Challenges and Opportunities for Efficient Serverless Computing at the Edge *

Phani Kishore Gadepalli[1,2], Gregor Peach[2], Ludmila Cherkasova[1], Rob Aitken[1], Gabriel Parmer[2]
[1] Arm Research, San Jose, CA 95134, USA
[2] The George Washington University, Washington, DC, USA
phanikishoreg@gwu.edu, peachg@gwu.edu, lucy.cherkasova@arm.com, rob.aitken@arm.com, gparmer@gwu.edu

***Abstract*—Serverless computing frameworks allow users to execute a small application (dedicated to a specific task) without handling operational issues such as server provisioning, resource management, and resource scaling for the increased load. Serverless computing originally emerged as a Cloud computing framework, but might be a perfect match for IoT data processing at the Edge. However, the existing serverless solutions, based on VMs and containers, are too heavy-weight (large memory footprint and high function invocation time) for operating efficiency and elastic scaling at the Edge. Moreover, many novel IoT applications require low-latency data processing and near real-time responses, which makes the current cloud-based serverless solutions unsuitable. Recently, WebAssembly (Wasm) has been proposed as an alternative method for running serverless applications at near-native speeds, while having a small memory footprint and optimized invocation time. In this paper, we discuss some existing serverless solutions, their design details, and unresolved performance challenges for an efficient serverless management at the Edge. We outline our serverless framework, called aWsm, based on the WebAssembly approach, and discuss the opportunities enabled by the aWsm design, including function profiling and SLO-driven performance management of users' functions. Finally, we present an initial assessment of aWsm performance featuring average startup time (12μs to 30μs) and an economical memory footprint (ranging from 10s to 100s of kB) for a subset of MiBench microbenchmarks used as functions.**

***Keywords*-Cloud computing, Serverless, FaaS, IoT, Edge computing, WebAssembly, Wasm, performance management, SLOs.**

I. Introduction

The surge of Industrial IoT and next generation technologies (such as self-driving cars, smart cities, smart grid, etc.) enable a growing number of novel applications that require data processing systems with new performance characteristics of low (near real-time) execution latency and high quality of service (QoS):

- Imagine a smart city with an automated street and traffic lights system that "senses" the traffic flows and aims to optimize the efficiency of the overall system as well as intelligently react to medical and fire department emergency requests for meeting their needs and Service Level Objectives (SLOs).
- Imagine a world where communicating cars and related data services can alert drivers about dangerous road conditions: "Black ice on the road in front of you (right lane in 200 meters)".

Both of these applications should process a large amount of new data coming from multiple diverse sources in near real-time in order to support the desired functionality. Real-time performance is expected for detection and control in many industrial and enterprise systems. Some scenarios require a response within 10ms [1]. If data analysis and control logic are implemented in the Cloud, such systems will be unable to meet the real-time service requirements. Time-critical applications, such as automotive, simply cannot rely on connectivity and the Cloud – such applications have to be run locally "on-device".

Multiple recent trends and related technological challenges motivate our interest to efficient serverless computing at the Edge. Though Cloud computing provides a good solution for applications designed at human perception speeds, it becomes inadequate for latency-sensitive IoT applications that rely on fast, automated decision making with no human in the loop. To satisfy the performance requirements of such workloads, we must provide a new way of processing their data closer to the source; i.e. providing Cloud functionality at the Edge. Additionally, this approach would be useful for supporting data security and privacy issues. No one wants their data breached, and the risk of that is higher when data is constantly shifted between Cloud and device. Users are increasingly demanding privacy and control of their data, and Edge computing is able to address these multiple concerns. The critical difference and challenge, when comparing Edge computing to Cloud, is that the Edge represents *a resource-constrained environment*, and therefore Edge resources need to be carefully shared between multiple tenants and efficiently managed to support a desirable IoT data processing functionality.

