RESEARCH ARTICLE OPEN ACCESS

A Distributed and Robust MicroVM Environment for Containerized Workloads

Adithya H*, Sagar P**, Shobha G***

*(Department of Computer Science, R.V. College of Engineering, Bangalore
Email: hadithya369@gmail.com)

** (Department of Computer Science, R.V. College of Engineering, Bangalore
Email:sagarperuvaje@gmail.com)

*** (Department of Computer Science, R.V. College of Engineering, Bangalore
Email:shobhag@rvce.edu.in)

--************************----------------------------------

Abstract:

Serverless computing is an emerging technology gaining the attention of cloud service providers and users alike. Many prior works have discussed the concept of function-as-a-service, surveyed the existing platforms, and evaluated their performances. Our work focuses on the Firecracker platform, and the specific advantages of using a lightweight microVM in environments that require robust components. Containerized workloads are executed with these lightweight virtual machines as the runtime environments orchestrated by Kubernetes. The proposed work creates a microVM cluster with computation, memory and networking capabilities configured based on requirements and evaluate the system using FunctionBench, a workload suite for evaluating serverless function services.The evaluation of our proposed environment depicted comparable latencies for video processing(2s for 1GB memory) and model serving(3s for 1GB memory) benchmarks with serverless platforms like AWS Lambda. FunctionBench workload results verifies the performance of our environment. The robustness of the environment and its performance earns an investigation into its applicability as a candidate in Massively Parallel Processing (MPP) in business and scientific applications.

Keywords **—Serverless computing, function-as-a-service,Firecracker, Kubernetes, FunctionBench**

--************************----------------------------------

I. INTRODUCTION

Serverless computing has emerged as one of the most relevant sectors of cloud computing and has been offered as a service by popular cloud service providers. The function-as-a-service model has gained popularity among various user groups due to the ease of execution of intended tasks without focusing on the setup of runtime for the execution of these tasks. Large distributed systems often involve the use of various virtual machines which host containers that can run user functions as a service which justify that Virtual machines are a better abstraction compared to functions [1].

The evolution of serverless systems has marked various advantages in executing independent application-based functions. These platforms have proven to be less expensive when compared to purchasing compatible servers for the different functions to be executed, or to renting the required servers for a given period. Users are also freed from analysing the amount of resources their functions take during their execution to choose a proper runtime, as the serverless providers automatically scale these depending on the requirements of the function. Moreover, task-level parallelism for functions could be automatically identified and implemented in the serverless platforms rather than the programmer having to analyse them.

Firecracker is a lightweight virtualization technology that runs on KVM, which enables the booting of multiple microVMs in a small time. When these VMs are used to host containers, the quick bring-up and shutdown of the hosts combined with instant launching of containers in the microVM bring the latency of the complete serverless computing system to a minimum. This also implies that many other boot-time overheads are eliminated, improving space and cost. The large number of concurrent user demands is satisfied by the scalability of Firecracker, which provides support for an equally numerous microVMs of various computation powers, all hosting various containers in them.

The proposed work makes use of Firecracker microVMs as nodes of a Kubernetes cluster, which uses docker containerization technology as the container runtime environment to run containers at the nodes. It also runs benchmarks from Function bench, in this system, that evaluates it with respect to the AWS Lambda platform.

II. LITERATURE SURVEY

Some previous works related to serverless architectures and their performance reviews.

J. Park, et al. introduce a type of infrastructure called “Serverless Clusters” that combine the functioning of serverless functions and Virtual Machines [1]. They argue that functions are not an appropriate abstraction for serverless execution, rather VMs are a better fit because they don’t suffer the shortcomings of a serverless function - they don’t provide auto-scaling by design, can’t guarantee concurrent execution and require workarounds to achieve communication between functions.

I. Pelle, et al.study the changes in the internal architecture of Function-as-a-Service (FaaS) systems, their improvements regarding ability to perform concurrent executions and the reduction in performance degradation which was seen in earlier generation FaaS systems [2]. They show that regardless of the workload, the latency is not affected much for a given memory configuration.

J. Kim, et al. provide a data-oriented workload suite for serverless functions that include arithmetic floating-point operations, image processing and ML Model Training processes for testing resource utilization in serverless setups provided by AWS Lambda and Google Cloud’s Functions [3].

Poth A. et al. provides a detailed measurement methodology for FaaS platforms and provide delay characteristics analysis of AWS components and investigate the functioning of a latency sensitive application - namely a drone controlling system which captures the latency needs of current applications [4].

