increasingly popular. microVMs use a slimmed down Linux kernel and offer a minimal legacy device model to achieve fast boot times and a small memory footprint. We analyze the performance of the only two hypervisors that currently implement the microVM machine type: Firecracker and QEMU. To create meaningful benchmarks, we evaluate the two solutions in the context of real usage scenarios. Our results indicate that Firecracker outperforms QEMU's microVM implementation in terms of kernel boot time, and scales better when running multiple microVMs concurrently.**

***Index Terms*—virtualization, microVM, hypervisor, serverless, cloud computing, FaaS, firecracker, qemu, kvm**

I. Introduction

Serverless computing is a new cloud-computing model that has gained popularity in recent years and is being offered to customers by multiple providers such as Amazon Web Services (AWS), Microsoft Azure and Google Cloud. Developers can upload functions which will be executed on demand and usage is billed with a granularity of a hundred milliseconds, for the duration the virtual machine is running to handle the request. As such providers execute code from multiple tenants simultaneously on the same physical hardware, isolation is very important to ensure no malicious customer can access or modify data from other tenants. Since containers do not provide sufficient isolation because they share the host OS kernel, full system virtualization is the only option to guarantee threat containment. However, full system virtualization comes at a price of much slower startup times, making it less suitable for the requirements of serverless applications.

To tackle these problems and reduce the exposed attack surface, Amazon developed a hypervisor or Virtual Machine Monitor (VMM), written in the Rust programming language. The Kernel-based Virtual Machine (KVM) is used to create small virtual machines, running a stripped down Linux kernel with only the necessary drivers loaded. Firecracker aims at offering the simplicity and performance of containers while also providing a secure virtualized execution environment. It follows a minimalist design principle and excludes unnecessary devices and guest functionality, in order to reduce its memory footprint and the attack surface of each microVM. This is intended to improve security while decreasing the virtual machine's startup time, and can be applied to increase hardware utilization. *AWS Lambda* and *AWS Fargate* are using Firecracker in production, which resulted in a cost reduction between *35-50%* due to improved infrastructure utilization, that was passed on to the customer [11].

Using Rust for the implementation is beneficial because the language aims to provide memory safety, which is a common source for vulnerabilities that is often abused for Remote Code Execution (RCE). QEMU has suffered from several critical vulnerabilities in the past that allowed for RCE and for guest to host escapes. An example of this is the VENOM (*CVE-2015-3456*) vulnerability, a buffer overflow in a virtual floppy device driver, that enables an attacker to escape from the confines of an affected virtual machine (VM) guest and potentially obtain code-execution access to the host [5]. Firecracker has the potential to prevent many attack vectors against the host system, both by avoiding to include any legacy drivers, only supporting recent kernels, and choosing a language for the implementation that provides memory safety. For these reasons, it is interesting to compare Firecracker's approach to the currently only alternative, the recently introduced microVM machine type for QEMU, in order to analyze the impact of using a memory safe programming language to the performance of the virtual machines in a realistic usage scenario.

This rest of this paper is organized as follows: in Section II, we will describe our research questions and limit the scope of this project. Section III briefly discusses hypervisors, their architecture and their security model. Section IV looks at related research and outlines our methodology for executing the experiments, as well as defining metrics of interest. Section VI will present our results and draw conclusions on which solution provides the better performance in our test scenarios. In Section VII we will discuss possible future additions to our benchmarking framework, metrics and methodology. Section VIII will critically reflect our results and name possible reasons for the observed data points. Finally, based on our measurements, we present our overall conclusions in Section

IX and give a recommendation on which solution to use.

II. Research Question

This research project aims to measure and compare the performance of the QEMU and Firecracker hypervisors when running microVMs, in the context of real world usage scenarios. The implications of implementing the Firecracker hypervisor in Rust, and the impact of this design decision are further subjects of interest.