Serverless computing, also known as *Function-as-a-Service* (FaaS), offers a new execution model, which enables users to run their code (a small application dedicated to a specific task; e.g. a single-unit function) without being concerned about operational issues. This way, the user is relieved from any involvement in the server provisioning and resource management. Serverless applications are intended to be event-driven and stateless. For example, FaaS could be instantiated (triggered by the described condition) to execute a predefined function and shut down when finished. In a commercial system, a user is charged on a per-invocation basis, without paying for unused or idle resources. Being supported by unlimited Cloud resources, serverless computing represents a form of utility computing with access to *elastic scaling* via available on-demand computational resources. Since the appearance of Amazon Lambda in 2014 [2], many major cloud providers have offered serverless platforms, including

* This work was mostly completed during P. K. Gadepalli's summer internship in 2019 at Arm Research. G. Parmer, P. K. Gadepalli, and G. Peach are partially supported by NSF No. CNS-1815690 and CPS-1837382.

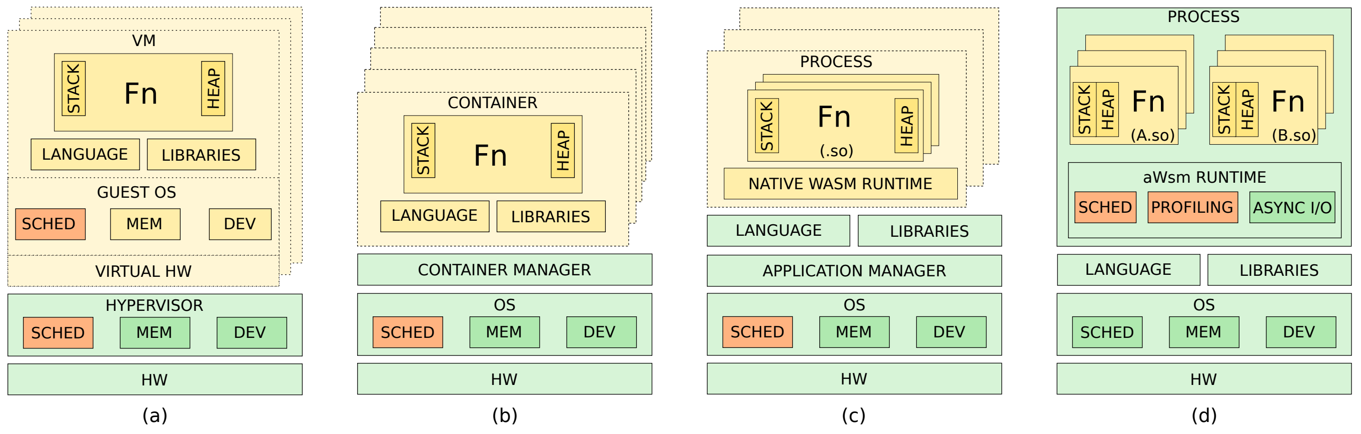

Fig. 1: (a) VM-based serverless. (b) Container-based serverless. (c) Native WASM-based serverless implementation. (d) aWsm approach for efficient serverless execution at the Edge. (*Yellow color* indicates the serverless function sandboxes, *Green color* represents shared layers and services between sandboxes in different approaches, and *Orange color* reflects control modules used for function performance management: scheduling and profiling.)

Google Cloud Functions [3], Microsoft Azure Functions [4], and IBM Cloud Functions [5].

The existing serverless frameworks host function instances in short-lived Virtual Machines (VMs) or containers, which support application process isolation and resource provisioning. These frameworks are somewhat heavy-weight for operating on Edge systems and not efficient for providing low latencies, especially when functions are instantiated for the first time. Startup delays vary across different platforms; e.g. from 125ms for AWS Lambda to 1sec for Microsoft Azure Functions [6]. To achieve better efficiency, these platforms cache and reuse containers for multiple function calls within a given time window; e.g. 5 minutes. Some users send artificial requests to avoid container shutdown. In the Edge environment, the long-lived and/or over-provisioned containers/VMs can quickly exhaust the limited node resources and become impractical for serving a large number of IoT devices. Therefore, supporting a high number of serverless functions while providing a low response time (say 10ms) is *one of the main performance challenges* for resource-constrained Edge computing nodes.

