WebAssembly for Edge Computing: Potential and Challenges

Mohammed Nurul Hoque and Khaled A. Harras

Abstract

Latency and privacy concerns, together with the spread of smart/IoT devices, have recently sparked interest in computational offloading to the edge. Portability and migratability are important requirements to achieve a stable edge-offloading platform. To that end, code compatibility is one of the core challenges toward achieving these goals due to the inherent heterogeneity of edge devices. In this article, we first examine existing edge-computing technologies, how they achieve portability and migratability, and the advantages and limitations, via experimentation, of each method. We then explore leveraging WebAssembly for edge computing. We present an overview of this rising technology, assess its performance, and discuss its potential compared to other solutions for edge offloading. In the end, we outline four potential methods to achieve migratability with WebAssembly and the trade-offs and costs of deployment for each method.

Introduction

Cloud computing has dominated the computing landscape for the last couple of decades. The recent trend of novel applications requiring high computation resources, increased privacy, and low latency, has driven the rise of edge computing [1]; the offloading of computations and data to devices on the edge of the Internet. Starting with cloudlets, to the more recent ideas of using peer mobile devices opportunistically, as in FemtoClouds [2] or mobile device clouds, the urge to push computations closer to the user is clear. On the other hand, the proliferation of IoT devices and smart gadgets in the modern environment made different approaches to edge computing possible on readily available platforms. These devices have considerable computing and storage capabilities in total and are idle most of the time [2]. Utilizing these untapped resources opens up a vast pool of computational power with no extra infrastructure. The term *edge computing* is used in literature to refer to a wide range of close proximity to the end-user. We use the term throughout to refer to the specific paradigm, also known as FemtoClouds [2], where edge devices are zero-hops away and belong to a single administrative entity, thus providing computational resources collaboratively as a single pool. Examples include households, schools, or workplaces where many member devices are controlled by the organization which can enable such cooperation.

Edge computing poses unique challenges different from the cloud because of differences in their characteristics. Edge devices tend to have higher heterogeneity and lower stability which has a direct impact on code offloading. We identify two criteria for any code execution platform to be viable for edge computing. First, it needs to be portable across possible targets, accounting for different operating systems and hardware architectures. Second, it should be able to migrate running code across devices so that partial computations can be saved when a device leaves or powers off, which is highly likely in various edge scenarios. Current code execution technologies offer different trade-offs between these two requirements.

In this article, we first provide an overview of the current state of technologies that achieve portability and migratability to various degrees. We classify the existing methods by their general approach into two categories: system-oriented technologies such as virtual machines and containers; and language-oriented technologies such as Java and JavaScript. We examine the advantages and drawbacks of each method and assess how each one performs on the scale of portability and migratability.

After the overview, we then present the potential of adopting WebAssembly for offloading computations on the edge. We begin by providing an overview of WebAssembly, the need for it, its code format, execution model, and runtime environments. Next, we demonstrate the portability of WebAssembly through a set of experiments that test different mechanisms of executing WebAssembly ranging from within the browser to standalone modes. We run the CoreMark benchmark on different OS and architecture platforms, in order to assess and compare the performance of these WebAssembly modes to the native performance of each platform. The results suggest that WebAssembly indeed provides ultra-portability with high performance when run in the browser, with mixed results for other methods.

In the end, we explore the current state of WebAssembly in the edge and emerging use cases and analyze the current main shortcoming, which is migratability. We identify the challenges that need to be overcome in order to achieve migratability while maintaining portability. We discuss several approaches to implement migration that have been tried for other technologies and shed some light on the potential speed, complexity, and control trade-offs that would be involved with each

Digital Object Identifier:
10.1109/MCOMSTD.0001.2000068

The authors are with Carnegie Mellon University.

2471-2825/22/$25.00 © 2022 IEEE

approach when applied to WebAssembly.

In summary, we contribute the following:

- Identifying portability and migratability as the main challenges of edge computing.
- An overview of WebAssembly, its purpose, how it works and its potential for edge computing.
- A comparison of the different WebAssembly runtimes and evaluating their performance.
- Outlining potential directions for migrating WebAssembly and the challenges with each method.

Portability and Migratability of Existing Technologies

Portability and migratability are old problems that have been tackled under two main paradigms, a system-oriented approach and a language-oriented approach. System-oriented approaches use system-level mechanisms to isolate programs from the host environment, providing a decent level of portability. The two examples we will consider are virtual machines and containers. Language-based approaches use a non-compiled language and rely on the existence of the language's runtime on all target platforms to run the same code.

