SERVERLESS COMPUTING: ADVANTAGES, DISADVANTAGES AND ARCHITECTURES

Jerad Hoy: j24x165
Saidur Rahman: j41s418
Montana State University

November 14, 2020

1 Introduction

Nowadays, Serverless Computing is becoming a popular technology. It is a system that provides backend services on an as-used basis. It also provides users to write and deploy code without worrying about the underlying support. It also handles all the system administration operations to make programmers' life more comfortable. If users use serverless technology for backend services, the user needs to pay based on their usage. They do not need to pay a fixed amount of bandwidth or number of servers because the service is autoscaling [15, 18]. Note that though the name is serverless, physical servers are still needed, but the developers do not need to be aware of it. Figure 1 shows the transformation of services from physical machines to Serverless. Serverless computing also permits much faster deployment of resources than other computing models.

In this project, we present a comprehensive review of the importance of serverless computing, different serverless architectures, performance evaluations and some challenges that need to be resolved. Section 2 presents some background on the history and application of physical servers, virtual machiness, containers, and serverless computing, and Section 3 provides a literature review on serverless computing. In Section 4, we outline some modern serverless computing architectures & their evaluation. We conclude the paper by summarizing and discussing future research in Section 5.

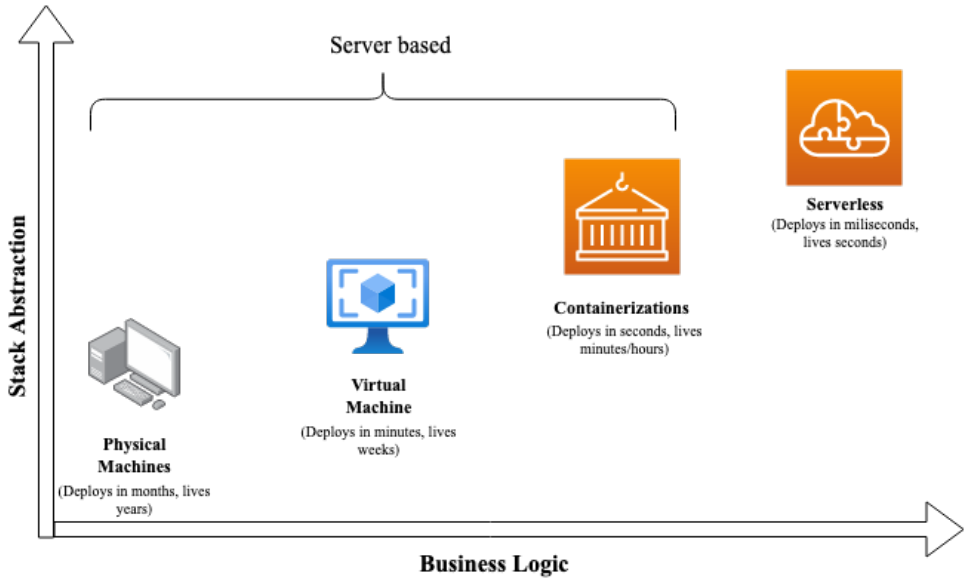


Figure 1: Road to Serverless

2 Background

Serverless computing is a relatively new innovation in the application deployment space. Initially, applications were run on single physical machines, with innovations and virtual machines and containers coming relatively recently and drastically improving commercial applications' deployment.

2.1 Physical Server

The first paradigm in application management involved running physical servers, either on-premise or with a cloud provider such as AWS. Dependencies and applications were installed manually with single applications running on a given server. Scaling these setups involves buying/renting additional physical servers and then configuring and installing dependencies, applications, and other settings.

Security problems arose when multiple applications were run on the same sever. When only a single application was run on a single machine on a single operating system, compute resources were often wasted on low-load applications. Virtual machines were invented to solve some of these problems.