We intend to answer the following questions:

1) How does the kernel boot time compare?
2) How does the machine startup time compare, including startup of a HTTP web service and responding to a request?
3) How do QEMU & Firecracker microVMs compare when performing a computationally intensive task?
4) How are the measurements affected when the host system starts multiple machines in parallel?

III. Hypervisors

A hypervisor is a layer of software that runs directly on the hardware of a machine. The hypervisor exports a virtual machine abstraction that resembles the underlying hardware. This abstraction is similar enough to the hardware that software which could run on the underlying hardware can also run in a virtual machine. VMMs virtualise hardware resources i.e. CPU, storage, memory, allowing multiple virtual machines to transparently and concurrently use the resources of the physical machine [6]. Conventionally, the operating system being run inside of a virtual machine is referred to as the guest OS.

Historically the hypervisor was the only code running in kernel space on a system, today we refer to these as type 1 hypervisors. This style of hypervisor inspired the newer hosted hypervisors (type 2) which run alongside a host OS. Hosted VMMs appear like a simple process running on the host and utilises the host's drivers, networking stack and other functionalities rather than implementing them from scratch.

There are three different approaches to virtualisation: Full Virtualisation (FV), Para Virtualisation (PV) and Hardware assisted Virtualisation (HV). PV requires modification to the guest OS in order to translate privileged instructions from the guest OS driver into hypercalls to the hypervisor. This simplifies the level of hardware abstraction and removes the need for a reverse device driver to translate instructions from the guest OS into instructions the hardware [7]. QEMU & Firecracker both leverage PV and rely on KVM to take advantage of the hardware's virtualisation technologies, namely Intel-VTx or AMD-V. [4]

In the design of Firecracker, Amazon improves performance by disabling the Peripheral Controller Interface (PCI) in the guest OS and is using virtIO devices instead. VirtIO is a standard for network and mass storage drivers where the guest knows it is running in a virtual environment and cooperates with the hypervisor, which results in a better overall performance as it allows for several optimizations.

IV. Related Work

Hypervisors have been extensively benchmarked in the current literature, with a focus on comparing performance for bare-metal hypervisors and lightweight virtualisation techniques such as containers [9].

Hwang et al. [7] compared four hypervisors (KVM, Hyper-V, vSphere and Xen) and concluded that there is no superior hypervisor. They find that different workloads are best suited for different hypervisors.

Morabito et al. [10] contrast running Linux on a KVM hypervisor versus Linux containers on Docker and Linux Containers (LXC), and they also cross compare with the *OSv* unikernel.

Koller et al. [8] compared unikernels and native performance on the host with container solutions, to check for which one performed best. Unikernels delivered much better performance than the container solution.

We did not encounter any related research to comparing performance of microVM hypervisors, which is likely due to the fact that microVMs are a novel concept with Firecracker being the first hypervisor designed to implement these ideas. Firecracker was open sourced mid 2018 so the scientific literature has yet to catch up. The QEMU microVM machine type has been released in a stable version in mid December 2019, at the time of writing no research related to QEMU's new microVM machine type could be found.

V. Experiment Design & Implementation

In order to compare both approaches and retrieve meaningful results, we used an identical kernel and identical custom root file system for both engine types. The default kernel for Firecracker did not work with QEMU because it was missing required drivers, therefore we've included those measurements for Firecracker, besides the measurements when both are using the exact same kernel.

A. Root File System

The root file system is based on *Alpine Linux version v3.10*, and was chosen due to its size and resource efficiency: a minimal installation to disk requires around 130M of storage [1]. The file system used for the experiments has an initial size of *58MB* and a capacity of *100MB* and is of type *ext4*. We used the *openRC* init system to initialize the network stack and start our agent process after system boot. The root file system contains all the necessary binaries and configurations to avoid any delays in transferring the files to the rootfs after system startup.

