Understanding the Performance of WebAssembly Applications

Yutian Yan[1], Tengfei Tu[2], Lijian Zhao[2], Yuchen Zhou[2], Weihang Wang[1]
[1]University at Buffalo, SUNY [2]Beijing University of Posts and Telecommunications
{yutianya, weihangw}@buffalo.edu {tutengfei.kevin, zhaolj, zhouyuchen7350}@bupt.edu.cn

ABSTRACT

WebAssembly is the newest language to arrive on the web. It features a compact binary format, making it fast to be loaded and decoded. While WebAssembly is generally expected to be faster than JavaScript, there have been mixed results in proving which code is faster. Little research has been done to comprehend WebAssembly's performance benefit. In this paper, we conduct a systematic study to understand the performance of WebAssembly applications and compare it with JavaScript. Our measurements were performed on three sets of subject programs with diverse settings. Among others, our findings include: (1) WebAssembly compilers are commonly built atop LLVM, where their optimizations are not tailored for WebAssembly. We show that these optimizations often become ineffective for WebAssembly, leading to counter-intuitive results. (2) JIT optimization has a significant impact on JavaScript performance. However, no substantial performance increase was observed for WebAssembly with JIT. (3) The performance of WebAssembly and JavaScript varies substantially depending on the execution environment. (4) WebAssembly uses significantly more memory than its JavaScript counterparts. We hope that our findings can help WebAssembly tooling developers identify optimization opportunities. We also report the challenges encountered when compiling C benchmarks to WebAssembly and discuss our solutions.

CCS CONCEPTS

• **Networks** → **Network measurement**; • **Information systems** → **Web applications**.

KEYWORDS

WebAssembly, web page performance, browser performance, just-in-time compilation

ACM Reference Format:
Yutian Yan[1], Tengfei Tu[2], Lijian Zhao[2], Yuchen Zhou[2], Weihang Wang[1]. 2021. Understanding the Performance of WebAssembly Applications. In *ACM Internet Measurement Conference (IMC '21), November 2–4, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3487552.3487827

1 INTRODUCTION

WebAssembly (abbreviated Wasm) is a low-level, portable, bytecode format for the web that aims to speed up web applications [30]. Recently, leading companies, such as eBay, Google, and Norton, are adopting WebAssembly in various projects to improve performance of their services that are typically written in JavaScript. To name a few, barcode readers [74], pattern matching [47], and TensorFlow.js machine learning applications [84] are the examples.

Before WebAssembly, JavaScript was the de facto standard client-side web scripting language for over 20 years [17]. While being prevalent and flexible to create powerful user interfaces, the performance of JavaScript is often considered a major limitation in practice. WebAssembly is designed to provide a better performance, aiming to unleash the potential of web applications. It differs significantly from JavaScript in two aspects. First, WebAssembly programs are delivered as compiled binaries that can be loaded and decoded faster than JavaScript programs which have to be parsed and compiled at runtime. Second, unlike JavaScript programs that are manually written by developers, WebAssembly programs are usually created by using compilers that compile existing programs in high-level programming languages, such as C/C++ and Rust, to the WebAssembly bytecode.

While WebAssembly is generally expected to be faster than JavaScript, there have been mixed results in practice [7, 12, 88]. For example, developers at eBay used WebAssembly to implement a barcode scanner, which boosted the performance of the JavaScript implementation by 50 times [74]. On the other hand, Samsung engineers observed that WebAssembly is slower than JavaScript on the Samsung Internet browser (v7.2.10.12) when performing multiplications on matrices of certain sizes [9].

The performance of WebAssembly programs is compiler- and environment-dependent. First, compilers that are used to generate WebAssembly programs can affect the performance, especially the compilers' optimization algorithms. For example, a WebAssembly program generated by a Rust compiler with the faster speed optimization option can run 20% faster than the same program compiled with the smaller code size optimization [73]. Second, the runtime environment, which includes web browsers and desktop/mobile platforms, also plays an important role. Benchmark results of a game emulator on different browsers showed that the performance advantage of WebAssembly over JavaScript is significant on Firefox (WebAssembly is 11.71x faster than JavaScript) but marginal on Chrome (1.67x faster) [87].

This paper conducts a systematic study to understand the performance of WebAssembly applications. We investigate the various factors that impact WebAssembly performance and compare it with JavaScript. We perform measurements on three sets of subject programs: (1) 41 WebAssembly binaries and 41 JavaScript programs compiled from 41 widely-used C benchmarks, (2) 9 compiler-generated WebAssembly binaries and 9 manually-written JavaScript programs, and (3) 3 real-world applications having implementations in both WebAssembly and JavaScript. These programs were

IMC '21, November 2–4, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-9129-0/21/11…$15.00
https://doi.org/10.1145/3487552.3487827

tested with diverse compiler optimizations and program inputs in various execution environments. Our findings include:

1. WebAssembly compilers are commonly built on top of existing compilers (e.g., LLVM) where their optimization techniques were not designed for WebAssembly. Our study shows that the optimizations are often ineffective for WebAssembly, leading to counter-intuitive results.
2. JIT optimization has a significant impact on JavaScript performance. However, we observed that there was no substantial performance increase for WebAssembly with JIT on both Chrome and Firefox.
3. We observe that the runtime performance of WebAssembly on Chrome, Firefox, and Edge browsers varies between desktop or mobile platforms. Specifically, Firefox has better performance (spends 0.61x time to run) in executing WebAssembly than Chrome on desktop computers while Edge performs worse (spends 1.28x time). Firefox takes 1.48x time to run compared to Chrome on mobile devices, but mobile Edge outperforms (takes 0.83x time) mobile Chrome. JavaScript also has significantly different performances on different platforms. Compared to Chrome, Firefox needs 1.06x time to execute on desktop but only needs 0.67x time to run on mobile. Edge spends 1.40x time to execute on desktop but needs 0.81x time to run on mobile compared to Chrome.
4. WebAssembly uses significantly more memory than JavaScript on Chrome, Firefox, and Edge. This is because JavaScript uses garbage collection that dynamically monitors memory allocations to determine when to reclaim memory that is no longer in use, while WebAssembly employs a linear memory model which does not reclaim memory automatically.

Our findings provide a deeper understanding of the contributing factors of the performance difference between WebAssembly and JavaScript. We hope that our analysis results can help WebAssembly tooling developers, including compiler developers and virtual machine developers, in identifying opportunities for improving runtime speed and reducing memory usage.

In summary, this paper makes the following contributions:

- We conduct a systematic performance comparison of WebAssembly and JavaScript in diverse settings.
- We analyze the contributing factors that influence WebAssembly and JavaScript performance in practice.
- We report the challenges we encountered during the compilation and discuss our solution.
- Our experiments show counter-intuitive results. We believe that our findings can help developers better understand when WebAssembly outperforms JavaScript and uncover opportunities in adopting WebAssembly.
- We make our data publicly available [2].

2 BACKGROUND

In this section, we present technical backgrounds relevant to our experiments. In particular, we focus on various factors affecting the performance of WebAssembly applications.

2.1 WebAssembly Compilers

Typically, WebAssembly programs are generated from source code written in high-level languages (e.g., C/C++ and Rust) using a WebAssembly compiler, such as *Cheerp* [86] or *Emscripten* [15]. As a result, the WebAssembly compiler and compilation options have a significant impact on the performance of generated WebAssembly programs.

2.1.1 C-to-WebAssembly Compilers. A core use case for WebAssembly is to port the existing ecosystem of C programs and allow them to be used on the web [29, 40]. Thus, in this paper, we focus on compiling C source programs to WebAssembly.

There are two C-to-WebAssembly compilers, *Emscripten* and *Cheerp*. Both of them can generate WebAssembly programs by using LLVM's backend stage [15, 86], and they offer varying levels of support for C libraries. However, their support for compiling C to JavaScript is very different: Cheerp supports standard JavaScript as a target; Emscripten does not produce standard Java-Script, but generates asm.js [5], an optimizable, low-level subset of JavaScript which was designed to allow C programs to be run as web applications with performance considerably better than standard JavaScript. As a precursor technology to WebAssembly, asm.js is also supposed to be created using compilers instead of manually written.

One important goal of this paper is to help developers solve the dilemma of choosing between JavaScript and WebAssembly for *developing* or *porting* a web application. For this purpose, we use Cheerp to compare the performance between WebAssembly and JavaScript (by creating WebAssembly and standard JavaScript from the same set of C benchmark programs). Additionally, to measure compilers' impact, we use both Emscripten and Cheerp to create WebAssembly (from the same C programs) and compare the performance differences (see Section 4.2.2).

2.1.2 Compiler Optimization Levels. C-to-WebAssembly compilers allow developers to specify optimization levels from command line options to determine how aggressive the target programs should be optimized.

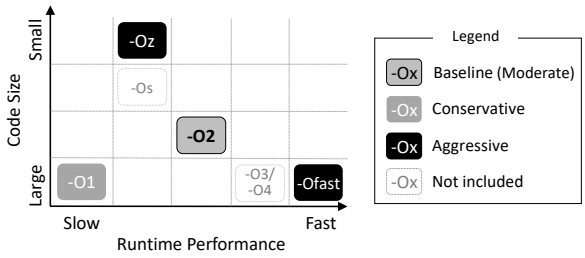

Figure 1: Compiler optimization options.

Fig. 1 illustrates the optimization levels supported by Emscripten and Cheerp with respect to the runtime performance (x-axis) and the code size (y-axis) of the compiled code. We describe the details of each optimization level below.

- `-O1`: applies basic optimizations. An example of optimizations applied at this level includes the pass (pass is the basic unit of LLVM-based compiler's optimization) `-globalopt`, which removes global variables that are never read.
- `-O2`: is an optimization level that balances the running time, code size and compilation time of produced code. We use

`-O2` as the baseline for most experiments (Sec. 4.3, Sec. 4.4, Sec. 4.5, and Sec. 4.6) in this study. An example pass applied at this level is `-vectorize-loops`, which reduces the loop frequency but increases the code size. This pass may reduce execution time as the loop structures are run less frequently.

- `-O3/-O4`: contains all optimizations in `-O2`, and enables optimizations that need more time to compile, or increase code size to reduce code running time. For Emscripten, `-O4` is a unique level and contains more optimization passes than `-O3`. For Cheerp, `-O3` and `-O4` are identical. An example pass applied in `-O3` is `-argpromotion`. The compiler will pass the value of an internal function argument into the function instead of the address of the value if the compiler can prove that this argument is only read but not written [10].
- `-Ofast`: aims for generating the fastest code. Besides optimizations in `-O2`, more aggressive optimizations such as inaccurate math calculations are used to further reduce execution time [85]. An example pass applied is `-fno-signed-zeros`. In math calculations, singed zero is required according to IEEE 754 standard [1]. However, this pass will ignore the sign bit of zeros to accelerate calculation.
- `-Os`: is built on top of `-O2`, with further optimizations for decreasing code size and the removal of optimizations that increase code size. An example pass used in `-O2` but removed in `-Os` is `-libcalls-shrinkwrap`. To avoid unnecessary calls, this pass wraps library calls whose results are not used with additional conditions. Because additional conditions increase code size, the pass is eliminated in `-Os`.
- `-Oz`: to reduce code size even more, `-Oz` adds more aggressive optimizations and eliminates certain optimizations from `-Os`. `-vectorize-loops` is an example pass that is no longer used at this level. This pass was discussed in `-O2`, and it will increase code size.

2.2 Execution Environment

Inside browsers, both WebAssembly and JavaScript run in the same engine – the JavaScript engine. However, the two languages are significantly different regarding their execution models and memory management mechanisms.

2.2.1 JavaScript. JavaScript source code is parsed, optimized, and compiled at runtime by JavaScript engines in browsers. Memory allocation in JavaScript is managed dynamically by the JavaScript engine's garbage collector.

JavaScript Engine. JavaScript source code first needs to be parsed to an abstract syntax tree which then will be used by the JavaScript engine for generating the bytecode. To speed up JavaScript program execution, the Just-in-time (JIT) compilation [38] can be applied on the sequences of frequently executed bytecode, translating them to machine code for direct execution on the hardware.

JavaScript Garbage Collection. JavaScript engine uses garbage collection to automatically monitor memory allocation and determine when a block of allocated memory is no longer in use and reclaim it. This form of automatic memory management makes JavaScript memory-efficient. As we observed in the experiments (see Section 4.3), unlike WebAssembly that allocates a large chunk of memory in the beginning, the memory occupied by JavaScript programs stays stable even when they process very large input.

2.2.2 WebAssembly. Unlike JavaScript programs, WebAssembly bytecode does not need to be parsed. WebAssembly also employs a linear memory model, which is very different from the garbage collection in JavaScript.

