2022 IEEE International Symposium on Workload Characterization (IISWC) | 978-1-6654-8798-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/IISWC55918.2022.00028

How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes

Wenwen Wang

University of Georgia

Abstract

WebAssembly was originally created to facilitate the deployment of non-web applications on web platforms. Because of its high speed, safety, portability, and language independence, WebAssembly is increasingly adopted outside the web. This is generally enabled by standalone WebAssembly runtimes, which are dedicated to running WebAssembly binary code without web browsers. However, the characteristics of existing standalone WebAssembly runtimes are not clear due to the limited study in academia. To fill up this knowledge gap, we conduct a comprehensive characterization study of standalone WebAssembly runtimes in this paper. Our study covers five predominant standalone WebAssembly runtimes. Besides, we also construct a benchmark suite for the study, named WABench, which contains not only benchmark programs from existing mature benchmark suites but also whole applications from various application domains. Our study reveals several interesting findings. For example, we find that all studied runtimes introduce an average of 1.59× − 9.57× performance slowdown when running WebAssembly binaries compared to native executions, meaning that effective yet low-cost dynamic optimizations are needed. We also observe that standalone WebAssembly runtimes can make significant architectural impacts, e.g., more branch prediction misses and higher cache pressure. We hope our study can pave the way for expanding WebAssembly to more broad non-web domains.

1. Introduction

WebAssembly [15] is a low-level binary code format. It was created to facilitate the deployment of non-web applications on web platforms. Since its first announcement in 2015, WebAssembly has attracted a great deal of attention because of its distinctive features, such as near-native speed, programming language independence, sandboxing-level execution safety, and cross-platform hardware compatibility. In fact, major web browsers, including Mozilla Firefox, Google Chrome, Microsoft Edge, and Apple Safari, have actively supported WebAssembly. For instance, Google Chrome has abandoned its own portable native client (PNaCI) sandbox and switched to WebAssembly [12].

The advanced features of WebAssembly have also encouraged researchers and developers to push WebAssembly beyond the web. For example, recent research work has extended the application domains of WebAssembly to embedded systems [24], edge computing [8, 16], security [29], serverless computing [26], and remote computation offloading [11]. It is anticipated that WebAssembly will become a new foundation for pervasive computing [5].

A key enabler of WebAssembly outside the web is *standalone WebAssembly runtimes*, which are specifically developed to run WebAssembly binary code on a host physical machine without the need of web browsers. Typically, these WebAssembly runtimes exploit either interpretation or just-in-time (JIT) compilation [27] techniques to execute WebAssembly code. Thanks to the great simplicity of the core specification of WebAssembly, standalone WebAssembly runtimes are generally much more lightweight than traditional fully-fledged language runtimes, e.g., JavaScript engines and Java virtual machines. Besides, by implementing the WebAssembly system interface (WASI), standalone WebAssembly runtimes can also enable WebAssembly code to access host system resources, e.g., files and network sockets.

Although standalone WebAssembly runtimes are critical in extending WebAssembly beyond the web, it is not clear what characteristics they exhibit in general. In particular, how is the performance efficiency of standalone WebAssembly runtimes when running WebAssembly binaries? What amount of extra memory resources is consumed by a standalone WebAssembly runtime? Do standalone WebAssembly runtimes have an impact on hardware branch prediction and cache management? Answering these questions is essential to estimate the benefits and potential risks of extending WebAssembly to a new application domain. Unfortunately, there is no existing characterization study of standalone WebAssembly runtimes. Previous performance studies of WebAssembly only focus on web browsers [17, 43], and thereby provide very limited insight into standalone WebAssembly runtimes.

The goal of this paper is to fill up this knowledge gap. More specifically, this paper conducts a comprehensive characterization study of standalone WebAssembly runtimes. Our study covers five predominant standalone WebAssembly runtimes, including Wasmtime [37], WAVM [38], Wasmer [36], Wasm3 [35], and WAMR [39]. These runtimes target various application scenarios of WebAssembly, and are actively maintained by independent organizations/communities. To conduct the study, we also construct a benchmark suite, named **WABench**, which not only includes a large number of benchmarks from existing mature benchmark suites, e.g., JetStream2 [18], MiBench [14], and PolyBench [25], but also assembles a set of *whole applications* from a variety of application domains, such as machine learning, natural-language processing, computer vision, gaming, database, and big data processing. By compiling the benchmarks into WebAssembly (with WASI) and running the compiled WebAssembly binaries on the standalone WebAssembly runtimes, we can systematically study the characteristics of the runtimes. Our study reveals several interesting and insightful findings about the runtimes. The source code of WABench is publicly available at https://github.com/wabench/wabench.

In summary, we make the following contributions in this paper:

- We conduct a comprehensive characterization study of standalone WebAssembly runtimes. To the best of our knowledge, this is the first time multiple standalone WebAssembly runtimes are thoroughly and profoundly studied.
- We construct a WebAssembly benchmark suite, named WABench, which includes not only benchmarks from existing benchmark suites but whole applications from various domains. WABench is also the first benchmark suite that targets standalone WebAssembly runtimes (to the best of our knowledge). It is publicly available to facilitate future research.
- We summarize several interesting and insightful findings from our study. These findings adequately depict the characteristics of existing standalone WebAssembly runtimes. They can also serve as a starting point of future research.

The rest of this paper is organized as follows. Section 2 presents the background knowledge of WebAssembly and standalone WebAssembly runtimes. Section 3 describes the methodology of our study. Section 4 conducts the performance efficiency study of WebAssembly runtimes. Section 5 performs the memory overhead study of WebAssembly runtimes. Section 6 is dedicated to studying the architectural implications of WebAssembly runtimes. Section 7 discusses our findings. Section 8 compares our work with related work. And Section 9 concludes the paper.

2. Background

In this section, we present the background knowledge of WebAssembly and standalone WebAssembly runtimes.

2.1. WebAssembly

WebAssembly was originally introduced to facilitate the deployment of existing applications on web platforms. Such applications are usually *not* developed with web programming languages like JavaScript. Rather, they are probably written in native languages, e.g., C/C++. Therefore, WebAssembly was designed as a *language-neutral* and *platform-independent* compilation target. In principle, source programs written in any high-level languages can be compiled to WebAssembly. This clearly differentiates WebAssembly from existing language-level intermediate representations, e.g., Java bytecode.

The WebAssembly core specification defines the instruction set architecture (ISA) of WebAssembly. This includes a list of WebAssembly *instructions*, e.g., memory load/store instructions, numeric instructions, and control flow instructions. Since WebAssembly adopts a structured *stack machine* for computation modeling, most WebAssembly instructions manipulate values on an implicit operand stack, though real implementations may not necessarily have a stack structure. WebAssembly binaries are organized into *modules*, which are the basic compilation and deployment unit. This is similar to executable files of native binaries. The major difference is that WebAssembly modules include the type information, which can be used to enhance the security of WebAssembly modules, e.g., enforcing control-flow integrity [40].

WebAssembly System Interface (WASI). An essential requirement to push WebAssembly beyond the web is to enable WebAssembly programs to access system resources, e.g., files and network sockets. This is necessary because many real-world applications depends on accessing system resources to complete their computation tasks. However, WebAssembly itself is just a binary code format and thus does not provide such a mechanism. To address this issue, WebAssembly developers proposed WASI, which specifies how a WebAssembly program can interact with the host platform. Specifically, WASI defines a set of *functions*, through which WebAssembly programs can access system resources on the host platform. These functions are typically implemented by standalone WebAssembly runtimes. They are very similar to the functions defined by the portable operating system interface (POSIX). But, it is worth pointing out that WASI is *not* exactly the same as POSIX. That is, a POSIX-compatible program may not be compiled directly to WASI. Nevertheless, it is possible to implement a POSIX-compatible library on top of WASI, e.g., WASI Libc [33].

To compile source programs written in high-level programming languages, e.g., C/C++, into WebAssembly with WASI, a developer can use the WebAssembly compiler toolchain provided by the WASI SDK [34]. Note that the Emscripten [6] compiler toolchain can also compile source programs to WebAssembly. However, our experience shows that the compiled WebAssembly binaries are not completely compatible with WASI [1]. This is probably because Emscripten mainly targets web platforms. Therefore, in this paper, we use the WebAssembly compiler of the WASI SDK to compile benchmark programs for our study.

2.2. Standalone WebAssembly Runtimes

Though WebAssembly binaries are low-level binary code, they are actually *not* executable and thus cannot be executed directly on a host machine. One possible way to run WebAssembly binaries is to rely on web browsers, which leverage JavaScript engines to execute WebAssembly code. Since web browsers are generally unavailable outside the web, standalone WebAssembly runtimes are required to run WebAssembly code in non-web domains.