Matthew Obetz, et al. propose a static call graph that captures the stateless and event-driven characteristics of serverless functions and provide a detailed method for their construction and demonstrate the applicability of such static call graphs in resource estimation, information flow, dead code elimination, security analysis and documentation purposes and incorporation of these features greatly benefit quality of service provided by native cloud platforms [5].

Müller, et al. Perform a comparative study of various hypervisors and containers which mainly includes Linux containers (LXC) which is also the one used in Docker, gVisor secure containers and the microVM hypervisor Firecracker that uses Kernel Virtual Machine (KVM) under the hood [6]. They show that these hypervisors move much of the functionality out of the kernel and yet execute substantially more code than a conventional linuxOS. The comparison concluded gVisor to be of light-weight when compared to firecracker.

Anjali, et al. studied various aspects of the firecracker environment including the serial handling of IO operations in the Firecracker environment causing a challenge to exceed 13,000 IOPS [7]. They also brought out the compromise in the durability of a write operation of a high performance caused by not enabling the flush-to-disk mechanism in Firecracker.

Wang, et al. surveyed the existing serverless platforms which perform file processing operations and the way they manage the load from users [8]. They identified the ways in which an instance state would be preserved when used by a platform to execute operations to deliver the desired services.

Shafiei, et al. made some observations regarding the requirements for serverless computing systems in terms of metrics like cost, scalability and fault tolerance [9]. They identify the limits set on runtime resource requirements of serverless code including the number of concurrent requests, and the maximum memory and CPU resources available to a function invocation.

Baldini I, et al. closely inspected the AWS Lambda platform and pinpointed some issues in the same including the increasing error rates and latencies experienced by the AWS Lambda users [10]. They concluded that this was due to a lack of strong service level agreements. They also implied that as a relatively new platform, AWS Lambda is not yet offering any uptime or operational level service. However the current status of AWS lambda has been widely viewed to have improved in levels of satisfaction among the users.

Adzic, et al. perform an industrial survey of early adopters which depict cost reduction in hosting in Lambda deployment [11]. Case studies related to Mindmup - a collaboration platform and Yubl - a networking company were performed to justify the same.

Ana, et al. analyse different storage services used in serverless with respect to performance and efficiency of said storage solutions and emphasize the need of a pay-as-you-need storage solution [12] to support high throughput parallel applications and infer that for applications that need throughput over latency Flash offers a good performance to cost solution and show promising scope for fine grained and elastic storage solutions.

Existing solutions demonstrate the advantages of using Firecracker, compare lightweight microVMs with containers and highlight the use and drawbacks of each. Existing clusters use Virtual Machines as a part of Kubernetes clusters that have larger start-up times (of the order of seconds to minutes) compared to a microVM such as Firecracker (of the order of milliseconds). The proposed solution runs containers on top of these microVMs that form a cluster providing robustness to a cluster setup that requires high availability.

III. METHODOLOGY

Figure 1 shows the architecture that is built for running the Firecracker microVMs as Kubernetes nodes.

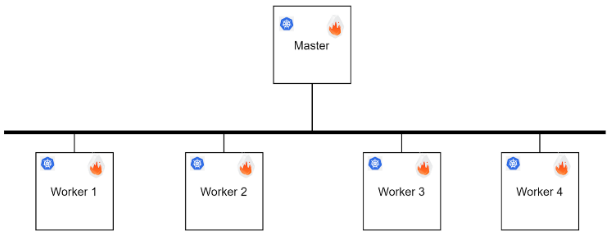

Figure 1: Firecracker microVMs are connected to a virtual bridge via virtual network taps

The following steps were followed for the bring up of Firecracker MicroVMs as a Kubernetes cluster.

A. *Host Server Setup*

The server on which the virtual machines are hosted needs to support the demands of the guest virtual machines with respect to processing, memory bandwidth, network throughput and robustness. The AWS i3 instances deem a good fit for this purpose. They offer bare metal instances that provide access to memory and processing elements that are often encapsulated from the user in traditional Amazon EC2 instances. Some specifications include Broadwell processors with 36 hyper-threaded cores, more than 500GB physical memory and Elastic Network Adapter(ENA) based networking that provides 25Gbps of network bandwidth, thus making it a suitable choice to run workloads that demand rapid scaling.

