Mewz: Lightweight Execution Environment for WebAssembly with High Isolation and Portability Using Unikernels

Soichiro Ueda
Kyoto University
Kyoto, Japan
ueda@inet.media.kyoto-u.ac.jp

Ai Nozaki
University of Tokyo
Tokyo, Japan
nozaki@hal.ipc.i.u-tokyo.ac.jp

Daisuke Kotani
Kyoto University
Kyoto, Japan
kotani@media.kyoto-u.ac.jp

Yasuo Okabe
Kyoto University
Kyoto, Japan
okabe@media.kyoto-u.ac.jp

Abstract

Cloud computing requires isolation and portability for workloads. Virtual machines, for isolation, and containers, for portability, are widely used to achieve these requirements these days. However, using VMs and containers together entails two problems. First, VMs and containers have overheads that degrade performance. In addition, container images depend on host operating systems and architectures. To solve these problems, we propose a new system that distributes applications as WebAssembly (Wasm) and runs them as unikernels. Additionally, we propose a mechanism to convert a Wasm application into a unikernel image with the Wasm binary AoT-compiled to native code. We evaluated the performance of the system by running a simple HTTP server compiled into Wasm. The result showed that it ran Wasm applications with lower overhead than existing technologies.

1 Introduction

These days, many application developers use cloud computing to provision computing resources and deliver applications as containers. However, Combining containers and cloud computing entails two problems. First, container images depend on host operating systems and CPU architectures. Second, the virtualization overhead of both containers and VMs degrades performance.

There are existing solutions to each of these challenges. For example, substitution of containers with WebAssembly is proposed to solve the portability problem[8]. To reduce the overheads of guest kernel on VMs, previous researches proposed unikernels, which are specialized kernels for each application[6]. However, no solution solves both problems.

We propose a new system where applications are distributed as WebAssembly and run them as unikernels. This system solves both the portability and virtualization overhead problems. However, the issue lies in converting a WebAssembly binary into a unikernel image, because WebAssembly cannot be simply linked with kernel codes. Moreover, it is preferable to compile WebAssembly ahead of time (AoT) into native code for performance reasons. We devise a new mechanism to do it by exploiting WebAssembly System Interface, which is the standardized API for WebAssembly to access system resources. We combine an AoT-compiled WebAssembly binary and kernel codes that provide WebAssembly System Interface by symbol resolution. We realized it by developing a unikernel that provides WebAssembly System Interface and an AoT compiler that converts WebAssembly to native code.

We evaluate our system with an HTTP server that distributes static files. The result shows that our system executes Wasm applications with lower overhead than existing technologies.

2 Background

There are two main requirements for cloud computing: isolation and portability. Each workload must be sufficiently isolated to avoid interference between users[14]. Cloud providers must ensure kernel-level isolation between workloads of different tenants[7]. Also, cloud applications should be easily moved to different cloud environments or even to on-premises servers. To achieve the two requirements, VMs and containers are usually used. Users deploy their applications as container images and cloud providers run them on VMs using hypervisors. Containers must be used with VMs because containers do not provide isolation of host kernels unlike VMs. However, there are two problems with this method. This study focuses on solving these problems at the same time.

First, the portability of container images is restricted to the same host operating system and the same architecture. Containers cannot run on different operating systems since

HotInfra '24, November 3, 2024, Austin, TX, USA
.

they share the same kernel with the host Container images also contain binaries compiled for a specific architecture, so they cannot be executed on different architectures. This enforces container images to be made for each operating system and architecture. It becomes a problem in environments such as cloud-edge continuum[12], where the same applications run on both cloud and resource-restricted edge devices with different ISAs and OSes. Applications must be distributed in such a way that they can run on any OSes and architectures.

Second, the overheads incurred by VMs and containers, respectively, occur simultaneously. One of the overheads of virtual machines is incurred when applications access hardware[5]. Applications issue system calls to the guest kernel and the guest kernel accesses the virtual hardware provided by the hypervisor. Then, the hypervisor emulates the device by operating the physical hardware. Thus, it requires more time to access hardware than running applications on physical machines. Network isolation is one of the major overheads for containers[15]. In containers packet processing occurs twice: once at the virtual network interface of the container and once at the physical network interface of the host. This results in larger packet processing time compared to the case without containers. Consequently, using containers on a virtual machine accepts both overheads, resulting in a significant performance degradation[17].