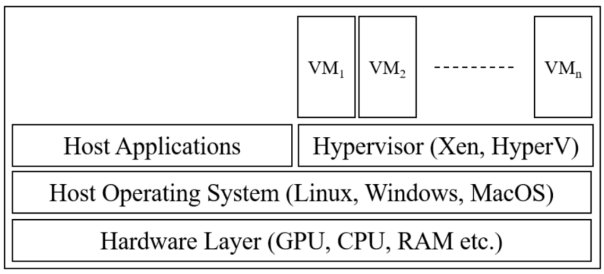


Figure 2: Virtual Machine application deployment architecture [11]

2.2 Virtual Machines

Virtual machines were introduced to allow multiple applications to be run on the same server and isolate them in different OS environments [11]. Figure 2 shows a typical architecture of a server running VM instances. On top of the host operating system, a hypervisor is a go-between for running instructions between the host operating system and each of the running VMs. Each VM consists of an entirely new operating system, which may differ from the host, running an application and its dependencies. Each of the VMs is isolated from one another, providing a strong level of security between applications.

Compared to physical servers, VMs provide a way to utilize more system resources on a single machine and to isolate applications running on different VMs from each other. While a significant gain in the ability to utilize CPU and memory resources is realized, a large amount of overhead is incurred in running each of the guest operating systems on top of the host. This is both in terms of the CPU and main memory required to simulate each guest OS but also in terms of the disk space required to save each guest OS and its state. To address these issues, containers were developed.

2.3 Containers

Containers run more like an application on the host OS. Figure 3 shows a typical architecture of a container running on a host machine. Each container runs in its own address space, allocated resources such as memory, CPU, and network resources by the Cgroups, and utilizes namespaces to provide an abstraction for kernel resources such as PIDs, network interfaces, and hostnames. This is usually done by a container engine such as Docker Engine [13].

There are several distinct advantages of using containers over Virtual Machines. The first is that they are much more lightweight. A container only needs to include the application executable to be run, and the dependencies needed to run it. The container uses the same OS as the host machine, with different containers sharing this OS, while virtual machines

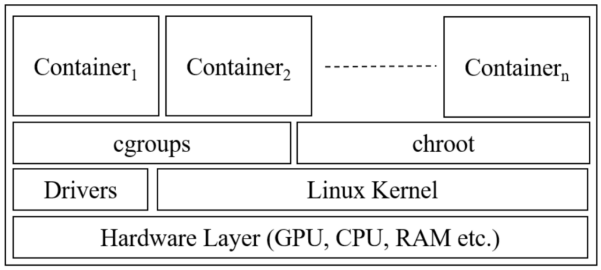


Figure 3: Container deployment architecture [11]

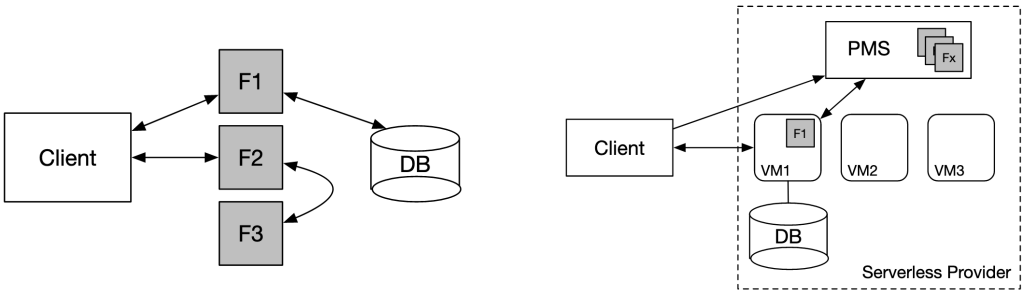
need to store and run several OS instances simultaneously. Virtual machines also have an overhead of the hypervisor, which translates VM instructions to host machine instructions, while containers interact directly with the host OS through system calls. This all results in less resource utilization by containers.

One of the primary disadvantages of using containers for application deployment is security [11]. Each VM has its own set of system resources, application libraries, and kernel resources, running independently. That creates strong isolation between separate VMs running on a single physical server. Therefore, if an application on one of the machines is hacked, the other VMs will generally be unaffected. If one VM fails, other VMs on the machine will likewise also be unaffected, providing greater security and failure redundancy than containers.

2.4 Serverless

The newest application deployment paradigm to be introduced is Serverless computing. In serverless, the OS, infrastructure, and platform in which the application is running, is abstracted away from the developer and managed by a service provider such as AWS [15]. To date, this typically comes in the form of serverless Function as a Service (FaaS). Here, developers simply deploy relatively narrow functions to a serverless platform, such as AWS Lambda, and use network interfaces to trigger and communicate with the function. Figure 4a shows an example of an application deployed to a serverless FaaS platform. The client application talks to one of many serverless functions that in turn may talk to each other or a another resource or service such as a database.