Before we present current technologies, we clearly specify our criteria for defining portability and migratability. Portability is concerned with the diversity of the platforms that support a technology, notably, hardware architectures and operating systems. There is also a development side of portability, of which we assess language Independence, the ability of the programmer to write in any source language and use the technology. Migratability, on the other hand, is the ability to interrupt running code, transfer its state, and resume execution on any other platform that supports the same technology.

System Based Portability

Virtual Machines: Virtual machines (VMs) virtualize a full machine by emulating the hardware and the operating system, completely isolating the running code from the underlying system. Because VMs are emulated, the hypervisor has full control over the state of the emulated system. This means migrating a VM is relatively simple since state data can be written to a file in a standardized format. This file can then be used to resume the emulation on any target that has a hypervisor that understands the same format. Emulation also means that VMs are portable as any guest can be emulated on any host. Ha et al., (2017) introduced VM handoff to allow for efficient migration of VMs between edge devices by splitting VM images into layers and tuning the balance between data volume and computations to minimize downtime and total completion time [3].

Despite their versatility, VMs suffer from high overhead especially when run cross-architecture. Because the whole system is emulated, even running a simple program requires emulating a lot of hardware components and a whole operating system, which can exceed 2GB in size and require a few minutes to migrate [4]. In addition, if the emulated hardware architecture is different from the host architecture, the guest architecture is simulated instruction-by-instruction which is not viable for practical purposes.

Containers: Containers use OS facilities to isolate container processes from host resources while still running the guest program as a native process. They are more lightweight than VMs as they virtualize only parts of the OS. Containers took off with data centers, for example, AWS, since Docker was introduced, because they package dependencies with applications so that they can run on any supporting OS without worrying about dependencies. Docker recently added support for CRIU (Checkpoint/Restore In Userspace), a feature of the Linux kernel that dumps the state of a paused process into files, which can be used to create a process resuming from that state on another device.

The following examples are representative state-of-the-art container-based solutions. Ma, Yi and Li leverage Docker's layered file system and Docker Hub to reduce transfer size between edge nodes by fetching base layers directly from the cloud and transferring base memory ahead of time, with only the new changes being sent when a migration request is issued [4]. CRIU is used by Teddybear [5] to migrate docker containers in areas with poor network bandwidths using both the Internet and the user's mobile device.

The containers' portability and migratability are, however, limited by OS features and hardware compatibility and still incur considerable overhead. Because of their reliance on OS features, containers are generally not portable across operating systems and require some form of virtual machines. Additionally, containers run as native processes and thus, are not portable across architectures without recompilation, and cannot migrate across architectures.

Language-Based Portability

Java: Java was the main "write once, run everywhere" platform and there were many attempts to reap its portability in the edge/IoT world such as Java applets in the early web and Eclipse foundation's enterprise IoT solutions. With the rise of WebAssembly, it is often compared to Java and its web implementation, Java applets. However, there are some key downsides to using Java in edge computing. First, Java binds the programmer to the programming languages that target JVM such as Java and Scala. Second, JRE is known to be a heavy and complex piece of software with many security issues. Finally, Java support is declining with modern platforms; iOS, for example, does not support Java since its first version and Java applets are now deprecated.

JavaScript: JavaScript is the dominant language of the web supported by all modern browsers, and as such enjoys the advantages of exceptional infrastructure. Even though JavaScript is semantically interpreted, in practice, it is just-in-time compiled and runs extremely fast, usually within 2x of native speed [6]. Also, JavaScript runs in browsers, which are ubiquitous and well maintained. JavaScript is not conventionally a migration platform but some attempts have been made to reap its portability. Gascon-Samson, Rafiuzzaman and Pattabiraman proposed ThingsMigrate, a system for migrating JavaScript code across Node.js runtimes [7]. It instruments the input code to track all the runtime state explicitly in a state object that is modified whenever a statement changes the state in the original code.

Portability and migratability are old problems that have been tackled under two main paradigms, a system-oriented approach and a language-oriented approach. System-oriented approaches use system-level mechanisms to isolate programs from the host environment, providing a decent level of portability.