The server has to support Kernel Based Virtualization (KVM) as Firecracker uses KVM under the hood to create micro Virtual Machines where the kernel acts as a hypervisor that hosts multiple Virtual Machines. KVM makes it possible for the host linux machine to acquire components used by operating systems to schedule MicroVMs as linux processes.

For establishing network connectivity between the MicroVMs and the host, between MicroVMs themselves, or between the MicroVMs and the internet, it is important to make use of adequate networking tools in the server. For Firecracker, setup of tap devices in the host enables their usage as virtual network interfaces. The host is able to connect to all the created taps whereas specific taps can be set as virtual interfaces of the MicroVMs during boot up.

B. Firecracker microVM

The micro Virtual Machines(microVMs) are created via API requests to the Firecracker Virtual Machine Manager API endpoint whose format is specified in the OpenAPI format. Configurations such as the number of CPUs, memory, network interfaces, CPU oversubscription, file-backed storage, logging and other metrics are performed via these API requests that are listened to by a socket created for every microVM that gets created. Rate Limiters can be configured for devices to restrict bandwidth and IO operations and to prevent overuse of the underlying resources of the host by any single microVM in a distributed setting. The microVMs require an uncompressed linux kernel image with version specifications and an ext4 root filesystem containing all the necessary files required for the microVMs execution.

The microVMs were configured for 4GB RAM with 4 vCPUs initially and later changed to different settings while running the microbench, had 1 network interface in the form of a virtual tap device, a blocking store containing the test-bench folder for later use without any rate limiters. The microVMs require an uncompressed linux kernel binary image that can be obtained by building it from the official Linux Repository and making sure that the kernel version is at least 4.14 as lower versions are not supported by Firecracker. The root filesystem was created using an Ubuntu system with all necessary requirements for the container environment installed for the workload's execution. Docker was used for the container runtime environment and Kubernetes for orchestration and all dependencies were pre-installed into the root filesystem. Currently firecracker only supports the procedure of supplying a root file system that contains all the libraries and files necessary for the workload execution along with a linux kernel binary that acts as the kernel for the microVM. It doesn't support any other image format such as the KVM's QCOW2(QEMU Copy on Write 2) images or any such format. There is also a requirement for the kernel binary to be in executable link format (ELF), but as the Linux Repository build is in such format and most binaries are required to be in ELF the requirement was satisfied. Tap Interfaces were created for networking which is a virtual layer 2 device that is used to provide networking capabilities. Networking is necessary for exchanging data between the different host VMs during execution and for Kubernetes to manage the microVMs essentially forming a cluster of microVMs that are extremely robust as they have very low startuptimes(of the order of few milliseconds) and if any node in the cluster goes down a new one can be brought up in its place in essentially no time by the Kubernetes control plane. It is to be noted that since all the microVMs are executing in a single host server it is not fully safe from attacks as any attack on the host renders the entire cluster to shut down. A better approach is to deploy multiple host bare metal instances that themselves deploy multiple microVMs forming a single Kubernetes cluster tapping the complete distributed potential of the system proposed and our single host server is just meant to be a demonstration of the capabilities of the microVM cluster architecture. Another benefit of such a distributed system apart from robustness is that of taking advantage of parallelism in the form of Massively Parallel Processors(MPP) and High Throughput Computing(HTC) that provide job level parallelism for data intensive workloads and is a candidate for further exploration.

C. *Container Environment setup*

Once the MicroVM is successfully launched in Firecracker, it needs to be configured to bring up a container. Using Docker as the container runtime environment, Kubernetes manages the container environment in the cluster.The master node is exclusively used for managing the worker nodes as a part of the Kubernetes control plane and it doesn't involve any computation. The control plane manges all nodes so that reliability of services is guaranteed and the desired state is maintained. One of the Firecracker MicroVMs is arbitrarily chosen as the master node in the Kubernetes cluster and others are chosen as workers. Multiple worker nodes were deployed that run Kubeadm, kubectl and kubelet which are installed in the Firecracker MicroVMs during the creation of the root file system. These nodes supported the process pods, the smallest deployable unit of computing in Kubernetes which manage the containers that run in them and maintain replica nodes if required. No replicas were used as our objective was to test this environment using the testbench mentioned in [3] to assess its use in distributed setups. Replicas can be set up easily in the configuration that describes the cluster. The master node then initiates the kubernetes cluster and a pod network is created. Every worker node is registered to the master, bringing up a Kubernetes cluster and then each of these workers run multiple docker containers depending on the pod configuration. A container is deployed in a pod, where it uses the host network settings to handle requests from users and is also the setup for measuring network bandwidth. Port forwarding in the MicroVM allows incoming traffic to be directed to a given container, so that multiple containers can simultaneously respond to requests from users outside the host network.

