Firecracker: A Case Study

Anshu Verma, Arpit Jain, Arun Jose - Group #17

Abstract

AWS Firecracker [3] is an open-source virtualization technology that is purpose-built for creating and managing secure, multi-tenant container and function-based services. It combines the best of containers and virtual machines by running containers withing microVMs and replacing type-2 hypervisors. gVisor [6] is a similar project by Google which provides a user-space kernel for containers. gVisor limits the host kernel surface accessible to the application while still giving the application access to all the features it expects. In this paper, we have analyzed the performance of Firecracker and compared it with gVisor. The performance metrics range from spinning up multiple instances to running CPU heavy matrix multiplication workloads on both the systems. Our findings should help the users in deciding between the two systems depending upon the features of their workloads

1 Introduction

Developers face a difficult choice when deciding between whether to use hardware-level virtualization via Virtual Machines or OS-level virtualization via Containers when deploying their applications. Containers are becoming increasingly popular due to their light-weight structure and efficient sharing of the host OS. However, this efficiency is obtained by trading off a secure isolation between workloads which could be maliciously exploited [14].

Amazon's AWS Firecracker attempts to get the "best of both worlds": offer the speed and resource efficiency of containers, while providing the security and workload isolation of traditional VMs [3]. It does this by implementing a Virtual Machine Monitor (VMM) that allows users to deploy lightweight micro Virtual Machines (microVMs). It has been developed optimizing for running transient and short-lived processes.

Firecracker uses the Linux Kernel-based Virtual Machine (KVM) [9] to create and manage microVMs on the system. In a typical Linux-based virtualization scenario, this type-1 hypervisor(KVM) is complemented by a type-2 hypervisor such as QEMU [11] that hosts a broad feature set. Firecracker is a minimalistic approach to replace QEMU and exclude any unnecessary overheads. This results in a low memory footprint and fast startup type for microVMs enabling users to pack thousands of such microVMs on a single machine. At the same time, every container group can be encapsulated with a virtual machine barrier, providing the necessary security guarantees that most containers lack.

We have outlined the details of the Firecracker architecture in more detail in § 2.1. This is followed by an introduction to Google's gVisor in § 2.2, which attempts to provide similar benefits but with a different architectural design. We then describe the workloads we intend to run in § 3 and outline our results with interpretations in § 4. Finally, we expand on some related work in § 5 and outline the scope for future work in § 6.

2 Background

2.1 Firecracker Architecture

Let us now understand Firecracker's architecture [2] to provide microVMs that combine the security and workload isolation properties of traditional VMs with the speed, agility and resource efficiency enabled by containers.

A Firecracker process runs three threads: API, VMM, and vCPU(s). The API thread is responsible for Firecracker's API server using which the client can set up and configure the microVM. The VMM thread exposes the machine model, minimal legacy device model, microVM metadata service (MMDS) and VirtIO device emulated Net and Block devices, complete with I/O rate limiting. Lastly, there can be one or more vCPU threads depending on the guest configuration. vCPU threads are created via KVM and run the KVM_RUN main loop. They execute synchronous I/O and memory-mapped I/O operations on devices models.

In order to run a microVM within a Firecracker process, the client first sets up the kernel and root file system for the microVM. Setting up kernel and root filesystem, and launching a microVM are all done by making API calls to the Firecracker process. Access to all of the emulated devices is rate limited by Firecracker to ensure that the hardware resources are used fairly by multiple microVMs.