WebAssembly Virtual Machine. The low-level WebAssembly bytecode does not need to be parsed as it is ready to be compiled into machine code. Moreover, WebAssembly has already gone through the majority of optimizations during compilation (except a few machine-dependent optimizations). However, the context switch between JavaScript and WebAssembly causes additional runtime overhead. WebAssembly requires JavaScript to access Web APIs (e.g., DOM and WebSockets). At the minimum, it requires JavaScript to instantiate the WebAssembly module.

WebAssembly Linear Memory Model. WebAssembly employs a linear memory model where linear memory is represented as a contiguous buffer of untyped bytes that can be read and modified by both WebAssembly and JavaScript [67]. A memory instance is a resizable JavaScript `ArrayBuffer`. When a WebAssembly module is instantiated, a memory instance is created (e.g., using `WebAssembly.Memory()` [58]) to allocate a chunk of linear memory for the module to use and emulate dynamic memory allocations. If the initial memory is fully occupied, the memory instance will be expanded to a bigger size. We observed that compared to JavaScript, WebAssembly consumes significantly more memory when processing large input.

2.2.3 Mobile vs. Desktop. The performance of WebAssembly and JavaScript may differ between browsers and platforms. For example, our experiments show that Chrome is the fastest on desktop for JavaScript, and Firefox is the fastest on desktop for WebAssembly. On mobile devices, however, Firefox is the fastest in executing JavaScript, and Edge outperforms others in running WebAssembly. In Sec. 4.5, we will discuss the performance implications of browsers and platforms.

2.3 Program Input Size

The value of a program's input that affects the amount of calculations is referred to as its *input size*. For example, the input size of the multiplication of two matrices is a tuple (M, N, K), where the dimensions of the first matrix are M×N, and the dimensions of the second matrix are N×K. Typically, programs taking a larger input would run for a longer time, executing a set of instructions repeatedly. Intuitively, such programs have higher chances to be better benefited from optimization techniques. For example, processors (e.g., x86 CPU) leverage code cache to optimize frequently executed instructions. JavaScript engines apply the JIT compilation on the JS statements that are frequently executed. These optimizations work well for JavaScript programs, especially those with hot loops. However, it is unknown whether WebAssembly programs are efficiently optimized. As WebAssembly is actively under development, its runtime performance is highly dependent on how browser engines optimize WebAssembly virtual machine. We will discuss the performance impact of input sizes in Sec. 4.3.

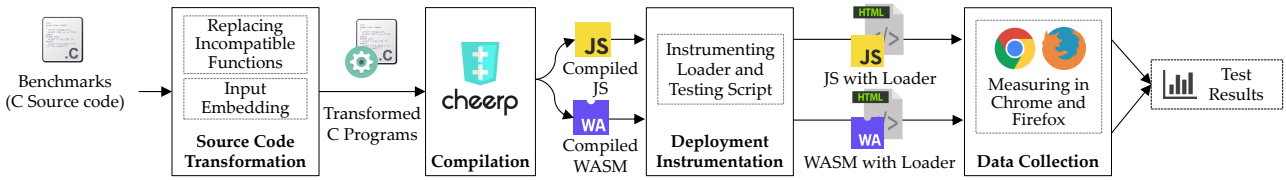

Figure 2: Process overview.

3 METHODOLOGY

Overview. Fig. 2 summarizes the procedure of measuring the performance of WebAssembly and JavaScript. It has four steps: (1) *Source Code Transformation*, (2) *Compilation to Wasm/JS*, (3) *Deployment Instrumentation*, and (4) *Data Collection*.

First, since there are 30 programs in our benchmarks having compilation errors, we resolve the errors by applying source code transformation so that these benchmarks can be compiled to WebAssembly and JavaScript successfully. The source code transformation essentially replaces incompatible C constructs that are not supported by Cheerp with comparable implementations. Second, we compile 41 C benchmarks using Cheerp to generate WebAssembly and JavaScript programs. Third, we instrument the generated programs to add time measurement code and create a minimal HTML page to load the WebAssembly/JavaScript programs. Finally, we run the generated WebAssembly/JavaScript program in HTML pages and collect execution time and memory usage using browsers' built-in developer tools.

3.1 Source Code Transformation

The testing process begins with transforming the source code to replace incompatible primitives, such as functions and data structures, with comparable compatible implementations.

```
try {
  ...
  if(matrix[i][j][k] <= 0)
    throw std::runtime_error(
     "Matrix elements must
      larger than zero");
  isFinished = 1;
} catch (...) {
  std::cout << e.what() <<
    " Input the data again" <<
    std::endl;
  isFinished = 0;
}
```

```
int error = 0;
...
if(matrix[i][j][k] <= 0)
    error = 1;
isFinished = 1;
...
if(error) {
    std::cout << "Matrix elements
    must larger than zero. Input
    the data again" << std::endl;
    isFinished = 0;
}
```

(a) Exception Handler

```
union { double d;
        unsigned long long ll; }
t;
t.d = ...;
t.ll = ...;
```

```
struct { double d; } t;
typedef struct {
   unsigned long long ll; } _T2;
t.d = ...;
((_T2*)(&t))->ll = ...;
```

(b) Union Data Type: union → struct and type casting

Figure 3: Code transformation examples.

Resolving Incompatible Primitives. One major technical challenge we encountered was that Cheerp compiler does not support all C/C++ features generally supported by mainstream C/C++ compilers such as GCC [36]. In particular, we discuss the following representative challenges that prevent us from compiling the C benchmarks to WebAssembly and JavaScript.

- **Exceptions.** Cheerp does not support exceptions correctly. Specifically, Cheerp blindly removes all the `catch` blocks in the `try-catch` statements. However, it does not remove the corresponding `throw` statements, leading to dangling exceptions at runtime. If an exception is thrown at runtime, the execution will crash (i.e., causing a segmentation fault). To resolve this incompatibility, we transform the source code to avoid using exceptions. As shown in Fig. 3(a), we remove the `try-catch` statement and replace a `throw` statement with a variable (`error` at line 10) that stores whether the exception occurs or not. Then, statements in the `catch` block are copied to the `error` predicate (lines 17-18).
- **Union.** Cheerp does not support the `union` data type. In C, `union` can be replaced with multiple `struct` definitions with proper casting operations on its uses. Fig. 3(b) shows the related transformation. Specifically, in addition to a structure `t` that includes `double d` at line 23, we define an additional structure (`_T2`) that contains `ll` (line 24). When `ll` is referred at line 26, we cast the original structure to `_T2` to implement the union functionality.

```
#include <stdio.h>
int fib(int i) {
  if (i < 3)
    return 1;
  else
    return fib(i - 1) + fib (i - 2);
}
int main() {
  printf("%d", fib(6));
}
```

(a) C Source Code


```
function fib(i) {
  if (i < 3) {
    return 1;
  } else {
    return (fib(i - 1)) +
           (fib(i - 2));
  }
}
function foo() {
  return fib(6);
}
let t1 = foo();
console.log(t1);
```

(b) Cheerp Compiled JavaScript Program

```
(module
 (type $t0 (func))
 (type $t1 (func (param i32) (result i32)))
 (type $t2 (func (result i32)))
 (func $f0 (type $t0)
   unreachable)
 (func $f1 (type $t1) (param $p0 i32)
  (result i32)
  (local $l1 i32) (local $l2 i32)
  local.get $p0
  i32.const -1
  i32.add
  ...
```

(c) Cheerp Compiled WebAssembly Program

Figure 4: Cheerp compiled JavaScript and Wasm programs.

3.2 Compilation to Wasm/JS

We use Cheerp to generate WebAssembly and JavaScript programs from the C source files of each program under test. Fig. 4 shows an example of Cheerp compiling a Fibonacci program written in C (Fig. 4(a)) to JavaScript (Fig. 4(b)) and WebAssembly (Fig. 4(c)). During the compilation, several parameters are used:

- **Input Size.** We use 41 C benchmark programs (see Section 4.1) in our experiment. For each benchmark, we compiled five sets of input with different sizes: *Extra Small (XS)*, *Small (S)*, *Medium (M)*, *Large (L)*, and *Extra Large (XL)*", defined by the benchmark frameworks (*PolyBenchC* [75] and

CHStone [44]). The input sizes for all benchmarks can be found on [2]. When compiling the benchmarks to Wasm/JS, macros are used to specify the input size.

- **Optimization Level.** We apply 4 optimization levels `O1`, `O2`, `Oz`, and `Ofast` during the compilation. We choose the `-O2` as the baseline as it achieves a balance between code running time, code size and compile time. We do not include `-Os`, `-O3`, and `-O4`, because their impact on performance and memory consumption is similar to the other options and thus are unrepresentative. Specifically, `-Os` is a subset of `-Oz` with a resulting code size in between `-Oz` and `-O2`. `-O3` and `-O4` are similar to `-Ofast` with respect to execution time.
- **Stack/Heap Limit.** Cheerp-compiled WebAssembly programs have a maximum 8 MB heap and 1 MB stack space by default. If a program uses heap/stack space more than the limit, it will throw runtime errors. To overcome this limit, we increase the heap size and stack size using flags `cheerp-linear-heap-size` and `cheerp-linear-stack-size`.

During the linking process of compiling the C benchmarks to WebAssembly/JavaScript, we encounter another major technical challenge that is closely related to the Cheerp compiler implementation. Specifically, we need to inspect all compiled results of the benchmarks to make sure the source code is properly compiled due to the following Cheerp-specific implementations.

Cheerp Pre-compiled Libraries. By default, Cheerp implicitly links pre-compiled common C/C++ libraries such as `stdlib.h` and `libc++`. When a target program is compiled and explicitly linked to the same library, there will be conflicting symbol definitions (i.e., one from the pre-compiled libraries implicitly linked and another one from the explicitly linked libraries). To avoid the issue, we configure Cheerp not to use pre-compiled libraries.

Missing Libraries Native to LLVM. While Cheerp is an LLVM-based tool, it does not support a few libraries that are supported in LLVM versions for typical target architectures (e.g., x86). For example, the `stdio.h` library that defines file input/output functions is not supported by Cheerp. Similarly, `stdlib.h` is also not supported. We have tried to compile the `libc`, but unfortunately, we were not successful. We find that multiple functions were not properly compiled, leading to empty functions, which can cause unexpected runtime behaviors for programs that use the library. To handle this issue, we look for alternative implementations of the functions in those missing libraries and use them if possible [14].

3.3 Deployment Instrumentation

3.3.1 Creating Web Page to Load Wasm/JS. We construct an HTML page to test the WebAssembly and JavaScript programs in browsers. To reduce the overhead imposed by other elements on the page, the HTML page is minimal and includes just the generated JavaScript program or the JavaScript loader (that generated for instantiating WebAssembly) using a '`<script>`' tag.

3.3.2 Instrumenting to Add Timers. To measure execution time, we use a JavaScript high-resolution timer `performance.now()`. The timer is added to the generated JavaScript program and the JavaScript loader. Specifically, we insert the timer calls before the target program starts, and after the program ends. Each benchmark was executed five times to get the average.

3.4 Data Collection

We test the performance of WebAssembly and JavaScript on three mainstream browsers (Chrome, Firefox, and Edge). For each experiment, we use browsers' developer tools (i.e., DevTools) to collect two metrics: *(1) Execution Time* and *(2) Memory Usage*. Note that the measured performance includes overhead caused by other components of web browsers such as page renderer. To reduce the overhead imposed by other tasks, we run only one browser tab that executes a single benchmark at a time.

4 EVALUATION

In this section, we first describe the three kinds of subject programs used in the study. Next, we measure the performance of WebAssembly and JavaScript: (1) compiled with various optimization levels, (2) with diverse inputs, and (3) when executed in different browsers and platforms. We evaluated the desktop performance and mobile performance of three mainstream browsers, Google Chrome (v79) [37], Mozilla Firefox (v71) [66], and Microsoft Edge (v79) [61]. The desktop experiments were done on a machine with Intel Core i7 processor and 16 GB memory, running Ubuntu 18.04.2. For experiments on mobile phones, we used a Xiaomi Mi 6 phone with an 8-core processor and 6 GB memory, running Android OS. We collected the execution time and memory usage on mobile browsers using Android Debug Bridge (adb) [4]. The parameters we used with Google Chrome in each subsection of the evaluation are described in Appendix A.

4.1 Subject Programs

Our study includes three kinds of subject programs: (1) 41 WebAssembly binaries and 41 JavaScript programs compiled from 41 widely-used C benchmarks, (2) 9 compiler-generated WebAssembly binaries and 9 manually-written JavaScript programs, and (3) 3 real-world applications having implementations in both WebAssembly and JavaScript.