Typically, there are two ways to execute WebAssembly code: *interpretation* and *just-in-time (JIT) compilation*. The interpretation way leverages a list of functions written in high-level languages to interpret WebAssembly instructions. Each time when a WebAssembly instruction needs to be executed, the corresponding function of the instruction will be executed to emulate the functionality of the instruction. In contrast, the JIT compilation way compiles WebAssembly code into native binary code on the fly. The functionality of WebAssembly code is then realized by executing the compiled native code. Since the interpretation way does not need to implement a code generator backend, it can be easily ported across different hardware platforms. Also, no code generation usually implies low compilation cost. But, the compilation overhead of JIT compilation is just a one-time cost and can be partially amortized if the compiled code is executed many times [31].

In addition, some standalone WebAssembly runtimes also support the ahead-of-time (AOT) compilation technique. The basic idea is to compile WebAssembly code into native binary code in advance. The compiled native code is stored as a disk file and loaded into the memory when the WebAssembly code needs to be executed. This can mitigate the performance overhead incurred by JIT compilation at runtime. We will investigate the impact of AOT compilation in our study.

3. Methodology

In this section, we describe the methodology of our study. We start with the selection of standalone WebAssembly runtimes, followed by the construction of the benchmark suite WABench and the description of our experimental environment.

3.1. WebAssembly Runtimes

To select target standalone WebAssembly runtimes for the study, we use the following selection criteria:

- The runtime is a standalone WebAssembly runtime and supports WebAssembly binary code compiled with WASI.
- The runtime is sufficiently mature to run a broad range of WebAssembly applications.
- The runtime is actively developed and maintained along with the evolution of WebAssembly and WASI.

After examining 33 existing standalone WebAssembly runtimes, we finally select 5 runtimes, which cover various application scenarios of WebAssembly and occupy the mainstream of standalone WebAssembly runtimes. Table 1 shows the five runtimes. We next describe more details about them.

Wasmtime. Wasmtime [37] is the official WebAssembly runtime developed by Bytecode Alliance [2], which is an organization of many well-known companies committed to establishing WebAssembly standards, foundations, and ecosystems. Wasmtime adopts JIT compilation to compile and run WebAssembly binaries. The JIT compiler is built on Cranelift [4], a retargetable code generator that translates WebAssembly code into executable native binary code.

TABLE 1: WebAssembly runtimes selected for our study.

Runtime	Lang.	LoC	Type	#Stars[a]	History
Wasmtime	Rust	314K	JIT compilation	7.6K	2 years
WAVM	C/C++	98K	JIT compilation	2.2K	2 years
Wasmer	Rust	154K	JIT compilation	12.4K	3 years
Wasm3	C	120K	Interpretation	5K	2 years
WAMR	C	145K	Interpretation	2.8K	2 years

[a]The number of stars received by the GitHub repository.

WAVM. WAVM [38] is another standalone WebAssembly runtime. It is mainly maintained by the open source community. It employs an LLVM-based JIT compiler to compile and run WebAssembly code. Specifically, WAVM lifts WebAssembly code into LLVM IR [22] and then invokes an LLVM JIT compiler to compile LLVM IR into native binary code. Benefiting from the rich optimizations in the LLVM infrastructure, WAVM is able to generate high-quality native binary code.

Wasmer. Wasmer [36] is also a WebAssembly runtime maintained by the open source community. In addition to being used as a standalone runtime, it can also be used as a library embedded in various languages. Wasmer supports multiple JIT compilers: SinglePass, Cranelift, and LLVM, which have tradeoffs between compilation speed and native code quality. Our study mainly uses Cranelift for Wasmer, as it is the default one. But, we will study the impact of different JIT compilers.

Wasm3. Different from previous runtimes, Wasm3 [35] is an interpretation-based WebAssembly runtime. Therefore, it naturally supports many different hardware platforms. It is also maintained by the open source community. Similar to Wasmer, it can also be used as a library embedded in different languages.

WAMR. WebAssembly Micro Runtime (WAMR) [39] is designed as a lightweight standalone WebAssembly runtime. It targets computer devices with low power and limited resources, e.g., embedded, IoT, and edge devices. Hence, it adopts interpretation to maintain portability for diverse hardware platforms. It is also a project of Bytecode Alliance.

3.2. WABench

To conduct the study, we also need to use a set of benchmark programs to exercise the standalone WebAssembly runtimes. Therefore, we create **WABench**, a benchmark suite that contains 50 benchmark programs for benchmarking standalone WebAssembly runtimes. Some benchmark programs in WABench are from existing mature benchmark suites, i.e., JetStream2 [18], MiBench [14], and PolyBench [25], while the others are constructed from scratch using whole applications in various application domains. Since the WASI SDK is still at a very early

TABLE 2: The benchmark programs included in WABench.

	Benchmark Program		Application Domain	Description	Workload
1	JetStream2	gcc-loops	Compilation	Loops used to tune GCC vectorizer	Default workload provided by the original benchmark suite
2		hashset	Hash table	Hash table operations of web page loading	
3		quicksort	Data Sorting	Quick sort algorithm implementation	
4		tsf	Data processing	Implementation of a typed stream format	
5	MiBench	basicmath	Automotive	Basic mathematical computations	Default workload provided by the original benchmark suite
6		bitcount	Automotive	Bit manipulations	
7		jpeg	Consumer multimedia	JPEG image compression/decompression	
8		stringsearch	Office automation	Searching given words in phrases	
9		blowfish	Security	Symmetric block cipher	
10		rijndael	Security	Block cipher with variable length keys	
11		sha	Security	Secure hash algorithm	
12		adpcm	Telecommunications	Adaptive differential pulse code modulation	
13		crc32	Telecommunications	32-bit Cyclic Redundancy Check	
14	PolyBench	correlation	Data mining	Correlation computation	Default workload provided by the original benchmark suite
15		covariance	Data mining	Covariance computation	
16		gemm	Linear algebra	Matrix multiplication	
17		gemver	Linear algebra	Vector multiplication and matrix addition	
18		gesummv	Linear algebra	Scalar, vector and matrix multiplication	
19		symm	Linear algebra	Symmetric matrix multiplication	
20		syr2k	Linear algebra	Symmetric rank-2k operations	
21		syrk	Linear algebra	Symmetric rank-k operations	
22		trmm	Linear algebra	Triangular matrix multiplication	
23		2mm	Linear algebra	Two matrix multiplications	
24		3mm	Linear algebra	Three matrix multiplications	
25		atax	Linear algebra	Matrix transpose and vector multiplication	
26		bicg	Linear algebra	BiCG sub kernel of BiCGStab linear solver	
27		doitgen	Linear algebra	Multiresolution analysis kernel	
28		mvt	Linear algebra	Matrix vector product and transpose	
29		cholesky	Linear algebra	Cholesky decomposition	
30		durbin	Linear algebra	Toeplitz system solver	
31		gramschmidt	Linear algebra	Gram-Schmidt	
32		lu	Linear algebra	LU decomposition	
33		ludcmp	Linear algebra	LU decomposition	
34		trisolv	Linear algebra	Triangular solver	
35		deriche	Image processing	Edge detection filter	
36		floyd-warshall	Physics simulation	Computing shortest paths in a graph	
37		nussinov	Dynamic programming	Sequence alignment	
38		adi	Systems control	Alternating direction implicit solver	
39		fdtd-2d	Physics simulation	2-D finite-difference time-domain kernel	
40		heat-3d	Physics simulation	Heat equation over 3D data domain	
41		jacobi-1d	Linear algebra	1-D jacobi stencil computation	
42		jacobi-2d	Linear algebra	2-D jacobi stencil computation	
43		seidel-2d	Linear algebra	2-D seidel stencil computation	
44	Whole Applications	bzip2	File management	File compression/decompression	Compressing a 120MB data file
45		espeak	NLP	Text-to-Speech synthesizer	Reading out a 200K text file
46		facedetection	Computer vision	Detecting human faces in images	Detecting faces in a photo
47		gnuchess	Gaming	Chess-playing game	Playing a single round game (depth 10)
48		mnist	Machine learning	A neural network for digit recognition	Training the network for 1000 iterations
49		snappy	Big data processing	Data compression/decompression library	Compressing 512MB memory data
50		whitedb	Database	Lightweight NoSQL database	Conducting a set of database operations

stage, many applications cannot be compiled with the provided WebAssembly compiler toolchain. This regretfully forces us to exclude many interesting benchmark programs in existing benchmark suites and whole applications. To summarize, we include an application in the benchmark suite mainly based on the following criteria:

- The application is actively developed and maintained.
- The application is a representative example in its problem domain.
- The application has no potential copyright/license issues.
- The application can be successfully compiled to WebAssembly + WASI and executed by all studied standalone WebAssembly runtimes.