Platform	Speed	Size	Cross OS	Cross arch.	Lang. indep.	migratablity
Native	1x	code	×	×	✓	×
VM	1.16x [8]	code + ~2 GB	✓	×	✓	✓
Containers	1.02x [8]	code + ~200 MB	×	×	✓	✓
JavaScript	1.88x [6]	code	✓	✓	×	×
Java	1.09–1.91x [9]	code	✓	✓	×	×
WebAssembly	1.1–1.45x [6]	code	✓	✓	✓	×

Table 1. Comparison of code execution environments.

JavaScript ranks high in our criteria of portability, but has limited migratability, lacks language independence, and is still not sufficiently fast in many cases [6]. Like Java, to run in JavaScript, code must be written in JavaScript or a few other languages for which transpilers to JavaScript exist. Second, the migratability of JavaScript is limited. The approach followed by Gascon-Samson et al. [7] assumes that code is event-driven and migration is processed as one of the events. This means no actively running computations can be migrated which reduces its utility for computationally intensive applications or preexisting libraries that are not tailored to that model. The complexity of JavaScript as a language makes the prospects of generalizing the approach to migrate any live JavaScript difficult. Finally, the performance of JavaScript, although impressive for a non-precompiled language, still lags considerably behind native performance.

The characteristics of these approaches discussed are summarized in Table 1. As the table shows, system-oriented approaches are generally migratable but less portable due to their reliance on host system services. On the other hand, language-based approaches tend to be more portable as they rely on a portable language runtime, but are slower, tie developers to languages, and are generally not migratable. The table also includes WebAssembly, which we expand on in the next section.

The Promise and Reality of WebAssembly

In the previous section, we presented many existing platforms where portability and migratability were achieved with different trade-offs. Next, we introduce WebAssembly, a platform that we argue can be potentially superior for edge computational offloading. WebAssembly is a low-level binary code format for the web [10], and in this section, we will introduce the goals and design of WebAssembly, and assess how it delivers on these goals.

The Origin of WebAssembly

As web content is becoming increasingly rich, more and more computations are taking place at the client-side, which makes fast code execution on the web a necessity. Nowadays, many desktop-class applications can be found as web applications such as photo editing software and multiplayer games. The high processing needs of these applications require a code execution technology that is more versatile than JavaScript. Several attempts were made over time to enable fast code on the client-side such as Google's PNaCl which used a subset of LLVM bytecode as a portable code format and Mozilla's asm.js which used a small low-level subset of javascript as a compilation target which itself can be compiled to efficient machine code. Browser vendors recognized the need for a unified solution, hence in 2017, all the major browser vendors announced the standardization of WebAssembly, a binary format for code to run on the web [8].

How WebAssembly Works

The WebAssembly Format: WebAssembly is a binary format designed to be a compilation target for high-level languages with size and load-time efficiency in mind [10]. WebAssembly also has an accompanying textual format (`.wat`) that is intended for debugging and testing. The low-level language only has four value types, `i32`, `i64`, `f32` and `f64` which correspond to integer and floating-point types of the given bit widths. The distributable unit of WebAssembly is a single-file module. This module consists of functions, globals, memory, tables, and imports and exports. Memory acts as a heap and is a bound-checked array indexed from 0. Each module can declare a memory with an initial size and a maximum size and can also import external memory. Imports in WebAssembly do not refer to a standard library or modules in a standard path. Instead, imports are just opaque names with a particular type and the WebAssembly runtime is free to initialize these imports with any compatible object, for example, a function in a library or allowing the user to choose what the name is linked to. Exports define the entities inside the module that can be externally accessed. Tables are arrays of pointers that are used to implement indirect function calls because WebAssembly disallows moving the program counter to arbitrary addresses in memory for security reasons.

WebAssembly is a stack-based language. Operators pop operands from the stack and push the result back. Similarly, function calls pop arguments from the stack and the return value is returned on the stack. WebAssembly is not purely stack-based, however. Similar to Java, it also has a set of locals for each function including its parameters. Even though a stack is used, WebAssembly instructions are typed with respect to their effect on the stack, which makes the stack offset at which an operation pushes or pops known statically. This allows for efficient compilation to register-machine architectures such as most physical hardware architectures.