Note that for the first two sets of subject programs, we develop WebAssembly by converting implementations from C rather than basing it on JavaScript. This is because C/C++ to WASM compilation is the more desirable way for WASM development, even if some JS to WASM compilation is possible. Currently there is no compiler that directly compiles generic JS to WASM, as several essential features in JavaScript, such as garbage collection, are not supported in WebAssembly. A subset of TypeScript to WASM compiler exists, but the project is not for generic JavaScript and has been inactive for several years [90]. By contrast, the support of compiling C/C++ features to WASM is relatively mature, as several components of WASM compilers are built atop the components of compilers targeting C/C++. Besides, it is worth mentioning that the original intention of WASM development is not to replace JS but as a way to complement it.

4.1.1 Compiler-Generated WebAssembly and JavaScript. First, we compile 41 C benchmark programs to WebAssembly and JavaScript, and measure the contributing factors of their performances (Sec. 4.2, 4.3, 4.4, and 4.5). As shown in Table 1, these 41 C programs are selected from two widely-used C benchmark suites: *PolyBenchC* (version 4.2.1) [75] and *CHStone* (version 1.11) [44]. The two benchmark

Table 1: Benchmark statistics.

	Benchmark	cLOC[1]	LOC	Description
PolyBenchC	covariance	175	958	Convariance computation
	correlation	201	984	Normalized covariance computation
	gemm	194	978	Generalized matrix multiplication
	gemver	215	997	Multiple matrix-vector multiplication
	gesummv	181	963	Summed matrix-vector multiplication
	symm	194	977	Symmetric matrix multiplication
	syrk	172	955	Symmetric rank k update
	syr2k	187	970	Symmetric rank 2k update
	trmm	171	954	Triangular matrix multiplication
	2mm	214	999	Two matrix multiplications
	3mm	229	1,015	Three matrix multiplications
	atax	170	953	A^T times Ax
	bicg	186	969	Biconjugate gradient stabilization
	doitgen	176	960	Numerical scientific simulation
	mvt	180	962	Matrix vector multiplication
	cholesky	170	952	Matrix decomposition
	durbin	163	945	Yule-Walker equations solver
	gramschmidt	185	974	QR Matrix decomposition
	lu	170	952	LU Matrix decomposition
	ludcmp	212	994	Linear equations solver
	trisolv	154	936	Triangular matrix solver
	deriche	227	1,010	Edge detection and smoothing Filter
	floyd-warshall	146	928	Shortest paths in graph solver
	nussinov	495	1,277	RNA folding prediction
	adi	205	988	2D heat diffusion simulation
	fdtd-2d	214	998	Electric and magnetic fields simulation
	heat-3d	171	954	Heat Equation w/ 3D space simulation
	jacobi-1d	157	940	Jacobi-style stencil computation (1D)
	jacobi-2d	160	943	Jacobi-style stencil computation (2D)
	seidel-2d	150	933	Gauss-Seidel stencil computation (2D)
CHStone	ADPCM	733	843	Speech signal processing algorithm
	AES	1,120	1,187	Cryptographic algorithm
	BLOWFISH	1,804	1,896	Data encryption standard
	DFADD	809	5,014	Addition for double
	DFDIV	644	2,689	Division for double
	DFMUL	622	2,487	Multiplication for double
	DFSIN	975	3,192	Sine function for double
	GSM	549	654	Speech signal processing algorithm
	MIPS	390	423	Simplified MIPS processor
	MOTION	1,007	1,040	Motion vector decoding for MPEG-2
	SHA	1,367	33,933	Secure hash algorithm

1: Excluding modification from researchers and generic benchmark harness.

suites include compute-intensive applications which represent common usage scenarios according to WebAssembly design goals [16]. In particular, PolyBenchC and CHStone include benchmarks that are relevant to applications such as scientific visualization, encryption, simulation, image/video/music editing/recognition, games, and virtual/augmented reality. For example, Tensorflow [31], one of the most popular AI/ML libraries, uses WebAssembly to achieve a ten times improvement in the performance of their models over the JS version. The benchmark software's matrix computations and other mathematical algorithms are directly relevant to this type of use case. Similarly, graphic editing tools and online games, such as Figma [35], a cloud-based graphic design tool for drawing, leverages WebAssembly to improve its load time by three times.

We list detailed attributions of individual benchmarks to the use cases as follows. (1) *PolyBenchC:* (1a) Scientific visualization and simulation: "floyd-warshall", "nussinov", "adi", "fdtd-2d", "heat-3d", "jacobi-1d", "jacobi-2d", and "seidel-2d". (1b) Image/video editing: "deriche". (1c) Image/video/signal processing applications: commonly use matrix computation benchmarks, including "gemm", "gemver", "gesummv", "symm", "syrk", "syr2k", "trmm", "2mm", "3mm", "atax", "bicg", "doitgen", "mvt", "cholesky", "lu", and "trisolv". (1d) Math-oriented applications and equation solvers: "correlation", "covariance", "durbin", "gramschmidt", and "ludcmp". (2) *CHStone:* (2a) Encryption: "AES", "BLOWFISH", and "SHA". (2b) Image/video editing: "MOTION". (2c) Scientific visualization and simulation: "ADPCM" and "GSM". (2d) Platform simulation/emulation: "MIPS". (2e) Signal processing that use intensive floating-point computations: "DFADD", "DFDIV", "DFMUL", and "DFSIN".

4.1.2 Compiler-Generated WebAssembly and Manually-Written JavaScript. The second experiment setting is to compare WebAssembly with native JavaScript (rather than JavaScript generated from C). To do so, we manually implement 9 benchmarks chosen from PolyBenchC and CHStone, each representing one category of computations (data mining, BLAS routines, linear algebra kernels, linear algebra solvers, algorithms in a graph, scientific simulation, two different cryptographic algorithms, and hashing)[1]. Note that one benchmark can be written in JavaScript in many different ways. To make these implementations better represent real-world JavaScript, we leverage popular JavaScript libraries, including math.js [50] (11.1k stars on GitHub) and jsSHA [8] (2k stars on GitHub), and use standard W3C APIs, such as Web Cryptography API [89] to perform SHA hashing, whenever possible. The list of manually-written JavaScript programs and their LOC are shown in Table 9.

4.1.3 Real-World Applications in WebAssembly and JavaScript. Finally, we look for real-world applications that are available in both WebAssembly and JavaScript from GitHub repositories. Specifically, we search for GitHub repositories with the topics 'WebAssembly' and 'wasm', rank these repositories by the number of stars, and then manually inspect these popular projects. Note that finding WebAssembly and JavaScript implementations of the same program on GitHub is nontrivial, and there aren't many of them available. After inspecting over 150 GitHub repositories, we find three widely-used libraries that have both WebAssembly and JavaScript implementations: *Long.js*, *Hyphenopoly.js*, and *FFmpeg*. We briefly describe each library below. The details of the libraries, including LOC, project size, and the input we used for the test, are given in Table 10.

Long.js defines a `Long` class to represent a 64 bit two's-complement integer value. According to ECMAScript, the JavaScript `Number` type cannot represent integers whose magnitude is greater than 2^{53} safely [59]. This library is commonly used for supporting full 64-bit integer values and reliable 64-bit integer arithmetic operations. Both WebAssembly implementation and JavaScript implementation [24, 25] of Long.js are available in the same repository [22].

Hyphenopoly.js hyphenates text if the user agent does not support CSS-hyphenation or has no support for a required language. For example, if the input is 'Hyphenation' in American English, the output should be 'Hy-phen-ation.' The WebAssembly implementation [65] (with 481 GitHub stars) and the JavaScript implementation [63] (with 593 GitHub stars) are from two different repositories [62, 64] but created by the same author.

FFmpeg provides functions and utilities to record, convert, and stream audio and video [32]. Compared to the other two projects, this project is much larger with over 9 million LOC and 23 MB. For this application, the WebAssembly implementation [34] (with 4.1k GitHub stars) and the JavaScript implementation [21] (with

[1]To the best of our knowledge, there is no official JavaScript implementation of PolyBenchC and CHStone.

Table 2: Geometric means of compiler optimization results (number less than 1 means it is faster/smaller than O2).

Metrics	Targets	JS	WASM	x86
Exec. Time	O1/O2	0.95x	0.88x	1.36x
	Ofast/O2	0.99x*	0.96x*	**0.97x**
	Oz/O2	**0.94x**#	**0.86x**#	1.22x
Code Size	O1/O2	**0.99x**	1.00x	1.00x
	Ofast/O2	1.00x	1.00x	1.11x
	Oz/O2	**0.99x**	**0.99x**	**0.99x**
Memory	O1/O2	1.00x	1.00x	-
	Ofast/O2	1.00x	1.00x	-
	Oz/O2	1.01x	1.00x	-

*: Ofast is unexpectedly slower than O1 and Oz.
#: Oz unexpectedly produces the fastest code.

433 GitHub stars) are from two different repositories ([33] and [20]) and are created by different developers.

4.2 Impact of Compilers and Compiler Optimizations

4.2.1 Compiler Optimizations. We first measure the impact of compiler optimizations on WebAssembly performance. As discussed in Section 2.1.2 (see Fig. 1), `-Ofast` is supposed to generate the fastest code; `-Oz` should generate the most compact code; `-O1` is supposed to produce large code that runs slowly; `-O2` should be faster than `-O1` and `-Oz` but slower than `-Ofast` in terms of execution time, and generate code which is smaller than `-O1` and `-Ofast` but larger than `-Oz`.

Optimization for WebAssembly and JavaScript. Fig. 5 shows the performance results of WebAssembly and JavaScript with four optimization levels, `-O1`, `-O2`, `-Ofast`, and `-Oz`. Table 2 summarizes the statistics of execution time, resulting code size, and runtime memory usage. Further statistical analysis on compiler optimization results are described in Appendix B.

Regarding execution time, we observe several counter-intuitive results. Specifically, `-Ofast`, which is supposed to produce fastest code, generated WebAssembly and JavaScript that execute slower (annotated by * in Table 2) than `-O1` and `-Oz`. `-Oz` unexpectedly produced the fastest WebAssembly (0.86x# compared to baseline optimization `-O2`) and the fastest JavaScript (0.94x#). Besides, the WebAssembly and JavaScript compiled with `-O2` run slowest, although `-O2` is supposed to generate faster target code than `-O1` and `-Oz`. Next, we use two benchmarks as examples to explain what causes the counter-intuitive results.

(1) ADPCM benchmark: The 'ADPCM' benchmark in WebAssembly compiled with `-Ofast` spends 1.50x time to run compared to that compiled with `-O2`. Fig. 7(a) shows the code snippet of the 'ADPCM' benchmark in C. Fig. 7(b) and (c) show the WebAssembly code compiled from the C code shown in Fig. 7(a) with `-O2` and `-Ofast`, respectively. Fig. 7(a) highlighted the statements at lines 4-5 which caused the counter-intuitive result. In particular, the global variable 'result' was never used, and therefore it should be eliminated in the compiled code. As shown in Fig. 7(b), there is no code generated for the C code at lines 4-5 with `-O2`. However, in Fig. 7(c), `-Ofast` added 14 extra instructions (lines 14-27). These extra instructions were executed 50 times during the experiment, leading to longer execution time. It means that Ofast misses dead code elimination. This is counter-intuitive because Ofast is supposed to include all of O3, which includes all of O2. From our further inspection, we believe that this might be a bug in the compiler. Specifically, we found a reported bug that is similar where O3 (and Ofast) perform worse than O2 [27].

(2) Covariance benchmark: The 'Covariance' benchmark in WebAssembly compiled with `-O1` takes 0.71x time compared to `-O2`. Fig. 8(a) and (b) show the WebAssembly code compiled with `-O2` and `-O1` respectively. As shown in Fig. 8(a), in `-O2`, a 64-bit float number is defined by first defining a 32-bit integer (`i32.const`) and then performing an i32-to-f64 type conversion (`f64.convert_i32_s`). In `-O1`, however, the same number was passed in as a function argument `$p0` (Fig. 8(b), line 9 and line 13). Because of the extra push and pop operations performed on WebAssembly's virtual stack, the two instructions (lines 5-6) in Fig. 8(a) are executed slower than the one instruction (line 13) in Fig. 8(b). We validated this intuition using a simple experiment that loops the two code snippets (lines 5-6 in Fig. 8(a) and line 13 in Fig. 8(b)) for 1 million times. The result shows that the one instruction in `-O1` takes 0.77x of the time to run than the two instructions in `-O2`.

