An Execution Model for Serverless Functions at the Edge

Adam Hall
Georgia Institute of Technology
Atlanta, Georgia
ach@gatech.edu

Umakishore Ramachandran
Georgia Institute of Technology
Atlanta, Georgia
rama@gatech.edu

ABSTRACT

Serverless computing platforms allow developers to host single-purpose applications that automatically scale with demand. In contrast to traditional long-running applications on dedicated, virtualized, or container-based platforms, serverless applications are intended to be instantiated when called, execute a single function, and shut down when finished. State-of-the-art serverless platforms achieve these goals by creating a new container instance to host a function when it is called and destroying the container when it completes. This design allows for cost and resource savings when hosting simple applications, such as those supporting IoT devices at the edge of the network. However, the use of containers introduces some overhead which may be unsuitable for applications requiring low-latency response or hardware platforms with limited resources, such as those served by edge computing environments. In this paper, we present a nomenclature for characterizing serverless function access patterns which allows us to derive the basic requirements of a serverless computing runtime. We then propose the use of WebAssembly as an alternative method for running serverless applications while meeting these requirements. Finally, we demonstrate how a WebAssembly-based serverless platform provides many of the same isolation and performance guarantees of container-based platforms while reducing average application start times and the resources needed to host them.

CCS CONCEPTS

• **Computer systems organization** → **Client-server architectures**; • **Networks** → *Cloud computing*.

KEYWORDS

Serverless, FaaS, Function-as-a-Service, WebAssembly, Edge Computing, Fog Computing

ACM Reference Format:
Adam Hall and Umakishore Ramachandran. 2019. An Execution Model for Serverless Functions at the Edge. In *International Conference on Internet-of-Things Design and Implementation (IoTDI '19), April 15–18, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3302505.3310084

IoTDI '19, April 15–18, 2019, Montreal, QC, Canada

ACM ISBN 978-1-4503-6283-2/19/04…$15.00
https://doi.org/10.1145/3302505.3310084

1 INTRODUCTION

Enabling next generation technologies such as self-driving cars or smart cities via edge computing requires us to reconsider the way we characterize and deploy the services supporting those technologies. Edge/fog environments consist of many micro data centers spread throughout the edge of the network. This is in stark contrast to the cloud, where we assume the notion of unlimited resources available in a few centralized data centers. These micro data center environments must support large numbers of *Internet of Things* (IoT) devices on limited hardware resources, processing the massive amounts of data those devices generate while providing quick decisions to inform their actions [44]. One solution to supporting emerging technologies at the edge lies in serverless computing.

Serverless computing (also known as *Function-as-a-Service* or *FaaS*) platforms allow developers to create and host single-purpose applications which scale dynamically. These applications are intended to perform a single function on demand and then stop running until needed again. Such platforms provide a significant cost savings to application owners, allowing them to pay for only the times during which their serverless functions are executing. Likewise, hosting providers may realize a substantial savings in resource consumption, since serverless applications do not require constantly running dedicated virtualized or containerized machines [43]. Although originally designed for the cloud, these platforms are well-suited for edge/fog computing environments, where resources are necessarily limited.

Serverless computing platforms can provide a strong complement to the edge, enabling a high degree of multi-tenancy while minimizing resource requirements. By collocating computing resources closer to the devices they support, edge computing allows for much lower latency responses to event triggers. Additionally, data may be filtered and processed locally, leading to a timely response on pertinent events while preventing extraneous data from saturating backhaul links to the cloud. The advantages provided by edge computing are not only desirable but also necessary when dealing with the requirements of large numbers of IoT devices [29].

State-of-the-art serverless computing platforms host application instances in short-lived containers for the purposes of lightweight process isolation and resource provisioning [32]. A container represents a distinct namespace within the underlying OS, complete with its own CPU, memory, and storage limits and without access to or from the rest of the system. When a serverless function is called, the platform instantiates a new container, populates that container with application files and dependencies, executes the application's function, and shuts down after a brief period of inactivity. To achieve more efficient operation, many of these platforms cache container resources and reuse containers if an application is accessed in succession (e.g., the container may stay active if used within a 5 minute window) [50].

While the optimizations employed by container-based serverless platforms do improve performance, these platforms still suffer from inefficiencies associated with container setup costs and resource provisioning. These inefficiencies lead to container startup delays, which in turn impact the time it takes for serverless applications to start. The startup delays incurred by existing serverless platforms stem from two main areas. Foremost, container-based solutions suffer from an issue known as the *cold start problem*. When a serverless function is first called, a new container must be instantiated before the function can be run. This involves provisioning resources, bringing the container to a running state, starting and running the function's application, and returning any output to the caller of the function [48]. Benchmarks show that this startup time can take 300 ms or more, depending on the serverless platform [13]. The second source of delays comes from attempting to provision resources [15]. For example, if a serverless function requires a container with 2 GB RAM and the platform is otherwise fully populated with 512 MB RAM containers, the system must wait for at least 4 containers to stop running before it can claim this resource. These issues are further exacerbated by the fact that application developers employ workarounds to avoid startup delays, including artificially activating functions (to avoid container shutdown) or overprovisioning resources required to run their functions (to secure a container that is harder to evict due to difficulty in acquiring resources) [21] [36].

On a container-based serverless computing platform in an edge computing environment, containers that are long-lived or overprovisioned can quickly degrade performance due to limited available resources. If an edge computing node consists of a single server with high multi-tenancy requirements (e.g., supporting a vast number of IoT devices), persistent containers are not only undesirable but also impractical. Supporting a large number of containerized serverless functions with limited hardware resources means that constant churn must occur, either through organic setup/tear down of containers upon function start/stop or forceful eviction of containers upon resource exhaustion. Such a high rate of churn exacerbates the cold start problem: whereas the slow start time of a container could normally be amortized over its lifetime, in an environment with constant turnover this delay quickly becomes an obvious impediment to low latency performance. When the majority of serverless applications at the edge require several hundred milliseconds to start, the advantages of collocating these applications closer to the end-user are quickly eroded. To fully reap the benefits of serverless platforms at the edge, the impediments posed by existing solutions must be reduced or eliminated while working within the constraints of edge computing environments.

In this paper, we present a new method for running serverless functions without the use of containers. This method leverages WebAssembly, a binary format that provides inherent memory and execution safety guarantees via its language features and a runtime with strong sandboxing capabilities. Our work provides two main contributions. First, we introduce a nomenclature for characterizing serverless access patterns. We use this nomenclature to decompose serverless workloads into their component behaviors to better describe and understand the requirements of a serverless computing runtime. Second, we demonstrate WebAssembly as a viable alternative to the use of containers in serverless platforms through the use of detailed experiments. In doing so, we provide an answer to the question of how to better orchestrate the execution of a serverless platform to fit the low latency/high multi-tenancy requirements of the edge.

2 BACKGROUND

Our WebAssembly-based solution is a middle ground between running unisolated serverless applications as separate processes on the underlying OS and running them inside of containers. WebAssembly provides several language and runtime features that make it well-suited as an alternative to container-based runtimes in serverless platforms. For the sake of completeness, we provide a background on WebAssembly and its supporting technologies in the remainder of this section. Readers already familiar with this information may wish to continue on to the next section, where we discuss how these features inform and support our design decisions.

WebAssembly. WebAssembly (sometimes abbreviated as *Wasm*) is a binary instruction format first announced in 2015 and released as a Minimum Viable Product in 2017 [41]. It is intended as a way to run portable executable code with near native performance in web browser environments. Developers who wish to leverage WebAssembly may write their code in a high-level language such as C++ or Rust and compile to a portable binary which runs on a stack-based virtual machine.