Another common type of serverless offering is Backend as a Service (BaaS) [15]. Here, specific tasks such as user authentication, notifications, or cloud storage are abstracted away so that the client does not have to worry about deployment or management of the servers in which these services run. In typical use, however, and for the remainder of this paper, when



(a) Serverless application example [15]

(b) Serverless provider example [15]

Figure 4: Serverless Computing

we discuss "serverless", we are primarily referring to serverless FaaS.

On the service provider side, functions are typically run on a VM instance, isolating different functions from each other. The provisioning of these instances and communication with the outside world is managed with a Provider Management System (PMS). These PMS systems can spin up, spin down, and allocate resources and network connectivity across machines running these services. This general architecture can be seen in Figure 4b.

The advantages of serverless deployment are primarily related to ease of deployment and management, and cost. Because developers don't have to worry about server setup, operating systems, and dependencies, the ease at which a serverless function can be deployed decreases developer overhead. Scaling the serverless function to more requests often happens automatically without any developer intervention. It reduces the amount of unused compute capacity reserved to handle fluctuations in application use and traffic. This both reduces cost and overhead in handling deployments.

The primary disadvantages of serverless relate to performance and state preservation. Because processes running on a single machine or VM instance can share and persist memory, there are many cases where they can achieve significantly better performance than serverless functions that may run on completely different machines, having to transfer state and communicate over the network. Additionally, when a serverless function hasn't been run in a while, there may be a significant "bootup" time, known as a "cold start", that can add latency to the response and decrease application performance. This paper discusses several modern serverless architectures that aim to ameliorate some of these disadvantages.

3 Related Work

Logistically, serverless application platforms involve multiple clients' workloads running on the same hardware with minimal overhead. However, the container provides weak security and minimal overhead, where the virtualization provides robust security and high overhead. So, Alexandru et al. developed Firecracker, a new open-source Virtual Machine Monitor (VMM) specialized for serverless workloads but generally useful for containers, functions, and other compute workloads within a reasonable set of constraints [4].

Serverless services impose severe restrictions for some applications, such as using a predefined set of programming languages or difficulty installing and deploying external libraries. The author [12] proposed a framework and a methodology to create Serverless Container-aware ARchitectures (SCAR). The SCAR framework creates highly-parallel event-driven serverless applications that run on customized runtime environments.

Hyungro et al. hypothesizes that the current serverless computing environments available can support dynamic applications running in parallel when a partitioned task is executable on a small function instance. To verify the hypothesis, they deployed a series of functions for distributed data processing to address the elasticity and then demonstrated the differences between serverless computing and virtual machines for cost efficiency and resource utilization based on throughput, network bandwidth, a file I/O [10].

Serverless FaaS functions are triggered by users and are provisioned dynamically through containers or virtual machines(VMs). However, the start-up delays of containers or VMs usually lead to relatively high latency of response to cloud users. Moreover, the communication between different functions generally depends on virtual net devices or shared memory and may cause extremely high-performance overhead. The authors [16] propose a lightweight approach to serverless computing named Unikernel-as-a-Function(UaaF) that offers extremely low start-up latency to execute functions and an efficient communication model to speed up inter-functions interactions. They also employ a new hardware technique VMFUNC to invoke functions in other unikernels seamlessly like IPC.

Linux containers offer a lightweight service to load applications into images, put them in isolated environments, and scan periodically to detect vulnerabilities using a vulnerability scanning service. The authors [7] investigate the same architecture for mitigating the threat to a serverless architecture.

Kalev et al. presents a method named Trapeze to provide security on serverless systems using dynamic information flow control (IFC). It encapsulates each serverless function in a sandbox, redirecting all the interactions to the security shim and enforces the global security

policy [5].