From our experiments, we observe that there is no silver bullet optimization flag for all target programs. For example, while Oz produced the fastest WASM binaries on average (15 out of 41 are the fastest), there are cases where other options (i.e., `-O1`/`-O2`/`-Ofast`) produced the fastest WASM binaries. Specifically, `-O1` generated the fastest binaries for "gesummv", "symm", "atax", "cholesky", "trisolv", "deriche", "jacobi-2d", and "SHA" (8 out of 41). `-O2` compiled "correlation", "gemm", "3mm", "dotigen", "gramschmidt", "ADPCM", "GSM", and "MIPS" are the fastest (8 out of 41). `-Ofast` produced the fastest binaries for "covariance", "syrk", "bicg", "durbin", "ludcmp", "floyd-warshall", "nussinov", "adi", "jacobi-1d", "seidel-2d", and "DFADD" (10 out of 41). Hence, our suggestion for application developers (who use WASM compilers) is that while `-Oz` may generally produce fast binaries, one should do a sufficient test and choose an optimization flag based on the result. This is because those optimizations typically target x86 binaries and seem to be not designed and implemented for WASM in mind. As a result, our takeaway for compiler developers is that there is a real demand to tailor the optimization techniques to WebAssembly.

In terms of resulting code size, compared to the baseline optimization `-O2`, programs produced with `-O1`, `-Ofast`, and `-Oz` optimizations have almost identical sizes (with less than 2% variance) for both WebAssembly and JavaScript. A few exceptions stem from the code sizes of two CHStone benchmarks, 'DFADD' and 'DFSIN'. These two benchmarks store the input data in global variables. Thus, a larger input size requires a larger data array, leading to a larger code size.

The memory usage of WebAssembly and JavaScript is mostly the same at all optimization levels. Note that we used medium-sized input for the tests, which did not trigger dynamic memory allocations extensively. The memory usage may differ if dynamic memory allocations occur more frequently.

Optimization for x86. To prove that the counter-intuitive results of WebAssembly and JavaScript are not compiler intended behaviors, we conduct the same experiments on x86. Specifically, we compile the 41 C benchmarks to x86 machine code using LLVM

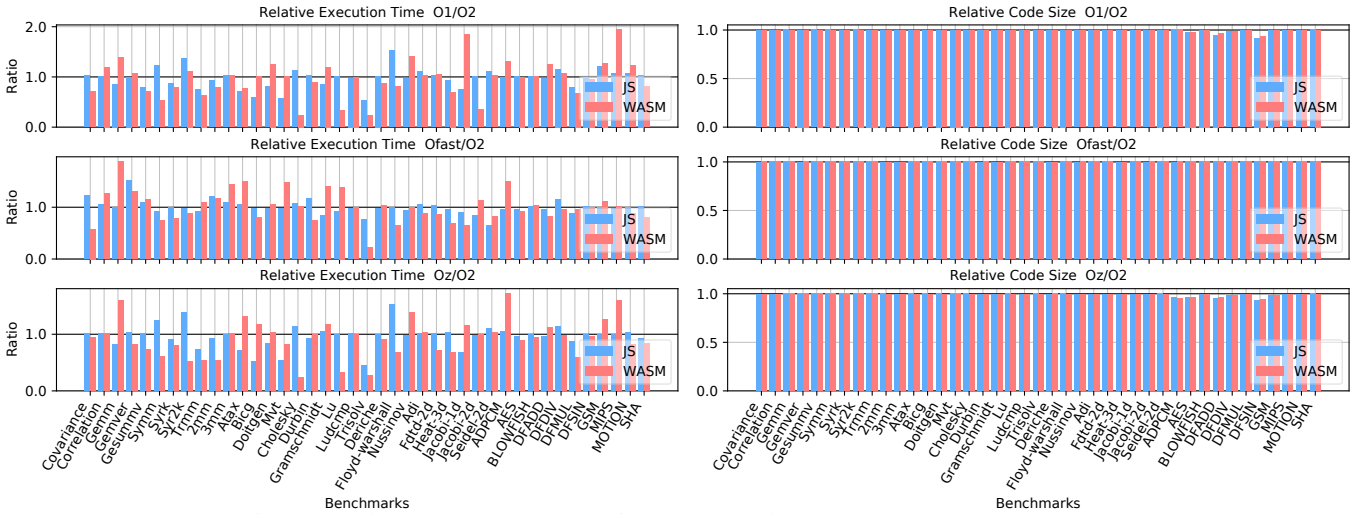

Figure 5: Execution time (the top row) and resulting code size (the second row) of WebAssembly and JavaScript with `-O1`, `-Ofast` and `-Oz`, compared to the result of `-O2`. Each benchmark was tested on Chrome v79 with the default input size.

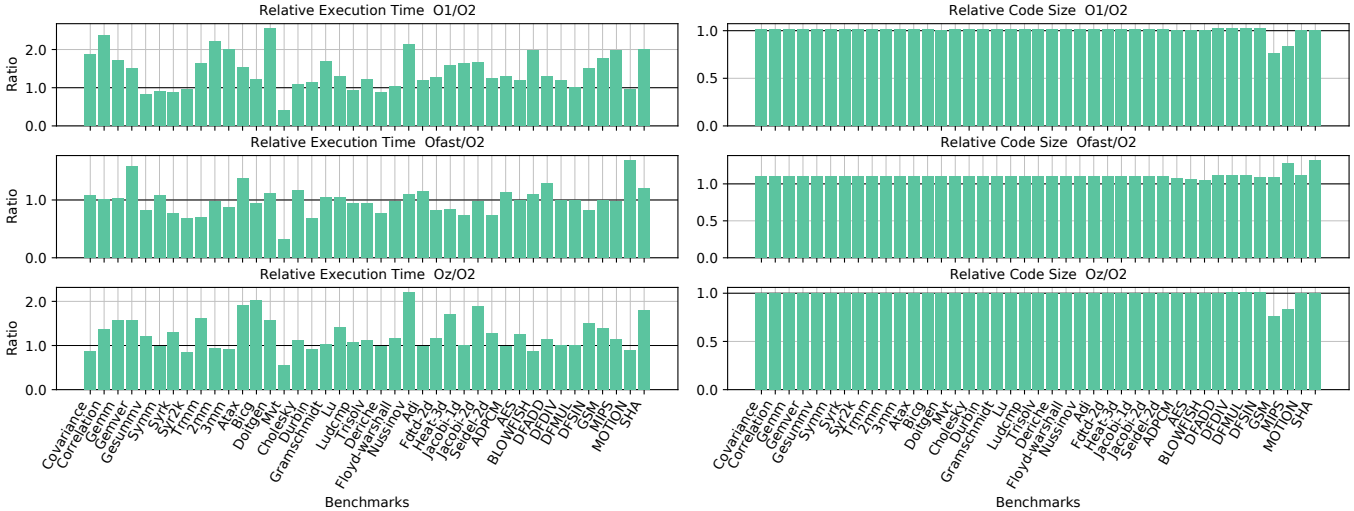

Figure 6: Execution time (the top row) and code size (the second row) of x86 code with `-O1`, `-Ofast` and `-Oz`, relative to `-O2`.

```
1  void adpcm_main() {
2    ...
3    decode(compressed[i / 2]);
4    result[i] = xout1;
5    result[i + 1] = xout2;
6    ...}
```

(a) C source code

-Ofast

```
11 (func $f5
12   ...
13   call $f3
14   local.get $10
15   i32.const 2
16   i32.shl
17   i32.const 1050296
18   i32.load
19   i32.store offset=1050300
20   local.get $10
21   i32.const 1
22   i32.add
23   i32.const 2
24   i32.shl
25   i32.const 1050700
26   i32.load
27   i32.store offset=1050300
28   local.get $10
29   ...)
```

(c) WebAssembly with -Ofast

-O2

```
7  (func $f5
8    ...
9    call $f3
10   local.get $10
11   ...)
```

(b) WebAssembly with –O2

Figure 7: *ADPCM* in WebAssembly with `-O2` vs. `-Ofast`.

with four optimization levels, `-O1`, `-O2`, `-Ofast`, and `-Oz`. To ensure that the results are comparable, we use LLVM v3.7.0, the same version as the one Cheerp is built upon.

```
1  (func $f41
2    ...
3    LOOP $L0
4      local.get $l10
5      i32.const 260
6      f64.convert_i32_s
7      f64.div
8      ...)
```

(a) WebAssembly with –O2

```
9  (func $f41 (param $p0 f64)
10   ...
11   LOOP $L0
12     local.get $l10
13     local.get $p0
14     f64.div
15     ...)
```

(b) WebAssembly with –O1

Figure 8: *Covariance* in WebAssembly with `-O1` vs. `-O2`.

Fig. 6 shows the execution time and the resulting code size of the compiled machine code. The result statistics shown in Table 2 (column 'x86') are aligned with the expected results described in Fig. 1. Specifically, `-Ofast` generated the fastest code (0.97x of the execution time relative to `-O2`). `-Oz` leads to the smallest target code size (0.99x relative to `-O2`). `-O2` produced code that takes less execution time than `-O1` (0.74x execution time) and `-Oz` (0.82x execution time) but, more execution time than `-Ofast` (1.03x execution time). In addition, the size of the code generated using `-O2` is smaller than `-Ofast` (0.90x) but larger than `-Oz` (1.01x).

Table 3: Chrome execution time statistics of Fig. 9.

Input Size	SD #[1]	SD gmean[2]	SU #[3]	SU gmean[4]	All gmean[5]
Extra-small	1	14.42x ↓	40	31.33x ↑	26.99x ↑
Small	2	4.78x ↓	39	9.92x ↑	8.22x ↑
Medium	18	1.71x ↓	23	6.70x ↑	2.30x ↑
Large	16	1.88x ↓	25	2.72x ↑	1.44x ↑
Extra-large	18	1.39x ↓	23	2.91x ↑	1.58x ↑

1: # of benchmarks which WebAssembly runs slower than JavaScript.
SD is short for the slowdown. 2: Geometric mean for SD. 3: # of benchmarks which WebAssembly runs faster than JavaScript. SU is short for speedup.
4: Geometric mean for SU. 5: Geometric mean for all 41 benchmarks.

Table 4: Chrome average memory usages (in KB) of Fig. 9.

Input Size	JavaScript	WebAssembly
Extra-small	879.41	2,001.54
Small	878.73	2,077.27
Medium	880.54	2,985.78
Large	883.10	26,991.05
Extra-large	889.20	100,943.88

Table 5: Firefox execution time statistics.

Input Size	SD #[1]	SD gmean[2]	SU #[3]	SU gmean[4]	All gmean[5]
Extra-small	33	4.75x ↓	8	2.04x ↑	3.05x ↓
Small	29	2.41x ↓	12	2.01x ↑	1.52x ↓
Medium	16	1.87x ↓	25	1.71x ↑	1.08x ↑
Large	12	1.52x ↓	29	1.85x ↑	1.37x ↑
Extra-large	6	1.13x ↓	35	1.86x ↑	1.67x ↑

1: # of benchmarks which WebAssembly runs slower than JavaScript.
SD is short for the slowdown. 2: Geometric mean for SD. 3: # of benchmarks which WebAssembly runs faster than JavaScript. SU is short for speedup.
4: Geometric mean for SU. 5: Geometric mean for all 41 benchmarks.

Table 6: Firefox average memory usages (in KB).

Input Size	JavaScript	WebAssembly
Extra-small	508.67	1,600.31
Small	492.02	1,674.03
Medium	525.02	2,583.72
Large	517.88	26,594.05
Extra-large	511.26	103,982.74

4.2.2 Compilers (Cheerp vs. Emscripten). To evaluate the impact of compilers on performances, we use both Emscripten and Cheerp to compile the 41 C benchmarks with the baseline optimization (`-O2`). The experiment was run on desktop Chrome with each benchmark's default input size (i.e., medium-sized input). The result shows that benchmarks compiled by Emscripten run faster (2.70x geometric mean) than benchmarks compiled by Cheerp, but they use 6.02x (geometric mean) more memory. Note that Emscripten uses 16MB as its page size, i.e., the smallest memory that needs to be allocated for instantiating WebAssembly modules. By contrast, the page size of Cheerp is 64KB. This difference makes programs compiled by Cheerp use less memory but run slower because of the overhead introduced by more frequent memory resizing requests (via invoking the JS function `memory.grow()` [67]).

4.3 Impact of Input Sizes

WebAssembly's compact code format and its low-level nature are designed to be faster than JavaScript. However, our experiments showed that JavaScript often outperforms WebAssembly, especially when the input of the program is large.

4.3.1 Chrome performance with diverse input sizes. We measure the execution time and memory usage of WebAssembly and JavaScript compiled from the 41 C benchmarks with five sets of input. Each benchmark was compiled using `-O2` and was tested on desktop Chrome v79.