Several language features make WebAssembly well-suited as an alternative to containers in serverless platforms [42]. First, each WebAssembly module is designed to be memory safe and execute deterministically within a sandboxed environment, providing per-application isolation. Second, a module's memory is laid out linearly and fixed at compile time, which prevents many well-known security vulnerabilities and errors arising from direct memory access. Third, developers may port existing code intended to be compiled natively to a WebAssembly compilation target with minimal effort. And finally, since WebAssembly is both source language and target platform agnostic, a WebAssembly module may be compiled once and moved freely between different hardware architectures with no reconfiguration.

Although WebAssembly shares many concepts with language runtimes such as Java's JVM or .NET's CLR, there are some key differences which make it more acceptable as a container replacement. Unlike Java or .NET, WebAssembly was not meant to serve as a compilation target for one particular language. Instead, it is intended to be as open as possible. This is further supported by the fact that the project is developed by the World Wide Web Consortium (W3C) with support from all major web browser vendors (Mozilla, Google, Microsoft, and Apple), thus increasing the likelihood that it will avoid serving the purposes of a single entity. WebAssembly's runtime was designed from the ground up with security and performance in mind. New language and runtime features are being added in small increments so as to avoid inadvertent design errors which may later lead to security holes, performance and stability bugs, or workarounds required to maintain backwards compatibility [40]. Strong industry support, ease of use by both developers and end users, and solid design decisions should ensure WebAssembly remains a safe, stable, and viable binary format in the long term.

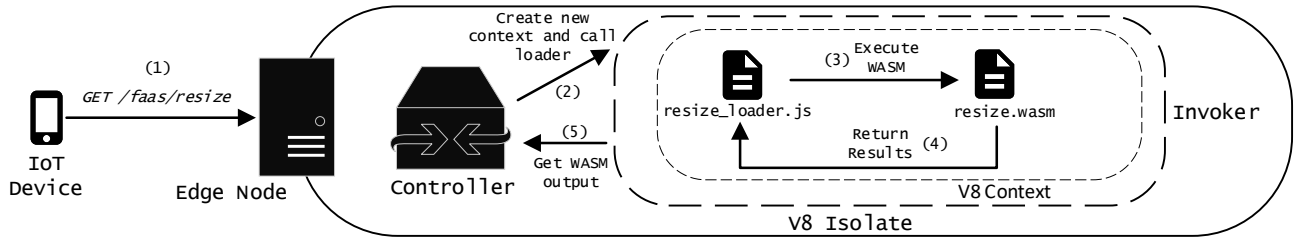

Figure 1: WebAssembly Serverless Platform Prototype Workflow

Runtime. Although one of WebAssembly's primary goals is to run in a web browser environment, such an environment is not strictly necessary. Despite its name, WebAssembly may run in any environment where the runtime contains an execution engine built to its specification. Several of these engines exist today, such as Mozilla's SpiderMonkey, Microsoft's Chakra, or Google's V8. The V8 engine [24] in particular has strong API support and is designed to be embedded in an external program, making it ideal for use in environments other than web browsers.

Applications which embed the V8 engine are referred to as *embedders* [14]. An embedder is responsible for telling V8 which code to execute and for making decisions on how that code may access resources such as the local filesystem or console. This flexibility allows V8 to run almost anywhere. For this reason popular embedders exist on both the client side, such as in the Chrome web browser, and on the server side, such as in NodeJS.

The NodeJS platform provides non-blocking, asynchronous I/O capabilities that allow it to support a large number of concurrent connections to applications written in JavaScript and WebAssembly. It is also highly extensible, providing strong support for additional functionality through its module system. Several built-in and third-party modules offer features such as filesystem access, V8 feature management, extended sandboxing, and web server API creation. As we will detail later, NodeJS forms the basis for our WebAssembly-based serverless platform prototype.

Compiler Toolchain. Emscripten [8] is the preferred method for creating WebAssembly binaries from C or C++. Its toolchain is backed by the *LLVM* [46] modular compiler infrastructure, which it relies on for translating high-level code from languages such as C++ to its own intermediate representation (IR). In addition to code translation, LLVM also provides several tools which aid in tasks such as optimization and dead code elimination. When source code files have been translated to the IR, Emscripten can then parse this representation to generate WebAssembly binary code and any other supporting files.

Since Emscripten's WebAssembly support is backed by LLVM, it is also possible to include external files that have been translated to LLVM's IR, *Bitcode*. For example, if a WebAssembly application developer wishes to leverage a popular library, they may link the library via Emscripten by first compiling it to Bitcode via LLVM's toolchain. This provides increased functionality for more easily porting existing code to WebAssembly modules.

When Emscripten generates a WebAssembly file, it also generates an associated JavaScript file which contains support for loading and executing the WebAssembly code. The features included in this file, such as enabling filesystem support or setting memory requirements, may be influenced by build time directives. Emscripten is generally capable of determining what features need to be added to the generated file based on its analysis of the source code, but for production builds it is safest to explicitly specify all desired options. The loader file generated by Emscripten is responsible for brokering calls to and from the WebAssembly file and instructing the execution engine (e.g., V8) to process and run the code.

3 PROTOTYPE DESIGN

In this section we discuss the design of our WebAssembly-based serverless computing platform prototype, including its goals, limitations, and implementation.

3.1 Design Goals

Our WebAssembly-based solution must meet several goals to be considered a viable alternative to the use of containers in a serverless platform. We address these goals as follows:

Strong Isolation. Our solution must provide isolation in the form of a distinct namespace in which applications can operate. This namespace must include memory and process segmentation (such that an application may not influence another's memory or execution) and filesystem segmentation (such that an application may only read and write its own files). Container-based solutions achieve these goals through the use of Linux namespaces [34], a kernel feature which provides an abstraction of system resources to a per-process granularity. We can achieve an equivalent functionality in WebAssembly through both its language features and its runtime, which we do as follows:

Memory Isolation: WebAssembly uses a memory representation that provides access to raw bytes without allowing direct memory access (i.e., pointers are not allowed). This memory is represented as an array of bytes (a *linear memory*) and is composed of 64 KiB pages. Access to memory occurs via basic load/store instructions which specify an offset into the array. In this respect, an application may not escape its own memory space, as doing so would create an array out-of-bounds condition. Modern WebAssembly execution engines represent these linear memories internally as a JavaScript *ArrayBuffer* [7].

Execution Integrity: The WebAssembly stack machine relies on *structured control flow* for code execution. This design decision prevents erroneous control instructions (e.g., *jmp*, *goto*) from creating irreducible loops or performing unsafe branching to other parts of

the program. As a result, the correctness of an application's control flow may be verified at compile time and its execution is guaranteed to be largely deterministic [27].

Filesystem Segmentation: The WebAssembly standard does not specify any guidelines for filesystem access. Instead, this functionality must be implemented by the runtime that is used to execute a WebAssembly application. Emscripten provides such functionality through its *FS* library and API. This library provides support for several virtual filesystem types, including that of NodeJS [9]. We rely on Emscripten's library coupled with the NodeJS module *vm2* [45] to provide filesystem isolation capability akin to *chroot*. This provides WebAssembly applications filesystem access restricted to a specific directory.

Runtime: Ultimately, it is the responsibility of the execution engine to extend functionality and enforce security for any WebAssembly application it runs. Our prototype relies on NodeJS, which in turn embeds V8 for the execution of WebAssembly code. V8 provides isolation between code executions via the notion of *contexts*. Each context represents a distinct namespace within V8 (similar to a container), such that an application executing within a context may only access resources associated with it. To restrict the execution of each WebAssembly module to its own context, we leverage *vm2*'s fine-grained sandboxing capabilities.