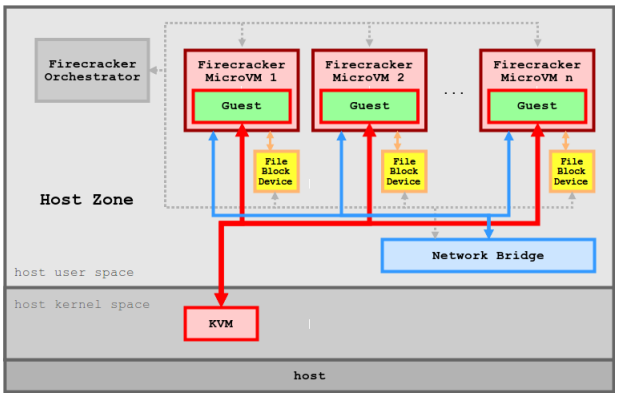

Figure 1: This diagram [2] depicts an example of a host running multiple Firecracker microVMs. Each Firecracker process encapsulates one and only one microVM. Both network and storage devices are emulated by Firecracker. The emulated network devices are backed by TAP devices on the host, and the emulated block devices are backed by (filesystem image) files on the host.

2.2 gVisor

gVisor [6] is an open-source OCI compliant container runtime initiative by Google. It runs containers with a new user-space kernel, delivering a low overhead container security solution for high-density applications. The main goals of the gVisor are:
Defense-in-depth: Each layer has its own user-space kernel providing multiple layers of security and defense against host kernel vulnerabilities.
Lightweight: The additional layer should not affect the performance of the containers
Zero configuration: It should be capable of running most Linux applications unmodified, with zero configuration.

gVisor runs on Sentry which both implements Linux and itself runs on Linux [16]. This provides gVisor defense-in-depth as a malicious user who has exploited a vulnerability in Sentry only has access to user-space and the host kernel is not breached. All the system calls from the containers to the kernel OS are intercepted by Sentry. It provides two modes to handle system calls. In the pTrace mode, all the system calls are intercepted by the Sentry. In the KVM mode, Sentry acts like a pass-through layer for these system calls. gVisor has another component 'Gofer', which handles all the commonly exploited syscalls like *socket* and *open*.

3 Methodology

3.1 Cluster Configuration

All experiments were run on the bare-metal CloudLab machines with Firecracker and gVisor running on separate instances.

Table 3.1 enumerates the guest configurations initialized in both Firecracker as well as gVisor.

Parameter	Value
OS Version	Linux Kernel Version 4.4.0
Memory	2 GB
vCPU	1
Model Name	Intel(R) Xeon(R) Processor @ 2.20GHz
Cache size	14080 KB

Table 1: Configuration for Firecracker and gVisor

3.2 Workload

We now will describe the seven workloads, ranging from spin up throughput to matrix multiplication, which we chose to run as our experiments.

- **Containers/microVM spin-up:** In this experiment, we have attempted to startup and shutdown containers and microVMs as fast as possible. Both containers and microVMs boot up an Alpine OS [8], which is one of the smallest possible OS images to boot. The guest configurations for both were kept as 1 vCPU and 2GB memory. Both were started in detached mode, as background processes, without any user logged into the OS. The key idea behind this workload is to measure the boot time difference between a container and a microVM, given that this is significantly higher in traditional VMs.

- **Network performance:** In this experiment, we have computed the network throughput of both the systems by downloading files of varying sizes from a speed test site using cURL. We experimented with downloading both small (1MB) files and large (1GB) files. A better approach for this experiment would have been to use Iperf, ensuring both client and host reside in the Cloudlab cluster. Due to time restrictions, we could not attempt the same.

- **System calls:** In this benchmarking test, we intend to invoke the system call *getpid()* on both gVisor and Firecracker for multiple iterations and compare the average time taken in both cases. Since Firecracker uses KVM directly, we expect to find some positive impact on the system calls.

- **File Open/Close:** To start this workload, we measure the latency of opening and closing files without any I/O operation. We ran this workload over multiple iterations and measured the average latency and throughput.
  gVisor handles the open file calls a bit differently than other containers and Firecracker. Since the *open()* method call is one of the most exploited methods, the call is handled by Gofer. The Gofer is able to open files on the Sentry's behalf and return the file handler to the Sentry via 9P. Once the Sentry receives the file handler, it can serve read and write operations for the application.

- **Malloc:** We have divided this workload into two parts. i) malloc without free, ii) malloc with free and tried to observe how Firecracker and gVisor are performing. We ran this experiment for different malloc size ranging from 1 KB to 1 MB to study the pattern of how the heap memory is being handled by Firecracker and gVisor.