WebAssembly uses structured control flow with no arbitrary jump instructions. The structured instructions are `if…else…end`, `block…end`

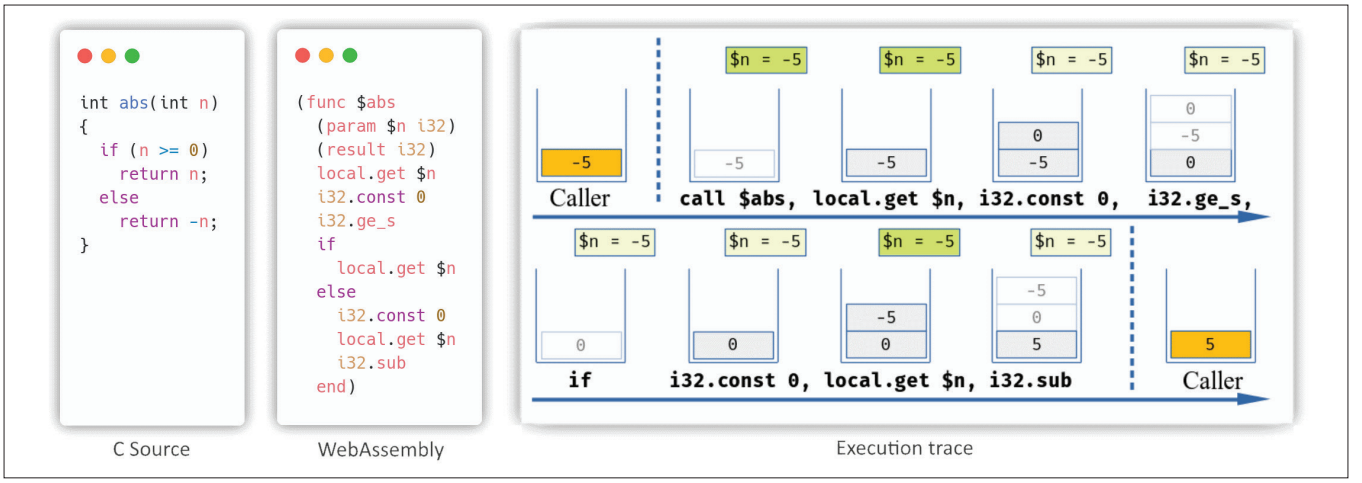

FIGURE 1. Absolute value function in WebAssembly and its execution.

`and loop…end` and they are typed according to their stack output. There are two structured jump instructions: break and conditional break. Break inside a block or if-else jumps to the end of that block, and break inside a loop jumps to the beginning of the loop. By default, control falls out of the loop if it does not hit a break. The break is more expressive than the C break as it allows breaking with a value and specifying a relative index which determines how many nested blocks to break from. This allows breaking from deeply nested blocks directly. Blocks and loops are typed with their net effect on the stack.

Figure 1 shows an example of tracing the execution of abstract WebAssembly for a function that computes the absolute value. This is for illustration purposes only, as WebAssembly actually has a built-in absolute value instruction. The light boxes indicate values that have just been popped, and the green box is the local variable *n*. After the caller places the arguments on the stack, it executes `call`, which pops the argument to the locals of the callee and transfers control to the first instruction in the callee. Next, `get` pushes a local and `const` pushes an inline constant to the stack. `ge_s` pops two values and pushes one if the first value (bottom) is greater than or equal to the second (top), else pushes 0 which is the case in our scenario. The condition of the `if` statement is popped from the stack and the if-branch is taken if the popped value is non-zero, otherwise, the else-branch is taken which is the case here. The rest is computing 0 – *n*. The final value that is on the stack at the end is the return value of the function.

Running WebAssembly: WebAssembly is only a specification of a format and its formal semantics which can run in practice in multiple ways including in-browser, in a standalone runtime, and through an interpreter. In all methods, the code starts as source code that is compiled to WebAssembly which is distributed and run using one of the methods above. Running in-browser is the originally intended way of running WebAssembly, which is illustrated in Fig. 2. WebAssembly runs inside the JavaScript engine which just-in-time compiles the WebAssembly to native code and executes it. Currently, WebAssembly requires JavaScript glue which loads the WebAssembly module, compiles the module to machine code, instantiates its

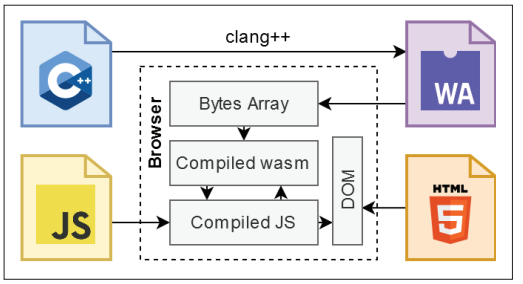

FIGURE 2. WebAssembly in the browser.

imports then invokes some exported function.