When combined, these language and runtime features of WebAssembly allow us to achieve isolation guarantees similar to those that containers provide to existing serverless computing platforms.

Resource Provisioning. We must provide a way for our solution to limit an application's execution time and maximum memory usage. In container-based platforms, this resource control is achieved via the Linux control groups feature. This feature allows one or more processes to be organized into groups which have their resources monitored and limited by the kernel [33]. We are able to achieve similar resource control in WebAssembly as follows:

Maximum Memory Usage: Each WebAssembly application has a single linear memory available to it. This memory is created with an initial size upon application load and may later be dynamically grown. Current runtimes rely on WebAssembly's JavaScript API to perform this memory creation operation. The *WebAssembly.Memory()* API function allows us to set initial and maximum sizes for the linear memory to be created [11]. We set these values on a per-application basis via our application loader code.

Execution Time: State-of-the-art serverless platforms provide some maximum execution time in which a function may be run before it is forcefully killed. We enforce the maximum runtime of WebAssembly applications via *vm2*'s *timeout* property. This property is configured upon context creation and restricts execution to a configured number of milliseconds.

Configuration directives for maximum memory usage and execution time are set via our application loader. Any errors encountered from out-of-memory or execution-time-exceeded conditions are handled gracefully by our application executor. Through the use of these features, our runtime may properly provision and control the use of resources for each serverless application it executes.

Application Creation and Portability. The use of WebAssembly in a serverless platform should not require application developers an undue amount of work in porting their existing code to the new runtime. Existing solutions provide tools to aid in serverless function development in a variety of programming languages [16]. WebAssembly can provide similar functionality. It is source code agnostic, meaning that potentially any programming language can be ported to WebAssembly. Currently, support for many popular programming languages (e.g., C#, Go, Python, Java) is in active development across numerous open-source projects. Additionally, its most popular toolchains automate much of the task of adapting code during the build process. For example, when translating source code to WebAssembly Emscripten will recognize and emit SIMD.js vector instructions for native SIMD code [10], providing vectorization support that is not tied to any particular hardware architecture.

In our experience with creating several new programs and adapting existing popular libraries for use in WebAssembly, the amount of manual work involved was minimal. For example, adapting OpenCV and several of its dependencies (*libjpeg*, *libpng*, and *zlib*) to WebAssembly required little more than modifying the build configurations for each library.

WebAssembly is also target platform agnostic, meaning that a WebAssembly binary may run on any architecture where a runtime exists. This allows applications to be easily moved between heterogeneous architectures (e.g., x86, ARM, RISC-V) without the need for maintaining multiple builds and toolchains. These features allow developers to create highly portable applications without the overhead of learning an entirely new paradigm.

3.2 Design Limitations

While our WebAssembly-based solution does provide many of the same advantages as containers, it also has some limitations worth mentioning:

Contexts vs. Isolates. Our prototype relies on V8 *contexts* for segmenting code. While the use of contexts does meet our goals by restricting code executions to unique namespaces and limiting access to resources, it does not represent the strongest form of segmentation offered by V8. The *isolates* feature of V8 provides finer grained control over segmentation and resource control. Each isolate contains one or more contexts, runs on a separate thread, and is restricted to a user-defined upper limit on memory. At the time of this writing, we were unable to locate a solution that allowed us to create and control V8 isolates via NodeJS while also meeting all of our requirements. Due to this limitation, we leave the implementation of hosting serverless functions inside isolates to future work.

Performance vs. Native Code. Although the WebAssembly specification does call for code that executes at near-native speeds, the current available runtimes introduce some overhead which can slow execution. WebAssembly continues to make strides in improving execution speed, but at present native code executes much faster than that of WebAssembly. We discuss this limitation further in the Evaluation section.

Hardware-Specific Features/Accelerations. Because WebAssembly is a hardware-agnostic format it lacks support for specific accelerations available via extensions on different architectures. Although some features such as SIMD are supported in a generic

manner, the exact extensions in their entirety are not. The WebAssembly project is currently working to support the most widely used features of the extensions across architectures while maintaining binary portability.

3.3 Implementation

Our goal is to create a prototype which represents basic serverless computing features and demonstrates the use of WebAssembly to execute functions. We model this prototype implementation after core features available in the Apache OpenWhisk platform. This decision is based on the open-source nature of OpenWhisk, which provides for introspection and access to design documentation. Such information allows us to most closely mirror select features of OpenWhisk so that our later prototype evaluation will be as fair as possible.

The OpenWhisk architecture consists of a user-facing reverse proxy web server; a Controller which serves a RESTful API that allows for the control, query, and invocation of functions; an Authentication and Authorization component; a message queue and load balancer; and an Invoker which executes functions within their own Docker containers. Of these features, we implement a Controller and an Invoker in our prototype. The reverse proxy and Authentication/Authorization components add unnecessary overhead and thus are not considered in our prototype. The message queue and load balancer components are unnecessary for the scale at which we evaluate our platform and are also not implemented. OpenWhisk allows us to monitor the performance of the Controller/Invoker components independent of other components, and as such we are able to directly compare their performance to that of our implementation.

We detail the implementation of our Controller and Invoker as follows:

Controller. Our Controller provides access to a RESTful API via a web interface. It is responsible for translating API calls into serverless function invocations and returning the status of these invocations to the caller. The basis for our Controller is the *Express* [17] web framework for NodeJS. Express is lightweight and provides several fundamental web application features which aid us in implementing a performant RESTful API frontend. We create a web server using Express which provides access to API endpoints via two HTTP methods:

- *GET* is used to call a serverless function with no arguments (i.e., the function will execute and return results, if any, to the caller)
- *POST* is used to call a serverless function with one or more arguments. These arguments will be passed to the function unmodified and output will be returned to the caller if available.

Each serverless application on the platform will have its own endpoint. For example, if the platform hosts an application named *send-alert*, this application may be accessed via the `/faas/send-alert` endpoint by issuing the *GET* command to the Controller. Note that a production system such as OpenWhisk would also include some authentication mechanism for calling applications, and applications would be differentiated by some unique ID so as to avoid naming collisions.

It is the responsibility of the Controller to validate any function invocation requests before calling the Invoker. This includes verifying that the function exists and confirming that any arguments passed with the request are valid for that function. If a request is invalid, an error code is returned to the caller. Otherwise, the Controller calls the Invoker with the function's name, WebAssembly file location, memory/filesystem limits, and arguments passed by the caller.

Invoker. The Invoker is responsible for loading and executing a function's WebAssembly representation, as well as gathering/returning any results. This process begins with setting up an execution context and loading the function's WebAssembly code. Currently, the most popular method for loading a WebAssembly application is via the use of its JavaScript API. Emscripten includes support for generating this boilerplate code when compiling a WebAssembly module. Our Invoker extends this code by first converting it to a JavaScript module, which allows us to call the file externally from other JavaScript applications (namely, the Controller). We create an entry function which will be called whenever the Invoker is needed (i.e., when an associated serverless function is called). This function takes as input the name of the WebAssembly file to be executed, an array of input to the WebAssembly file (such as variables passed from clients via the web API), and a callback function that should be called after the WebAssembly code finishes executing.