WebAssembly (Wasm) [7] is a nascent technology that provides a strong memory isolation (through sandboxing) at near-native performance with a much smaller memory footprint. Wasm enables the users to write functions in different languages, which are compiled into a platform-independent bytecode. Wasm runtimes have a power to leverage various hardware and software technologies for providing isolation [8] and managing desirable resource allocations.

In this paper, we discuss existing serverless technologies and their related performance challenges, and highlight the recent trend of utilizing the Wasm approach to enable efficient serverless computing at the Edge. We outline the design of aWsm (pronounced as *awesome*) – a novel Wasm-based framework for serverless execution at the Edge, that we are working on. The proposed design opens up a set of interesting opportunities for elaborate profiling and SLO-driven performance management of users' functions. Finally, we present an initial performance characterization of aWsm with average function startup time ($12\mu s$ to $30\mu s$) and its economical memory footprint (10s to 100s KB) for a subset of MiBench microbenchmarks [9] used as functions. The remainder of the paper presents our results in more detail.

II. Existing Traditional Approaches to Serverless

In this section, we describe two existing, traditional approaches used for serverless implementation: VM-based and Container-based frameworks.

A. *VM-based Approaches*

Virtualization leverages software to abstract the physical hardware from guest operating system(s) [10]. A hypervisor or a virtual machine monitor (VMM) allows creation and deletion of virtual machines (VMs) and multiplexing the hardware resources among those VMs. Infrastructure as a service (IaaS) is a cloud framework which offers VMs, storage, network and other compute resources to the customers as a service. These customers manage only their operating systems, storage, network, and applications, but not the underlying cloud hardware infrastructure that enables IaaS. Though VM-like isolation is the current standard for multi-tenant hardware sharing [6], VMs are too heavy-weight for serverless function execution.

Figure 1(a) depicts a serverless function execution in a VM environment. An instance of a function is a combination of its code, memory (stack and heap), its language runtime and the library dependencies (depicted as yellow boxes in Figure 1(a)). Function footprint is significantly smaller compared to a footprint of a provisioned VM, which includes guest OS, virtual hardware resources (provisioned to a VM a priori and not necessarily required by a function).

All the yellow boxes in Figure 1(a) constitute a function in a VM environment. The serverless functions are transient and short-lived. They typically have a small memory footprint. While VMs are often the opposite – requiring gigabytes of memory and significantly longer startup and shutdown times. Additional virtualization overheads for control transfer between the guest OS and the hypervisor as well as for supporting the hierarchical scheduling (orange boxes in Figure 1(a)) are quite significant too.

Since VM provisioning might take tens of seconds, serverless computing providers use inventive optimization techniques for provisioning the function execution environment faster. For example, via maintaining different VM pools: (1) a *warm pool* of VMs (i.e., already instantiated VMs) which could be assigned to a new tenant and his functions, and (2) an *active pool* of VMs that were recently used for executing a function and are saved to serve the function future invocations.

Microsoft Azure Functions [4] provide multiple serverless hosting plans for the customers with an option to host functions in VMs. In a VM-based hosting plan, the customers are charged hourly for having the VM instance running – much like the traditional IaaS billing approach.

There have been other efforts to provide lighter-weight VMs and significantly faster startup times than traditional VMs, by Amazon Firecracker [11] and Kata Containers [12]. Amazon Firecracker is a minimalistic VMM that uses Linux kernel-based virtual machine (KVM) to manage its light-weight microVMs with memory footprint of 5MB and startup latency of about 125ms. It is one of the enabling technology behind the leading serverless cloud platform AWS Lambda [2].