Previous researches have proposed solutions to each of these problems. Our research aims to solve both problems at the same time, by combining two solutions.

Unikernels[1] are a kind of lightweight kernel designed to run in cloud. With unikernels, a single application is statically linked with the kernel into a single kernel image at build time. Thus, the application can call kernel functions directly, which reduces the overhead of system calls. However, since unikernel is built for an ISA, the kernel image of unikernel depends on the architecture. Running the same application with unikernels on different architectures requires building a unikernel image for each architecture. Therefore, unikernels do not solve the portability problem.

WebAssembly[3] (Wasm) is a portable binary format for executable code. Wasm can be executed on any host operating system and CPU architecture using a Wasm runtime. The OS/CPU architecture-independence of Wasm makes it a good candidate for solving the portability problem. Instead of distributing applications in the form of container images, applications may be compiled into Wasm and deployed. This allows the distribution of a single Wasm binary regardless of OSes and architectures of target environments. However, Wasm alone cannot provide the isolation required for cloud computing[13]. This is because Wasm shares the host kernel as well as containers. If developers run Wasm in cloud, virtual machine isolation will be required. This means that using Wasm as an alternative to containers does not solve the virtualization overhead issue.

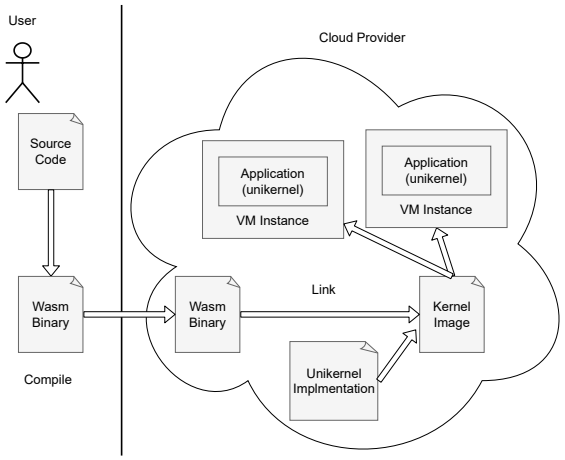

Figure 1. Architecture of the system

3 Design and Implementation

This section proposes a new system that combines Wasm and unikernels to solve the problems of portability and overheads.

First, applications are compiled into Wasm binaries and distributed. In cloud environments, Wasm binaries are linked into unikernel images. The unikernel images are then run on VMs. The figure 1 shows the architecture of the system.

This architecture benefits from the lightweightness of unikernels while ensuring the portability of binaries. With conventional unikernels, users must build a kernel image from source codes of applications, and distribute it to cloud. In this way, users cannot deploy applications to environments with other architectures quickly and easily. Some unikernels can execute Linux binaries[11]. Even with these unikernels, the built binary is dependent on the OS and the architecture. Thus, it is difficult to distribute portable binary enjoying the advantages of unikernels. With this system, developers can run applications with low overheads on cloud while they can deploy the same binary to any environments, such as edge devices with other architectures.

To implement this system, we need to convert Wasm binaries into unikernel images. But Wasm is a virtual ISA so it cannot be linked with kernel codes directly. Additionally, it is desirable to AoT compile Wasm to native code for better performance. Thus, it is a challenge to combine a Wasm binary and kernel codes into a unikernel image while converting Wasm to native codes.