To guarantee application isolation, a new V8 context needs to be created for each serverless function call. By default, NodeJS does not provide any strong mechanism for isolating untrusted code running within a context, meaning it may be possible for malicious code to escape its current context and adversely affect code in unrelated contexts. To avoid this issue, we utilize the *vm2* third-party NodeJS module which allows for the creation of separate strongly sandboxed contexts running untrusted code. This also enables us to specify which NodeJS features are available to each context with fine granularity. By containing each application's code within a dedicated V8 context custom tailored to its needs, we effectively achieve our goal of separation from the underlying operating system and other processes. An instance of the loader prepares this new context for executing the WebAssembly code by setting memory, execution time, and filesystem limits and then makes the appropriate API calls to instruct the V8 engine to begin executing the function's WebAssembly code within the context. When the WebAssembly code finishes its execution, it returns control to the loader file. The loader file then passes a status message and any output from WebAssembly back to the Controller via a callback function. Finally, the Controller returns this status and any available output to the client.

We illustrate the workflow of our prototype in Figure 1, which represents the call of a serverless function named *resize*. The workflow begins with an IoT device contacting the Controller on an edge node (Step #1). The device calls the function at its API endpoint by issuing a *GET* command to `/faas/resize`. The Controller confirms this request as valid, then calls the Invoker with details of the *resize* function's configuration (Step #2). The Invoker creates a new context configured with memory, execution time, and filesystem limits specific to the function being executed. It then executes loader code

within that context (*resize_loader.js*). The loader code makes the appropriate V8 API calls to begin execution of the WebAssembly code (Step #3, *resize.wasm*) and waits for a return. When execution completes, *resize.wasm* returns the results of its execution to the loader (Step #4). The loader then checks for a valid return code and either returns the output of *resize.wasm* (upon success) or an error code (upon failure) to the Invoker. Finally, the Invoker shuts down the function's context and passes this output back to the Controller (Step #5). At this point the Controller may return an appropriate status to the IoT device and terminate the connection.

4 CHARACTERIZING SERVERLESS FUNCTION ACCESS PATTERNS

Our experimentation suggests that access patterns to serverless functions may be characterized in three basic ways. To describe these access patterns, we provide three scenarios from a real-world campus camera network consisting of approximately 1,000 devices. These devices provide real-time high-definition video streams to the campus police department for use in identifying threats as they occur. Our group collaborates with this department in an ongoing effort to develop an edge computing-based infrastructure which automatically identifies and tracks suspicious objects via computer vision. The scenarios we present center around our design of serverless platforms at the edge of the network which support up to 50 cameras within the same geographic region. These platforms are responsible for first stage processing, determining which video streams contain relevant data that should be further considered by humans in the loop or servers in the cloud. The remainder of this section outlines the three access patterns clients may use when accessing serverless computing platforms.

Single Client, Multiple Access. The first access pattern we describe is *Single Client, Multiple Access*. This pattern is illustrated in Figure 2. We consider a scenario where an object of interest (e.g., a vehicle) has entered a smart camera's field of view. This camera first needs to gather the features of this object before it can proceed with image recognition, and to do so it will query a serverless function to perform feature extraction. During the period where the object remains in the camera's field of view, the camera captures multiple frames in which the object appears. Before the camera can send these frames to the feature extraction function, it must first resize them to a resolution suitable for processing. To do this, it repeatedly queries the *resize* serverless function, sending a different frame with each successive query. The first call to this function incurs the cold start penalty, creating a delayed start to execution, but subsequent calls execute without delay. In this scenario, the camera is a client which accesses the same instance of the serverless function multiple times in close succession. This access pattern allows for already warm resources (such as an already running container or populated cache) to be reused for subsequent requests, thereby increasing performance and decreasing overall response latency. A *Single Client, Multiple Access* pattern where an already warm container may be reused multiple times represents the best case scenario for a serverless platform.

Multiple Client, Single Access. The second access pattern we describe is *Multiple Client, Single Access*. This pattern is illustrated in Figure 3. For this access pattern, we consider a scenario where multiple license plate reader cameras monitor the entrances to a parking deck during peak morning hours. Each of these readers queries the same serverless function a single time to determine whether a captured license plate is of interest to the campus police. In this scenario, the various license plate readers are the multiple clients which each access separate instances of the same serverless function concurrently. Since this access pattern requires a separate instance of the serverless function to be spawned to handle each request, each instantiation will contribute some initial delay to function response latency (e.g., the cold start penalty for container-based systems). A *Multiple Client, Single Access* pattern where all functions incur the cold start penalty represents the worst case scenario for a serverless platform.

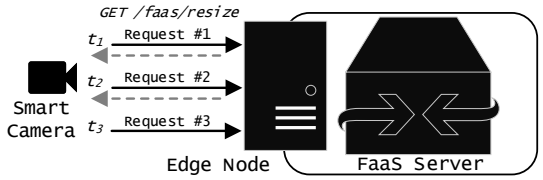

Figure 2: Single Client, Multiple Access Pattern

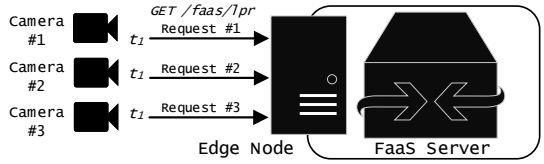

Figure 3: Multiple Client, Single Access Pattern

Multiple Client, Multiple Access. The third access pattern we describe is *Multiple Client, Multiple Access*. This pattern is illustrated in Figure 4. It is a combination of the first two access patterns and is most representative of real-world workloads. We consider a scenario where multiple smart cameras have already gathered the features of objects within their fields of view and now wish to perform image recognition on those objects. To perform image recognition, each smart camera queries a serverless function with the features of an object and receives a response with some classification (e.g., "blue car"). Some cameras will only need to perform image recognition on one object, meaning they will access a serverless function only once. Other cameras will need to perform image recognition on multiple objects, meaning they will access the same serverless function many times in succession. In this scenario, the many cameras are the multiple clients which access separate instances of the same serverless function one or many times. Depending on the ratio of *Single Client, Multiple Access* requests to *Multiple Client, Single Access* requests, more or less opportunities for optimization may exist. For example, with an even mix of request types, containers created for clients with single requests may be quickly recycled for clients with multiple successive requests, leading to an overall speedup. *Multiple Client, Multiple Access* represents the average case scenario for a serverless platform.

Name	Source Language	Size (Native)	Size (WASM)	Function	Libraries
License Plate Reader	C++	2.1 MB	271 KB	Accepts license plate identifier as input and determines whether identifier is on a list of suspicious vehicles	None
Image Recognition	C++	8.4 MB	1.4 MB	Accepts image as input to a Deep Neural Network and returns name and confidence level of objects recognized in image	tiny-dnn [47]
Image Resize	C++	2.5 MB	481 KB	Resizes input image by a given percentage	Boost.GIL [38], libjpeg [25]

Table 1: Example Serverless Functions

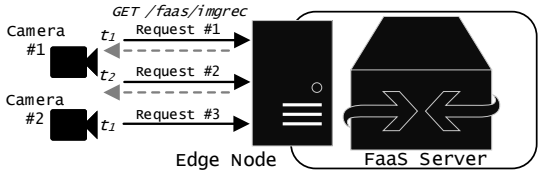

Figure 4: Multiple Client, Multiple Access Pattern

5 EVALUATION

In this section we describe our methods for evaluating WebAssembly as an alternative to the use of containers in serverless computing platforms.