Here, we would like to point out that we do not consider whether an application is typically running on a standalone WebAssembly runtime as a criterion. The reason is that pushing WebAssembly to non-web domains is still at a very early stage. Instead of limiting the benchmark suite to existing applications already running on standalone WebAssembly runtimes, we can provide more insights for standalone WebAssembly runtime users/developers by including more diverse applications, especially when they want to apply WebAssembly to unprecedented problem domains. Table 2 lists the benchmark programs in WABench. Next, we discuss them in detail.

JetStream2. This benchmark suite was developed to benchmark the speed and smoothness of web browsers. It contains both JavaScript and WebAssembly benchmarks in different web application scenarios. We only include WebAssembly benchmarks into WABench to study the support of web applications in standalone WebAssembly runtimes.

MiBench. As an embedded benchmark suite, MiBench contains embedded workloads in various domains, e.g., automotive and industrial control, security, office automation, telecommunications. We include all MiBench benchmarks in WABench except those that cannot be compiled to WebAssembly due to the fledgling WASI. Through these benchmarks, we can understand the support of embedded applications in existing standalone WebAssembly runtimes.

PolyBench. This benchmark suite includes 30 numerical computation kernels extracted from a wide range of application domains. It was initially created to evaluate the effectiveness of polyhedral compilation techniques. Since WebAssembly runtimes also involve the compilation of WebAssembly code, we include the kernels into WABench to study the JIT compilation processes in standalone WebAssembly runtimes.

bzip2. bzip2 [3] is a popular data compression tool available on many Unix/Linux systems. It can achieve a very high compression ratio, while maintaining a high execution speed. We include it into WABench to study how standalone WebAssembly runtimes support such a commonly used tool.

eSpeak. eSpeak [7] is a compact open source speech synthesizer. It can synthesize clear speeches in English or other languages from a text file. The speech speed and voice can also be adjusted through the command line interface. The purpose of including this application into WABench is to understand how natural language processing (NLP) applications are supported in WebAssembly runtimes.

Face Detection. Nowadays, face detection has been widely used in many different applications, e.g., access control, security, photography, healthcare, and marketing. Given an image, a face detection algorithm is able to find human faces in the image. The face detection benchmark in WABench leverages a convolution neural network (CNN) model to perform the detection [20]. This application allows us to study how WebAssembly runtimes support a computer vision application.

GNU Chess. As one of the earliest chess games that is still actively maintained nowadays, GNU Chess [10] aims to serve as a research basis for chess game development. It can perform at the senior master/weak international master strength. It is often used in conjunction with a graphical user interface (GUI), e.g., XBoard [42] or GNOME Chess [9]. Through it, we can study how gaming applications are supported by standalone WebAssembly runtimes.

TABLE 3: The hardware and software configurations of our experimental environment.

	Configuration
CPU	Intel Xeon CPU E5-1620 v4 @ 3.5GHz 4 cores, 8 threads L1-D Cache: 32K L1-I Cache: 32K L2 Cache: 256K L3 Cache: 10240K
Main Memory	32GB DDR4 2400MHz
Storage Device	1TB SATA hard disk drive
Operating System	Ubuntu 20.04 with Linux-4.15.0
Native Compiler	LLVM-10.0.0
WASI SDK	Version 10.0 (based on LLVM-10.0.0)

MNIST. The recent advances in machine learning techniques have benefited a broad range of applications. To understand how machine learning applications are supported with standalone WebAssembly runtimes, we include the mnist application [23] into WABench. It is an implementation of a neural network algorithm to recognize handwritten digits using the MNIST data set [19]. The algorithm is able to reach an accuracy of around 92% after training with 1000 iterations.

Snappy. This is a big data compression/decompression library developed by Google [13]. Rather than achieving maximum compression ratios, the major purpose of the Snappy library is to provide a good balance between high compression speeds and reasonable compression ratios. It has been extensively used to compress and decompress petabytes of data in Google's production environment. We include it into WABench to study how standalone WebAssembly runtimes support applications in real production environments.

WhiteDB. As a lightweight and portable NoSQL database library written in C, WhiteDB [41] operates data records completely in main memory. It has a very small memory footprint and can be easily integrated into existing applications. By including it in WABench, we can study the support of a database application in standalone WebAssembly runtimes.

3.3. Experimental Environment

Table 3 shows the detailed configurations of our experimental environment. The hardware platform is equipped with an Intel Xeon E5-1620 CPU and 32GB DDR4 main memory. The operating system is Ubuntu 20.04 with Linux-4.15.0 and the native C/C++ compiler is LLVM-10.0.0, which is used to compile the benchmark programs to native binaries. The version of the WASI SDK is 10.0, which was developed based on LLVM-10.0.0. Note that in each experiment we use the *same* compiler option to compile native and WebAssembly binaries. During our study, the platform is occupied exclusively by our experiments to reduce the potential impact of random factors.

4. Performance Efficiency

In this section, we study the performance efficiency of standalone WebAssembly runtimes. We first give an overview of the overall performance of the runtimes and then study the performance impact of JIT compilers, AOT compilation, and WebAssembly compiler optimizations.

4.1. Overall Performance

Figure 1 shows the normalized execution times of the benchmark programs in WABench running with different standalone WebAssembly runtimes. The baseline here is the execution times of native binaries. Note the WebAssembly binaries and the native binaries are compiled from the benchmark programs using the same compiler optimization level, i.e., `-O2`.

As we can see from the figure, for all benchmarks, the studied standalone WebAssembly runtimes introduce more or less performance slowdown compared to the native execution. This is as expected because every runtime needs to do some extra work in order to execute WebAssembly binaries, e.g., parsing and compiling WebAssembly code. Among all evaluated benchmark programs and runtimes, the highest performance slowdown happens when WAVM executes the *jpeg* benchmark, i.e., 135.11×. Interestingly, WAVM also exhibits the lowest performance slowdown when executing the *adi* benchmark, i.e., 1.01×. This clearly shows the significant performance difference when running *different* WebAssembly binaries with the *same* runtime. In other words, the performance of a standalone WebAssembly runtime is substantially affected by what WebAssembly binary to run. Overall, compared to the native execution, the five standalone WebAssembly runtimes introduce an average performance slowdown of 1.67× (Wasmtime), 3.54× (WAVM), 1.59× (Wasmer), 6.99× (Wasm3), and 9.57× (WAMR), respectively.

Another interesting observation we can make from Figure 1 is that, in most cases, the performance of the JIT compilation-based runtimes, i.e., Wasmtime, WAVM, and Wasmer, have a better performance than those interpretation-based runtimes, i.e., Wasm3 and WAMR. This is probably because WebAssembly JIT compilers are capable of generating native binary code with a better performance efficiency than the interpreter functions written in high-level programming languages. However, JIT compilation is not free [30]. In particular, JIT compilation time may account for a substantial portion of the total execution time of a benchmark program. This is demonstrated by benchmarks that have extremely short running times, e.g., *quicksort*, which finishes the execution in less than one second. Our further investigation shows that, for short-running benchmarks, the three JIT-based runtimes usually exhibit a higher performance slowdown than long-running benchmarks. In contrast, the performance slowdown of the

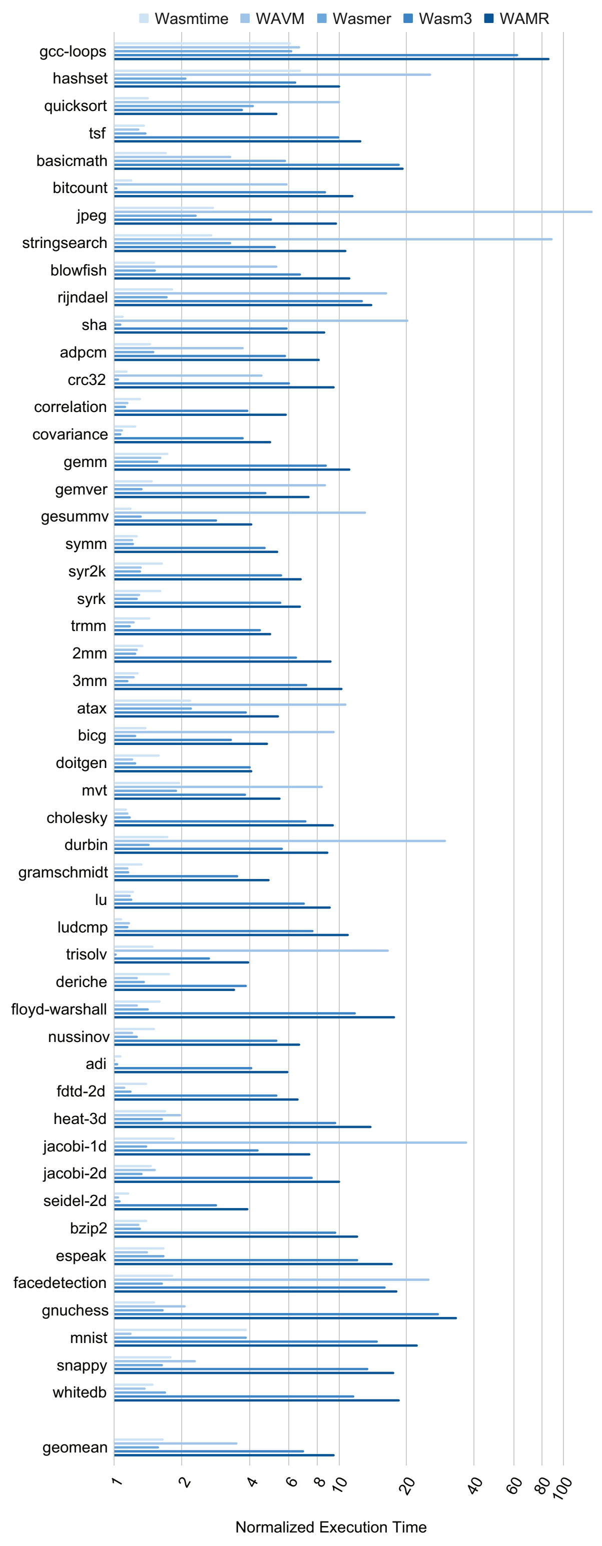

Figure 1: Normalized execution times of the benchmarks in WABench with different standalone WebAssembly runtimes. The baseline is the native execution.