Execution Time. Fig. 9 shows execution time results and Table 3 shows the statistics of the results. In Table 3, "speedup" is the ratio of execution speed of a faster program to the execution speed of a slower program, and "slowdown" is the ratio of the execution time of a slower program to the execution time of a faster program.

When benchmarks were tested with XS or S input, WebAssembly is faster than JavaScript for almost all benchmarks (97.6% and 95.1% for XS and S respectively). On average, WebAssembly achieves a speedup of 26.99x for XS inputs and 8.22x for S inputs.

However, when the input size increases to M, there are 18 benchmarks where WebAssembly becomes slower than JavaScript. For example, the benchmark 'Lu' in WebAssembly was 62.50x and 2.84x faster than JavaScript for XS (N=40) and S (N=120) input. However, it became 2.49x slower for M input (N=400). For the remaining 23 benchmarks, the performance gap between WebAssembly and JavaScript also drops significantly (6.70x on average). For example, the WebAssembly version of the '3mm' benchmark is 47.71x, 10.54x, and 1.12x faster than its JavaScript version, with XS input, S input, and M input respectively. When the input size further increases to L or XL, the number of benchmarks that JavaScript performs faster is not increasing anymore.

Memory Usage. Table 4 shows the statistics of the memory result presented in Fig. 9. As shown in Table 4, the memory usage of JavaScript stays fairly stable (between 878.73KB and 889.20KB) with diverse inputs. By contrast, the WebAssembly programs consume significantly more memory when the input size increases to L (increases by ≈24MB) and XL (increases by ≈74MB). This is because WebAssembly does not support garbage collection [39]. When a WebAssembly module was instantiated, a large chunk of linear memory was initialized to emulate memory allocations. If the linear memory is fully occupied, instead of reclaiming memory that is no longer in use, the linear memory is further extended to a bigger size. By contrast, JavaScript employs garbage collection which dynamically monitors memory allocations and reclaims the memory that is no longer needed. The result shows that JavaScript is more memory-efficient than WebAssembly.

In addition, we observe that all PolyBenchC benchmarks compiled to JavaScript have similar memory usage (between 882 and 908 KB, the yellow line in sub-graphs from Covariance to Seidel-2d in Fig. 9) regardless of input sizes. The structure of benchmarks: a unified test framework with different calculation core, may lead to this result. JavaScript's memory management system introduced above can handle all cores in benchmarks with similar memory usage. With a fixed amount of unified test framework memory usage, different benchmarks finally have similar memory usage. On the contrast, CHStone benchmarks did not have a unified framework, so their JavaScript memory usage vary.

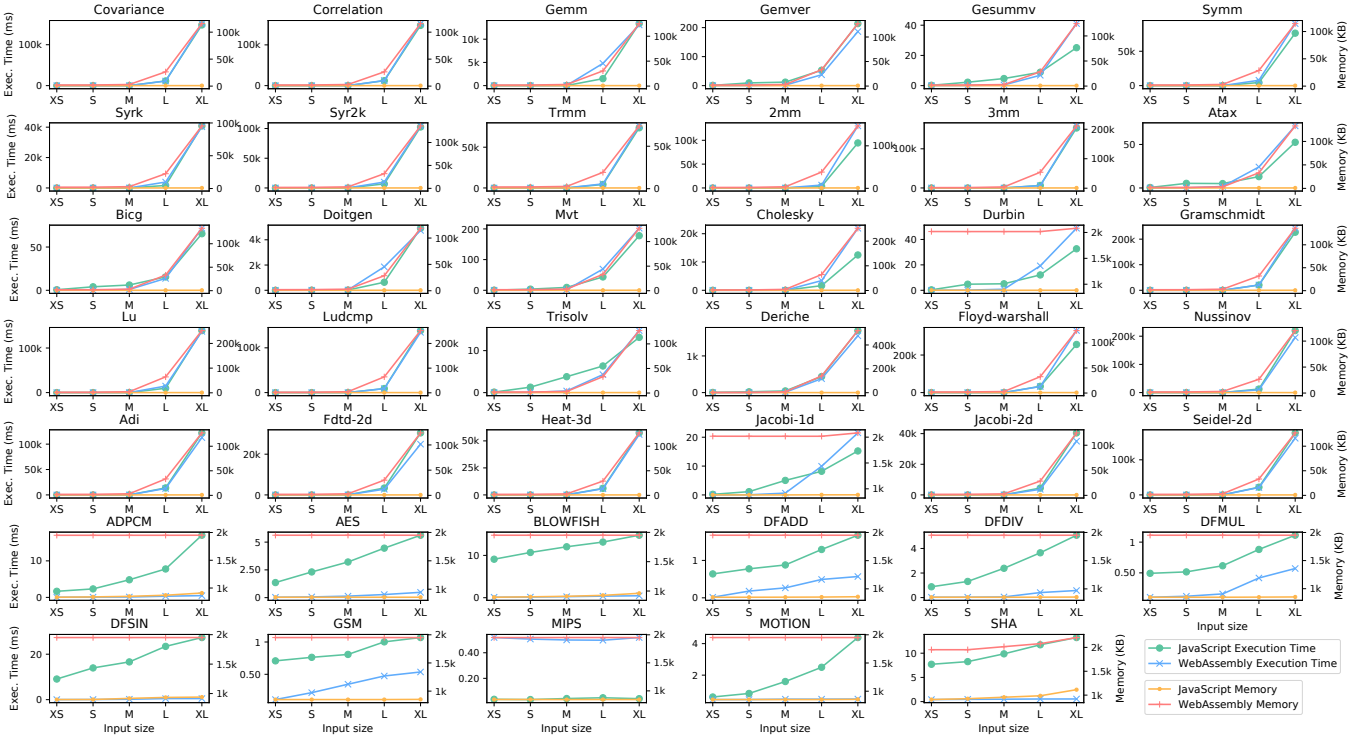

Figure 9: Execution time (left y-axis) and memory usage (right y-axis) of WebAssembly and JavaScript of the 41 benchmarks with five sets of input. Each benchmark was compiled with optimization level `-O2` and was tested on Chrome v79.

4.3.2 Firefox performance with diverse input sizes. We also measure execution time and memory usage of WebAssembly and JavaScript with five input sizes on Firefox v71.

Table 5 shows the statistics of the execution time on Firefox. When input sizes are M, L, and XL, similar to Chrome, WebAssembly achieves better performance than JavaScript (1.08x speedup for M, 1.37x speedup for L, and 1.67x speedup for XL). However, different from Chrome, the percentage of benchmarks where WebAssembly runs faster than JavaScript becomes higher when the input size increases (60.1% for M, 70.7% for L, and 85.4% for XL). When benchmarks were tested with XS or S input, most JavaScript benchmarks are faster than WebAssembly (80.5% and 70.7% for XS and S respectively), which is different from Chrome where most JavaScript benchmarks were slower than WebAssembly. On average, WebAssembly performs 3.05x slowdown for XS input and 1.52x slowdown for S input on Firefox.

The memory usage in Firefox is shown in Table 6. In general, Firefox and Chrome memory usage has a similar trend. The JavaScript memory usage is relatively stable (between 492.02KB and 517.88KB) with different input sizes. By contrast, the WebAssembly programs have significantly more memory usage when the input size increases from M to L (increases by ≈24MB) and from L to XL (increases by ≈77MB). Another noticeable point is Firefox's JavaScript memory usage is smaller than Chrome for all input sizes. For WebAssembly, Firefox uses less memory than Chrome when executing XS, S, M, and L benchmarks, but uses more memory when executing XL benchmarks.

4.4 Impact of JIT Optimization

4.4.1 JIT Optimization for JS vs. WASM. The JavaScript engines in modern browsers leverage JIT compilation to improve the performance of the frequently executed code (e.g., hot-loops) in JavaScript/WebAssembly programs. To better understand the correlation between performance and JIT, we compare the execution time of JS/WASM between JIT-enabled Chrome and JIT-less Chrome. Specifically, we use the '`-no-opt`' [41] flag and '`-liftoff-no-wasm-tier-up`' to disable the JIT optimization (i.e., TurboFan optimizing compiler) for JavaScript and WebAssembly in Chrome.

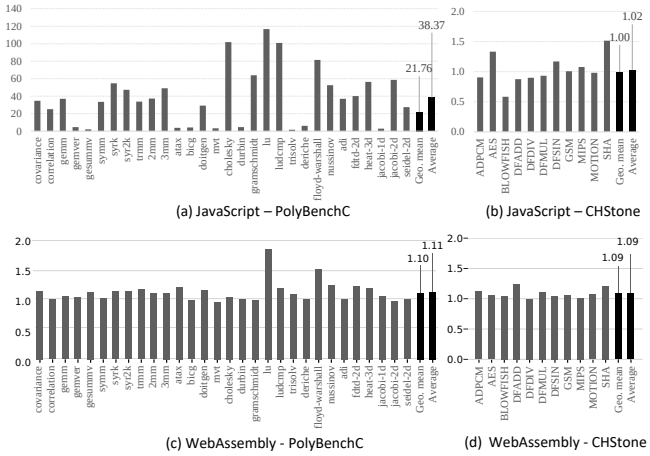

Figure 10: Performance improvement with JIT optimization.

Table 7: WASM performance improvement with JIT on Chrome vs. Firefox (numbers are execution speed ratio of default setting to only enabling basic/optimizing compiler).

Benchmark	Metric	Basic only		Optimizing only	
		LiftOff	Baseline	TurboFan	Ion
PolyBenchC	Geo. mean	1.10x	1.15x	0.88x	0.90x
	Average	1.11x	1.20x	0.90x	0.90x
CHStone	Geo. mean	1.09x	1.03x	1.07x	0.92x
	Average	1.09x	1.04x	1.07x	0.93x
Overall	Geo. mean	1.09x	1.12x	0.93x	0.91x
	Average	1.10x	1.16x	0.95x	0.91x

Fig. 10(a) and Fig. 10(b) show performance improvement of JavaScript compiled from PolyBench and CHStone, respectively. Fig. 10(c) and Fig. 10(d) shows the results of WebAssembly compiled from both benchmark suites. The x-axis presents target programs under test and the y-axis represents the performance improvement with JIT optimization compared with the executions without JIT. Specifically, we run each program with and without JIT and compare the measured execution times of them. For example, a value of 20 in the graph means the program runs 20x faster with JIT than the one without JIT. In each graph, the last two bars represent the geometric mean and average.

In general, the performance of JavaScript programs is affected significantly by JIT optimization. Programs from CHStone are affected less than the programs from PolyBench. Our manual inspection shows that this is because, in part, the programs and inputs of CHStone benchmarks are too small to trigger JIT at runtime. However, the performance improvement ratios of benchmarks in WebAssembly are mostly near 1, meaning that there is no significant performance difference with and without JIT.

4.4.2 JIT Optimization for WebAssembly on Chrome vs. Firefox. As shown in Fig. 10(c) and Fig. 10(d), no significant performance improvement for WebAssembly with JIT was observed on Chrome. To investigate if the same behavior can be observed on a browser with different execution engine, we repeat the study for WebAssembly on JIT-enabled Firefox and JIT-disabled Firefox.

In particular, both Chrome v79 and Firefox v71 have a two-layer compiler structure for WebAssembly: a basic compiler ('LiftOff' in Chrome and 'Baseline' in Firefox), which aims for quick compilation at the expense of less effective code, and an optimizing compiler ('TurboFan' in Chrome and 'Ion' in Firefox), that performs JIT compilation to generate high-performance code while taking more time to compile. The basic and optimizing compilers are both enabled by default in Chrome and Firefox. To understand the effectiveness of the two compilers, we perform experiments with three different settings: only enabling basic compiler (optimizing compiler disabled, i.e., JIT disabled), only enabling optimizing compiler (basic compiler disabled), and enabling both compilers (which is the default browser setting) on Chrome and Firefox.

Table 7 shows the performance improvement of WebAssembly with three experiment settings in Chrome (columns 3 and 5) and Firefox (columns 4 and 6). The numbers in the table are the execution speed ratio of the default setting (that uses both compilers) to the setting that only enables the basic/optimizing compiler. As can be seen, we observed similar results on both Chrome and Firefox for JIT-enabled browser vs. JIT-disabled browser. Specifically, enabling both compilers (i.e., the default setting) is slightly faster than the JIT-disabled setting (i.e., only enabling the basic compiler) on both Chrome and Firefox (1.09x geometric mean on Chrome and 1.12x on Firefox). Additionally, we observed that enabling both compilers is slightly slower than the setting that only enables the optimizing compiler (0.93x on Chrome and 0.91x on Firefox), with the exception that CHStone benchmarks in WebAssembly runs faster with the default setting on Chrome (1.07x faster).