5.1 Setup

We evaluate our WebAssembly-based serverless computing platform against Apache OpenWhisk, an open-source serverless computing platform which uses the Docker container engine for hosting applications. Our decision to use OpenWhisk as a comparison platform stems from the fact that its open-source nature allows for careful introspection and configuration of its inner workings. Our evaluations consist of benchmarking three example applications representing serverless functions of varying complexity. Both evaluation platforms were installed on identical hardware consisting of an Intel Xeon E5-2680 v2 CPU and 16 GB of RAM running Ubuntu Linux 16.04 LTS. Our WebAssembly evaluation platform is backed by NodeJS version 11.0.0-pre running V8 version 7.0.276.24-node.4 and our OpenWhisk evaluation platform is backed by Docker version 18.06.1-ce.

The execution time statistics used in our evaluations are gathered from the serverless platforms themselves. Our WebAssembly-based platform records the total time taken to instantiate a context and execute a WebAssembly function, and returns this information when responding to each request. OpenWhisk provides similar information (known as *annotations* [39]) with its responses, recording the duration a function executes, the initialization time required to create a container, and the waiting time incurred by other platform operations (e.g., authentication, accounting, etc.). Since our benchmarks are concerned with the cold start time of containers and since our WebAssembly-based prototype does not include operations such as authentication and accounting, we measure only the duration and initialization times of OpenWhisk when determining how long it requires to execute a serverless function. This methodology provides the closest comparison of the two platforms' abilities to execute serverless functions while reducing extraneous data such as network latency or overhead incurred by unrelated services.

Benchmarks were conducted from the client-side using the Apache JMeter [19] load testing tool. This tool was installed on a separate server equipped with a Xeon E5-2430 CPU and 16 GB of RAM and connected to our serverless computing platforms via a 1 Gbps link. Separate configurations were created for each testing scenario, with identical workloads applied to both platforms under evaluation. The size and scope of these configurations are based on the example scenarios described in Section 4 and are intended to demonstrate the types of workloads a serverless computing platform at the edge might experience. Results gathered by the JMeter client were compared with results gathered server-side to ensure accuracy.

5.2 Example Applications

For our evaluations, we created three custom applications representative of serverless functions from the scenarios described in Section 4. These applications were written in C++ and statically compiled to native x86 and WebAssembly binaries using clang 6.0.1-x and Emscripten 1.38.x, respectively. Any dependent libraries were first either statically compiled to native code using clang or compiled to Bitcode using the Emscripten toolchain and later linked during build time. Details of these applications can be found in Table 1 on page 7.

Although both native and WebAssembly applications were statically compiled from the same code, the resulting sizes of their output binaries vary significantly. This is due in large part to Emscripten's use of *dead code elimination* when compiling source code to WebAssembly. Emscripten, through its LLVM backend, analyzes and removes code that it determines will be unused during run time. This system is highly effective, but at times may inadvertently eliminate code that is actually needed. As a remedy, the dead code elimination feature of Emscripten can either be disabled entirely or configured to explicitly include portions of code that the developer is certain will be needed. Creating smaller WebAssembly binaries enables faster load times and more efficient code profiling, leading to an overall speedup in execution.

5.3 Native vs. WebAssembly Execution Time

We begin our evaluations with a comparison of the execution speed of native vs. WebAssembly binaries. The execution of each binary file was recorded over 50 runs via the Linux *time* utility and a timer internal to each application. Binaries compiled to native x86 code were executed directly and binaries compiled to WebAssembly code

were executed via NodeJS through their JavaScript loader files. The average execution times for each binary can be seen in Table 2.

In this scenario, WebAssembly provides no clear advantage. The overhead of executing WebAssembly code via NodeJS causes delays that far outstrip the execution time of native code. Although the WebAssembly specification calls for performance similar to native code [26], current mainstream runtimes do not achieve this goal in full.

Despite its slower performance compared to native execution at the time of this writing, the WebAssembly project is still making strides in closing this gap. For example, during the course of our research Google's V8 team released a new baseline compiler, *Liftoff* [22], that improved our WebAssembly execution times by approximately 50%. There are also ongoing efforts to create new runtimes for WebAssembly, some of which have shown early benchmarks with execution time on par with native binaries [30]. However, these runtimes are still in nascent stages and lack the more fully featured offerings of NodeJS. For this reason, we leave the exploration of alternative runtimes to future work.

App	x86	wasm
License Plate Reader	1 ms	6 ms
Image Recognition	30 ms	160 ms
Image Resize	60 ms	115 ms

Table 2: Native vs. WebAssembly Execution Speeds

5.4 Single Client, Multiple Access Workload

Container-based serverless computing platforms perform most efficiently when processing successive requests from the same client. For example, an IoT device may call a serverless function every 10 ms for 50 times in succession without closing its initial connection. The efficiency gained comes from the fact that the container instantiated to handle the initial request can be recycled for subsequent requests, thereby amortizing the long cold start time over the lifetime of the container.

To demonstrate the speedup from successive single client requests, we created a workload in JMeter simulating 50 subsequent requests from a single worker to the same serverless function. From this workload, we measured the time taken to receive a response from the function. We also applied this workload to our WebAssembly-based platform for comparison.

The results from our benchmark can be seen in Figure 5 and Table 3. As expected, the first requests for all three sample applications experience startup delays associated with container instantiation (in the case of OpenWhisk) or context creation (in the case of WebAssembly). The OpenWhisk platform experienced delays during the initial calls only, as indicated by the maximum latency value recorded for each function call. Subsequent requests were much faster, executing at approximately native speed. This initially slow response is due to the cold start penalty associated with creating the first containers, and the subsequent speedup is due to those containers being recycled to service the remaining calls. The WebAssembly platform also experienced a small delay during the first calls, but subsequent function calls did not receive the same speedup as those served by containers. This behavior caused the WebAssembly platform to perform in a more predictable manner, but slower on average relative to the average latencies of the container-based platform. Any speedups gained by WebAssembly came as a result of code profiling and code hotspot caching, both of which occur during the first few runs of an application.

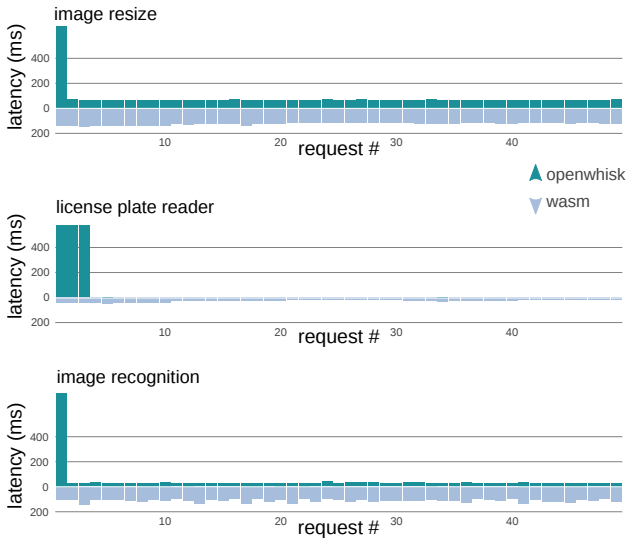

Figure 5: Single Client, Multiple Access Latencies

App	Avg / Min / Max (OpenWhisk)	Avg / Min / Max (WebAssembly)
Image Resize	79 / 64 / 658 (ms)	123 / 114 / 145 (ms)
License Plate Reader	38 / 3 / 580 (ms)	27 / 19 / 48 (ms)
Image Recognition	45 / 29 / 744 (ms)	112 / 94 / 144 (ms)

Table 3: Single Client, Multiple Access Latencies

When compared to serverless functions executing in already warm containers, WebAssembly still lags behind. Although WebAssembly's initial startup time is much better than that of OpenWhisk, over the lifetime of a long-running container this advantage is eroded by native execution speeds resulting in an overall lower average response time. However, it is important to note that *Single Client, Multiple Access* traffic patterns are not representative of a typical serverless computing workload. In an edge computing environment it is more likely that a large number of devices will make concurrent, simultaneous requests which require the instantiation of a separate container or context to process.