Some docker containers require additional settings to be configured in the MicroVM. Enabling huge pages in the MicroVM is essential to running containers that have large database requirements thereby reducing latency. A Linux machine conventionally has a normal page size of 4Kb, forcing a container to run with the same page size. However, enabling huge pages provides flexibility for the container to use a page size thereby reducing page faults and additional IO overhead that follows. Note that huge pages need to be set up such that the underlying memory can accommodate it.

D. *Running the benchmark*

The performance of our cluster environment was tested on FunctionBench which is a workload suite for serverless function evaluation in [3]. It includes 4 categories of workloads - Microbenchmark, which is a series of arithmetic and logic operations to test the computational capabilities of the environment, IO operations and network bandwidth. Network bandwidth is tested using iperf3 that connects different microVMs and data is passed between them.

The image processing application that processes image and video workloads that imposes moderate workload on the compute capability as well as moderate to high IO operation overhead.

The ML Model Training workload aims to demonstrate the utility of our cluster in machine learning tasks as a possible solution to training models that require large datasets and this is the workload that has large data and high network overhead. The details regarding model training and dataset can be accessed in [3].

The last of the workloads include serving arbitrary machine learning and deep learning models that are used to capture the effectiveness of the environment of running a model for a learning task.

The workbench functions that were used to test AWS Lambda were recreated so that it could be executed in our clusters with the program files saved in the root file system and the image and video datasets were provided through a shared backing store. The memory settings were configured to match the AWS Lambda's execution metrics and the results of execution were compared.

IV. IMPLEMENTATION

Firecrackers requires and uncompressed kernel binary that is at least 4.14 that is in Executable Link Format(ELF) which was obtained from Linux's official repository and a root file system was created with docker, kubectl, kubeadm and kubelet and other dependencies pre-installed with the benchmark programs. The datasets containing images and videos present in the benchmark's repository were provided through a shared backing store. Networking was configured with tap devices for each microVM that provided connectivity between the virtual machines. A Kubernetes cluster was set up with one master and 4 worker nodes with the worker nodes configured to have 8vCPUs with memories varied from 128MB to 3008MB for comparing latencies. No rate limiters were set for the network interfaces and were allowed to use the complete bandwidth available. The microVM's configurations served as API requests to the Firecracker Manager that processed these configurations to create microVMs. The programs that were part of the FunctionBench were run with the latencies logged into a result file and the results were compared.

V. RESULTS

The microVMs were configured with physical memories ranging from 128MB to upto 3008MB and the microbenchmark workload was executed and the latencies were measured and compared to the latency of AWS Lambda's execution results obtained from [3]. The latencies show that the performance of our single host cluster is similar to that of AWS Lambda for the given memory configurations, Figure 2 shows that latencies of the cluster are within 5 ms of the Lambda execution latencies for video processing applications. Figure 3 shows that latencies of the cluster are within 6 ms of the Lambda execution latency for 128MB of memory and reduces as the memory increases for model serving applications. The almost identical results emerge from the fact that AWS Lambda uses Firecracker microVMs for its serverless execution service and that AWS Lambda and serverless deployments run in a similar environment as that of ours.

The cluster of microVMs are very robust with low node startup times owing to the light-weight microVMs that startup in the order of milliseconds due to the reduction of the attack surface by design of the firecracker VMs.

In both the video processing and model serving workloads it is seen that for smaller physical memories AWS Lambda performs slightly better than our cluster owing to the excess overhead of containerization and container orchestration tasks. But as the workloads run on increasingly larger physical memories the performance is identical to AWS Lambda.