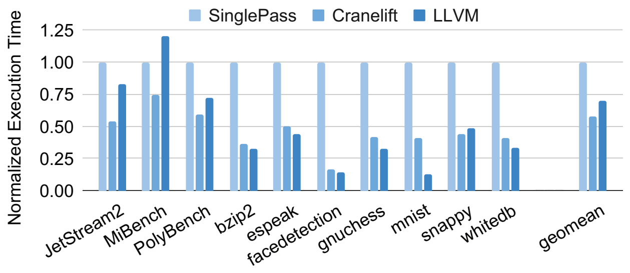

Figure 2: The normalized execution times of Wasmer with different JIT compilers. The baseline is SinglePass.

two interpretation-based runtimes is relatively stable for both short-running and long-running benchmarks.

Finding 1. All five standalone WebAssembly runtimes introduce additional performance overhead when executing WebAssembly binaries, compared to the native execution. The average performance slowdown ranges from 1.59× (Wasmer) to 9.57× (WAMR).

4.2. Impact of WebAssembly JIT Compilers

To study the impact of JIT compilers on the performance efficiency of a standalone WebAssembly runtime, we use Wasmer as the target runtime, because it supports three different JIT compilers: SinglePass, Cranelift, and LLVM. Figure 2 shows the normalized execution times of the benchmark programs with different JIT compilers in Wasmer. Here, the baseline is SinglePass. Due to the space limitation, we aggregate the results of the benchmarks from JetStream2, MiBench, and PolyBench, respectively, and only show their geometric means. See Figure 11 in Appendix for more detailed results.

As shown in the figure, for JetStream2, MiBench, and PolyBench, the Cranelift JIT compiler achieves the best overall performance. However, for all whole applications except *snappy*, the LLVM JIT compiler outperforms the other two JIT compilers. This shows that it is not easy for a WebAssembly JIT compiler to always achieve the best performance for different applications. On the other side, this also implies that there is still room for all the three JIT compilers to further improve the performance. Overall, compared to the SinglePass JIT compiler, the Cranelift JIT compiler can reduce the execution time by 42.49%, while the LLVM JIT compiler can reduce 29.93% on average.

Finding 2. Among the three WebAssembly JIT compilers, Cranelift and LLVM have the best performance for different sets of benchmark programs. On average, compared to SinglePass, Cranelift has a performance speedup of 1.74× while LLVM has a speedup of 1.43×.

4.3. Impact of AOT Compilation

We next study the impact of ahead-of-time (AOT) compilation on the performance efficiency of WebAssembly runtimes. Figure 3 shows the performance speedup achieved with AOT enabled for the three JIT compilation-based runtimes. The baseline is the original performance of the corresponding runtime without AOT. Here, as before, we aggregate the results of benchmarks from JetStream2, MiBench, and PolyBench, respectively, and only show their geometric means. See Figure 12 in Appendix for more detailed results. As shown in the figure, AOT compilation achieves substantial performance improvements for WAVM. In particular, for *facedetection*, the performance speedup is as high as 14.19×. Our further investigation shows that benchmarks like *facedetection* have very short running times, while their dynamic code footprints are rather large. Hence, AOT compilation can effectively mitigate the overhead incurred by JIT compilation, especially when massive optimizations need to be applied. On average, the performance speedup achieved by AOT compilation is 1.02× for Wasmtime, 1.73× for WAVM, and 1.02× for Wasmer.

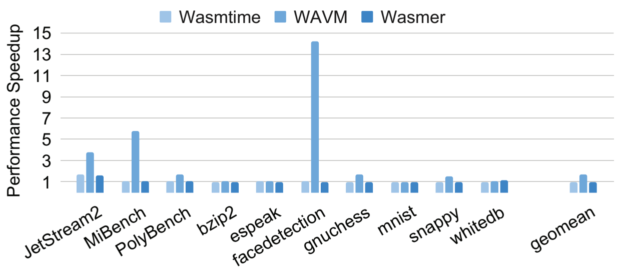

Figure 3: Performance speedup achieved by enabling AOT compilation in different WebAssembly runtimes. The baseline is the executions without AOT compilation.

TABLE 4: AOT compilation times of different WebAssembly runtimes (in seconds). The numbers in parentheses show the percentage of AOT compilation times in total execution times of WebAssembly runtimes without AOT.

	Wasmtime	**WAVM**	**Wasmer**
JetStream2	0.49 (8.36%)	3.43 (45.86%)	0.14 (2.46%)
MiBench	0.07 (42.86%)	0.89 (87.55%)	0.07 (25.68%)
PolyBench	0.04 (0.40%)	0.41 (4.04%)	0.04 (0.43%)
bzip2	0.09 (0.46%)	1.38 (7.30%)	0.10 (0.53%)
espeak	0.08 (0.27%)	2.14 (8.36%)	0.12 (0.39%)
facedetection	0.30 (72.09%)	5.31 (93.79%)	0.26 (69.57%)
gnuchess	0.07 (2.56%)	1.67 (43.06%)	0.07 (2.37%)
mnist	0.05 (0.02%)	0.47 (0.60%)	0.05 (0.02%)
snappy	0.05 (3.65%)	0.57 (34.09%)	0.05 (4.08%)
whitedb	0.05 (0.43%)	0.96 (8.40%)	0.05 (0.38%)
Average	**0.09 (0.67%)**	**0.93 (9.52%)**	**0.06 (0.48%)**

Table 4 illustrates the detailed AOT compilation times of the three WebAssembly runtimes. As we can see from the table, for most benchmarks, the AOT compilation times of the three WebAssembly runtimes are less than one second. The only exception is WAVM, which may have a relatively long AOT compilation process for some benchmarks, e.g., *facedetection* and *hashset* in JetStream2 (not shown in the table). This is because these benchmarks have a large

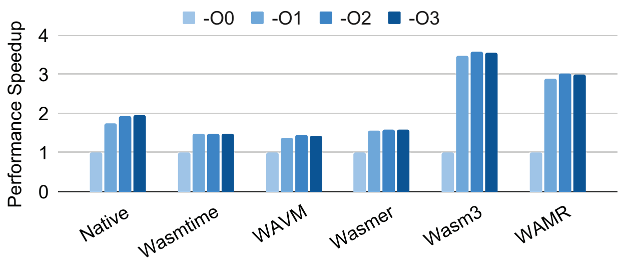

Figure 4: Performance speedup achieved by WebAssembly compiler optimizations. The baseline is `-O0`. The results are geometric means of *all* benchmarks in WABench.

WebAssembly binary code size. Overall, the average AOT compilation time is less than one second for all the three WebAssembly runtimes.

Finding 3. AOT compilation has a substantial impact on the performance of WAVM (1.73× performance speedup), while the impact is quite limited for Wasmtime (1.02× speedup) and Wasmer (1.02× speedup).

4.4. Impact of Compiler Optimizations

In general, a high compiler optimization level, e.g., `-O3`, aggressively applies more optimizations than a low optimization level, e.g., `-O1`, and thereby can generate binary code with better performance efficiency. However, it is not clear how WebAssembly compiler optimizations affect the performance efficiency of standalone WebAssembly runtimes when running the compiled WebAssembly binaries. Therefore, to study the impact, we first compile each benchmark program into multiple WebAssembly binaries using different compiler optimization levels, including `-O0`, `-O1`, `-O2`, and `-O3`, and then compare the performance of the standalone WebAssembly runtimes when running them. In this way, we can understand the impact of WebAssembly compiler optimizations on the performance of standalone WebAssembly runtimes.