Table 8: Arithmetic average statistics of Fig. 12 and Fig. 13.

	JavaScript			WebAssembly		
	Chrome	Firefox	Edge	Chrome	Firefox	Edge
D.[1] Exec. Time (ms)	45.57	48.26	63.62	65.23	39.65	83.53
M.[2] Exec. Time (ms)	249.60	167.03	201.68	233.08	345.98	192.87
D.[1] Memory (KB)	885.10	505.41	871.27	2,999.63	2,493.02	2,996.20
M.[2] Memory (KB)	406.71	692.63	966.80	2,522.37	2,894.20	3,087.24

1, 2: D means Desktop and M means Mobile.

4.5 Impact of Browsers and Platforms

To measure the performance impact of browsers and platforms, we test WebAssembly and JavaScript in six deployment settings: desktop Chrome (v79), desktop Firefox (v71), desktop Edge (v79), mobile Chrome (v79), mobile Firefox (v68), and mobile Edge (v44). Table 8 shows the statistics of execution time and memory usage results. Detailed performance results can be found in Appendix C.

Execution Time of JS/WASM Across Browsers. On desktop, Chrome is the fastest in executing the tested JavaScript programs. Firefox is slightly slower, with 1.06x execution time, compared to Chrome. However, Firefox executes WebAssembly much faster (0.61x execution time) than Chrome. Such differences may indicate Firefox's WebAssembly implementations are more optimized for performance. For example, in October 2018, Firefox released a new version that has made the calls between WebAssembly and JavaScript much faster by getting rid of unnecessary work to organize stack frames and taking the most direct path between functions [11]. To quantify the context switch overhead, we measure the time used for switching between WebAssembly and JavaScript in three desktop browsers. The result shows that Firefox performs much faster (only 0.13x execution time) than Chrome and Edge, indicating that the optimization made by Firefox for function calls between WebAssembly and JavaScript is efficient.

On mobile devices, the performance comparison of the three browsers is different from the result on desktop. Specifically, Firefox has better performance on executing JavaScript programs compared to Chrome (0.67x execution time), but it takes more time (1.48x execution time) to execute the WebAssembly counterparts. Similarly, Edge performs worse than Chrome for both JavaScript (1.40x execution time) and WebAssembly (1.28x execution time) on desktop. However, Edge outperforms Chrome on mobile for JavaScript (0.81x execution time) and WebAssembly (0.83x execution time).

Execution Time of JS vs. WASM Across Browsers. As can be seen, the performance of WebAssembly on Firefox and Chrome differs significantly between mobile platform and desktop platform. Unlike Chrome that uses the same codebase for both mobile and desktop versions, Firefox for Desktop uses the Gecko web engine

Table 9: Results of Manually-Written JavaScript Programs.

	Benchmark	LOC	Time (ms)			Memory (KB)		
			Manual	Cheerp	WASM	Manual	Cheerp	WASM
PolyBenchC	3mm	18,387	179.680	59.050	52.577	3,986	885	4,321
	Covariance	18,367	51.278	25.346	34.145	2,738	885	2,977
	Syr2k	18,361	54.670	13.021	24.460	3,007	882	2,849
	Ludcmp	18,400	73.050	39.878	23.440	4,367	883	4,513
	Floyd-warshall	18,351	729.535	202.807	308.663	2,771	882	2,977
	Heat-3d (W3C)							
	Heat-3d (math.js)	18,367	786.975	69.456	100.325	3,446	883	2,977
CHStone	AES	896	2.405	3.210	0.136	827	858	1,951
	BLOWFISH	723	36.705	12.039	0.245	856	910	1,951
	SHA (W3C)	44	1.575	9.866	0.500	790	956	2,015
	SHA (jsSHA)	342	13.120	9.866	0.500	804	956	2,015

and Firefox for Android uses the GeckoView engine [68]. GeckoView is a lightweight implementation of Gecko suited for mobile devices. This difference in deployment between Chrome and Firefox could explain the differences in performance. In addition, Firefox's JavaScript engine, SpiderMonkey, has some differences for mobile architectures as well. SpiderMonkey features a two-tier compilation system for WebAssembly. A quick, less-performant baseline compilation is performed first, and then a more-optimized JIT compilation is performed. Normally, SpiderMonkey uses the BaldrMonkey engine [69] to perform the tier-2 compilation. However, on ARM64 platforms (such as the MI 6 used in our mobile evaluation), this engine is not supported and is replaced with Cranelift for code generation. This difference in engines also contributes to the performance difference. The performance of Mobile Chrome and Edge browsers are relatively similar because they both are forks of the Chromium Blink engine.

Memory Usage. The memory usage results on desktop and mobile browsers are shown in Table 8. On desktop, Firefox uses less memory than Chrome for both JavaScript (0.57x) and WebAssembly (0.83x). Edge uses similar memory as Chrome (0.98x for JavaScript and 1.00x for WebAssembly). On mobile, Chrome uses less memory than Firefox (0.59x for JavaScript and 0.87x for WebAssembly) and Edge (0.42x for JavaScript and 0.82x for WebAssembly).

For all desktop browsers, WebAssembly uses more memory (3.39x on Chrome, 4.93x on Firefox, and 3.44x on Edge) than JavaScript. Mobile browsers show a similar result: WebAssembly uses 6.20x more memory on Chrome, 4.18x on Firefox, and 3.19x on Edge. As we discussed in Sec. 4.3, unlike JavaScript that uses garbage collection to reclaim memory no longer in use automatically, WebAssembly allocates a large chunk of memory at the instantiation time for the module to use. WebAssembly memory is a growable array of bytes, and the default size of the array is large compared to JavaScript applications. A potential improvement on WebAssembly memory usage is to implement more adaptive memory management (e.g., by leveraging memory allocators) rather than creating a giant memory block at the beginning of the execution.

4.6 Impact of Source Programs

To show that our findings are valid for more diverse programs, besides the 41 compiled benchmarks, we study two additional program sets: (1) 9 benchmarks (chosen from PolyBenchC and CHStone) that were manually reimplemented in JavaScript and (2) 3 real-world applications obtained from open-source GitHub repositories.

Table 10: GitHub Repository Data.

Benchmark		Input	LOC	Proj. Size	WA Time*+	JS Time*	Ratio
Long.js	multiplication	10,000 mul(36,-2)	1,501	44KB	13.365	18.305	0.730
	division	10,000 div(-2,-2)	1,506	44KB	42.190	81.130	0.520
	remainder	10,000 mod(36,5)	1,501	44KB	7.910	13.675	0.578
Hyphen-	en-us	18 KB English Text	2,264	95KB	308.105	328.550	0.938
opoly.js	fr	18 KB French Text	2,277	96KB	310.600	323.560	0.960
FFmpeg - mp4 to avi		296 MB MP4	9,167,136	23,910KB	154,170.000	560,243.000	0.275

+: WA Time: WebAssembly execution time. *: Time unit: ms.

4.6.1 Benchmarks Manually-Implemented in JavaScript. We run the manually implemented JavaScript programs on desktop Chrome. Table 9 shows the results. Observe that most manually reimplemented programs are slower that Cheerp generated programs. There are two exceptions, AES and SHA (W3C), which outperform Cheerp generated versions in terms of execution speed. Besides, all manually written PolyBenchC benchmarks consume more memory than the versions produced by Cheerp. However, reimplemented versions of CHStone consume slightly less memory.

We make two observations. First, careful implementation of JavaScript can outperform certain types of computations (e.g., AES and SHA), echoing the findings in the previous sections. Second, it is challenging to build optimal JavaScript programs in practice, which means that compiler-generated versions may be beneficial for developers to design efficient JavaScript programs (in terms of both runtime and memory space overhead).

4.6.2 Real-World Applications. We selected three real-world applications from open-source GitHub projects, Long.js, Hyphenopoly.js, and FFmpeg, and conducted six experiments: three experiments for Long.js, two for Hyphenopoly.js, and one for FFmpeg. Table 10 shows the experiment input, the sum of LOCs of HTML, JavaScript, and WAT (human-readable WebAssembly Text) files, the exeuction time of WebAssembly and JavaScript, and execution time ratio of WebAssembly to JavaScript.

Long.js. We test three operations using Long.js, multiplication, division, and remainder, in both WebAssembly and JavaScript. Rows 1-3 in Table 10 show the execution time result. In all three experiments, WebAssembly executes faster than JavaScript. We manually inspect the three programs to identify the number of arithmetic operations executed by them. Our inspection shows that the JavaScript versions run many more instructions than the WebAssembly versions because of the different mechanisms of implementing 64-bit operations in JavaScript and WebAssembly. The count of arithmetic operations executed is presented in Appendix D.

Hyphenopoly.js. We test Hyphenopoly.js in WebAssembly and JavaScript using two input languages, English (en-us) and French (fr). As shown in Table 10 rows 4-5, WebAssembly and JavaScript have similar execution time while WebAssembly is marginally faster than JavaScript. Our manual investigation shows that a significant amount of time is spent on input and output operations in which WebAssembly is not specialized.

FFmpeg. We measure the performance of this library in WebAssembly and JavaScript by converting a 296 MB video file in MP4 to AVI. Table 10 row 6 shows that WebAssembly executes much faster than JavaScript. This is because the WebAssembly implementation uses multiple WebWorkers to parallelize the conversion, while the JavaScript implementation has no parallelization.

5 LIMITATIONS AND FUTURE WORK

Threats to Validity. Our study is potentially subject to several threats, namely the representativeness of the chosen benchmarks and the generalization of the results. According to [16, 70], WebAssembly was designed to be used in a variety of applications, including compression, cryptographic libraries, games, image processing, numeric computation, and others. In our experiment, we choose 41 widely used C benchmark programs that perform numeric computation, image processing, data compression, and cryptographic algorithms. While we believe the programs we tested can well represent some common WebAssembly use scenarios, we do not include large standalone programs such as games in the comparison. This is because of the complexity of their source code and unsupported features that are incompatible with the compiler, Cheerp is not able to compile these programs. In the future, we plan to overcome the incompatible issues to support the evaluation of complex real-world applications by modifying the compiler or refactoring the source programs. Another threat concerns the generalization of the performance results. The benchmarks used in the study were tested on three mainstream browsers, Google Chrome, Mozilla Firefox, and Microsoft Edge. These browsers are evolving quickly, releasing updates frequently. Thus, the results of this study may not reflect the up-to-date performance of the browsers. To reduce the bias introduced by different browsers, we ensure three browsers were stable release versions and were released around the same time (Dec. 2019).

Future Work. We discuss several future directions that are worthy of pursuing based on our empirical findings: First, we observed that JavaScript performance was significantly affected by JIT optimization. However, no substantial performance increase was seen for WebAssembly with JIT. This is because the current browser engine can identify hot code in JavaScript to substantially improve its speed, but not so much in WebAssembly, suggesting that more effort should be spent on optimizing WebAssembly code execution. Second, our experiments show that compiler optimizations do not work as intended for WebAssembly. For example, `-Ofast`, which is supposed to create the fastest target code, is slower than `-Oz` and `-O1` for WebAssembly. As described in Section 2.1.2, such compiler inefficiencies are pervasive. These findings call for more research effort on designing new compiler optimization techniques for WebAssembly.

6 RELATED WORK

WebAssembly Performance Measurement and Studies. Our work is closely related to WebAssembly performance measurement and studies [43, 46, 48, 70, 77, 81]. [43] measured the performance of WebAssembly, asm.js, and native C implementations. [48] focused on performance comparison of WebAssembly and C programs. [81] studied WebAssembly performance for applications performing sparse matrix-vector multiplication. [70] studied the prevalence of WebAssembly in Alexa Top 1 Million Websites. Hilbig et al. [46] presented an empirical study of 8,461 real-world WebAssembly binaries and analyzed their security properties, source languages, and use cases. To the best of our knowledge, our work conducts a first comprehensive study on the performance of both generic JavaScript and WebAssembly with diverse settings.

WebAssembly Analysis Tools, Protections, and Extensions. Prior works on WebAssembly analysis tools, protections, and extensions [28, 49, 53, 54, 71, 72, 78–80, 91] are also related. Wasabi [54] is the first general-purpose framework for dynamically analyzing WebAssembly. Lehmann et al. [53] analyzed how vulnerabilities in memory-unsafe source languages are exploitable in WebAssembly binaries. Swivel [71] presented a new compiler framework for hardening WebAssembly against Spectre attacks. CT-wasm [91] introduced a type-driven, strict extension to WebAssembly to facilitate the verifiable secure implementation of cryptographic algorithms. MS-Wasm [28] extended WebAssembly to enable developers to capture low-level C/C++ memory semantics in WebAssembly at compile time.