B. OS Kernel

The choice of Linux kernel to use during these experiments will have an impact on the results, because any enabled and unused features will impact the kernel boot time. In order to reduce the kernel boot time difference strictly down to the hypervisor's performance and KVM setup strategy, we decided to use a kernel supported by both QEMU and Firecracker. Since Firecracker's provided kernel has too many disabled features to be able to run in QEMU, we've compiled the Linux kernel version *v5.3.0* based on the configuration file provided by the initial microVM machine type contributor in the QEMU developer mailing list. We slightly modified this configuration to facilitate debugging, by enabling early print messages and support for DOS partitions.

C. Host Hardware

Our host system is a *Dell PowerEdge R210*, running *x86_64 GNU/Linux system version 4.15.0-70-generic 79-Ubuntu SMP*, with 8GB of DDR3 RAM and a *Intel(R) Xeon(R) CPU L3426 @ 1.87GHz*, with 128KiB L1 cache, 1MiB L2 cache and 8MiB L3 cache.

D. Benchmarking Framework

Our bechmarking framework is written in Golang. It handles orchestration of the experiments, communication and startup of the guest machines, as well as saving experiment results in log files. To allow for alternation of parameters during the experiments, the host component offers a command line interface with various options, ranging from the hypervisor to run (QEMU or Firecracker), to general settings like the number of virtual cores, the amount of memory per guest and the number of concurrent virtual machines to run. Our orchestrator on the host side communicates with an agent component that is running inside of the guests over a HTTP REST API. This design allows us to execute commands inside the guests and retrieve the output from those operations on the host. On the host side, measurement data is persisted in structured logfiles for each microVM execution.

For the guest microVMs to accessible from our host machine, we setup a local subnet for each microVM. For security purposes, every guest VM must be on a separate network. This is to avoid guest VMs from being able to communicate with each other. For simplicity and for easy naming, we used a */24* subnet per microVM. Since microVMs are designed to be completely isolated from each other, we programmatically create a new tap interface on the host i.e. the first guest VM will be assigned a static IP of *10.0.0.2*. The tap interface is assigned the address *10.0.0.1*. The next guest VM will be assigned *10.0.1.2* and the associated tap interface will be *10.0.1.1*, and so on.

This setup allowed us to reach the limit of concurrently running microVMs with QEMU on our host system. Launching more than 20 virtual machines caused hangs at some point and prevented the experiments from completing. This is because QEMU/KVM runs exactly like a normal Linux program. It allocates memory by using a *malloc()* or *mmap()* call e.g. if a QEMU guest is launched and we specify that its physical memory should be 1GB, the QEMU/KVM will do a *malloc()* call, and allocate 1GB from the host's virtual space. Just like a regular *malloc()* program, there is no physical memory allocated at the time of the *malloc()*, it will only be allocated the first time it is touched. Once the QEMU guest VM is launched, it sees the allocated memory as being its physical memory. Therefore the number of microVM that can be launched by QEMU will be limited by the available physical memory of the host.

Firecracker on the other hand, allows for over-subscription [2], which enables optimal distribution of workload across the physical server infrastructure.

E. Experiment Procedure

We chose to evaluate two different variations: a single machine and a multi machine scenario. Each machine will be bootstrapped using the custom root file system, kernel and tap device, as visualized by figure 1. Each virtual machine is assigned a single virtual processor core and 512MB of RAM.

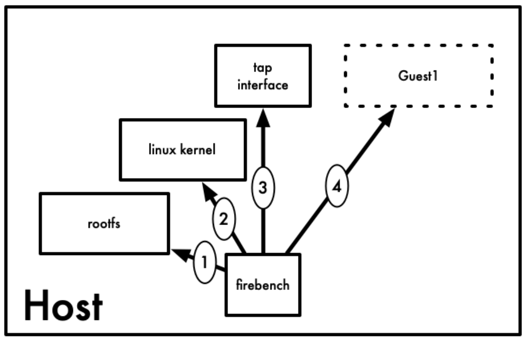

Fig. 1. Setup

1) Sequential Execution: In the sequential scenario, the root file system for the microVM is prepared once, the virtual machine is started, the experiments executed and the machine shutdown. This procedure is repeated *ten* times in a sequence, always ensuring only a single microVM is running at the same time. Sequential execution is not a real world scenario, as there potentially several machines might be started in parallel. We use these experiments to establish a performance baseline and observe deviations from this in the concurrent scenario. Figure 2 shows the network setup for communicating with the virtual machine from the host.