WebAssembly on its own has no concept of a main function. Instead, all globals are initialized at instantiation, then WebAssembly acts as a library of functions that can be called from JavaScript. There are also standalone WebAssembly runtimes that can run standalone WebAssembly modules outside the browser. These are usually just-in-time compilers that run WebAssembly locally without the JavaScript glue. Finally, WebAssembly can also be run using an interpreter which comes with the usual trade-off of interpreters, lower performance for higher portability. Non-browser solutions, including standalone runtimes and interpreters, may also support a standard extension proposal called WebAssembly System Interface (WASI), which defines a standard interface for WebAssembly to perform system calls across different platforms which allows for standalone WebAssembly executables.

Assessment of Performance and Portability

WebAssembly makes many bold promises, combining web-class portability with near-native performance. We devise experiments using CoreMark [11] to test the extent to which different implementations of WebAssembly achieve these goals and the trade-offs between the different implementations in terms of portability and performance. Different use cases might benefit from using different implementations based on these trade-offs.

We use the CoreMark® benchmark since it is an industry standard for CPU benchmarks and measures pure processor throughput. A WebAssembly runtime in a way acts as a core that executes WebAssembly instructions and as such, is

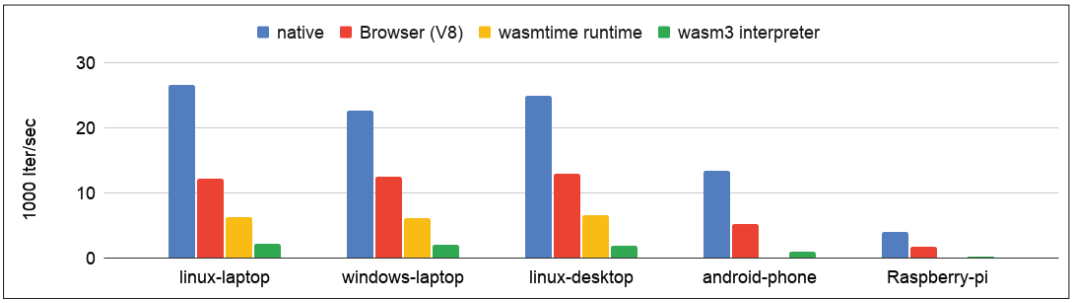

FIGURE 3. CoreMark benchmark results.

similar to a CPU processor in this regard. Also motivating the use of CoreMark is the fact that it is standardized, widely used, portable even to microcontrollers, and reports a single number allowing for quick comparisons. All of this will help any future work to compare with our results and other published CoreMark scores.

We use five platforms in our tests:

- Linux laptop using intel Core i5-8250u,
- Windows laptop using intel Core i5-8250u,
- Linux desktop using intel Core i7-3770,
- Android phone using Snapdragon 845 and
- Raspberry Pi model B V1.1.

We also investigated three different methods of runningWebAssembly: using the Google Chrome JavaScript engine V8, using wasmtime which is a standalone WebAssembly runtime by Bytecode Alliance, and using wasm3 which is an interpreter for WebAssembly. We compiled CoreMark using the `–O3` flag on all of these platforms.

Setting up the testbed involved a few challenges and was not the same for all platforms. Browsers — thanks to the huge standardization efforts — ran smoothly on all platforms. Standalone WebAssembly runtimes, even though conceptually a just-in-time compiler similar to browsers, are newer and mostly in beta development. We used wasmtime (version 0.21) by bytecode alliance, an open-source community founded by corporations with interests in WebAssembly development. The interpreter we used is wasm3, which is written in C. It ran without modification on the laptop, the desktop and the Pi. On Android, we had to make source code modifications and reconfiguration of compiler flags to make it work, as the original source relies on conditional compilation for portability. We configure the interpreter to compile as a shared library that is loaded through the Android NDK at runtime.

The sample results depicted in Fig. 3 confirm the portability of WebAssembly and its high performance in the browser case, but it is a mixed bag for the other two methods. The standalone runtime wasmtime

achieves a performance similar to the browser on supported platforms. However, as it is relatively new and under active development, not all of our platforms are yet supported by wasmtime. It is worth noting that an earlier version (0.7) which is only a year old, scores 45 percent-50 percent of the performance of the newer version above, which shows the rapid progress happening in WebAssembly tooling. Interpreters are about an order of magnitude slower but portable to all platforms. Nevertheless, browser engines achieve the goal of WebAssembly; they are ubiquitous and can potentially serve as a platform for portable offloading with high performance.