The key to solving this challenge is WebAssembly System Interface (WASI). WASI[4] is a specification of API that provides system resources, such as network and file system. In other words, WASI acts like a system call in Wasm. Unlike general system calls, WASI provides its interface in the form of functions. WASI is defined only as a specification of its API, and its implementation is left to each runtime that uses

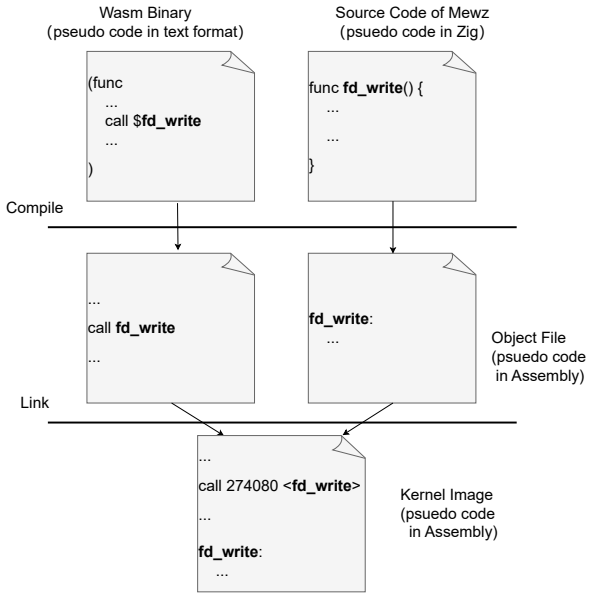

Figure 2. Linking process of Wasm and Mewz

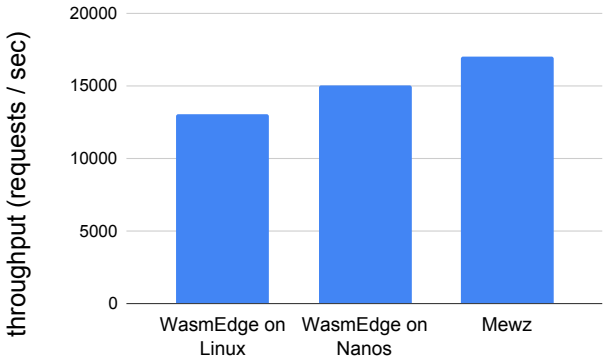

Figure 3. Result of the performance evaluation

WASI. Therefore, Wasm binaries have instructions that call WASI functions. As shown in the figure 2, we can build a unikernel image containing a Wasm application. First, we compile a Wasm binary into an object file with WASI function symbols unresolved. Then, we link the object file with kernel codes that have WASI functions. Thus, WASI acts as a boundary between an application and kernel codes, enabling Wasm to be linked with kernel codes. No other portable binary format, such as JVM bytecode and LLVM IR, has this feature. To do the above steps, we developed two softwares: Mewz and Wasker.

Mewz[9] is a unikernel that provides WASI API. Mewz is a unikernel specialized for Wasm execution so its functions are provided only through WASI. Mewz has only the functionality necessary for WASI implementation, which minimizes the functionality. It has features of memory management, network, a read-only file system, and so on. However, it does not implement thread functionality, since the current version of WASI does not have thread API. This shortage can be covered by scaling out the number of VM instances. It reduces the overhead of thread creation, switching, and scheduling.

Wasker[16] is an AoT compiler that converts a Wasm binary to native code on a target CPU architecture. Wasker compiles a Wasm binary into an object file, leaving WASI functions as unresolved symbols. This means the object file converted from Wasm does not contain WASI implementations. After AoT compilation by Wasker, the object file can be linked with Mewz by symbol resolution, as shown in the figure 2. Wasker uses LLVM as a backend for AoT compilation.

4 Performance Evaluation

For evaluation, we ran a simple HTTP server compiled into Wasm that distributes static files on the system. We compared the performance of the following three environments: Mewz, WasmEdge on Linux, and WasmEdge on Nanos.

Nanos is a unikernel that executes Linux application binaries[10]. Therefore it can run WasmEdge compiled for Linux.

For benchmarking, we used an HTTP benchmarking tool called wrk[2]. wrk continuously sends HTTP requests to a specified URL and measures the throughput. wrk keeps sending requests in parallel with a specified number of threads while maintaining a specified number of connections. In this evaluation, 16 threads and 800 connections were loaded for 60 seconds.

We prepared two physical machines and wrk was executed on one of them. A KVM virtual machine was created on the other machine and the target was executed on it. The two physical machines were connected with an L2 switch. On the KVM host, a Linux Bridge was created and connected to the TAP device assigned to the virtual machine.