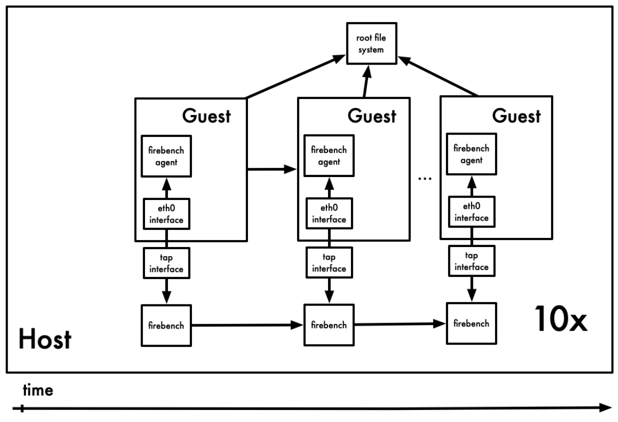

Fig. 2. Sequential microVM execution

2) Concurrent Execution: In the concurrent execution scenario, the root file systems for multiple virtual machines will be prepared simultaneously. Once all root file systems are ready, all virtual machines are started at once and execute the experiments. Figure 3 shows a simplified overview of this.

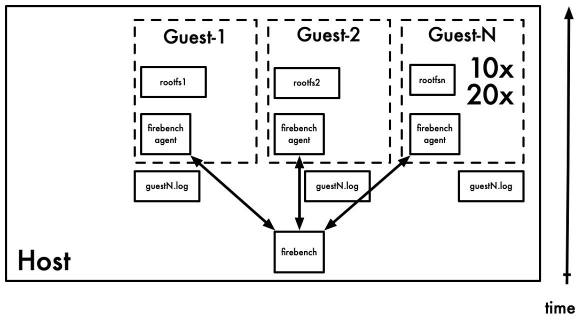

Fig. 3. Concurrent microVM execution

F. Metrics

In order to draw conclusions from our experiments, we defined the following metrics of interest:

1) Kernel Boot Time: During boot, the kernel prints messages to the kernel log by using the *printk* function. From the relative timestamps of the log messages, we determine the amount of time it took to initialize the OS kernel *after* the root file system has been successfully mounted. We define the end of the kernel boot operation to be the point in time of logging the last message in the systems *dmesg* log. In our case this always turned out to be the log message *random: fast init done*, logged after successful initialization of the pseudo random cryptographic number generator. Our agent process takes care of parsing this log message and sending back the time delta to the orchestrator on the host.

2) Web Service Startup Time: A common use case of serverless applications is to expose functionality over a HTTP REST API. We chose the scenario of serving static files from the root file system, while bypassing any OS level caching and forcing direct reads from disk, by setting the *O_DIRECT* flag during the *read()* syscall. For this, the agent process inside the guest offers a route that will be constantly requested from the orchestrator on the host as soon as the microVM has been started. We define the web service startup time to be the delta between launching the virtual machine and receiving a reply with a 200 status code to the HTTP GET request(s). Each HTTP request has a timeout of *100 milliseconds*, requests are sent every *10 milliseconds*.

3) CPU Usage: To evaluate the scheduling of CPU resources we will benchmark the operation duration of hashing pseudo-random data in a loop using the *SHA-256* hashing algorithm. A pseudo random blob of data from of size *1MB* will be hashed in a loop *100* times. This functionality is also provided by the agent process via a REST route, which will execute the hashing operation inside the guest and write back the result to the host.