- **Read performance:** In this workload, we measure the read performance of gVisor and Firecracker on external tmpfs and the guest file system. We extended the file open workload by the read workload. The experiment had a range of *read()* block size from 1 KB to 1 MB.

- **Write performance:** Our experiment on the storage aspect of Firecracker is further extended by adding the writing layer. We had a range of block sizes varying from 1 KB to 1 MB writing on tmpfs and the guest file system. In the case of a container, multiple containers can share access to the same image, and make container-specific images on a writable layer which is deleted when the container is removed.
  We expect Firecracker to perform better in this experiment as it has a host CPU core dedicated to the Firecracker device emulation thread.

- **Matrix multiplication:** The Matrix multiplication workload can be described as a highly parallel, compute-intensive workload with little to no reuse of input data. We wrote a C program to randomly generate a matrix and multiply with itself. The workload is performed by providing the matrix size as an input argument and the average latency over multiple iterations is measured.

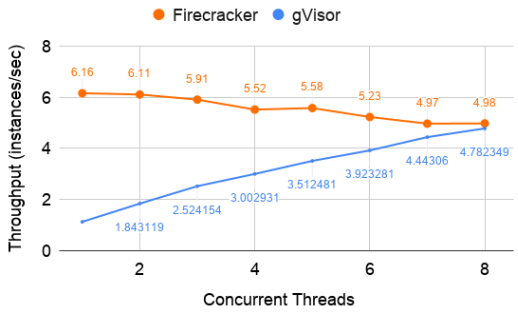

Figure 2: Varying number of concurrent threads to spin up and destroy containers and microVMs.

4 Evaluation

4.1 Containers/microVM spin-up

Observation: Figure 2 shows the spin-up and tear down throughput with varying number of concurrent threads. Firecracker displays a higher throughput as compared to gVisor. We also observed that with more concurrent threads, the throughput decreases for Firecracker whereas it increases for gVisor.

Explanation: Firecracker provides a minimal required device model to the guest operating system while excluding non-essential functionality (only 5 emulated devices are available: virtio-net, virtio-block, virtio-vsock, serial console, and a minimal keyboard controller used only to stop the microVM). This, along with a streamlined kernel loading process enables a short ($<$ 125ms as per specification) startup time and a small ($<$ 5 MiB as per specification) memory footprint. On the other hand, the startup time of gVisor is significantly higher, with most of the time overhead associated with Docker itself.
In the case of Firecracker, the kernel and rootfs image is shared across all the microVMs, which might be the reason for the reduced throughput with more concurrent threads as a result of more contention. This is our hypothesis and we are still evaluating other potential reasons behind this.

Implication: Firecracker can be preferred w.r.t spinup time if the number of concurrent threads is low, whereas gVisor might have an edge as the number of such threads is higher.

4.2 Network performance

Both Firecracker and gVisor have a restricted architecture for accessing the network. The network devices are emulated by Firecracker. This is done by setting up tap devices in the host which are then attached to the guest for accessing the network. Hence, there can potentially

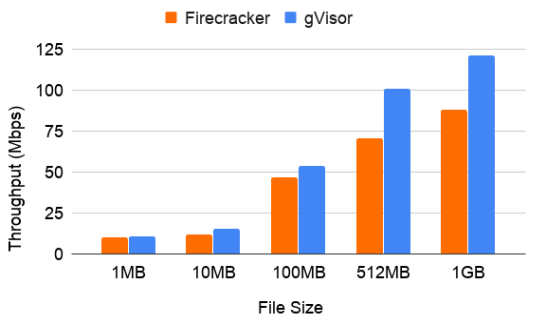

Figure 3: Downloading files of varying size using *cURL*.