AWS Lambda has two models of isolation and sandboxing of serverless functions [13]:

- Using dedicated AWS EC2 instances (traditional VMs) to isolate different AWS accounts, while individual function invocations are sandboxed using the traditional container approaches [14];
- Using Firecracker microVMs to provide sandboxing environments for different function invocations.

B. Container-Based Approaches

Containers provide a standard way to package the user application's code, configurations, and all the dependencies into a single object. Containers share an operating system installed on the server hardware as shown in Figure 1(b). Containers are run as resource-isolated processes. The containers are isolated from each other using the underlying system using security features such as Linux chroot, control groups (cgroups), and namespaces.

Linux Containers (LXC) [14], Dockers [15], and Windows Containers [16] are different container runtimes, which containarize the applications into processes with a dedicated filesystem, providing a full execution environment, while sharing the host binaries and libraries, where appropriate. Many leading cloud providers like Amazon, Google, Microsoft, IBM and others use this technology for enabling platform as a service (PaaS) and more recently the FaaS [17], [18], [19], [6], [13].

Figure 1(b) depicts a serverless function execution in a containerized environment. Each function (yellow boxes) is packaged into a container image with the language runtime and the dependencies including libraries. The language runtimes and the library dependencies are not shared between multiple functions or even multiple container sandboxes of a single function, thereby impacting the memory footprint of each function and the function latencies. The green layers are shared between different functions and the underlying OS controls container/function scheduling in this framework. The container manager controls creation, deletion, starting, and stopping containers in the serverless environment. In some cases, to enable strong isolation between customers, these containers are hosted in dedicated per customer VMs [2], [4].

We believe that the existing VM/container solutions are highly inefficient for enabling low latency functions and economically utilizing the limited Edge resources.

III. Performance Management Challenges

The *startup latency* (i.e. the latency for initiating a function instance) in a FaaS platform can vary significantly even for the same function, from a few milliseconds to several seconds, depending on a variety of factors. A function execution might be a "*warm-start*", reusing a VM/container from the previous event of the same function, or it might be a "*cold-start*", where a new VM/container has to be launched, the function host process needs to be started, etc. Cold-start latency is one of the major concerns when there is a need to support a predictable function performance. Moreover, cold-start latency depends on many additional factors: the language used for programming the function, the size of the allocated memory, the amount of code, the number and types of libraries and their dependencies, the configuration of the function environment itself, etc. While many of these factors are under a developer's control and are considered for optimization [20], [21], one can achieve only a limited reduction of the startup latency incurred as a result of a cold-start.

The next challenging issue is to *choose the right amount of resources (memory and compute power)* for executing your function. Requesting the "*right*" amount of compute is not always possible in the serverless frameworks. For example currently in AWS Lambda, the user can only provide a memory setting. The minimum allocated amount of memory is 128MB, with 64MB of increments (with maximum memory size of 3GB). Once a user selects a memory size, AWS allocates the CPU quota proportionally. Does the function runtime performance scale proportionally to the memory setting? Multiple papers and experiments [22], [6], [23], [24] show that while, generally, function performance scales with a memory size, it often exhibits inconsistent behavior. What if your function has a small memory footprint but you would like it to finish faster? Current resource provisioning schemas can be inflexible, inefficient, and lead to unpredictable performance outcomes.

Many current serverless computing platforms suffer from a lack of performance isolation [6] between the functions, which makes their performance less consistent and predictable.

Moreover, function performance depends on the type of hardware used for the deployed function instance. The Cloud infrastructure is often heterogeneous [6], [25] as a result of data center infrastructure upgrades, and this heterogeneity may significantly impact performance of different invocations of the same function.

To enable efficient serverless computing at the Edge, a serverless framework has to support performance management capabilities such as:

- Profiling function performance and needed resources (on the underlying hardware) for support of a desirable function response/execution time;
- Guaranteeing predictable function performance;

- Efficiently utilizing multi-core hardware;
- Intelligently scheduling submitted function requests: assigning and managing the right amount of computing resources, with possible preemption of functions already running in the system, to achieve specified function SLOs (Service Level Objectives), e.g., in a form of soft execution deadlines.