Figure 4 shows the results of the performance speedup achieved through WebAssembly compiler optimizations. Here, for each standalone WebAssembly runtime, the baseline is the performance of the runtime running WebAssembly binaries compiled with `-O0`. The results are geometric means of *all* benchmarks from WABench. We also include the results of native binaries here for reference. As we can see from the figure, WebAssembly binaries compiled with higher optimization levels can generally lead to better performance of WebAssembly runtimes. For example, the WebAssembly binaries compiled with `-O2` can bring in 3.57× performance speedup for Wasm3. An interesting observation we can make from the figure is that the two interpretation-based runtimes, i.e., Wasm3 and WAMR, are *more* in favor of WebAssembly binaries compiled with high optimization levels than the three JIT compilation-based runtimes, i.e., Wasmtime, WAVM and

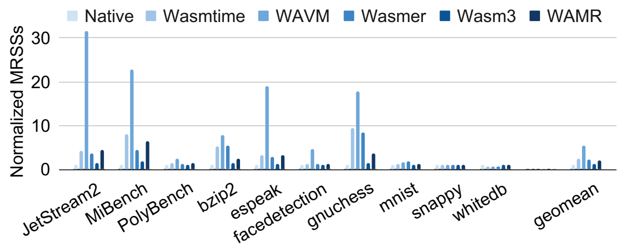

Figure 5: Normalized maximum resident set sizes (MRSSs) of WebAssembly runtimes. The baseline is the native execution.

Wasmer. A possible explanation for this phenomenon is that the interpretation method is more sensitive to the quality of the input WebAssembly binary code due to less optimizations applied during the interpretation process.

From Figure 4, we can also conclude that the same high optimization level can usually bring in more performance benefits for native binaries than the JIT compilation-based WebAssembly runtimes. For instance, the `-O2` optimization level can lead to 1.94× performance speedup for native binaries, while it only results in 1.44× performance speedup for WAVM. In other words, this suggests that the WebAssembly binaries compiled with a high optimization level may still miss some optimizations that are applied when native binaries are compiled with the same optimization level. Further investigation is necessary to understand why such optimizations are missed when WebAssembly binaries are compiled, as well as figure out whether it is feasible to enable them in either the WebAssembly compiler or the JIT compiler of a standalone WebAssembly runtime.

Finding 4. WebAssembly compiler optimizations can lead to considerable performance improvements for different WebAssembly runtimes, e.g., 1.44× − 3.57× performance speedup with the `-O2` optimization level compared to `-O0`.

5. Memory Overhead

In this section, we study the memory overhead incurred by standalone WebAssembly runtimes. Since a standalone WebAssembly runtime is actually a dynamic execution environment of WebAssembly binary code, it inevitably consumes extra memory resources when running WebAssembly binaries, compared to native executions. Therefore, the memory overhead is also an important characteristic of standalone WebAssembly runtimes.

In our study, to understand the memory overhead of standalone WebAssembly runtimes, we measure the *maximum resident set size* (MRSS), which represents the peak size of physical memory consumed by an application. Figure 5 shows the normalized results, where the baseline is the result of the native execution. Similarly, we aggregate the results of JetStream2, MiBench, and PolyBench, respectively, and only show their geometric means. See Figure 13

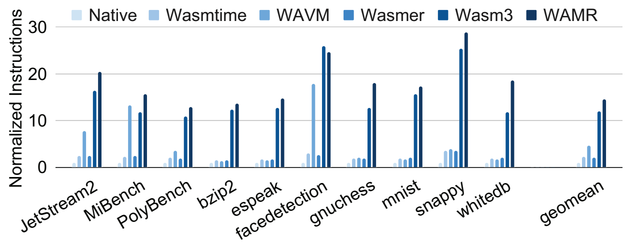

Figure 6: Normalized numbers of dynamic instructions of WebAssembly runtimes. The baseline is the native execution.

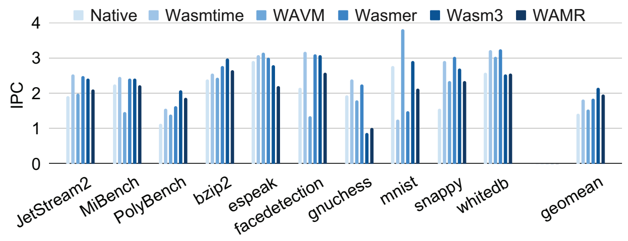

Figure 7: The instructions per cycle (IPC) of the native execution and WebAssembly runtimes.

in Appendix for more detailed results. As shown in the figure, for all benchmarks except *whitedb*, the standalone WebAssembly runtimes consume more memory resources compared to the native execution. Surprisingly, the three JIT compilation-based runtimes consume less memory for *whitedb*. A possible reason for this phenomenon is that the memory allocated by the application is actually not used and therefore optimized by these runtimes. In most cases, WAVM consumes most memory resources among the five runtimes, while Wasm3 consumes least memory resources. For example, WAVM consumes 31.66× physical memory for JetStream2 benchmarks compared to the native execution, while Wasm3 only consumes 1.55×. This might because a JIT compilation-based runtime is more heavyweight than an interpretation-based runtime.

Another interesting observation we can make from Figure 5 is that the additional memory resources consumed by WebAssembly runtimes are very incremental for some whole applications, e.g., *mnist* and *snappy*. Our further investigation shows that these whole applications consume a lot of memory themselves. For instance, *snappy* needs more than 1GB memory resource. As a consequence, the extra memory consumed by WebAssembly runtimes only account for a small portion of the total memory.

Finding 5. The memory resources consumed by standalone WebAssembly runtimes when running WebAssembly binaries are 1.26× − 5.50× of the native execution.

6. Architectural Characteristics

In this section, we study the architectural characteristics of standalone WebAssembly runtimes. To this end, we utilize the Linux `perf` tool [21] to collect the architectural statistics of the studied runtimes when running WebAssembly binaries.

6.1. Machine Instructions

We first study the dynamically executed machine instructions of the runtimes. Figure 6 shows the normalized results, where the baseline is the number of machine instructions executed by the corresponding native binary. Again, we aggregate the results of benchmarks from JetStream2, MiBench, and PolyBench, respectively, and only show their geometric means. See Figure 14 in Appendix for more detailed results. From the figure, we can clearly observe that more machine instructions are executed when running WebAssembly binaries through the runtimes, compared to the native execution. It is also interesting to see that, in most cases, the two interpretation-based runtimes, i.e., Wasm3 and WAMR, execute significantly more instructions than the other three JIT compilation-based runtimes. In general, the additional instructions introduced by a JIT compilation-based runtime mainly come from the online compilation process, which is just a one-time cost. On the contrary, an interpretation-based runtime introduces extra instructions each time when a WebAssembly instruction is interpreted and therefore needs to execute more instructions. Overall, the machine instructions executed by the standalone WebAssembly runtimes range from 2.03× to 14.61× of the native execution.

We next study instructions per cycle (IPC), which indicates how much work is done on average by a processor in a clock cycle. Figure 7 shows the measured results. As shown in the figure, for all benchmarks except *gnuchess* (due to Wasm3), both the native execution and the standalone WebAssembly runtimes can achieve an IPC value larger than 1. Actually, the largest IPC value can reach up to 4.07 for *gcc-loops* in JetStream2 by Wasmer (not shown in the figure). This demonstrates that standalone WebAssembly runtimes can effectively utilize CPU resources to complete the execution of WebAssembly binaries. Interestingly, the IPC values of the runtimes are generally higher than the native execution. This implies that more work needs to be done by the runtimes. At the same time, this may potentially raise concerns about power consumption and thermal dissipation when deploying standalone WebAssembly runtimes in real-world environments, especially for battery-powered embedded processors.

Finding 6. WebAssembly runtimes not only execute more machine instructions, i.e., an average of 2.03× − 14.61× of the native execution, but also exhibit higher IPC values than the native execution.

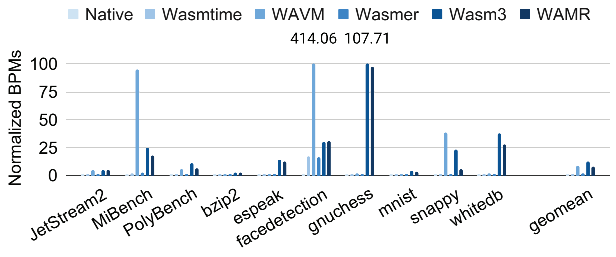

Figure 8: Normalized branch prediction misses (BPMs) of WebAssembly runtimes. The baseline is the native execution.

TABLE 5: Branch prediction miss ratios of the native execution and WebAssembly runtimes.