5.5 Multiple Client, Single Access Workload

One advantage of serverless computing platforms is their ability to scale dynamically with demand. For example, if 50 IoT devices

simultaneously request the same serverless function, the platform hosting that function may need to create a separate instance of the function to concurrently serve each request. On a container-based platform this means up to 50 new containers must be created, all of which will incur the cold start penalty. We refer to this access pattern as *Multiple Client, Single Access*. To benchmark this workload, we created a JMeter configuration with 50 clients which access the same serverless function once simultaneously over a 10 second ramp-up period.

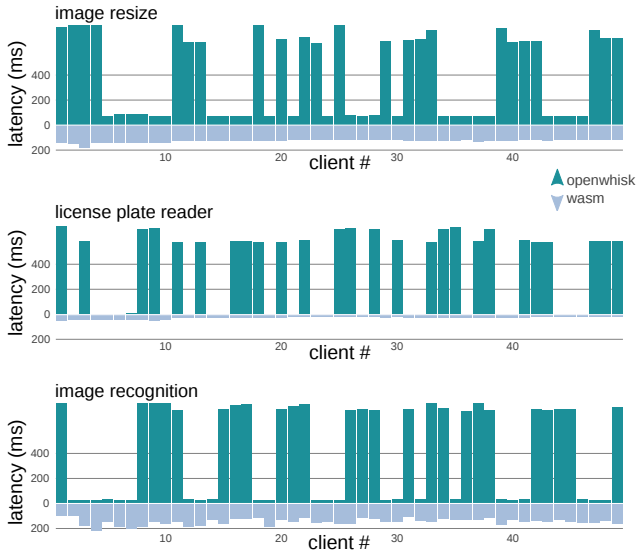

Figure 6: Multiple Client, Single Access Latencies

App	Avg / Min / Max (OpenWhisk)	Avg / Min / Max (WebAssembly)
Image Resize	394 / 68 / 904 (ms)	124 / 114 / 179 (ms)
License Plate Reader	334 / 3 / 701 (ms)	28 / 19 / 54 (ms)
Image Recognition	432 / 30 / 1073 (ms)	146 / 102 / 219 (ms)

Table 4: Multiple Client, Single Access Latencies

Figure 6 demonstrates the latency distributions for the 50 clients in our benchmark. At least half the clients accessing applications hosted on the OpenWhisk platform suffered from cold start delays, with other clients benefiting from warm containers. Since concurrent accesses in our benchmark could not occur at exactly the same time and since our example applications finished executing very quickly, the OpenWhisk platform was able to recycle some containers to serve other clients when those containers became available. This was an unexpected result, but speaks to the efficiency of OpenWhisk in reducing latency whenever possible. In the worst case, each request would have required a separate container and thus 50 cold starts would occur. The worst case scenario would be much more likely given serverless functions with longer running times, which would greatly reduce the opportunity for container reuse during the brief period where an influx of concurrent requests is processed. Our WebAssembly platform exhibited lower startup times overall, but was not able to achieve the same level of performance as already warm containers running native code. However, WebAssembly did provide the advantage of more stable, predictable latencies and much lower average latencies for all function calls. Details from our benchmarks can be found in Table 4.

5.6 Multiple Client, Multiple Access Workload

Although benchmarks of the *Single Client, Multiple Access* and *Multiple Client, Single Access* workloads both yield interesting results, neither is representative of a real-world workload. Instead, access patterns to serverless functions exhibit properties of both workloads: some clients access a function multiple times in succession and other clients access a function only once. Our goal is to determine whether WebAssembly provides any advantage in reducing the average latency when accessing functions on platforms processing this type of workload.

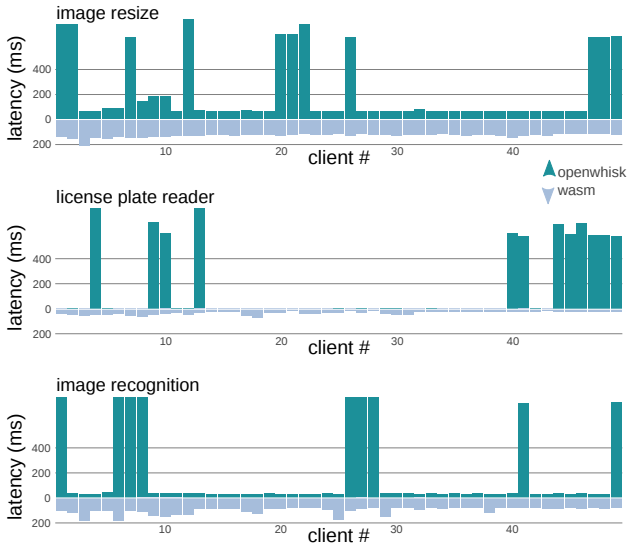

Figure 7: Multiple Client, Multiple Access Latencies

App	Avg / Min / Max (OpenWhisk)	Avg / Min / Max (WebAssembly)
Image Resize	221 / 66 / 1039 (ms)	129 / 116 / 208 (ms)
License Plate Reader	173 / 3 / 912 (ms)	34 / 19 / 75 (ms)
Image Recognition	247 / 30 / 1799 (ms)	98 / 75 / 181 (ms)

Table 5: Multiple Client, Multiple Access Latencies

We demonstrate the effect of such mixed access patterns by creating a workload in JMeter which provides equal representation to

both access patterns. This workload consists of 1 worker accessing the same serverless function 25 times (a *Single Client, Multiple Access* pattern) and 25 workers accessing the same function 1 time over a 5 second ramp-up period (a *Multiple Client, Single Access* pattern). Workers with little to no delay between accesses will realize the benefit of an already warm container during subsequent requests, while workers with a large delay will likely incur the cold start penalty each time. We apply this same workload to our OpenWhisk and WebAssembly platforms.

The results from our benchmark can be found in Figure 7 and Table 5. Given this type of access pattern, the OpenWhisk container-based platform incurred cold start penalties for approximately 20-25% of requests. The WebAssembly platform demonstrated more uniform results, and showed performance consistent with previous benchmarks of other workloads.

5.7 Discussion of Results

Benchmarks. We designed our benchmarks to demonstrate the performance of a container-based serverless platform and a WebAssembly-based serverless platform given workloads from real-world scenarios. The example serverless functions we created from these scenarios represent two tasks of moderate complexity (Image Resize and Image Recognition) and one task of basic complexity (License Plate Reader). To prove a viable alternative to the use of containers for hosting applications, WebAssembly should provide at least similar performance on average to a container-based solution when presented with the same workloads. The results gathered from our tests suggest that WebAssembly is indeed a viable alternative, despite having its own disadvantages in certain scenarios.

During our discussion of these results, we abbreviate several repeated references for the sake of simplicity. We refer to the container-based solution as *OpenWhisk* and the WebAssembly-based solution as *Wasm*. When describing application executions we refer to the Image Resize and Image Recognition applications as *moderate tasks* and the License Plate Reader application as a *basic task*. And when discussing execution latencies we refer to the minimum, maximum, and average latencies as the *best*, *worst*, and *average* cases.