The benchmark machine was equipped with an AMD Ryzen 9 5900HX CPU, 32GB of memory, and a 2.5Gbps ethernet interface. The KVM host machine had Intel Xeon E3-1245v5 CPU, 16GB of memory, and a 1Gbps ethernet interface.

The result of the evaluation is shown in the figure 3. The throughput of Mewz is 1.3 times higher than that of WasmEdge on Linux and 1.1 times higher than that of WasmEdge on Nanos. This result shows that Mewz has a lower overhead than WasmEdge on Linux and Nanos. Mewz allows Wasm to call kernel functions directly, which reduces the overhead of system calls. Additionally, Mewz entails no overhead of a Wasm runtime or other functionality that is not necessary for Wasm execution.

5 Conclusion

This paper proposed a new system that combines Wasm and unikernels. It solves the problems of portability and overheads when using containers and VMs in cloud computing. By adopting Wasm, it enables applications to be run on any host operating system and CPU architecture, unlike container images. Running Wasm as a unikernel reduces the overhead of guest OS accompanying virtual machine isolation. To implement this architecture, we developed a unikernel with WASI API and an AoT compiler that converts Wasm to native code. We evaluated the performance of the system by running a simple HTTP server compiled into Wasm. The result showed that it ran Wasm applications with lower overhead than existing technologies.

References

[1] Nabil El Ioini, Ayoub El Majjodi, David Hastbacka, Tomas Cerny, and Davide Taibi. 2023. Unikernels Motivations, Benefits and Issues: A Multivocal Literature Review. In *Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum*. 39–48.

[2] Will Glozer. 2022. *wrk - Modern HTTP benchmarking tool.* visited on 2024-01-28.

[3] WebAssembly Community Group. 2019. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/. visited on 2024-01-28.

[4] WebAssembly Community Group. 2020. WebAssembly System Interface. https://github.com/WebAssembly/WASI/blob/d8b286c697364d8bc4daf1820b25a9159de364a3/phases/snapshot/docs.md. visited on 2024-01-28.

[5] Zheng Li, Maria Kihl, Qinghua Lu, and Jens A Andersson. 2017. Performance overhead comparison between hypervisor and container based virtualization. In *2017 IEEE 31st International Conference on advanced information networking and applications (AINA)*. IEEE, 955–962.

[6] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 461–472.

[7] Ilias Mavridis and Helen Karatza. 2023. Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud. *Concurrency and Computation: Practice and Experience* 35, 11 (2023), e6365. https://doi.org/10.1002/cpe.6365 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6365

[8] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. WebAssembly as a Common Layer for the Cloud-edge Continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*. 3–8.

[9] Mewz. 2024. *A unikernel designed specifically for running Wasm applications and compatible with WASI.* https://github.com/mewz-project/mewz visited on 2024-09-09.

[10] Nanos. [n. d.]. *nanos - A kernel designed to run one and only one application in a virtualized environment.* visited on 2024-01-28.

[11] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 59–73. https://doi.org/10.1145/3313808.3313817

[12] Adrián Orive, Aitor Agirre, Hong-Linh Truong, Isabel Sarachaga, and Marga Marcos. 2022. Quality of Service Aware Orchestration for Cloud–Edge Continuum Applications. *Sensors* 22, 5 (2022). https://doi.org/10.3390/s22051755

[13] Aaron Schlesinger. 2024. *What is Hyperlight?* https://arschles.com/blog/hyperlight-overview-1/ visited on 2024-01-28.

[14] S. Subashini and V. Kavitha. 2011. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications* 34, 1 (2011), 1–11. https://doi.org/10.1016/j.jnca.2010.07.006

[15] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. 2018. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 189–197.

[16] Wasker. 2024. *WebAssembly AoT compiler for your favorite Operating System.* https://github.com/mewz-project/wasker visited on 2024-09-09.

[17] Fei Xu, Fangming Liu, Hai Jin, and Athanasios V Vasilakos. 2013. Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proc. IEEE* 102, 1 (2013), 11–31.