	Native	Wasmtime	WAVM	Wasmer	Wasm3	WAMR
JetStream2	3.15%	1.68%	1.47%	1.74%	0.73%	0.71%
MiBench	0.61%	0.62%	2.96%	0.71%	0.92%	0.64%
PolyBench	1.17%	0.84%	1.65%	1.07%	0.83%	0.53%
bzip2	1.46%	0.80%	1.58%	0.81%	0.27%	0.31%
espeak	1.12%	0.70%	1.08%	1.00%	0.73%	0.81%
facedetection	0.30%	1.68%	3.02%	1.71%	0.18%	0.24%
gnuchess	2.86%	2.36%	3.00%	2.56%	20.89%	18.13%
mnist	0.30%	0.17%	0.17%	0.17%	0.03%	0.03%
snappy	0.11%	0.03%	0.89%	0.03%	0.07%	0.02%
whitedb	0.31%	0.31%	0.42%	0.34%	1.33%	0.73%
Geomean	**1.01%**	**0.77%**	**1.69%**	**0.92%**	**0.76%**	**0.53%**

6.2. Branch Prediction

We next study branch prediction, the accuracy of which is of great importance for application performance. Figure 8 shows the normalized numbers of branch prediction misses of the standalone WebAssembly runtimes. The baseline is the native execution. As shown in the figure, more branch prediction misses are experienced by the WebAssembly runtimes. For instance, when running *facedetection*, WAVM shows 414.06× branch prediction misses of the native execution. On average, the branch prediction misses are 1.52× (Wasmtime), 8.99× (WAVM), 1.56× (Wasmer), 12.64× (Wasm3), and 8.14× (WAMR) of the native execution.

To more accurately understand the impact of branch instructions, we further study the branch prediction miss ratio, which represents the percentage of branches that are mispredicted during the execution of an application. Table 5 shows the results. It is surprising that even though the WebAssembly runtimes have more branch prediction misses, their branch prediction miss ratios are very close to the native execution except *gnuchess* (due to Wasm3 and WAMR). This demonstrates the effectiveness of the hardware branch predictor when running WebAssembly runtimes. For *gnuchess*, further investigation is required to understand why Wasm3 and WAMR have such high branch prediction miss ratios. Also, both of them have a very large number of branch prediction misses.

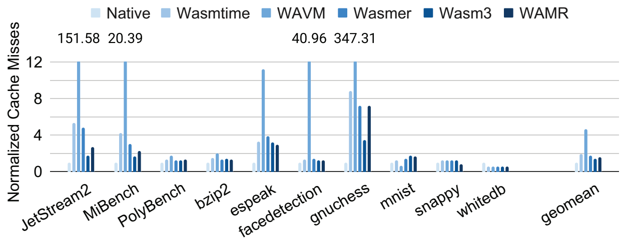

Figure 9: Normalized cache misses of WebAssembly runtimes. The baseline is the native execution.

Finding 7. WebAssembly runtimes exhibit a higher number of branch prediction misses than the native execution, ranging from 1.52× to 12.64×, but their branch prediction miss ratios are generally very close to the native execution.

6.3. Cache Misses

We finally study cache misses. Figure 9 shows the normalized cache misses of the standalone WebAssembly runtimes, where the baseline is the native execution. As illustrated in the figure, all runtimes introduce more cache misses when running WebAssembly binaries, compared to the native execution. For example, the cache misses of WAVM for *gnuchess* are 347.31× of the native execution. On average, the five WebAssembly runtimes have 1.91×, 4.60×, 1.73×, 1.39×, and 1.60× cache misses of the native execution, respectively. This shows the higher pressure on caches caused by standalone WebAssembly runtimes.

We further study the cache miss ratio, which means the percentage of cache misses in total cache references. A higher cache miss ratio of an execution usually indicates the execution has worse cache locality. Figure 10 shows the cache miss ratio results. Interestingly, the standalone WebAssembly runtimes have similar cache miss ratios compared to the native execution, although they have more cache misses. The average cache miss ratio of the native execution is 11.13%, while the average cache miss ratios of the standalone WebAssembly runtimes are 12.98% (Wasmtime), 5.57% (WAVM), 13.26% (Wasmer), 7.97% (Wasm3), and 8.99% (WAMR), respectively. This demonstrates that even though the runtimes introduce more cache references and misses, the hardware can still effectively serve the reference requests with reasonably low cache miss ratios. On the other side, this also shows that the impact of standalone WebAssembly runtimes on the cache locality of the input WebAssembly applications is very limited.

Finding 8. WebAssembly runtimes have an average of 1.39× – 4.60× cache misses compared to the native execution, while their cache miss ratios are typically similar.

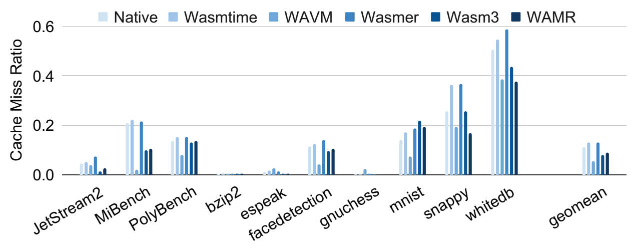

Figure 10: Cache miss ratios of the native execution and WebAssembly runtimes.

7. Discussion

Our study has revealed some interesting findings about the characteristics of standalone WebAssembly runtimes. In this section, we discuss several implications of our findings.

Selecting An Appropriate WebAssembly Runtime. According to our findings, we can conclude that different WebAssembly runtimes have quite different characteristics. As a result, it is highly recommended for WebAssembly users to select an appropriate WebAssembly runtime when applying WebAssembly to a non-web application domain. For example, JIT compilation-based runtimes usually have higher performance efficiency than interpretation-based runtimes. However, they also consume more physical memory resources in general. Therefore, it might not be appropriate to adopt a JIT compilation-based WebAssembly runtime on resource-constrained platforms, e.g., IoT devices. In addition, due to the inevitable performance and memory overhead incurred by WebAssembly runtimes, it is also necessary to evaluate whether WebAssembly is a better choice than native binaries for particular application scenarios.

Tuning WebAssembly Runtime Performance. From our findings, we can also see that the performance of a specific WebAssembly runtime is actually affected by many different factors, e.g., the JIT compiler adopted by the runtime, the AOT compilation, and the optimization level of the input WebAssembly binaries. Therefore, it is necessary for WebAssembly runtime developers to carefully tune the performance of a WebAssembly runtime in various usage scenarios. For example, a WebAssembly runtime may leverage an interpreter for occasionally-executed WebAssembly code while a JIT compiler for frequently-executed code. This way, the runtime can mitigate the high memory/cache pressure of the JIT compilation process without the loss of the high performance efficiency. That is to say, tuning the performance of a WebAssembly runtime by identifying and fixing the performance bottlenecks in different usage scenarios can improve the adaptability, robustness, and popularity of the runtime.

Developing WebAssembly-Specific Optimizations. The current WebAssembly compiler is mainly constructed based on LLVM, which was created to compile native binaries rather than WebAssembly binaries. Given the inherent differences between the execution models of WebAssembly binaries and native binaries, it is of necessity for WebAssembly compiler developers and WebAssembly runtime developers to work together to exploit *WebAssembly-specific* optimization opportunities. For example, based on our findings, the high optimization levels of the WebAssembly compiler are not as effective as those of the native compiler when using the JIT compilation-based WebAssembly runtimes to run the compiled WebAssembly binaries. This may provide a unique opportunity for a WebAssembly compiler to embed some essential information into the compiled WebAssembly binaries to open up more optimization opportunities for WebAssembly runtimes. Here, it is worth pointing out that WebAssembly compiler developers and WebAssembly runtime developers need to collaborate together to figure out what kind of information is required in order to enable the optimizations.

8. Related Work

There is very limited related work about characterizing standalone WebAssembly runtimes. A previous study [17] attempts to understand the performance efficiency of WebAssembly binaries. However, the study is limited to web browsers and thus does not provide sufficient insights into standalone WebAssembly runtimes. Similarly, a recent study [43] seeks to understand the performance differences between WebAssembly applications and JavaScript applications. Again, it only focuses on web applications instead of broad applications in non-web domains. Besides, both the studies in [28] and [32] are limited to PolyBench. Although PolyBench is a successful compiler benchmark suite, it apparently *cannot* represent applications in more diverse non-web domains.

Our study explicitly differentiates from these studies, as our goal is to systematically discover previously unknown characteristics of *standalone WebAssembly runtimes*. Moreover, our study assembles a *new* benchmark suite, namely WABench, which includes not only benchmarks from existing benchmark suites, i.e., JetStream2, MiBench, and PolyBench, but a broad range of whole applications. We believe our study considerably deepens and extends our understanding of standalone WebAssembly runtimes, especially in non-web domains.

9. Conclusion

In this paper, we conduct a comprehensive characterization study of standalone WebAssembly runtimes. Our study reviews five predominant standalone WebAssembly runtimes. Besides, we also construct a new benchmark suite through this study, named WABench. Our study leads to several interesting and insightful findings about standalone WebAssembly runtimes, including their performance efficiency, memory overhead, and architectural characteristics. These findings are valuable for both WebAssembly users, runtime developers, and compiler developers. We hope our findings can pave the way for future research on standalone WebAssembly runtimes.