4 Different Serverless Architectures & Performance

4.1 SCAR: Serverless Computing for Container-based Architectures

This paper proposed a methodology to create Serverless Container-aware ARchitectures (SCAR) [12] to solve the installation and deployment of external libraries that are not predefined. The SCAR can be used to serve event-driven and highly-parallel serverless applications that run on customized runtime environments defined as Docker images on top of AWS Lambda. Figure 5 shows the architecture of SCAR. SCAR enables users to assign Lambda functions. When an invocation occurs, a container from the Docker image, which is stored in a Docker hub, will be executed. The architecture has two sections: a SCAR client and a SCAR server. The SCAR Client is a python script to validate input, create the deployment including udocker, create the Lambda function with SCAR supervisor, manage the configuration to trigger events from S3 [2] to Lambda. The SCAR server represents the Lambda function's code (Python 3.6) and retrieves the Docker image using udocker from Docker hub. First, the user selects a Docker image available in Docker Hub and generates the Lambda with the user's

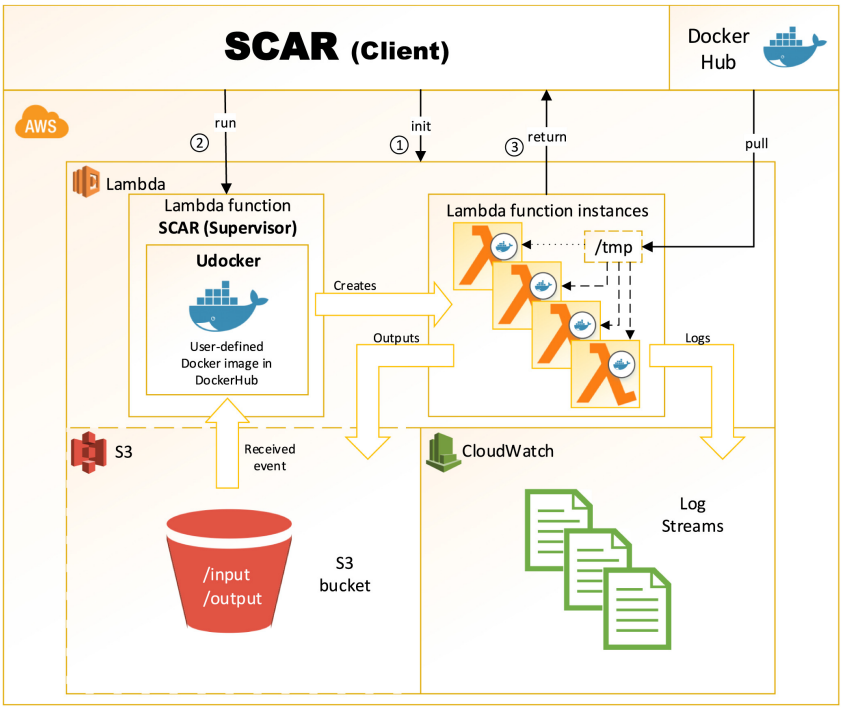
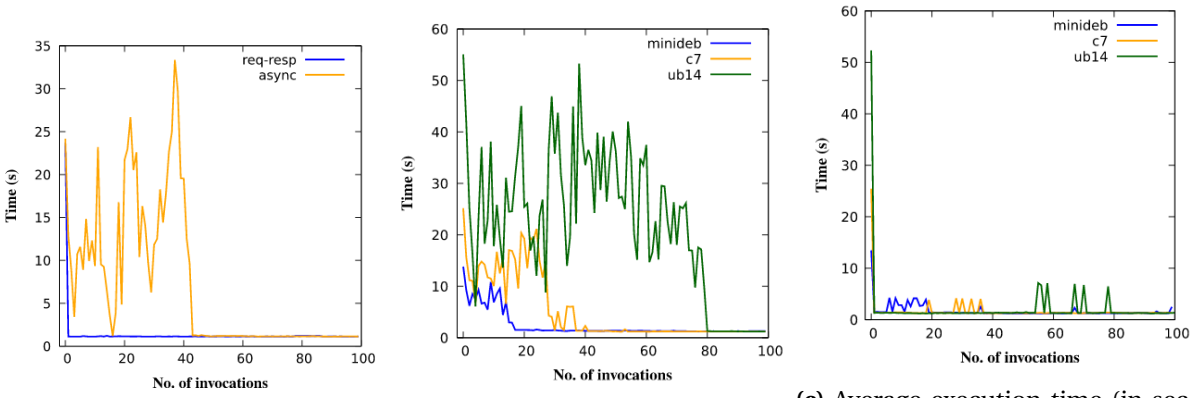