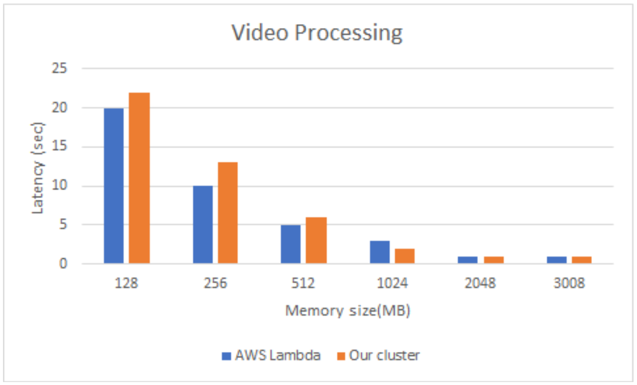

Figure 2: Comparison of Latencies of Proposed Cluster with AWS Lambda for Video Processing

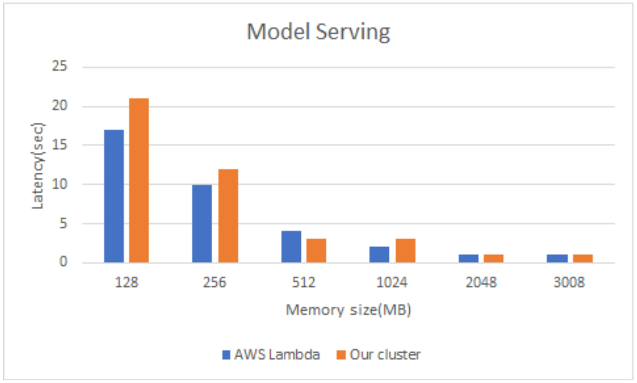

Figure 3: Comparison of Latencies of Proposed Cluster with AWS Lambda for Model Serving

VI. CONCLUSION AND FUTURE WORKS

The proposed work has demonstrated the utility of microVMs as nodes in a Kubernetes cluster and how their quick startup times prove to be useful in providing a highly available distributed system. The performance of the cluster on various computationally intensive tasks and tasks that require network bandwidth were compared with that of serverless functions provided by AWS Lambda, verified through FunctionBench.

The utility of our setup has to be tested in distributed computing as a High Throughput Computing(HTC) system with multiple bare metal instances running clusters of microVMs connected as a single large cluster to achieve task level and request level parallelism which may prove useful in business and scientific computation.

VII. REFERENCES

[1] J. Park, H. Kim and K. Lee, "Evaluating Concurrent Executions of Multiple Function-as-a-Service Runtimes with MicroVM," 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), Beijing, China, 2020, pp. 532-536, doi: 10.1109/CLOUD49709.2020.00080.

[2] I. Pelle, J. Czentye, J. Dóka and B. Sonkoly, "Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS," 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 2019, pp. 272-280, doi: 10.1109/CLOUD.2019.00054.

[3] J. Kim and K. Lee, “Practical cloud workloads for serverless faas,” inProceedings of the ACM Symposium on CloudComputing, ser. SoCC ’19.New York, NY, USA: ACM,2019.

[4] Poth A., Schubert N., Riel A. (2020) Sustainability Efficiency Challenges of Modern IT Architectures – A Quality Model for Serverless Energy Footprint. In: Yilmaz M., Niemann J., Clarke P., Messnarz R. (eds) Systems, Software and Services Process Improvement. EuroSPI 2020. Communications in Computer and Information Science, vol 1251. Springer, Cham. https://doi.org/10.1007/978-3-030-56441-4_21

[5] Matthew Obetz, Stacy Patterson, & Ana Milanova (2019). Static Call Graph Construction in AWS Lambda Serverless Applications. In 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19). USENIX Association.

[6] Müller, Ingo & Bruno, Rodrigo &Klimovic, Ana & Wilkes, John &Sedlar, Eric & Alonso, Gustavo. (2020). Serverless Clusters: The Missing Piece for Interactive Batch Applications?. 10.3929/ethz-b-000405616.

[7] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. 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 (VEE '20). Association for Computing Machinery, New York, NY, USA, 101–113.

[8] Wang, Liang, et al. “Peeking behind the Curtains of Serverless Platforms.” USENIX ATC ’18 Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, 2018, pp. 133–145.

[9] Shafiei, Hossein &Khonsari, Ahmad & Mousavi, Payam. (2019). Serverless Computing: A Survey of Opportunities, Challenges and Applications. 10.13140/RG.2.2.32882.25286.

[10] Baldini I. et al. (2017) Serverless Computing: Current Trends and Open Problems. In: Chaudhary S., Somani G., Buyya R. (eds) Research Advances in Cloud Computing. Springer, Singapore

[11] Adzic, Gojko&Chatley, Robert. (2017). Serverless computing: economic and architectural impact. 884-889. 10.1145/3106237.3117767.

[12] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, &Animesh Trivedi (2018). Understanding Ephemeral Storage for Serverless Analytics. In 2018 USENIX Annual Technical Conference (USENIX ATC 18) (pp. 789–794). USENIX Association.