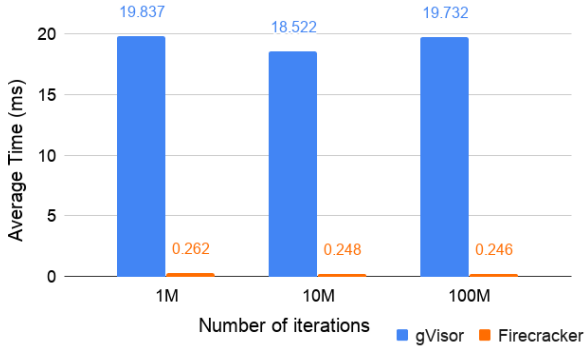

Figure 4: *getpid()* syscall Performance

be some overhead while accessing the network from the Firecracker guest instance. In the case of gVisor, networking is provided using a user-space networking stack, called netstack. This is done to safeguard the user against any networking attacks. As expected, this also results in some overhead while accessing the network from the gVisor guest instance.

Observation: As shown in Figure 3, we observed that both Firecracker and gVisor have similar performance with respect to the network throughput. For larger file sizes, gVisor marginally outperforms Firecracker.

Explanation: One reason for this could be the in-built rate-limiting in Firecracker to avoid overuse of any resource by any of the guests. We also compared the results of gVisor network performance with previously reported results in [16], and observed that the network stack has improved over time.

As explained in § 3, our experiments obtained the throughput by downloading files of varying sizes from a speed test website. This setup adds redundant noise over the network because of various factors like availability and scalability of the speed test website, network throttling by Cloudlab, network contention by other cluster users, etc. A better set up for this experiment would have been to use *Iperf* which would have initiated the host and the client within the Cloudlab cluster. As a result, a lot of noisy factors would have diminished and provided better results. Such a set up has been described as part of our future work in § 6.

Implication: Users can use either gVisor or Firecracker as both of them have similar performance for network-intensive workloads.

4.3 System calls

Observation: Figure 4 shows that *getpid()* syscall runs about 75x slower in gVisor than Firecracker.

Explanation: gVisor provides defense in depth by adding a layer called Sentry between the containers and the guest OS. All the system calls executed in the containers go to the guest OS via Sentry to provide added protection. This creates an additional overhead and impacts gVisor's performance for system calls.

In the case of Firecracker, all the system calls reach the guest OS via Firecracker which acts as an orchestration layer between microVMs and the guest OS.

To add protection against vulnerable system calls in Firecracker, seccomp filters are used to limit the system calls in Firecracker. The filters are loaded in the Firecracker process, immediately before the execution of the untrusted guest code starts. It provides three levels of filtering.

- 0: disabled.
- 1: whitelist a set of trusted syscalls by their identifiers.
- 2: whitelists a set of trusted system calls with trusted parameter values.

Implication: If the workload involves heavy system call usage, then Firecracker is recommended to be used over gVisor.

4.4 File Open/Close

Observation: Figure 5 shows the *open()* call is about 60x times slower in gVisor than Firecracker. It also shows that Firecracker has a similar performance for external tmpfs and guest OS.

Explanation: In the case of gVisor, *open()* call goes to the host system via Gofer[6]. We run this experiment on the guest and external tmpfs. External tmpfs is created by mounting a tmpfs in the host file