4) Shutdown Time: The time from issuing a machine shutdown command to the guest via HTTP GET request to the agent process, until the machine is powered off and the control process on the host exits. Since Firecracker does not implement a power model for the virtual machines, the only way to stop them is to issue a *reboot* command on the system shell.

5) Concurrent Virtual Machines: In order to understand how QEMU and Firecracker's performance differs when running multiple microVMs, we repeated all experiments by running *10* and *20* microVMs concurrently. We expect this to reveal differences in the virtual machine manager's scheduling of the microVMs.

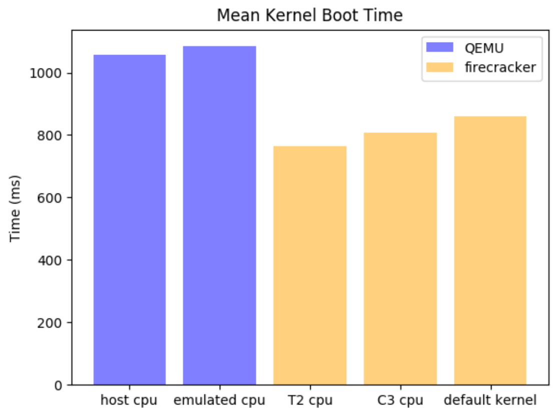

Fig. 4. Kernel boot time benchmark

VI. Results

A. Kernel Boot Time

In the Firecracker documentation, Amazon names an achievable boot time of less than 125ms [3]. To verify this claim we calculated the mean boot time for Firecracker and QEMU, which are displayed in Figure 4. In our experiments the mean kernel boot time of Firecracker microVM is *800ms* in the sequential experiments, and *1000ms* in the concurrent scenario. QEMU boots the Linux kernel *18%* slower on average. For Firecracker, the T2 CPU template delivers the best results on our hardware, the C3 template being slightly slower. The slowest mean boot time for Firecracker occurs with the default kernel built accordingly to the instructions from the official Firecracker repository. It is important to note that the network stack setup during takes additional time, without initialising the network stack the machine is able to boot in *150ms-200ms*. The reduced boot time of Firecracker can be explained by the fact that Firecracker only emulates five devices: virtio-net, virtio-block, virtio-vsock, serial console, and a minimal keyboard controller used only to stop the microVM [3].

Furthermore, by counting the numbers of log entries from the kernel boot process from *dmesg*, it becomes visible that less components for the kernel are initialized with firecracker, as shown in Figure 5. Interestingly, although showing the highest kernel boot time for Firecracker VMs, the default kernel with Firecracker writes the least log entries to *dmesg* during boot. Kernels booted via QEMU log slightly more messages during boot, therefore we assume additional components are initialized, which leads to the increased time needed for QEMU to boot the kernel.

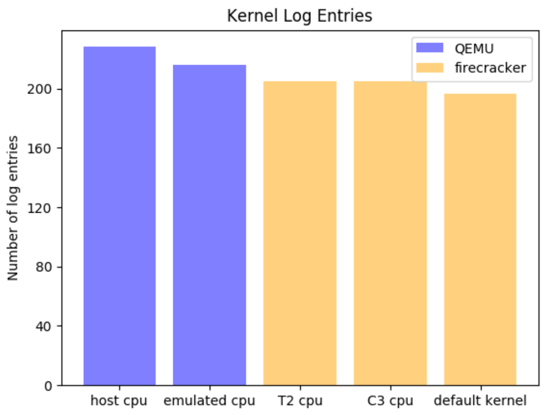

Fig. 5. Mean Kernel Log Messages

In Figure 4 we can see that there is a significant statistical difference between the QEMU and Firecracker when looking at how concurrency affects the kernel boot time. We believe that this large overhead in QEMU boot time is because QEMU uses libvirt to isolate each process from one another whereas we did not compare this to Firecracker's performance with jailer. Every QEMU process will be allocated the assigned amount of memory i.e. 512MB in our experiments whereas Firecracker microVMs will only consume the memory required by the VM process.