Acknowledgments

We are very grateful to anonymous reviewers for their valuable feedback. This work was supported in part by the M. G. Michael Award funded by the Franklin College of Arts and Sciences at the University of Georgia and a faculty startup funding offered by the University of Georgia.

References

[1] Alon Zakai, "Outside the web: standalone WebAssembly binaries using Emscripten," 2019, https://v8.dev/blog/emscripten-standalone-wasm.

[2] Bytecode Alliance, 2022, https://bytecodealliance.org.

[3] bzip2 and libbzip2, 2022, https://www.sourceware.org/bzip2.

[4] Cranelift, 2022, https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/README.md.

[5] David Bryant, "WebAssembly Outside the Browser: A New Foundation for Pervasive Computing," 2020, https://icwe2020.webengineering.org/wp-content/uploads/2020/06/ICWE2020_keynote-David_Bryant.pdf.

[6] Emscripten Documentation, 2022, https://emscripten.org.

[7] eSpeak text to speech, 2022, http://espeak.sourceforge.net.

[8] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 265–279.

[9] GNOME Chess, 2022, https://wiki.gnome.org/Apps/Chess.

[10] GNU Chess, 2022, https://www.gnu.org/software/chess.

[11] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting," in *Proceedings of the 20th International Middleware Conference*, ser. Middleware '19. ACM, 2019, p. 123–135.

[12] Goodbye PNaCl, Hello WebAssembly!, 2017, https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html.

[13] Google, "Snappy," 2022, https://google.github.io/snappy.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.

[15] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 185–200.

[16] A. Hall and U. Ramachandran, "An Execution Model for Serverless Functions at the Edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19. New York, NY, USA: ACM, 2019, p. 225–236.

[17] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 107–120.

[18] JetStream2, 2022, https://browserbench.org/JetStream.

[19] Y. LeCun, C. Cortes, and C. J. Burges, "THE MNIST DATABASE of handwritten digits," 2022, http://yann.lecun.com/exdb/mnist.

[20] libfacedetection, 2022, https://github.com/ShiqiYu/libfacedetection.

[21] Linux Kernel, "perf: Linux profiling with performance counters," 2022, https://perf.wiki.kernel.org/index.php/Main_Page.

[22] LLVM Language Refernece, 2022, https://llvm.org/docs/LangRef.html.

[23] MNIST Neural Network in C, 2022, https://github.com/AndrewCarterUK/mnist-neural-network-plain-c.

[24] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An Embedded Trusted Runtime for WebAssembly," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 205–216.

[25] PolyBench/C: the Polyhedral Benchmark suite, 2022, https://web.cse.ohio-state.edu/~pouchet.2/software/polybench.

[26] S. Shillaker and P. Pietzuch, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX, Jul. 2020, pp. 419–433.

[27] C. Song, W. Wang, P.-C. Yew, A. Zhai, and W. Zhang, "Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX, Jul. 2019, pp. 77–90.

[28] B. Spies and M. Mock, "An Evaluation of WebAssembly in Non-Web Environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10.

[29] J. Sun, D. Cao, X. Liu, Z. Zhao, W. Wang, X. Gong, and J. Zhang, "SELWasm: A Code Protection Mechanism for WebAssembly," in *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 2019, pp. 1099–1106.

[30] W. Wang, J. Wu, X. Gong, T. Li, and P.-C. Yew, "Improving Dynamically-Generated Code Performance on Dynamic Binary Translators," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 17–30.

[31] W. Wang, P.-C. Yew, A. Zhai, and S. McCamant, "A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT)," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 591–603.

[32] Z. Wang, J. Wang, Z. Wang, and Y. Hu, "Characterization and Implication of Edge WebAssembly Runtimes," in *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, 2021, pp. 71–80.

[33] WASI Libc, 2022, https://github.com/WebAssembly/wasi-libc.

[34] WASI SDK, 2022, https://github.com/WebAssembly/wasi-sdk.

[35] Wasm3, 2022, https://github.com/wasm3/wasm3.

[36] Wasmer, 2022, https://github.com/wasmerio/wasmer.

[37] Wasmtime, 2022, https://github.com/bytecodealliance/wasmtime.

[38] WAVM, 2022, https://github.com/WAVM/WAVM.

[39] WebAssembly Micro Runtime (WAMR), 2022, https://github.com/bytecodealliance/wasm-micro-runtime.

[40] WebAssembly Security, 2022, https://webassembly.org/docs/security.

[41] WhiteDB, 2022, http://whitedb.org.

[42] XBoard, 2022, https://www.gnu.org/software/xboard.

[43] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the Performance of Webassembly Applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21. ACM, 2021, p. 533–549.

Appendix

We present more detailed experimental results here.

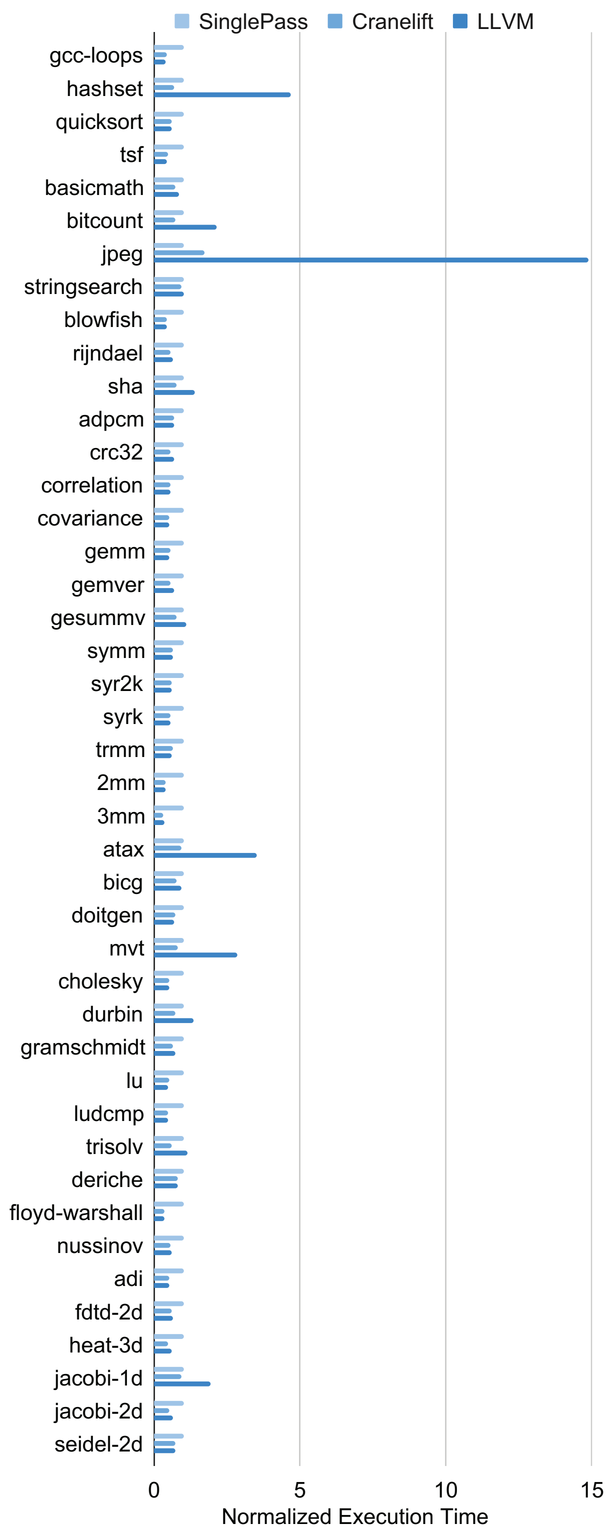

Figure 11: The normalized execution times of Wasmer with different JIT compilers for benchmarks from JetStream2, MiBench, and PolyBench. The baseline is SinglePass.