In the next section, we outline the design of our serverless solution for supporting these desirable capabilities.

IV. Leveraging WebAssembly for Serverless at the Edge

A. Applying Wasm to Serverless Computing

The existing techniques for enabling serverless in the Cloud computing environments are often inefficient for resource-constrained Edge environments and low-latency support of executed serverless functions. In this section, we describe a recent trend of applying WebAssembly for serverless computing to enable fast function startup, efficient handling of invocations, minimizing memory consumption, and supporting strong security, while also providing fine-grain and accurate billing to the customers.

1) Introducing WebAssembly: WebAssembly (Wasm) [7] is a language-, architecture-, and platform-independent low-level bytecode designed to serve as a compilation target for code written in various high-level languages like C, C++, Rust, and many others for deployment of client and server applications on the Web. Wasm was intended to serve as an alternative to JavaScript for web browser execution while providing strong memory-safe, sandboxed execution environment. A key goal of Wasm is to enable near-native application speed by taking advantage of common hardware capabilities available on a wide range of platforms. Google's V8 [26] and Emscripten [27] enable Wasm execution in a browser by compiling a program written in Wasm supported lanaguage or in WebAssembly Text Format (.wat) [28] to a JavaScript interpreter using the sandboxing and execution environment based on JavaScript VMs.

2) WebAssembly using JavaScript VMs: A serverless framework [29] based on V8, Emscripten, and VM2 [30] demonstrates Wasm ability to enable the serverless function execution at the Edge. Though the JavaScript-backend framework demonstrates significant improvements in startup latencies of Wasm functions, the overall function performance is significantly slower (2x to 5x times of native performance) [29]. This is due to the overheads and complexities in JavaScript virtual machine backend and its sandboxing mechanisms. The serverless functions are often registered once and have thousands of invocations per day, therefore the use of JavaScript JIT compilation introduces the additional (significant) overhead in the execution time of each function invocation.

3) Native WebAssembly Runtimes: There has been a significant effort in adopting Wasm for a native execution, as it is a portable target for compilation of various high-level languages. Wasm standard does not necessarily make web-specific assumptions and there has been significant work to standardize the WebAssembly System Interface (WASI) [31] to run Wasm outside Web. The goal of WASI is to create a system interface that allows the Wasm binaries to be truly platform independent (by being able to run across different native platforms).

There are a number of Wasm runtimes [32] for programs written in different languages: all focused on enabling the native execution of Wasm. Fastly (www.fastly.com) has announced Lucet [33], [34] that provides a compiler and runtime to enable native execution of Wasm applications, and Lucet can instantiate a Wasm module within $50\mu s$, with just a few kilobytes of memory overhead. Fastly's Terrarium project [35] offers a multi-language, browser-based editor and a deployment platform based on Lucet.

These existing systems demonstrate the ability of Wasm runtimes to enable serverless functions have lightweight sandbox isolation and pushes the edge computing beyond the limitations of existing serverless implementations.

Figure 1(c) depicts execution of a Wasm function using a native Wasm runtime. The Wasm functions are Ahead-of-Time (AoT) compiled to enable the compiler and the language runtimes to leverage the platform software and hardware mechanisms [8] for providing strong inter- and intra-sandbox memory-safety. The AoT compilation to shared-objects significantly improves the startup latencies of native Wasm function executions and enables code sharing between multiple Wasm invocations.

The application manager is typically a separate process that enables multiple Wasm-based function runtimes as separate processes on these systems. These native Wasm runtimes enable multiple sandboxes of a function in a single process and are significantly lighter weight than VMs/containers (yellow boxes in Figure 1(c)). The native language runtime, the libraries, and the OS are shared among multiple functions and sandboxes (shown as green boxes in Figure 1(c)). Often the performance and execution properties of different function invocations are not controlled in these existing Wasm runtimes. A system interface support in the native Wasm runtime in Figure 1(c) enables the serverless functions to leverage the OS functionality. However, it is important to note that the OS scheduler has control (orange box in Figure 1(c)) over functions executing as seperate processes, and therefore the performance of different functions.