B. Web Service Startup Time

Since the web application startup time is measured from the moment we start the VM up until the HTTP endpoint becomes available, the kernel boot time, network stack initialisation and startup time of the web server are also included in this metric. Figure 6 displays the results of the measurements.

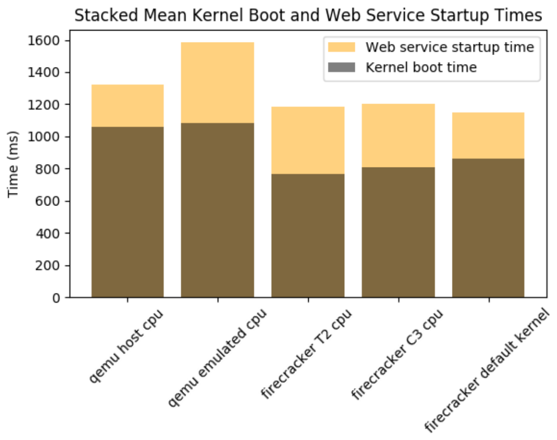

Fig. 6. Stacked kernel boot and webservice startup time

The kernel boot time has been stacked, but is not accumulated with the web application startup time, in order to illustrate the impact of kernel boot to the total time until the web application became reachable. When using the emulated CPU with QEMU in the sequential scenario, the time needed to start the web service it takes almost twice as long compared to QEMU using the host CPU. For Firecracker, using the different CPU templates T2 and C3 did only make a slight difference on our host system, namely T2 being faster by *10-30ms*. The default kernel for Firecracker achieved the best performance for a full machine launch and starting the web service, although its kernel boot time is the highest of all compared Firecracker variations. Figure 7 shows how the measured values change when executing multiple virtual machines in parallel. The first run executed *10* machines at once, the second run started *20* microVMs and the measurements.

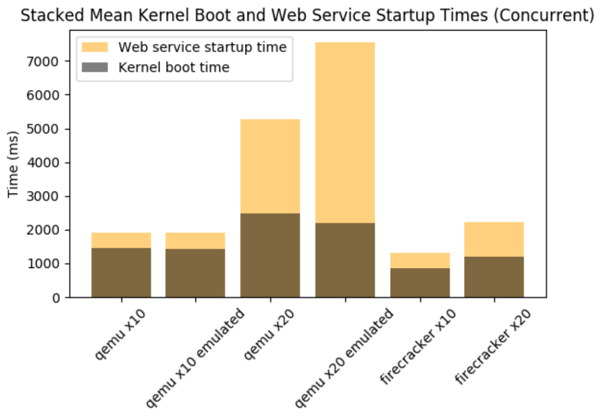

Fig. 7. Stacked mean kernel boot and webservice startup time (Concurrent)

For Firecracker the startup time for the web service doubled from running *10* virtual machines, to running *20* microVMS, while being generally between *66%* faster for the SHA-256 hashing calculations inside the guest, compared to Firecracker's performance. Figure 8 displays those results, showing the best performance for QEMU microVMs using the host CPU.

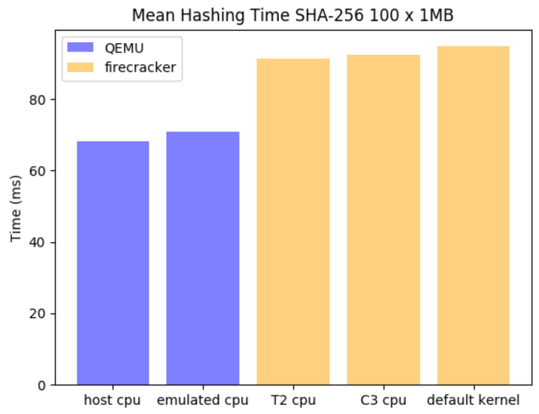

Fig. 8. Mean hashing time benchmark

Figure 9 shows the benchmarks in the multi-vm scenario. It becomes visible again that Firecracker scales much better when running *20* machines in parallel, while QEMU's performance suffers greatly and takes more than twice as long as Firecracker. For the benchmark with *10* parallel machines however, QEMU is still slightly faster than Firecracker.