Web Performance Measurement. There have been several prior works on testing web page performance and analyzing JavaScript, PHP, and other web technologies [42, 55, 76, 82]. Besides, our work is also relevant to studies [3, 18, 19, 26, 45, 52, 56, 57, 60, 83, 92], researching the performance of operating systems, mobile applications, and virtual machines. The closest previous work is [45] which also compares WebAssembly and JavaScript on desktop and mobile devices. However, our work covers more diverse applications and inputs, and tests on new versions of the browsers (i.e., our target browsers are released two years later than those used in [45]). Our results also differ from it where WebAssembly only performs better on desktop Firefox, mobile Chrome, and mobile Edge.

Compiler Optimization Studies. [6] conducted a case study using the Intel Core 2 Duo processor to analyze the compiler optimizations required to obtain high performance on modern processors. [51] leveraged machine learning techniques to predict the best optimization flags for creating efficient programs. [13] researched the impact of compiler optimizations on high-level synthesis. By contrast, we investigate the impact of compiler optimizations on the performance of compiled WebAssembly programs.

7 CONCLUSION

This paper conducts the first systematic empirical study to understand the performance of WebAssembly applications along with JavaScript. We perform measurements on different types of subject programs, including compiler-generated programs, manually-written programs, and real-world applications, with diverse settings. Our findings provide insights for WebAssembly tooling developers to optimize for performance improvement. We make our data publicly available [2].

8 ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Balakrishnan Chandrasekaran, for their constructive feedback. We greatly appreciate the time and effort spent by our shepherd and other reviewers in helping us improve our paper.

REFERENCES

[1] 2019. IEEE Standard for Floating-Point Arithmetic.
[2] 2020. Project Website. https://benchmarkingwasm.github.io/BenchmarkingWebAssembly/
[3] Aldeida Aleti, Catia Trubiani, André van Hoorn, and Pooyan Jamshidi. 2018. An efficient method for uncertainty propagation in robust software performance estimation. *Journal of Systems and Software* 138 (2018), 222–235.

[4] Android. 2020. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb

[5] asm.js. 2020. asm.js - an extraordinarily optimizable, low-level subset of JavaScript. http://asmjs.org/

[6] Aart JC Bik, David L Kreitzer, and Xinmin Tian. 2008. A case study on compiler optimizations for the Intel® Core TM 2 Duo Processor. *International Journal of Parallel Programming* 36, 6 (2008), 571–591.

[7] Stack Overflow Contributor Blindman67. 2018. Why is webAssembly function almost 300 time slower than same JS function. https://stackoverflow.com/questions/48173979/why-is-webassembly-function-almost-300-time-slower-than-same-js-function

[8] Caligatio. 2021. Caligatio/jsSHA. https://github.com/Caligatio/jsSHA

[9] Winston Chen. 2018. Performance Testing Web Assembly vs JavaScript. https://medium.com/samsung-internet-dev/performance-testing-web-assembly-vs-javascript-e07506fd5875

[10] Clang. 2020. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html#argpromotion-promote-by-reference-arguments-to-scalars

[11] Lin Clark. 2018. Calls between JavaScript and WebAssembly are finally fast. https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast-%F0%9F%8E%89/

[12] Stack Overflow Contributor ColinE. 2017. Why is my WebAssembly function slower than the JavaScript equivalent? https://stackoverflow.com/questions/46331830/why-is-my-webassembly-function-slower-than-the-javascript-equivalent/46500236#46500236

[13] Jason Cong, Bin Liu, Raghu Prabhakar, and Peng Zhang. 2012. A study on the impact of compiler optimizations on high-level synthesis. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 143–157.

[14] Emscripten Contributors. 2015. File System Overview — Emscripten 1.39.17 documentation. https://emscripten.org/docs/porting/files/file_systems_overview.html#file-system-overview

[15] Emscripten Contributors. 2020. Emscripten 1.39.4 documentation. https://emscripten.org/

[16] WebAssembly Contributors. 2020. Webassembly Use Cases. https://webassembly.org/docs/use-cases/

[17] Netscape Communications Corporation and Inc. Sun Microsystems. 1995. Netscape and Sun Announce JavaScript, the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet. https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html

[18] Luis Cruz and Rui Abreu. 2017. Performance-based guidelines for energy efficient mobile applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 46–57.

[19] Mariana Cunha and Nuno Laranjeiro. 2018. Assessing Containerized REST Services Performance in the Presence of Operator Faults. In *2018 14th European Dependable Computing Conference (EDCC)*. IEEE, 95–100.

[20] Damianociarla. 2021. Damianociarla/node-ffmpeg. https://github.com/damianociarla/node-ffmpeg

[21] Damianociarla. 2021. Damianociarla/node-ffmpeg/lib/ffmpeg.js. https://github.com/damianociarla/node-ffmpeg/blob/master/lib/ffmpeg.js

[22] DcodeIO. 2021. DcodeIO/Long.js. https://github.com/dcodeIO/Long.js/

[23] DcodeIO. 2021. Long.js Avoiding Overflow. https://github.com/dcodeIO/long.js/blob/master/src/long.js#L56-L59

[24] DcodeIO. 2021. Long.js JavaScript Source Code. https://github.com/dcodeIO/long.js/blob/master/src/long.js

[25] DcodeIO. 2021. Long.js WebAssembly Source Code. https://github.com/dcodeIO/long.js/blob/master/src/wasm.wat

[26] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. 2004. Early performance testing of distributed software applications. In *Proceedings of the 4th international workshop on Software and performance*. 94–103.

[27] Mozilla developers. 2021. Bugzilla – Bug 37449 – llvm performs less inlining in -O3 than in -O2. https://bugs.llvm.org/show_bug.cgi?id=37449

[28] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. 2019. Position Paper: Progressive Memory Safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy* (Phoenix, AZ, USA) *(HASP '19)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3337167.3337171

[29] MDN Web Docs. 2020. Compiling an Existing C Module to WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly/existing_C_to_wasm

[30] Haas et al. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[31] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[32] FFmpeg. 2021. FFmpeg. https://www.ffmpeg.org/

[33] ffmpegwasm. 2021. ffmpegwasm/ffmpeg.wasm. https://github.com/ffmpegwasm/ffmpeg.wasm

[34] ffmpegwasm. 2021. ffmpegwasm/ffmpeg.wasm/dist/ffmpeg.min.js. https://unpkg.com/@ffmpeg/ffmpeg@0.10.0/dist/ffmpeg.min.js

[35] Inc. Figma. 2021. The collaborative interface design tool. https://www.figma.com/

[36] Free Software Foundation (FSF). 2020. GCC, the GNU Compiler Collection. https://gcc.gnu.org/

[37] Google. 2020. Google Chrome - Download the Fast, Secure Browser from Google. https://www.google.com/chrome/

[38] Google. 2020. V8 JavaScript Engine. https://v8.dev/

[39] WebAssembly Group. 2020. WebAssembly/design. https://github.com/WebAssembly/design/blob/master/FutureFeatures.md

[40] WebAssembly Community Group. 2020. Use Cases - WebAssembly. https://webassembly.org/docs/use-cases/

[41] Jakob Gruber. 2021. JIT-less V8. https://v8.dev/blog/jitless

[42] Antonio Guerriero, Raffaela Mirandola, Roberto Pietrantuono, and Stefano Russo. 2019. A Hybrid Framework for Web Services Reliability and Performance Assessment. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 185–192.

[43] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.* 52, 6 (June 2017), 185–200.

[44] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. 2009. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing* 17 (2009), 242–254.

[45] David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. 2018. WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices. *University of McGill, Montreal: QC, Technical report SABLE-TR-2018-2* (2018).

[46] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2696–2708. https://doi.org/10.1145/3442381.3450138

[47] Raymond Hill. 2019. gorhill/ublock. https://github.com/gorhill/uBlock

[48] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not so fast: analyzing the performance of webassembly vs. native code. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 107–120.

[49] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. Trust, but verify: SFI safety for native-compiled Wasm. In *NDSS*. Internet Society.

[50] Josdejong. 2021. Josdejong/mathjs. https://github.com/josdejong/mathjs

[51] Yuriy Kashnikov, Jean Christophe Beyler, and William Jalby. 2012. Compiler optimizations: Machine learning versus o3. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 32–45.

[52] Heejin Kim, Byoungju Choi, and W Eric Wong. 2009. Performance testing of mobile applications at the unit test level. In *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*. IEEE, 171–180.

[53] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[54] Daniel Lehmann and Michael Pradel. 2018. Wasabi: A Framework for Dynamically Analyzing WebAssembly. *CoRR* abs/1808.10652 (2018). arXiv:1808.10652 http://arxiv.org/abs/1808.10652

[55] Kai Lei, Yining Ma, and Zhi Tan. 2014. Performance comparison and evaluation of web development technologies in php, python, and node. js. In *2014 IEEE 17th international conference on computational science and engineering*. IEEE, 661–668.

[56] Zhiming Liu, Nafees Qamar, and Jie Qian. 2013. A quantitative analysis of the performance and scalability of de-identification tools for medical data. In *International Symposium on Foundations of Health Informatics Engineering and Systems*. Springer, 274–289.

[57] Goran Martinovic, Josip Balen, and Bojan Cukic. 2012. Performance Evaluation of Recent Windows Operating Systems. *J. UCS* 18, 2 (2012), 218–263.

[58] MDN. 2020. WebAssembly.Memory(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Memory/Memory

[59] MDN. 2021. Number.MAX-SAFE-INTEGER - JavaScript: MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER

[60] Tianhui Meng, Katinka Wolter, and Qiushi Wang. 2015. Security and performance tradeoff analysis of mobile offloading systems under timing attacks. In *European Workshop on Performance Engineering*. Springer, 32–46.

[61] Microsoft. 2020. Download New Microsoft Edge Browser: Microsoft. https://www.microsoft.com/en-us/edge

[62] Mnater. 2021. Mnater/Hyphenator. https://github.com/mnater/Hyphenator

[63] Mnater. 2021. Mnater/Hyphenator/Hyphenopoly-Loader.js. https://github.com/mnater/Hyphenator/blob/master/Hyphenator_Loader.js

[64] Mnater. 2021. Mnater/Hyphenopoly. https://github.com/mnater/Hyphenopoly

[65] Mnater. 2021. Mnater/Hyphenopoly/Hyphenopoly-Loader.js. https://github.com/mnater/Hyphenopoly/blob/master/Hyphenopoly_Loader.js
[66] Mozilla. 2020. Firefox: Internet for people, not profit. https://www.mozilla.org/en-US/
[67] Mozilla. 2020. WebAssembly Memory. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/WebAssembly/Memory
[68] Mozilla. 2021. Geckoview. https://mozilla.github.io/geckoview/
[69] Mozilla. 2021. SpiderMonkey JavaScript/WebAssembly Engine. https://spidermonkey.dev/docs/
[70] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 23–42.
[71] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. 2021. Swivel: Hardening WebAssembly against Spectre. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1433–1450. https://www.usenix.org/conference/usenixsecurity21/presentation/narayan
[72] Shravan Narayan, Tal Garfinkel, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2019. Gobi: WebAssembly as a practical path to library sandboxing. *arXiv preprint arXiv:1912.02285* (2019).
[73] Wasm pack contributors. 2019. Wasm Speed Are No Faster Than JS. https://github.com/rustwasm/wasm-pack/issues/558
[74] Senthil Padmanabhan and Pranav Jha. 2020. WebAssembly at eBay: A Real-World Use Case. https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/
[75] Louis-Noël Pouchet, U Bondugula, and T Yuki. 2016. PolyBench/C 4.2. Polyhedral Benchmark Suite.
[76] Raghu Ramakrishnan and Arvinder Kaur. 2020. An empirical comparison of predictive models for web page performance. *Information and Software Technology* (2020), 106307.
[77] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. 2021. An Empirical Study of Bugs in WebAssembly Compilers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[78] Alan Romano and Weihang Wang. 2020. WASim: Understanding WebAssembly Applications through Classification. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1321–1325. https://doi.org/10.1145/3324884.3415293
[79] Alan Romano and Weihang Wang. 2020. WasmView: Visual Testing for WebAssembly Applications. In *Proceedings of the 42nd International Conference on Software Engineering Companion* (Seoul, South Korea) *(ICSE'20 Companion)*. Association for Computing Machinery, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382155
[80] Alan Romano, Yunhui Zheng, and Weihang Wang. 2020. MinerRay: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1129–1140. https://doi.org/10.1145/3324884.3416580
[81] Prabhjot Sandhu, David Herrera, and Laurie Hendren. 2018. Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in JavaScript and WebAssembly. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. 1–13.
[82] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in JavaScript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*. 61–72.
[83] Yuliang Shi, Xudong Zhao, Shanqing Guo, Shijun Liu, and Lizhen Cui. 2016. SRConfig: An Empirical Method of Interdependent Soft Configurations for Improving Performance in n-Tier Application. In *2016 IEEE International Conference on Services Computing (SCC)*. IEEE, 601–608.
[84] Daniel Smilkov, Nikhil Thorat, and Ann Yuan. 2020. Introducing the WebAssembly backend for TensorFlow.js. https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html
[85] The Clang Team. 2020. clang - the Clang C, C++, and Objective-C compiler — Clang 11 documentation. https://clang.llvm.org/docs/CommandGuide/clang.html#cmdoption-o0
[86] Leaning Technologies. 2020. Cheerp | C/C++ to WebAssembly compiler. https://leaningtech.com/pages/cheerp.html
[87] Aaron Turner. 2018. WebAssembly Is Fast: A Real-World Benchmark of WebAssembly vs. ES6. https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193
[88] Vladimir. 2018. WebAssembly vs. the world. Should you use WebAssembly? https://blog.sqreen.com/webassembly-performance/
[89] W3C. 2021. Web Cryptography API. https://w3c.github.io/webcrypto/
[90] Evan Wallace. 2016. Evanw/thinscript: A low-level programming language inspired by TypeScript. https://github.com/evanw/thinscript
[91] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *Proc. ACM Program. Lang.* 3, POPL, Article 77 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290390
[92] Junjun Zheng, Hiroyuki Okamura, and Tadashi Dohi. 2016. Performance Evaluation of VM-based Intrusion Tolerant Systems with Poisson Arrivals. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE, 181–187.