Figure 5: SCAR Architecture [12]



(a) Average execution time(in seconds) for each invocation type [12]

(b) Average execution time (in seconds) for different container sizes (all synchronous) [12]

(c) Average execution time (in seconds) for different container sizes (First invocation req-res and subsequent are asynchronous) [12]

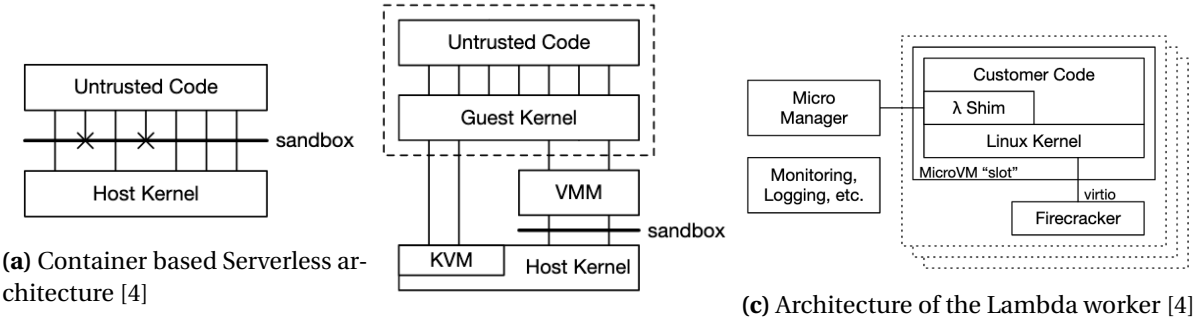
Figure 6: SCAR Performance

configuration. The user can directly invoke the Lambda function where the SCAR supervisor is triggered, which ends up executing a container out of a Docker image and optionally runs a user-defined shell script. Data staging to and from S3 is automatically managed by the SCAR supervisor and diverted the logs into CloudWatch.

Figure 6 shows the performance of SCAR where Figure 6a refers the average execution time for each invocation type. The *req-resp* means the request-response invocation type, and the *async* means asynchronous invocation type. Here, the *req-resp* initially takes 25s to download the container image from the docker hub, and after that next invocations execute instantly. On the other hand, the *async* shows unpredictable behavior for the first 40 invocations. As SCAR caches the file system in the Lambda invocation's temporary space, it is normal to run instantly after the first invocation for the req-resp. However, the *async* shows erratic behavior because all the invocations execute simultaneously. The invoked function cannot find the container cache and download from the docker hub and create a container. Figure 6b shows the average execution time for different container sizes for all asynchronous invocations. The authors want to show how the container size affects the caching performance for all asynchronous invocations. The *minideb* means docker image *bitnami/minideb* (size: 22MB), the *c7* means *centos:7* (size: 70MB) and the *ub14* means *grycap/jenkins:ubuntu14.04-python* (size: 153MB). The graph points out that the increasing container size takes higher time and higher invocations to store the temporary disk space's cache. Figure 6c shows the SCAR's idea to use first req-resp invocation and then asynchronous invocations. As a result, the unpredictable behavior of caching has gone and reduces execution time and invocations to store the container cache.