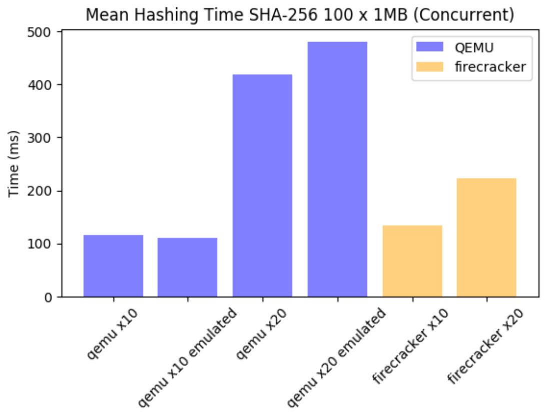

Fig. 9. Mean hashing time benchmark (Concurrent)

C. Shutdown Time

Figure 10 displays the results of measuring the shutdown time for stopping virtual machines in sequential order. We've observed a significant overhead for QEMU, for both running with the host and emulated CPU. While Firecracker manages to shut down the machine in about *2000ms* on average, QEMU takes more than *10000ms* for the same task.

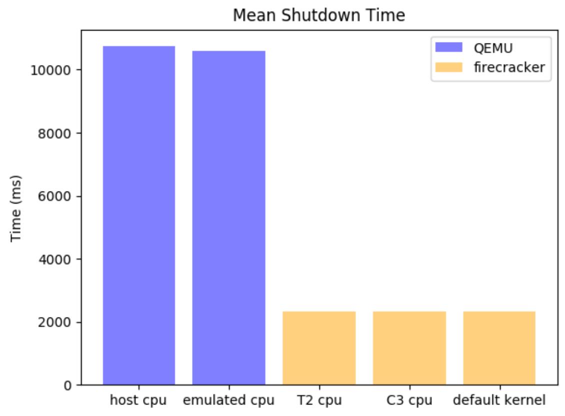

Fig. 10. VM shutdown time benchmark

Figure 11 shows how these measurements change when stopping several machines at once. We can observe a significant impact on QEMU again, especially when running *20* virtual machines in parallel, while Firecracker delivers a stable performance.

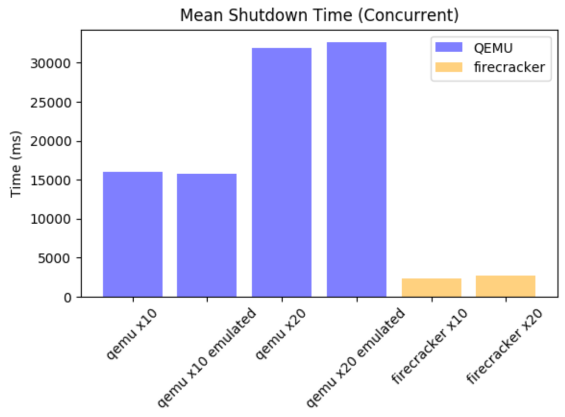

Fig. 11. VM shutdown time benchmark (Concurrent)

On average, QEMU microVMs take between *85%* longer until they are powered down, compared to shutting down Firecracker microVMs. Looking into the reason for this could be subject to future research.