Given the *Single Client, Multiple Access* workload, Wasm provides little advantage. In the best case, Wasm performed approximately 1.5-3x slower than OpenWhisk when executing moderate tasks and approximately 6x slower executing a basic task. Despite applications hosted on OpenWhisk incurring the cold start penalty during their initial calls, subsequent calls all executed on warm containers at native speed. These initial cold starts increased average application latencies by approximately 25-50% for the moderate tasks and approximately 400% for the basic task. These results suggest that although OpenWhisk is generally a solid performer for the *Single Client, Multiple Access* workload, very simple applications will require far more subsequent executions before the cost of the initial cold start is amortized enough to reduce average execution latency.

Wasm begins to show its benefit over OpenWhisk when tasked with the *Multiple Client, Single Access* workload. We initially expected this workload to be problematic for OpenWhisk, with each concurrent request forcing a cold start. However, we found that OpenWhisk handled these requests gracefully and was able to recycle approximately half the containers to avoid cold start penalties. There are two reasons for this behavior. The first reason is that no two requests can be exactly concurrent, which means there was a small window of opportunity for a warm container to become available during the brief period where the concurrent requests were being sent from client to server. The second reason is the short running times of our moderate and basic tasks. Our applications finished execution quickly, which freed up their containers to be recycled. If our applications had taken longer to run, it is very likely that OpenWhisk would have suffered from a majority of cold starts. Wasm processed this workload with execution speeds very similar to those seen with *Single Client, Multiple Access*. When compared to OpenWhisk's best case speeds, Wasm is still much slower. However, when we look at the average case we begin to see the effect the cold start problem has on overall performance. OpenWhisk's average latency was roughly 6x-14x higher than the best case for the moderate tasks and around 100x higher than the best case for the basic task. Comparatively, Wasm executed the moderate tasks approximately 70% faster and the basic task approximately 90% faster on average.

The *Multiple Client, Multiple Access* workload exposed each platform to an even mix of requests from each of the first two workloads. Since OpenWhisk realizes a strong advantage when faced with the *Single Client, Multiple Access* workload, we expected that this advantage would offset some of the disadvantage caused by the concurrent requests of the *Multiple Client, Single Access* workload. The results from our benchmarks were in line with our expectations. Most requests to OpenWhisk were served from warm containers and approximately 25% of function calls suffered from cold start delays. Almost all these delays can be attributed to the concurrent requests from the *Multiple Client, Single Access* pattern. Interestingly, we note that this workload consisting of 50% *Multiple Client, Single Access* requests results in approximately 25% cold starts, which is consistent with the results from our previous benchmark where a workload consisting of 100% *Multiple Client, Single Access* requests resulted in approximately 50% cold starts. Wasm continued to perform in a consistent manner throughout this workload, achieving best, worst, and average case latencies very similar to the previous two workloads. OpenWhisk demonstrated similar best case latencies and decreased average case latencies due to less cold starts. We note that a *Multiple Client, Multiple Access* workload will not always achieve an even split between the two access patterns, and that a skew toward one or the other type of request can easily cause more or less cold starts to occur. For example, a mixed workload with a majority of *Single Client, Multiple Access* requests would show fewer cold starts, while a workload with a majority of *Multiple Client, Single Access* requests would show more cold starts. Our workload in this benchmark represents an ideal case so that we may fairly demonstrate the effect of such access patterns on both OpenWhisk and Wasm.

Our benchmarks demonstrate that Wasm performs well over the three given workloads. Although at times this performance is slower than that of OpenWhisk, it is consistent and on average faster. Even when faced with workloads which cause OpenWhisk to experience spikes in latency, Wasm provides relatively stable response times. These results do not necessarily suggest that Wasm

is better than OpenWhisk. Certain workloads that are favorable to container reuse still provide superior performance due to Wasm's slower-than-native execution speeds. However, Wasm does appear to at least be a peer to OpenWhisk and even a viable alternative given certain conditions.

Performance vs. Containers. Wasm's primary advantage over container-based solutions is the absence of a large cold start penalty. There are two reasons for this advantage. First, container runtimes such as Docker incur a large amount of overhead in ensuring their support for containers is as broad as possible. This support includes several features (e.g., advanced networking, container checkpoint/restore) which are unnecessary for the minimal container configurations used by serverless functions. Setup and control of these features requires multiple syscalls and IPC between container parent and child processes before a serverless function can begin executing. In contrast, Wasm is focused on executing sandboxed code as quickly as possible. Many of the necessary isolation and resource management features needed for serverless functions are inherent to Wasm. The additional overhead we need to add to use Wasm in a serverless platform is minimal, allowing for a reduced mean time to function execution. Second, each Docker container consists of one or more separate processes, whereas each Wasm instance is contained within the same V8 process. This provides a Wasm-based solution the advantage of warmer caches and reduced context switch penalties, creating better potential for speedup when concurrently executing multiple serverless functions. Although in general a container's overhead may be acceptable for long-running applications, this overhead quickly proves an impediment to meeting the low-latency demands of serving emerging IoT applications.

It is important to keep in mind that WebAssembly is still in a nascent stage and continues to improve at a rapid pace. At present, its biggest advantage over containers is consistent performance and lower average latency when cold starts exist. However, as the project continues to progress execution times will improve significantly. When these execution times reach near-native performance, WebAssembly will provide an even greater benefit over containers with or without regard to the cold start problem.

6 RELATED WORK

The oldest and arguably most popular serverless computing platform, Amazon's AWS Lambda [2], was first introduced in 2014. Since then, other major cloud providers have followed suit with offerings such as Google's Cloud Functions [23], IBM's Cloud Functions [31], and Microsoft's Azure Functions [4]. Interest and research in serverless computing has also spawned open source projects such as OpenLambda [12] and Apache OpenWhisk [18]. Offerings also extend to the edge of the network, with commercial platforms such as AWS Greengrass [3] and Azure IoT Edge [5] and open-source platforms such as EdgeXFoundry [20] specifically targeting serverless for IoT. As mentioned in the Introduction, these platforms rely on containers for function isolation and are thus susceptible to cold start delays.

Several approaches to improving container-based serverless platforms have been proposed. SAND [1] introduced the notion of grouping functions related to the same application within the same container, thereby greatly increasing the opportunity for reuse of already warm resources. Slacker [28] and SOCK [37] introduced new methods for loading and caching dependencies upon container instantiation, decreasing the time to initial application execution after a container has been provisioned. And McGrath, et al. [35] proposed the use of an intelligent queuing system where worker platforms indicate the availability of cold and warm containers to enable better placement for executing incoming function requests.

Our work takes a different approach to improving serverless computing performance by adapting WebAssembly, a technology from the client-side web browser space, to work on the server-side. In the future, WebAssembly may serve as an alternative runtime for state-of-the-art serverless platforms. Its features are especially beneficial to those platforms at the edge of the network which need to serve vast numbers of IoT devices with low latency response times. Additionally, the hardware-agnostic nature of WebAssembly allows its applications to be readily portable across a wide array of devices with different architectures within the IoT space.

To our knowledge, our study is the first to quantitatively compare container-based and WebAssembly-based solutions for serverless function execution. We are not aware of the existence of any formal research which has explored WebAssembly for containing and executing serverless functions as an alternative to containers. However, at the conclusion of our research a commercial product, Cloudflare's Service Workers [49], began to offer support for creating and hosting serverless functions in WebAssembly. There is no explicit documentation indicating what runtime is used for these serverless functions, but available information implies that Cloudflare's solution may function similar to ours. We view this development as a validation of our research ideas and look forward to the emergence of similar projects in the future.