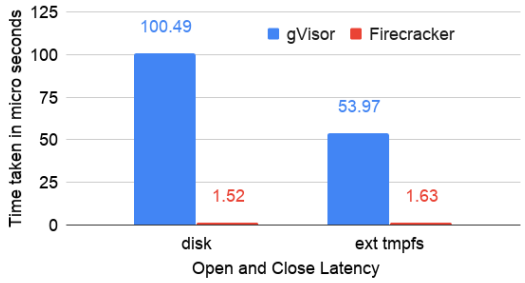

Figure 5: File Open/Close Performance

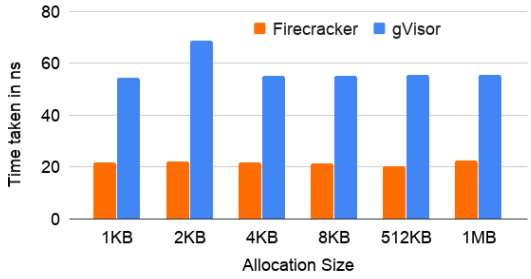

Figure 6: Malloc with free call after every iteration

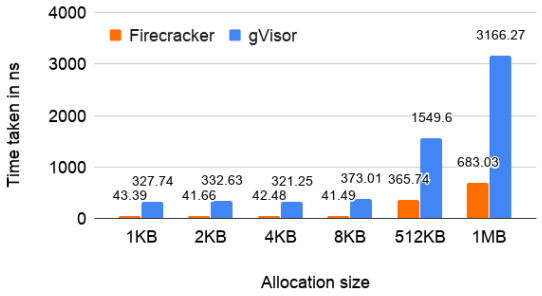

Figure 7: Malloc without freeing memory after every iteration

system. This mount point was exposed to Firecracker and gVisor while starting up the micro VM and container respectively.

Although the tmpfs is intended to appear as a mounted file system, all the writes are stored in volatile memory. We notice that gVisor is significantly slower than Firecracker which implies that Gofer is a major bottleneck for *open()* method calls.

Implication: A workload involving calls that are accessed via Gofer will be a major bottleneck in the case of gVisor and it is recommended to use Firecracker instead.

4.5 Malloc

4.5.1 Malloc without free()

Observation: Figure 6 shows that memory allocation in heap using *malloc()* is 7x faster in Firecracker than gVisor when we are not freeing memory.

4.5.2 Malloc with free()

Observation: Figure 7 shows that memory allocation in heap using *malloc()* is 2.5x faster in Firecracker than gVisor when we are freeing memory after every iteration.

Explanation: Firecracker[3] and gVisor[6] are designed to have smaller memory footprints. If an application grows its memory footprint beyond the original allocation then it results in page faults which is an expensive operation.

In both cases, we ran over 100K iterations for size ranging from 1 KB to 1 MB to see the impact on the performance of Firecracker and gVisor.

In the first set of experiments without freeing the memory, we noticed that gVisor was almost 7x slower than Firecracker. In the second round of the experiment, we stressed the system with over 100K of *malloc()* and *free()* call in every iteration i.e. every iteration comprises of *malloc()* and a *free()* method call. Despite having two methods call every iteration as compared to one in the first round of experiments, the *malloc()* with free is 2x faster than *malloc()* without free. In this set of experiments, we notice that Firecracker consistently performs better gVisor.

The performance for both systems degraded when we started stressing the systems with 512 KB and 1 MB allocation size. The reason for this degradation stems from the fact that while allocating memory greater than MMAP_THRESHOLD, *malloc()* allocates the memory as a private anonymous mapping using *mmap()*. MMAP_THRESHOLD is 128 KB by default in Linux, therefore, we see a performance degradation for allocation size greater than 128 KB[10].

Implication: Firecracker is advisable to be used over gVisor for memory-intensive applications. This is because in the longer run, we see gVisor getting significantly affecting the performance of such applications.