4.2 Firecracker: Lightweight Virtualization for Serverless

To meet both security and minimal overhead, Amazon AWS team developed Firecracker, a new open-source VMM (Virtual Machine Monitor) for the serverless platform. Firecracker uses the Linux Kernel's KVM virtualization architecture to provide MicroVMs but removes QEMU to build a new VMM. It also supports current Linux hosts and Linux and OSv guests, a REST-based configuration API, device emulation for disk, networking, and serial console; and configurable rate-limiting for network and disk throughput and request rate. To provide security isolation, one Firecracker process runs per MicroVM. Firecracker with KVM gives a new foundation for implementing isolation between containers and functions. Providing an API allows Firecracker to control more carefully the life cycle of MicroVMs. The block device and network devices proposed by Firecracker provides built-in rate limiters via the API. These rate limiters provide limits on operations per second (IOPS for disk, packets per second for network) and bandwidth for each device attached to each MicroVM. For the network, separate limits can be set to receive and transmit traffic. Limiters are implemented using a simple in-memory token bucket, optionally allowing short-term bursts above the base rate and one-time burst to accelerate booting. Figure 7 shows the comparison of the security approaches between Linux containers and virtualization. An untrusted code can directly call the host kernel in Linux containers with the restricted kernel surface area shown in Figure 7a. It also interacts directly with other host kernel services, like filesystems and the page cache. On the other hand, an untrusted code is usually permitted to access a guest kernel in virtualization fully and is allowed to use all kernel features but treat the guest kernel as untrusted shown in Figure 7b. The VMM and hardware virtualization limit the guest kernel's access to the privileged domain and host kernel. Figure 7c shows the Lambda worker's architecture, where Firecracker gives the significant security boundary required to run a large number of different

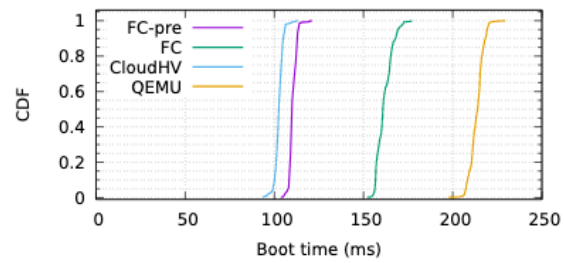


(a) Container based Serverless architecture [4]

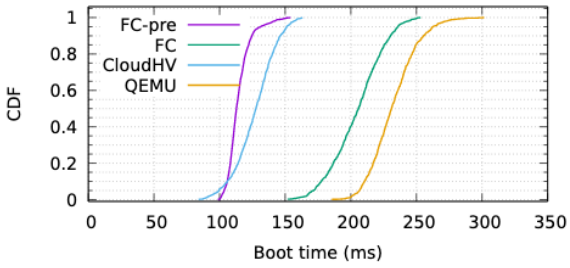
(b) KVM virtualization model [4]

(c) Architecture of the Lambda worker [4]

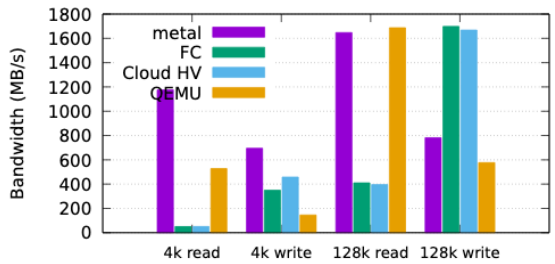
Figure 7: Firecracker Architecture



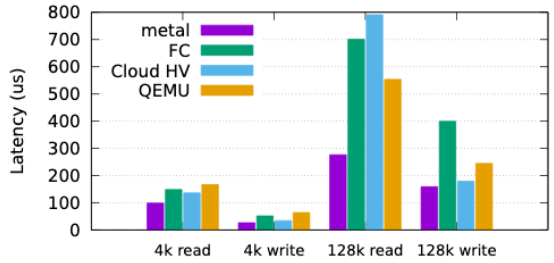
(a) Cumulative distribution of wall-clock times for starting MicroVMs in serial [4]



(b) Cumulative distribution of wall-clock times forstarting 50 MicroVMs in parallel [4]



(c) I/O throughput on EC2 m5d.metal and running inside various VMs [4]



(d) 99th percentile IO latency on EC2 m5d.metal and running inside various VMMs [4]

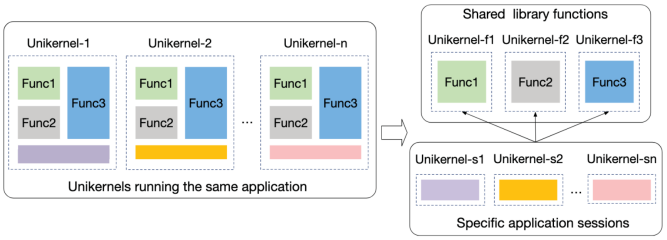
Figure 8: Firecracker Performance

workloads on a single server.