7 CONCLUSIONS AND FUTURE WORK

Our goal in benchmarking WebAssembly was to determine if it could provide similar or better performance than containers given the same workloads. The results from these benchmarks showed it to perform consistently across all access patterns. Although its execution speed vs. container-based native binaries is slower, when the cold start penalty of containers is factored in to this calculation its performance is faster on average. WebAssembly is a new technology, and advances are still being made toward enabling performance similar to native execution speeds. Its execution speed has continued to improve since the release of the Minimum Viable Product in late-2017. We believe the results from our benchmarks indicate WebAssembly is a viable alternative to the use of containers in serverless platforms, and that further exploration in this space is warranted.

In addition to its viability as a serverless computing runtime for the Edge, WebAssembly can also prove useful to the IoT domain in general. WebAssembly's platform-neutral nature lends itself to building applications that can execute across the myriad architectures of devices which comprise the Internet of Things as well as servers in the cloud. This is in line with the vision of the Fog Computing paradigm [6], which describes a computational continuum that extends the power of the cloud to the network edge.

We intend to continue our exploration of WebAssembly as an execution environment for serverless functions in three areas:

Full Serverless Platform: The WebAssembly-based serverless platform we created for this paper is a simple prototype. It provides the most basic functionalities of serverless platforms, but is not robust enough to be considered production-ready. As future work we intend to further develop this platform to mirror the features of state-of-the-art offerings such as OpenWhisk. This work would include a more robust web API, authorization and accounting features, and stronger orchestration features to allow provisioning and migration among different instances of the serverless platform.

Custom Runtime: Our prototype is based on NodeJS and several of its third-party modules, which limits us to the features that they offer. We intend to explore the creation of our own WebAssembly runtime which will provide much more robust functionalities. In particular, we hope to achieve finer-grained control over resource provisioning, execution of code, and process isolation.

Benchmarks at Scale: With our custom runtime and full serverless platform in place, we hope to revisit our benchmarks and test our WebAssembly-based solution against popular container-based solutions at scale. These tests would incorporate workloads from the access patterns described in Section 4 and better demonstrate WebAssembly's efficacy for executing serverless functions over a wide range of scenarios.

ACKNOWLEDGEMENTS

This work was funded in part by an NSF Award (NSF-CPS-1446801) and a grant from Microsoft Corp. We thank members of Georgia Tech's Embedded Pervasive Lab, our anonymous reviewers, and our shepherd Yogesh Simmhan for their feedback and help in improving this presentation.

REFERENCES

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: towards high-performance serverless computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*.
[2] Amazon. 2019. AWS Lambda. https://aws.amazon.com/lambda.
[3] Amazon AWS. 2019. AWS IoT Greengrass. https://aws.amazon.com/greengrass.
[4] Microsoft Azure. 2019. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[5] Microsoft Azure. 2019. Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge/.
[6] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 13–16.
[7] Lin Clark. 2017. Memory in WebAssembly (and why it's safer than you think). https://hacks.mozilla.org/2017/07/memory-in-webassembly-and-why-its-safer-than-you-think/.
[8] Emscripten Contributors. 2018. emscripten. http://www.emscripten.org.
[9] Emscripten Contributors. 2018. File System API - Emscripten 1.38.25 documentation). https://emscripten.org/docs/api_reference/Filesystem-API.html.
[10] Emscripten Contributors. 2018. Porting SIMD code - Emscripten 1.38.25 documentation. https://emscripten.org/docs/porting/simd.html.
[11] MDN Contributors. 2018. WebAssembly.Memory(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Memory.
[12] OpenLambda Contributors. 2019. OpenLambda. https://open-lambda.org.
[13] Yan Cui. 2018. I'm afraid you're thinking about AWS Lambda cold starts all wrong. https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f.
[14] V8 Developers. 2018. Getting started with embedding V8). https://emscripten.org/docs/api_reference/Filesystem-API.html.
[15] Amazon AWS Documentation. 2018. AWS Lambda Execution Context. https://docs.aws.amazon.com/lambda/latest/dg/running-lambda-code.html.
[16] Amazon AWS Documentation. 2019. Serverless Application Developer Tools. https://aws.amazon.com/serverless/developer-tools/.
[17] expressjs.com contributors. 2018. Express - Node.js web application framework. https://expressjs.com.
[18] Apache Foundation. 2019. Apache OpenWhisk. https://openwhisk.apache.org.
[19] The Apache Software Foundation. 2018. Apache JMeter. https://jmeter.apache.org.
[20] The Linux Foundation. 2019. EdgeX Foundry. https://www.edgexfoundry.org.
[21] Sam Goldstein. 2018. How Cold Starts Impact Serverless Performance. https://thenewstack.io/how-cold-starts-impact-serverless-performance/.
[22] Google. 2018. Liftoff: a new baseline compiler for WebAssembly in V8. https://v8project.blogspot.com/2018/08/liftoff.html.
[23] Google. 2019. Google Cloud Functions. https://cloud.google.com/functions/.
[24] Inc. Google. 2018. V8 JavaScript Engine. https://chromium.googlesource.com/v8/v8.git.
[25] Independent JPEG Group. 2018. Independent JPEG Group. https://www.ijg.org.
[26] WebAssembly Working Group. 2018. WebAssembly High-Level Goals - WebAssembly. https://webassembly.org/docs/high-level-goals/.
[27] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 185–200.
[28] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers.. In *FAST*, Vol. 16. 181–195.
[29] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2016. Quantifying the impact of edge computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 5.
[30] Rian Hunter. 2018. Kernel Mode WebAssembly Runtime for Linux. https://github.com/rianhunter/wasmjit.
[31] IBM. 2019. IBM Cloud Functions. https://www.ibm.com/cloud/functions.
[32] IBM. 2019. Platform architecture. https://console.bluemix.net/docs/openwhisk/openwhisk_about.html.
[33] Linux Programmer's Manual. 2018. cgroups - Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.
[34] Linux Programmer's Manual. 2018. namespaces - overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html.
[35] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*. IEEE, 405–410.
[36] Goncalo Neves. 2018. Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues. https://serverless.com/blog/keep-your-lambdas-warm/.
[37] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*.
[38] Boost C++ Library Project. 2018. Boost.GIL - Generic Image Library. https://github.com/boostorg/gil.
[39] OpenWhisk Project. 2018. Annotations on OpenWhisk assets. https://github.com/apache/incubator-openwhisk/blob/master/docs/annotations.md.
[40] WebAssembly Project. 2018. Design Rationale - WebAssembly. https://www.webassembly.org/docs/rationale/.
[41] WebAssembly Project. 2018. Minimum Viable Product - WebAssembly. https://www.webassembly.org/docs/mvp/.
[42] WebAssembly Project. 2018. Security - WebAssembly. https://www.webassembly.org/docs/security/.
[43] Larry Seltzer. 2018. Serverless computing explained | HPE. https://www.hpe.com/us/en/insights/articles/serverless-computing-explained-1807.html.
[44] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.
[45] Patrik Simek. 2018. vm2. https://github.com/patriksimek/vm2.
[46] LLVM Admin Team. 2018. The LLVM Compiler Infrastructure. https://llvm.org.
[47] tiny-dnn Project. 2018. tiny-dnn: header only, dependency-free deep learning frakework in C++14. https://github.com/tiny-dnn/tiny-dnn.
[48] Colby Tresness. 2018. Understanding Serverless Cold Start | Azure App Service and Functions team Blog. https://blogs.msdn.microsoft.com/appserviceteam/2018/02/07/understanding-serverless-cold-start/.
[49] Kenton Varda. 2018. WebAssembly on CloudFlare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.
[50] Tim Wagner. 2014. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.