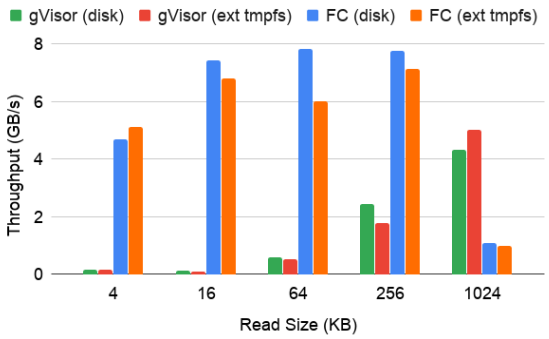

Figure 8: Read Performance

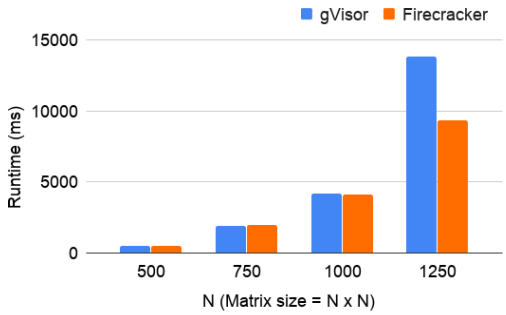

Figure 10: Matrix multiplication

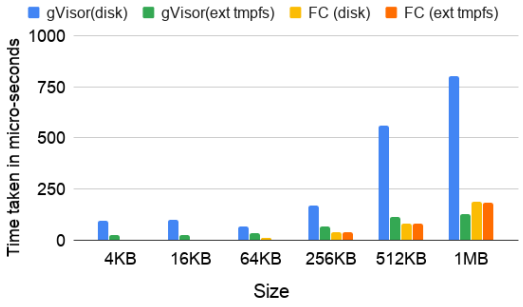

Figure 9: Write Performance

4.6 Read performance

Observation: Figure 8 shows that the Firecracker throughput increases as the read size increases, but drops significantly for a large read size of 1 MB by around 75%.

Explanation: At larger read sizes, there is likely to be an increase in the number of page faults. This could explain why there is a drop in throughput for Firecracker at large read sizes. For gVisor, along with the rise in page faults, there is also a decrease in the number of user-space KVM exits. As a result, there is a gradual increase in throughput for gVisor, and no drop at larger read sizes[16].

Implication: Firecracker read throughput suffers at large read rates, so it should only be preferred for small read rates. gVisor can be preferred at higher read rates.

4.7 Write performance

Observation: Firecracker is 4x faster than gVisor for persistent writes as observed in Figure 9.

Explanation: In this set of experiments, we stressed gVisor and Firecracker with writes ranging from 1 KB to 1 MB over 100K iterations. We ran these experiments on external tmpfs of both and guest OS of Firecracker and the writable layer of gVisor. It is important to note that all the writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted and the underlying image remains unchanged.

All the writes to the gVisor's writable layer are accessed via Sentry and Gofer which adds additional overhead suggesting Gofer is a major bottleneck[6]. We also observe that gVisor and Firecracker give an almost similar performance while writing to external tmpfs.

Implication: If the workload contains heavy writes then it is advisable to use Firecracker than gVisor. gVisor can be used for a write-heavy workload while using sandbox overlay (external tmpfs).

4.8 Matrix multiplication

Observation: Figure 10 shows an almost comparable performance for low to moderate compute-intensive workloads. Firecracker is around 32% faster as the computational intensity is increased.

Explanation: A traditional VM takes about 10% additional system CPU overhead due to the type-2 hypervisor like QEMU present in them. As a result, they are inherently slower than most containers running on similar computational workloads. Firecracker replaces this hypervisor with a much lighter version, which could alleviate any such CPU overhead. On the other hand, the Sentry manages the memory and CPU in gVisor [6]. This introduces a small overhead which could be the reason for the slightly better performance of Firecracker.

Implication: Firecracker can be preferred for

computationally-intense workloads.

5 Related Work

Kata Containers & QEMU: Similar to Firecracker, Kata Containers [4] also perform like containers and provide the workload isolation and security advantages of VMs. It is an OCI-compliant container runtime that executes containers within QEMU [11] based virtual machines. QEMU is a type-2 hypervisor that can emulate virtual resources to the guest OS running within the VM. It does so by translating every privileged mode system call made by the guest OS. Firecracker is a cloud-native alternative to QEMU. Firecracker focuses only on running containers safely and efficiently. To do so, it removes all the non-essential functionality and provides a minimal required device model to the guest operating system. This is how it differs from Kata Containers and QEMU.

gVisor: gVisor is a container runtime by Google with a focus on isolation and defense in depth. It has two components Sentry and Gofer which acts as an intermediate layer between the containers and host OS. gVisor allows only 211 of 319 *syscalls* and the critical calls like *socket* and *open* are intercepted and handled by Gofer for additional defense. Many prior studies have been done to compare lightweight VMs and containers [1, 7, 15]. A recent study was done to compare native and gVisor in pTrace and KVM mode [16]

6 Future work

6.1 Scalability

A single host can run multiple Firecracker processes, and thus it should be evaluated if the performance of a workload on a single microVM in isolation gets degraded when the host is shared by multiple microVMs. For this, we will start up multiple microVMs on the same host and run the same workload in all of them. We will make comparisons of these results with the corresponding isolation run results.