B. The Novel aWsm Framework and Its Approach to Serverless Performance Management

The aWsm is a native Wasm compiler and a runtime framework. The aWsm compiler is Rust-based and uses LLVM intermediate representation (IR) to enable hardware- or software-based sandbox isolation checks. The aWsm runtime leverages the AoT compiled Wasm shared-objects to enable multiple functions and their invocations within a single process. Figure 1(d) depicts the aWsm runtime and its sandboxing mechanism. Multiple functions (e.g., A.so and B.so) are dynamically loaded into the aWsm runtime. The yellow boxes in Figure 1(d) are instances of different functions in separate light-weight Wasm sandboxes. The language runtime, library dependencies, and the OS functionality are shared and available to different functions running in the aWsm runtime. The runtime *bypasses the Linux kernel* to take full control over the execution properties of different function invocations. Asynchronous, event-based I/O is used to enable the sandboxes to wait on I/O completion while allowing different sandboxes to efficiently utilize the CPU.

Function Execution Control in aWsm: The complexity of Linux kernel has been growing quadratically with time [36] which has a significant impact on the performance of the applications and security of the system (increased attack surface). The pervasive integration of Internet of Things (IoT) devices makes the traffic at edge devices significantly higher and susceptible to the edge system's processing delays. The overheads in the Linux kernel interactions significantly impact the serverless computation and therefore the end-to-end latency for every invocation. This forms a strong motivation for aWsm runtime to take a drastic approach, and provide its own isolation and scheduling facilities for the management and servicing of function executions.

Providing scheduling of requests and function execution along with sandbox-based isolation has multiple benefits. Some of the benefits include,

- *Flexible scheduling policy:* The runtime has the complete power in scheduling the sandboxes. The aWsm runtime scheduling can be configurable (is independent of the underlying OS scheduler) to facilitate different temporal guarantees to functions of various importance and criticality requirements. The orange box "SCHED" in Figure 1(d) is a part of the aWsm runtime unlike Figures 1(a), (b), (c), where the OS scheduler controls the execution properties of the function sandboxes.
- *Fast sandbox switches:* The sandboxing mechanism in aWsm enables fast context-switches between sandboxes at user-level thus removing the system call overhead in the case of co-operative sandbox switches similar to M:1 user-level threads (or green threads) [37].
- *Control over preemption:* The runtime scheduler leverages POSIX "signals" to control the preemption in the sandbox executions. This enables the runtime to have fine-grained control over the function execution, to provide accurate accounting and performance profiling of multiple functions and their invocations. The OS-level software interrupts (like SIGALRM and SIGUSR1) are leveraged by the aWsm runtime to enable control over preemption.
- *Multi-core execution control:* Sharing of resources and locks in the Linux kernel could induce significant cache-coherency and inter-processor interrupt (IPI) interference in a multi-core execution [38] which could directly impact the function execution latencies. The aWsm runtime partitions the function invocation workload across cores thus limiting sharing of resources and providing better predictability properties.
- *Management control:* The single shared aWsm runtime captures the performance characteristics of different function executions in its profiling module ("PROFILING" box in Figure 1(d)).

We believe that the ability of performance profiling and control over execution properties of multiple functions will enable aWsm runtime to provide strong SLO-driven latency and predictability guarantees to functions at the edge.

C. Initial Performance Evaluation of aWsm: Startup Latency and Memory Footprint

To evaluate the function startup performance and memory footprint characteristics in our aWsm prototype, we use a subset of benchmarks in MiBench [9] and a *null* function.

The *null* function is a simple C program that returns from the *main* and has no other library dependencies.