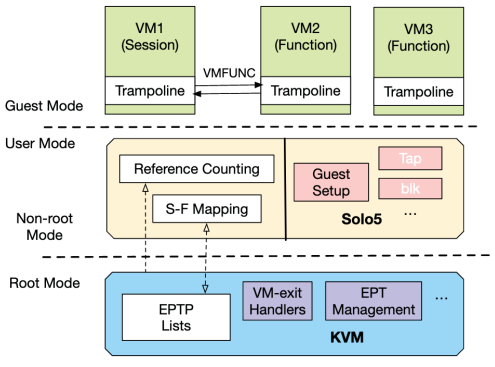
Figure 8 shows the performance of Firecracker compared to other systems. *FC-pre* means re-configured Firecracker where Firecracker has already been configured through the API, *FC* means the Firecracker without configured, *CloudHV* means Cloud Hypervisor which is an open source VMM proposed by Intel [3] and *QEMU* means a system proposed by [6]. Figure 8a refers the cumulative distribution of wall-clock times for starting MicroVMs in serial where *Pre-FC* and *CloudHV* perform better because of VM emulation. Figure 8b describes the cumulative distribution of wall-clock times for starting 50 MicroVMs in parallel and again *Pre-FC* and *CloudHV* perform better. Still for both graphs, *FC* performs better than QEMU. Figure 8c shows I/O throughput on EC2 m5d.metal and running inside various VMs. The graph displays some limitations of Firecracker and CloudHV's current block IO implementation. When high write performance needs, those systems don't perform well because both Firecracker and CloudHV don't use flush to disk and limit to handle IOs serially. Figure 8d displays 99th percentile IO latency on EC2 m5d.metal running inside various VMMs. Firecracker performs pretty well for small blocks however have increased overhead for large blocks like CloudHV.

4.3 UaaF: Lightweight Serverless Computing via Unikernel

The startup delays of VMs or containers create a higher response latency to users and performance overhead. This paper introduces UaaF (Unikernel-as-a-Function) to provide a lightweight approach to serverless computing that offers a low startup latency to execute functions and a communication model for better inter-function interactions [16]. The existing serverless frameworks install all the libraries by an application at startup in every sandbox. As a result, the application finds a high-level redundancy in the memory footprint and high latency in loading libraries. Around 2ms-800ms is needed to load the runtime library when a lambda invocation occurs for the first time, and 80% of the startup latency is the library installation time [17]. To avoid a cold start, the authors abstract fine-grained functions from codes and use these shareable libraries in different unikernels and use a special unikernel named “session” as a proxy of a serverless application. It holds the sequence of library function invocations and represents the serverless application in the task scheduler. As a result, a session can be executed more quickly. Moreover, UaaF installs separate stacks for different sessions within the virtual address space; therefore, the only copy of library functions need to be stored in the memory that decreases memory repetition across the serverless application shown in Figure 9a. Here is the architecture of UaaF shown in Figure 9b. The white modules are proposed by the authors and rests exist in original KVM [9] and Solo5 [1]. An EPTP (Extend Page Table Pointer) list is a special memory page in the KVM kernel space that has at most 512 EPTP entries pointing to the EPT of other unikernels. The list stores EPT entries used by the VMFUNC instructions. Each unikernel has a trampoline that is used to handle the remote calls between unikernels. UaaF has two modules in user mode. The session-function (S-F) mapping module maintains the mappings between the session ID and the EPTP ID

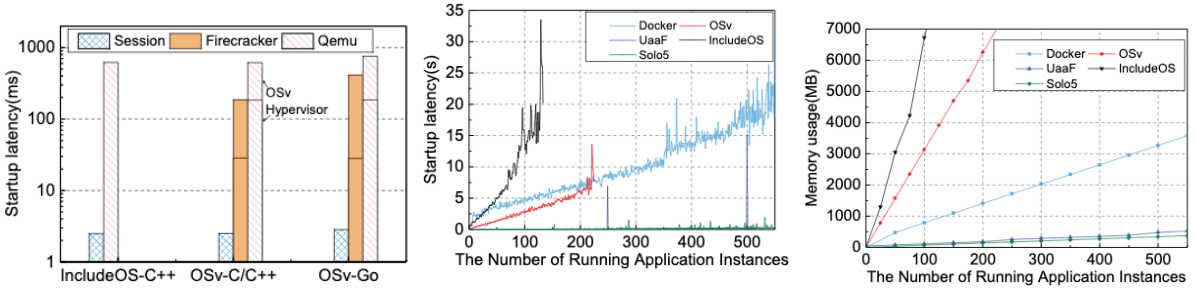


(a) Functions are abstracted form applications and shared by multiple tasks in UaaF