WebAssembly for Edge Computing: Alternatives and Trade-Offs

As we have seen, WebAssembly exhibits an unmatched portability-performance trade-off that makes it a worthwhile platform for edge offloading. Even though WebAssembly's main goal is to support client-side code in the web, adoption for general computing was anticipated [10] for its portability and the good tooling web technologies usually enjoy. In this section, we describe the current state of WebAssembly in edge computing and potential directions to truly achieve edge computing and its other fundamental requirements we identified at the beginning: migratability. Some of these are readily deployable with existing infrastructure while others require modifying existing WebAssembly toolchains. They also require different size and performance overheads as we shall discuss in this section.

State of WebAssembly on the Edge

Considering the portability of WebAssembly, attempts have been made to harness its potential in computational offloading at the edge. Particularly, WebAssembly is witnessing increasing adoption in serverless computing where standalone computational functions are offloaded, a paradigm that lends itself to edge computing due to its scalability and low maintenance complexity. Fastly built Compute@Edge [10], a serverless compute environment using WebAssemby. Hall and Ramachandran in [11] extensively test WebAssembly startup time under different use cases in a serverless environment and find significant advantages in some cases against containers.

Migrating WebAssembly

As we argued at the beginning, migratability is key in edge computing to remedy the relative instability of infrastructure compared to the cloud. Migratability across binary-incompatible platforms is a difficult challenge and an active area of research. We identify four possible methods for migrating WebAssembly that have been used for other platforms, and discuss the particular challenges with adapting them to WebAssembly.

Cold Migration: In the simplest form, static code is started as a new process on the target but persistent side-effects such as writes to disk are migrated. This fits paradigms where loads are standalone and small, making the occasional restarting of some loads inexpensive. This

approach can be improved to allow for semi-live migration as done by Jeong et al. in [14]. In their approach, WebAssembly modules are cold-migrated but the JavaScript runtime that encapsulates them is statefully migrated. The memory state of the JavaScript runtime which can include outputs of prior invocations to WebAssembly to persist while any actively running WebAssembly code at migration time needs to be restarted.

Interpreter-Based: When live migration is necessary, running WebAssembly in an interpreter allows straightforward live migration as the interpreter is omniscient to the state of the running program. This is effectively equivalent to running VMs in a non-native architecture which emulates the instruction set, in this case, the WebAssembly bytecode. The large cost with this approach is with performance, since interpreted execution is much slower than compiled execution. On the other hand. this approach allows debugging and inspecting the running program straightforwardly, and easier enforcement of security and isolation. Also, using an interpreter causes smaller memory and startup time overheads, and interpreters are more portable as they do not have to generate platform-specific output.

WebAssembly Instrumentation: When performance is also necessary, a possible approach is WebAssembly instrumentation. The WebAssembly input is instrumented to keep track of its own state at runtime. The instrumented code is run with an optimizing compiler, for example, the browser. On a migration request, the instrumented code on the source will produce a dump of the interrupted state which is used by the instrumented code on target to restore the state and resume execution. This approach was employed in [9] to instrument JavaScript. One drawback with their implementation is that the code size grows significantly with each migration. Moreover, there are a few challenges to consider when applying it to WebAssembly:

- WebAssembly is low-level. Whereas in JavaScript, we can add a state object which we populate with elements to save state, in WebAssembly we have to decide its address in memory and serialize the data.
- Instrumenting the call stack is tricky. The approach in [9] simply assumes an empty call stack during migration. Instrumenting will have to restore the call-stack at the target which is an architecture-dependent structure. WebAssembly does not support unstructured jumps (goto) making it difficult to jump to a saved call-stack address.

Binary Instrumentation: Another level at which instrumentation can occur is the native binary. The WebAssembly compiler can produce a binary instrumented with migration code such as responding to migration interrupts at safe migration points and running the instrumented binary. For each target platform, the compiler produces an instrumented binary such that the migration runtime can identify the equivalent points at migration time. This method requires enforcing a uniform data and code layout across architectures as discussed in [13], which implements this migration method for LLVM IR instead of WebAssembly.