The benchmark applications are used as a proxy for serverless functions, without the external interface for request/response. We randomly select a subset of MiBench benchmarks from different categories with the same input data set from [9]:

- *basicmath* test performs simple mathematical calculations with the input data containing fixed set of constants.
- *bitcount* tests the bit manipulation abilities by counting the number of bits in an array of integers and the input data is an array of integers with equal numbers of 1's and 0's.
- *crc3* performs a 32-bit Cyclic Redundancy Check (CRC) on a file. The input file is a large speech file.
- *qsort* sorts a large array of strings into ascending order using quick-sort algorithm with the input data containing a small list of words.
- *sha* benchmark is a the SHA secure hash algorithm that produces 160-bit message digest for a given input. The input data is a large ASCII text file of an article found online.
- *stringsearch* benchmark searches for given words in phrases using a case insensitive comparison algorithm.
- *susan* benchmark is an image recognition package and was developed for recognizing corners and edges in brain MRI. The input data is a complex picture.

As a testbed environment we use Intel i5-5200U CPU running at @2.20GHz with 8 GB physical memory on a single core (all the experiments were executed on a single core).

Benchmark Name	Memory Footprint	Startup time (in μs)		Cold-start (in μs)
		Average	95%-tile	
null	18KB	12	15	193
basicmath	90KB	29	39	186
bitcount	120KB	29	39	185
crc32	103KB	30	35	395
qsort	148KB	19	22	260
sha	103KB	26	30	187
stringsearch	179KB	26	29	297
susan	177KB	20	24	209

TABLE I: Memory footprint and function invocation startup times over thousand iterations of different benchmarks in aWsm.

Table I shows the average startup cost of different benchmarks and the memory footprint of these benchmarks compiled and executed in aWsm. The startup latency of each benchmark is a measure of time taken to create a sandbox, allocate a fixed size stack (32MB), linear memories (16MB initially, 4GB maximum), and populate the linear memory. The function sandboxes are allocated a dedicated stack memory and reserved a contiguous virtual address space of maximum of 4GB for linear memory using `mmap`. The linear memory is initially allocated 16MB and expands within the reserved contiguous virtual address space.

Discussion: As shown in Table I, the memory footprint of a *null* function is 18KB, and the memory footprints of selected benchmarks range between 90KB to 180KB, reflecting much smaller memory footprints of serverless functions compared to the size of the VM/container execution environment required to capture all the necessary dependencies.

The average startup latency of a *null* function is approximately 10μs, and within 20μs-30μs for other selected benchmarks. Additionally to the average startup latencies Table I shows the 95%-tile of startup times, which falls within 22μs-39μs for the selected benchmarks. This narrow latencies range reflects a much higher performance predictability in the functions performance delivered by the aWsm serverless framework.

In the Edge environment, the aWsm runtime loads a predefined set of functions and their library dependencies at the startup, and therefore, a function invocation will only incur the "cold-start" latency once: for the first-time after the start of the aWsm runtime. The "cold-start" latencies of functions in aWsm are significantly smaller (0.2ms-0.4ms) than for traditional VMs/containers cold-starts.

These numbers reflect a promising aWsm runtime performance. We believe that aWsm runtime together with customizable profiling, modeling, and scheduling intelligence will provide an attractive implementation option for an efficient serverless execution at the Edge.

V. Conclusion and Future Work

Serverless computing is emerging as a new compelling framework for IoT data processing at the Edge. In this paper, we discuss the state-of-the-art serverless platforms, based on VMs and containers, and list the number of challenges related to running these solutions at the Edge. Then we analyze the latest serverless implementation based on WebAssembly, which is proposed as an alternative method for running serverless applications at near-native speeds, while also having a small memory footprint and optimized invocation time. We outline our design of aWsm – a novel Wasm-based framework for serverless execution at the Edge. aWsm bypasses the Linux kernel and takes a full control on functions executions and their management: including profiling, scheduling, and resource allocations. The proposed design opens up a set of interesting opportunities for elaborate and easily customizable SLO-driven performance management of users' functions.