(b) The architecture of UaaF

Figure 9: Unikernel-as-a-Function [16]



(a) The startup latency of sessions and functions in UaaF

(b) The startup latency varies with the number of application instances

(c) The memory usage of different-functions and containers

Figure 10: The performance of UaaF [16]

in a table. The reference counting module stores the number of references to each of the functions. The functions that are not referenced, are destroyed by the UaaF. Figure 10 shows the performance of UaaF. Figure 10a shows the startup latency of sessions and functions in UaaF. As UaaF preloaded the functions in memory, the front-end sessions can be started instantly, and UaaF can also limit the application's startup latency below 3ms, which is faster than Firecracker and other traditional unikernels. Figure 10b shows the startup latency varies with the number of application instances. As each instance consisted of one session function, and multiple library functions were shared among all instances in UaaF, it can scale large instances similar to Solo5. Figure 10c describes the memory usage of different-functions and containers. As we know, IncludeOS, Docker and OSv do not support instances to share execution environments. So, their memory increases linearly with the increasing number of instances. On the other hand, UaaF provides similar memory usage to Solo5 since all library functions were shared.

5 Discussion & Future Research

Serverless computing makes the developers' life more comfortable by proposing a simplified programming and deployment environment. As a result, it is becoming a more popular system to use. In the paper, we dove deep into the general concept of serverless computing, the importance, advantages and disadvantages of serverless computing compared to virtual machine and containers, and then explored some serverless architectures proposed by researchers along with their experimental evaluation.

Though it may be a breakthrough technology that is becoming increasingly popular, some challenges need to be solved to make the technology robust and reliable [8].

- Serverless computing offers limited running time for functions, which can be problematic when an application may take a while to compute [14].
- Serverless short-lived storage needs low latency and high IOPS at a reasonable cost; however, it does not provide long-term storage.
- There is a debate between container-based serverless vs a VM based serverless architecture. Container-based serverless provides low overhead but low security. VM-based serverless provides high overhead but high security. Though Firecracker provides a lightweight virtualization solution, however, it performs poorly when high write performance is needed.
- For cloud functions with many cores like VM instances, multiple tasks can be combined and shared before sending over the network or receiving it instead of doing it serially.

In future, we will extend the work to compare those architectures for real applications. Moreover, Saidur is working on Serverless Computing for his PhD research where he is currently trying to solve the limited runtime issue for serverless platform [14].

References

[1] A sandboxed execution environment for unikernels . `https://github.com/Solo5/solo5`.

[2] Amazon Simple Storage Service Documentation. `https://docs.aws.amazon.com/s3/`.

[3] Intel Cloud Hypervisor. `https://github.com/cloud-hypervisor/cloud-hypervisor`.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[5] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[6] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05. USENIX, 2005.

[7] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef. Leveraging the serverless architecture for securing linux containers. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 401–404, 2017.

[8] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[9] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

[10] H. Lee, K. Satyam, and G. Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450, 2018.

[11] Sumit Maheshwari, Saurabh Deochake, Ridip De, and Anish Grover. Comparative study of virtual machines and containers for devops developers. *arXiv preprint arXiv:1808.08192*, 2018.

[12] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50 – 59, 2018.

[13] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[14] Saidur Rahman, Mike P Wittie, Ahmed Elmokashfi, Laura Stanley, and Stacy Patterson. MicroLambda -Packetized Computation for 5G Mobile Edge Computing. 2020.

[15] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: A survey of opportunities, challenges and applications, 2019.

[16] B. Tan, H. Liu, J. Rao, X. Liao, H. Jin, and Y. Zhang. Towards lightweight serverless computing via unikernel as a function. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2020.

[17] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[18] M. Wu, Z. Mi, and Y. Xia. A survey on serverless computing and its implications for jointcloud computing. In *2020 IEEE International Conference on Joint Cloud Computing*, pages 94–101, 2020.