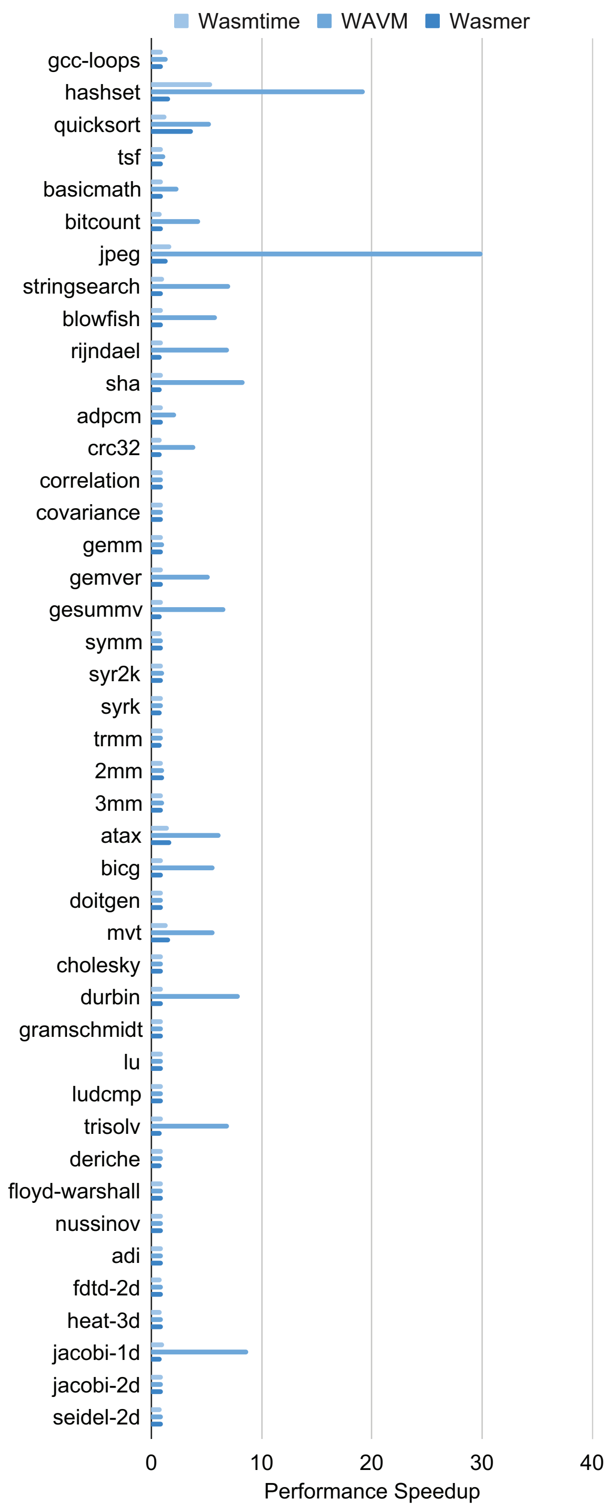

Figure 12: Performance speedup achieved by enabling AOT compilation in different WebAssembly runtimes for benchmarks in JetStream2, MiBench, and PolyBench. The baseline is the executions without AOT compilation.

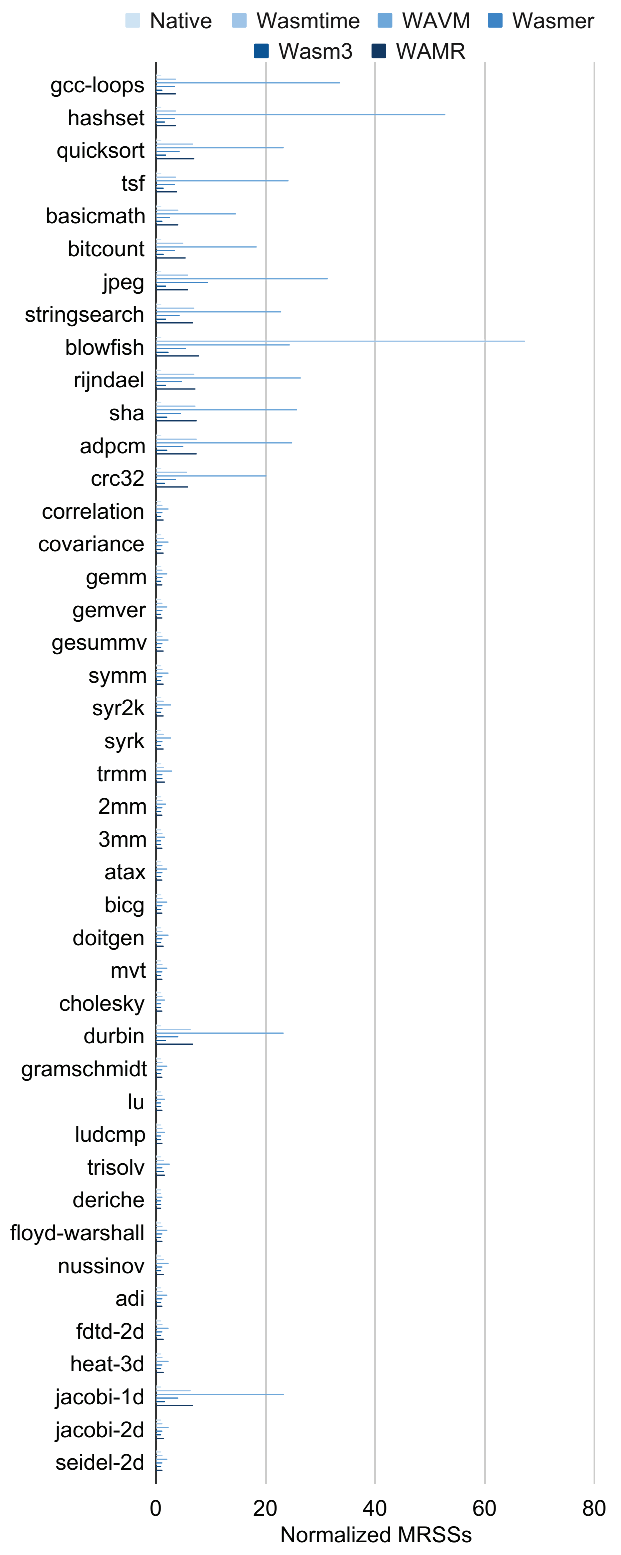

Figure 13: Normalized maximum resident set sizes (MRSSs) of WebAssembly runtimes for benchmarks in JetStream2, MiBench, and PolyBench. The baseline is native execution.

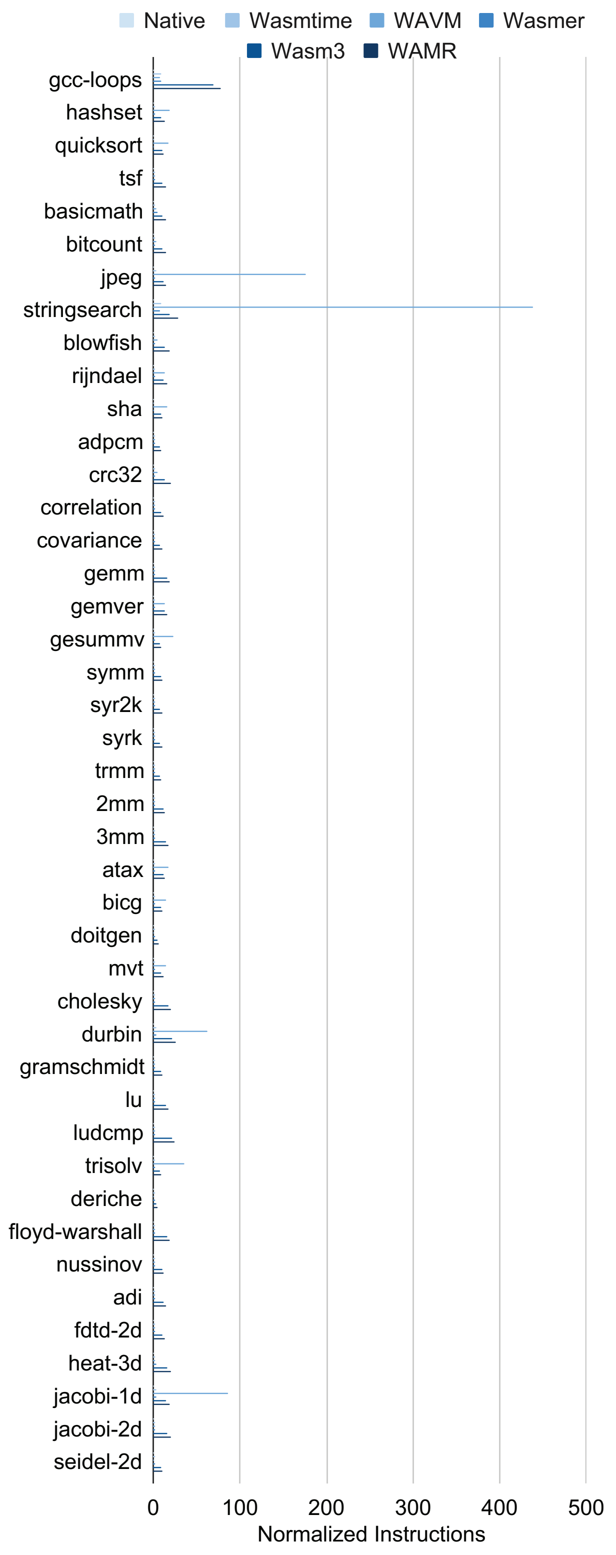

Figure 14: Normalized numbers of dynamic instructions of WebAssembly runtimes for benchmarks in JetStream2, MiBench, and PolyBench. The baseline is native execution.