In our future work, we plan to further utilize these opportunities: *(i)* build a profiling module to track and analyze functions' performance, *(ii)* experiment with different scheduling strategies to support the SLO-based classes of functions, *(iii)* provide customized interfaces to offer a variety of function scheduling and admission control policies, and *(iv)* leverage Arm hardware features to improve memory-safety of function sandboxes and their execution performance.

References

[1] "White Paper of Edge Computing Consortium, https://www.iotaustralia.org.au/wp-content/uploads/2017/01/White-Paper-of-Edge-Computing-Consortium.pdf," 2017.

[2] "AWS Lambda: https://aws.amazon.com/lambda/," 2019.

[3] "Google Cloud Functions: https://cloud.google.com/functions/," 2019.

[4] "Microsoft Azure Functions: https://azure.microsoft.com/en-us/services/functions/," 2019.

[5] "IBM Cloud Functions: https://cloud.ibm.com/functions," 2019.

[6] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *USENIX ATC*, 2018.

[7] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web Up to Speed with WebAssembly," in *PLDI*, 2017.

[8] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, "Position Paper: Progressive Memory Safety for WebAssembly," in *HASP*, 2019.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *WWC-4*, 2001.

[10] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, ""Xen and the Art of Virtualization"," in *SOSP*, 2003.

[11] "Firecracker: https://firecracker-microvm.github.io/," 2019.

[12] "Kata Containers: https://katacontainers.io/," 2019.

[13] "Security Overview of AWS Lambda," 2019. [Online]. Available: https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf

[14] "Linux Containers: https://linuxcontainers.org/," 2019.

[15] "Docker: https://www.docker.com/," 2018.

[16] "Windows Containers: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/," 2019.

[17] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *HotCloud*, 2016.

[18] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," 2019. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[19] W. Wong, "VM, Containers, and Serverless Programming for Embedded Developers," 2017. [Online]. Available: https://www.electronicdesign.com/embedded-revolution/vm-containers-and-serverless-programming-embedded-developers

[20] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *USENIX ATC*, 2018.

[21] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS containers," in *USENIX ATC*, 2018.

[22] "Choosing the right amount of memory for your AWS Lambda Function: https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e ," 2018.

[23] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in *CLOUD*, 2019.

[24] "The Occasional Chaos of AWS Lambda Runtime Performance: https://medium.com/@raupach/choosing-the-right-amount-of-memory-for-your-aws-lambda-function-99615ddf75dd," 2018.

[25] Z. Zhang, L. Cherkasova, and B. T. Loo, "Platform Selection for MapReduce Processing in the Cloud," in *ICCAC*, 2015.

[26] "Google V8: https://v8.dev/," 2019.

[27] "Emscripten: https://emscripten.org/index.html," 2019.

[28] "WebAssembly Text Format: https://webassembly.org/docs/text-format/," 2019.

[29] A. Hall and U. Ramachandran, "An Execution Model for Serverless Functions at the Edge," in *IoTDI*, 2019.

[30] "VM2: https://github.com/patriksimek/vm2," 2019.

[31] L. Clark, "Standardizing WASI: A System Interface to Run WebAssembly Outside the Web," 2019. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[32] "WebAssembly Runtimes: https://github.com/appcypher/awesome-wasm-runtimes," 2019.

[33] "Announcing Lucet: Fastly's native WebAssembly compiler and runtime," 2019. [Online]. Available: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[34] "Lucet: https://github.com/fastly/lucet," 2019.

[35] "Terrarium: https://wasm.fastlylabs.com/," 2019.

[36] R. Koller and D. Williams, "Will Serverless End the Dominance of Linux in the Cloud?" in *HotOS*, 2017.

[37] "Many-to-One / Green Threads: https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqe/," 2019.

[38] P. K. Gadepalli, G. Peach, G. Parmer, J. Espy, and Z. Day, "Chaos: a System for Criticality-Aware, Multi-core Coordination," in *RTAS*, 2019.