A EXPERIMENT PARAMETERS USED WITH GOOGLE CHROME

Table 11: Google Chrome Parameters.

Section	Figures/Tables	Parameter	Impact
Sec. 4.2	Figure 5, 6 Table 2	chrome.exe –incognito	Prevent the browser from caching the benchmark.
Sec. 4.3	Figure 9 Table 3, 4, 5, 6	chrome.exe –incognito	Prevent the browser from caching the benchmark.
Sec. 4.4	Figure 10 Table 7	chrome.exe –incognito	Prevent the browser from caching the benchmark. By default (without extra parameters), both LiftOff and TurboFan compilers are enabled.
	Figure 10	chrome.exe –js-flags="–no-opt" –incognito	"–no-opt" enables the LiftOff compiler only for JavaScript benchmarks.
	Figure 10 Table 7	chrome.exe –js-flags="–liftoff –no-wasm-tier-up" –incognito	"–liftoff –no-wasm-tier-up" enables the LiftOff compiler only for WebAssembly benchmarks.
	Table 7	chrome.exe –js-flags="–no-liftoff –no-wasm-tier-up" –incognito	"–no-liftoff –no-wasm-tier-up" enables the TurboFan compiler only for WebAssembly benchmarks.
Sec. 4.5	Figure 11, 12 Table 8	chrome.exe –incognito	Prevent the browser from caching the benchmark.
Sec. 4.6	Table 9, 10, 11	chrome.exe –incognito	Prevent the browser from caching the benchmark.

Table 11 shows the parameters we used with Google Chrome in each subsection of Sec. 4 and discuss their impacts on the results.

B STATISTICAL ANALYSIS OF COMPILER OPTIMIZATION RESULTS

Fig. 11 shows the statistics of execution time, code size, and memory usage of JS, WASM, and x86 with different optimization levels on desktop Chrome. The x-axis represents the execution time, code size, and memory usage results and the y-axis represents the five-number summary of the result: the minimum, first quartile, median, third quartile, and maximum.

In general, the execution time of JS, WASM, and x86 varies across optimization levels. While the execution time medians of JS and WASM across optimization levels are close to 1, the execution time medians of x86 with `O1/O2` and `Oz/O2` are higher than 1 (1.29 with `O1/O2` and 1.16 with `Oz/O2`). This result is in line with the geometric means of x86 execution time with `O1/O2` and `Oz/O2` as shown in Table 2 (1.36x and 1.22x, respectively). On the other hand, the code size and memory usage has little variation and is close to 1x except for 'x86 Code Size `Ofast/O2`'. According to Table 2, the

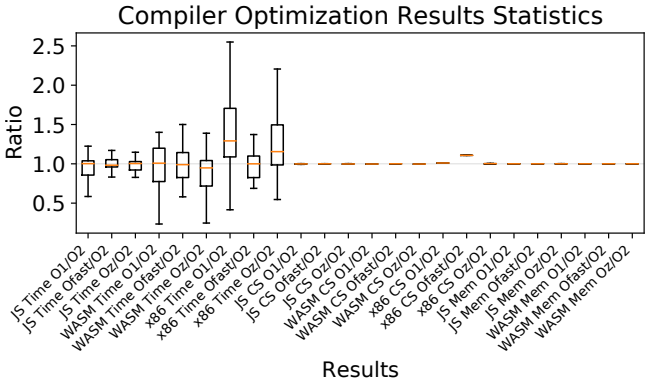

Figure 11: Execution time (Time), code size (CS), and memory usage (Mem) of JS, WASM, and x86 with different optimization levels on Chrome. Each box and its whiskers represent the five-number summary of the result: the minimum, first quartile, median (shown above each bar), third quartile, and maximum.

geometric mean of x86 code size with `Ofast/O2` is 1.11x, which is consistent with the result in Fig. 11.

C RESULTS OF BROWSERS AND PLATFORMS

Fig. 12 shows the execution time result of WebAssembly and JavaScript on desktop/mobile Chrome, desktop/mobile Firefox, and desktop/mobile Edge. Fig. 13 shows the memory usage result. The statistics of these results are summarized in Table 8.

D OPERATIONS IN LONG.JS

Table 12: Long.js Number of Operations

Benchmark	JS/WASM	ADD	MUL	DIV	REM	SHIFT	AND	OR	Total
Multiplication	JS	160k	100k	0	0	120k	110k	20k	510k
	WASM	0	10k	0	0	30k	0	20k	60k
Division	JS	80k	100k	160k	0	10k	0	0	350k
	WASM	0	0	10k	0	30k	0	20k	60k
Remainder	JS	170k	110k	20k	0	120k	110k	20k	550k
	WASM	0	0	0	10k	30k	0	20k	60k

To obtain the number of arithmetic operations in Long.js programs, we manually instrument both JavaScript and WebAssembly programs' arithmetic operations. Table 12 shows the result. Observe that the JavaScript versions run more many more instructions than the WebAssembly versions.

This is because these 64-bit operations involve fewer calculations in WebAssembly. Specifically, WebAssembly supports 64-bit arithmetic operations by treating each 64-bit integer input as two 32-bit integers to perform the calculation, and merging the results to a single 64-bit integer. By contrast, the Long.js library supports 64-bit integer arithmetic operations in JavaScript by splitting one 64-bit integer into four 16-bit integers to avoid overflow [23].

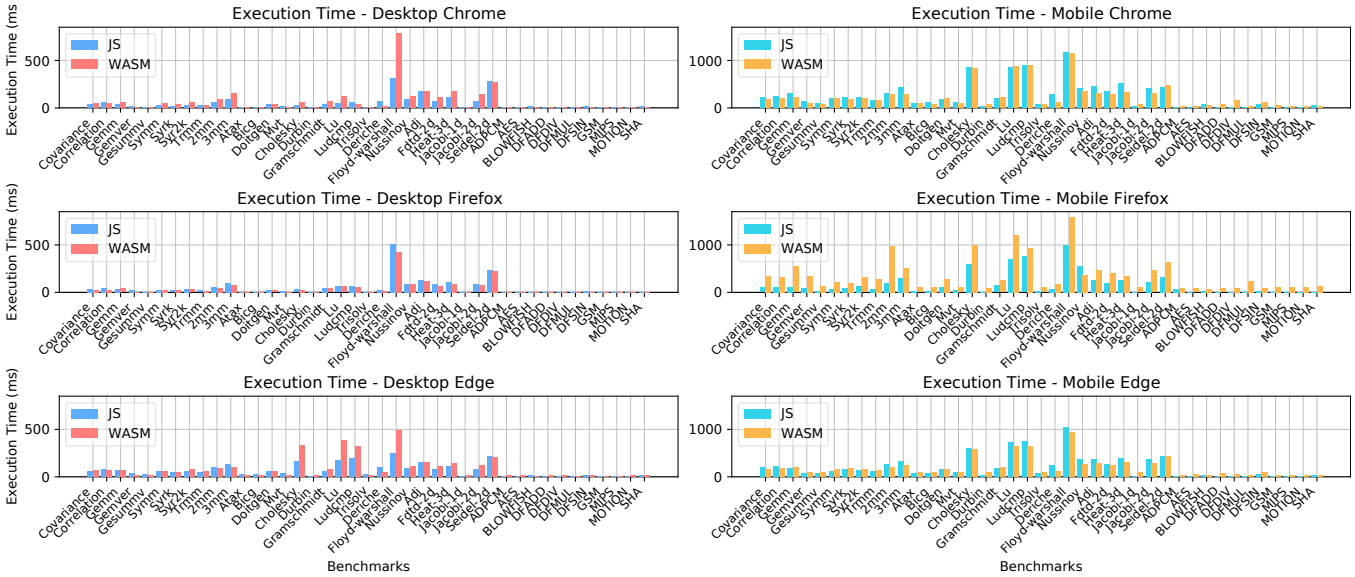

Figure 12: Execution time of WebAssembly and JavaScript on Chrome for desktop, Firefox for desktop, Edge for desktop, Chrome for mobile, Firefox for mobile, and Edge for mobile. Each benchmark was tested with default input using the baseline compiler optimization (-O2).

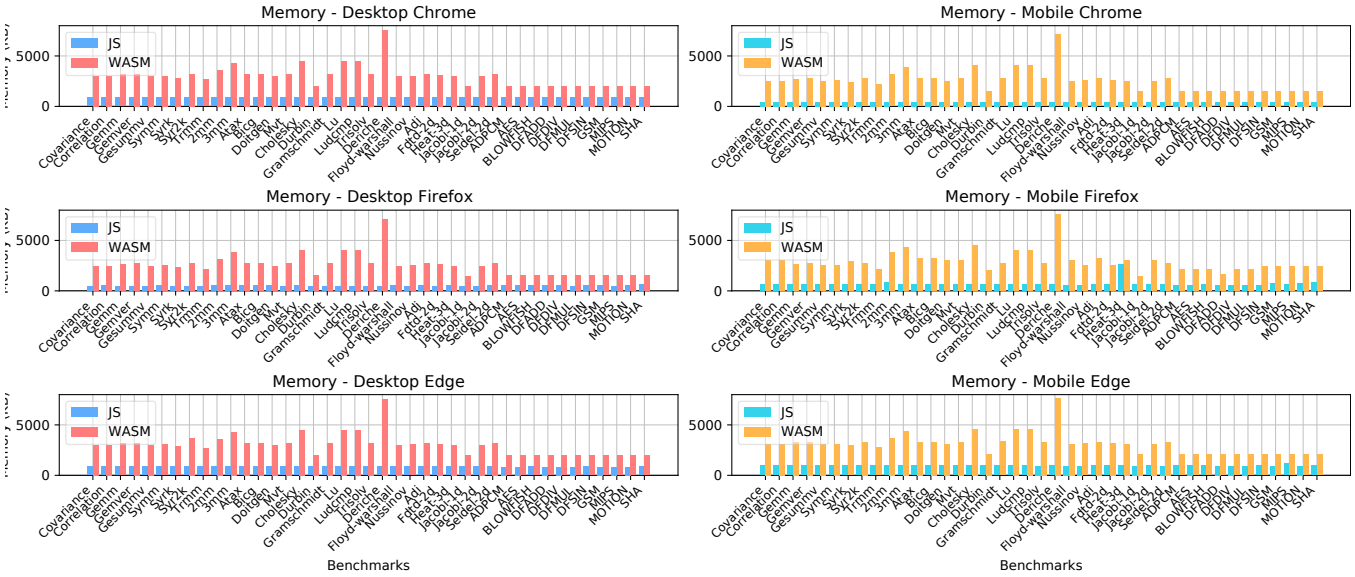

Figure 13: Memory usage of WebAssembly and JavaScript on Chrome for desktop, Firefox for desktop, Edge for desktop, Chrome for mobile, Firefox for mobile, and Edge for mobile.