Conclusion

In this article, we have identified code compatibility as a core problem in edge computational offloading and its requirements of portability and migratability. We surveyed existing technologies that achieve these goals either through a system-oriented approach or a language-oriented approach, and discussed their merits and drawbacks. Next, we presented WebAssembly and how it works and determined that it exceeds all other formats in terms of portability without losing performance. We identified migratability as the currently missing feature in WebAssembly to achieve edge offloading and devised four possible methods to achieve it with different trade-offs: using cold migrations, using an interpreter, instrumenting WebAssembly, and instrumenting native binary.

References

[1] M. Satyanarayanan, "The Emergence of Edge Computing," *Computer*, vol. 50, no. 1, 01 2017, pp. 30–39.

[2] K. Habak *et al.*, "Workload Management for Dynamic Mobile Device Clusters in Edge Femtoclouds," *Proc. SEC '17*, 2017.

[3] K. Ha *et al.*, "You Can Teach Elephants to Dance: Agile Vm Handoff for Edge Computing," *Proc. SEC '17*, 2017, pp. 12:1–14.

[4] L. Ma, S. Yi, and Q. Li, "Efficient Service Handoff Across Edge Servers via Docker Container Migration," *Proc. SEC '17*, 2017, pp. 11:1–13.

[5] A. E. Elgazar and K. A. Harras, "Enabling Seamless Container Migration in Edge Platforms," *Proc. CHANTS '19*, 2019.

[6] A. Jangda *et al.*, "Not So Fast: Analyzing the Performance of Webassembly vs. Native Code," *Proc. USENIX ATC '19*, 07 2019, pp. 107–20.

[7] J. Gascon-Samson *et al.*, "ThingsMigrate: Platform-Independent Migration of Stateful JavaScript IoT Applications," *Proc. ECOOP '18*, 2018, pp. 18:1–33.

[8] W. Felter *et al.*, "An Updated Performance Comparison of Virtual Machines and Linux Containers," *Proc. ISPASS '15*, 03 2015, pp. 171–72.

[9] L. Gherardi, D. Brugali, and D. Comotti, "A Java vs. C++ Performance Evaluation: A 3d Modeling Benchmark," *Lecture Notes in Computer Science*, vol. 7628, 11 2012.

[10] A. Haas *et al.*, "Bringing the Web Up to Speed With Webassembly," *Proc. PLDI '17*, 2017, p. 185–200.

[11] S. Gal-On and M. Levy, "Exploring Coremark™ – a Benchmark Maximizing Simplicity and Efficacy," EEMBC, Tech. Rep; available: https://www.eembc.org/techlit/articles/coremarkwhitepaper. pdf.

[12] P. Hickey, "How Fastly and the Developer Community Are Investing in the Webassembly Ecosystem;" available: www.fastly.com/blog/how-fastly-and-developer-communityinvest-in-webassembly-ecosystem.

[13] A. Hall and U. Ramachandran, "An Execution Model for Serverless Functions at the Edge," *Proc. IoTDI '19*, 2019, pp. 225–36.

[14] H.-J. Jeong *et al.*, "Seamless Offloading of Web App Computations From Mobile Device to Edge Clouds via html5 Web Worker Migration," *Proc. SoCC '19*, 2019, p. 38–49.

[15] A. Barbalace *et al.*, "Edge Computing: The Case for Heterogeneous-Isa Container Migration," *Proc. VEE '20*, 2020, p. 73–87.

Biographies

Mohammed Nurul Hoque (mntalateyya@live.com) graduated from Carnegie Mellon University Qatar (CMUQ) in 2020 with a bachelor's degree in computer science with outstanding academic achievement award. Previously a teaching assistant at CMUQ, he is currently a CPU compiler engineer at the UK-based Imagination Technologies. His research interests include edge computing, computational offloading and software execution environments.

Khaled Harras [SM] (kharras@cs.cmu.edu) is a Professor oc Computer Science at Carnegie Melon University Qatar (CMUQ), and Area Head of the Computer Science program there. He is the founder and director of the Networking Systems Lab. He has more than 140 refereed publications in various conferences, journals, and magazines. Along with his research group in the past few years, he has won the best national computing research award twice, received two best paper awards, and his work has been featured online various venues like MIT Tech Review and Tech the Future. To date, he has been involved in or managing research grants that amount to more than 4M USD, and has supervised over 30 different personnel. He is a Senior member of ACM..

Another level at which instrumentation can occur is the native binary. The WebAssembly compiler can produce a binary instrumented with migration code such as responding to migration interrupts at safe migration points and running the instrumented binary.