6.2 Complex Workloads

In this paper, we have only experimented with the workloads which are contained in a single container or microVM. We can extend these experiments by including some workloads which spread over multiple containers and microVM and covers network communication and shared data access.

6.2.1 Map and Reduce - Ephemeral Data

Traditional compute-centric frameworks [5, 17] process batches of data in multiple stages often represented by a DAG abstraction. Each stage reads input data, performs some compute, and writes some intermediate data to the disk. Post this, the next stage reads data from the disk, once again performs some computation, and once again writes out intermediate data to the disk. It is termed as ephemeral data because the data is no longer needed once downstream stages have consumed them. We model an ephemeral data workload by modeling it as a variant of iterative MapReduce [5], The mapper reads data from disk, then writes out intermediate data to disk, which is then read by a variable number of reducers. The reducers once again write out data to disk which is then read by the mappers in the next iteration. The ephemeral data written out by mappers and reducers can be deleted when they have been consumed by their downstream stage. Thus, ephemeral data presents a workload that involves a high number of writes, reads, and deletions.

6.2.2 pyWren

AWS Firecracker is created as an optimized virtualization layer for event-driven, short-lived nature kind of workloads. It provides hardware virtualization-based security boundaries of virtual machines while maintaining the smaller package size and agility of containers and functions. It's a fairly simple serverless execution model designed specifically for stateless functions with a small memory footprint and limited access to virtIO devices.

As future work, we can move pyWren[13] to AWS Firecracker from AWS lambda functions and test its performance against the existing pyWren prototype.

The rationale behind moving pyWren to AWS Firecracker are:

- The existing limit for AWS Lambda is extremely low. If we move it to AWS Firecracker, we can get greater control over resource allocation.
- Launch overhead is an issue for the AWS lambda functions. AWS Firecracker can launch 5-6 instances per seconds which is fairly good when compared against 30 seconds of AWS lambda function invocations.

6.2.3 Hot and Cold

[12] presents an observation that users operate for a large fraction of their time on a small subset of files. We refer to the frequently accessed data as hot data and the less frequently accessed data as cold data. We define skewness as the percentage of total data that is hot data. For instance, a skewness of 10% implies that 10% of the data

is accessed frequently while 90% of the data is accessed infrequently. We perform a mixture of reads and writes on both hot and cold data in our workload.

References

[1] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder. Performance evaluation of microservices architectures using containers. In *Proceedings of the 2015 IEEE 14th International Symposium on Network Computing and Applications (NCA)*, NCA '15, pages 27–34, Washington, DC, USA, 2015. IEEE Computer Society.

[2] AWS. AWS Firecracker Design. https://github.com/firecracker-microvm/firecracker/blob/master/docs/design.md.

[3] AWS. Firecracker. https://firecracker-microvm.github.io/.

[4] K. Containers. Kata Containers. https://katacontainers.io/.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[6] Google. gVisor. https://gvisor.dev/.

[7] Z. Kozhirbayev and R. Sinnott. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68, 03 2017.

[8] Linux. Alpine. https://alpinelinux.org/.

[9] Linux. Kernel Virtual Machine. https://www.linux-kvm.org/page/Main_Page.

[10] Linux. Linux. http://man7.org/linux/man-pages/man3/malloc.3.html.

[11] QEMU. QEMU. https://www.qemu.org/.

[12] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[13] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 1–8, New York, NY, USA, 2018. ACM.

[14] D. J. Walsh. Are Docker Containers Really Secure? https://opensource.com/business/14/7/dockersecurity-selinux, July 2014.

[15] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[16] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The true cost of containing: A gvisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.