D. Choice of Programming Language

Based on the gathered results we conclude that the choice of a memory safe programming language for the hypervisor implementation, did not affect the performance of our metrics in a negative way. While outperforming QEMU's implementation in terms of startup time and making the webservice available, Firecracker scaled much better. We believe that preventing memory corruption exploits in such complex systems is important, in order to guarantee data confidentiality and integrity for customers. Using rust for the implementation seems to be a solid choice here, the codebase of Firecracker consists of about

55000 lines of rust code, as of version *v0.20.0*, while QEMU comes at about *1,6 million* lines of C code, as of version *v4.2.0*. Lines of code have been counted using the *cloc* tool. It should be noted however, that it is possible to slim down QEMU by suppliying various options during compilation, possibly reducing the total lines of code involved.

VII. Future Work

We will opensource our benchmarking framework, to allow for independent verification of our results and further extension. The repository will be made available at *github.com/dreadl0ck/microbench*. Once new versions are being released, it is trivial to repeat the experiments and regenerate the plots to compare the impact of changes to the programs. Possible areas for extension include but are not limited to: IO performance, network throughput and scaling impact.

Another possible area of research is to modify the Linux kernel to use I/O writes so that events in the firmware and Linux kernel can be traced. This would allow breaking our measured kernel boot time accordingly and trace points such as: the first kvm_entry, to determine how long QEMU and Firecracker took to finish initialising. Furthermore, the microVMs' firmware and kernel initialisation time could be traced as well as the time to start the init process within the VM.

Most importantly, we would like to add benchmarks for Firecracker running using the provided jailer program, and see if this causes any deviations from the observed values. Another interesting benchmark would be running the experiemtns in a container solution such as docker, to see how much the results differ to this technology. Unikernels and technologies such as light-weight contexts should also be added to the comparison to get a better picture of the existing technologies.

VIII. Discussion

When measuring the number of syscalls the control process on the host is issuing with *strace*, we observed that QEMU makes about *730 000* system calls, while Firecracker manages to issue only about *5,000*, for both the most frequent one being the *ioctl* call. By default QEMU utilises *libvirt* to perform operations on virtual machines such as starting and stopping. All software consuming KVM in QEMU does so through *libvirt*. Because the QEMU process handles input and commands from the guest, it can be exposed to malicious activity and therefore it should run in an isolated environment. This is the principle of least privilege [4].

On the other hand, Firecracker's management tool (firectl) does not confine virtual machines and instead, Firecracker provides a binary called jailer to configure the namespace of the running Firecracker process. The experiemnts should be repeated on different hardware and compared to the current results. Further program instrumentation could provide more precise and additional metrics.

IX. Conclusion

Firecracker outperformed QEMU's current implementation in the kernel boot time and web service reachability experiments, as well as scaling better in the concurrent scenarios. We observed drastic differences in shutdown time, which can lead to problems in environments with several hundred of machines. Due to the smaller codebase of Firecracker, the use of a memory safe programming language and the demonstrated scaling capabilities, we recommend the use of Firecracker for the operation of microVMs. In the cloud business, this technology will cause in a cost reduction for the customer, and provide a better infrastructure usage for the providers. Both solutions are open source and under active development, the experiments have been designed to be easily repeated, in order to check for improvements or changing trends in the gathered data.

Appendix

References

[1] Alpine. *Alpine About Page*.

[2] Amazon. *Firecracker Design Documentation*.

[3] Amazon. *Firecracker Frequently Asked Questions*.

[4] Paolo Bonzini. All you need to know about kvm userspace. 10 2019.

[5] Jason Geffner. Venom vulnerability.

[6] Robert P Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[7] Jinho Hwang, Sai Zeng, Frederick y Wu, and Timothy Wood. A component-based performance comparison of four hypervisors. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 269–276. IEEE, 2013.

[8] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 169–173, New York, NY, USA, 2017. ACM.

[9] Zheng Li, Maria Kihl, Qinghua Lu, and Jens A Andersson. Performance overhead comparison between hypervisor and container based virtualization. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 955–962. IEEE, 2017.

[10] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. 03 2015.

[11] Nathan Peck. Aws fargate price reduction up to 50%. 01 2019.

[12] RedHat. *KVM memory*, 2 2010.