**WebAssembly and Its Performance Evaluation**

## Introduction

WebAssembly (Wasm) is a binary instruction format created for high performance execution on web browsers. It was originally developed to help web applications run at speeds close to native programs. Over the course of time, its uses have expanded beyond browsers to include cloud computing, server-side applications, and even blockchain technologies.

The goal of this project is to study how WebAssembly performs compared to native execution in languages like C, C++, Java, and Python. Although WebAssembly is faster than JavaScript, its performance compared to compiled languages is still being studied. This project aims to measure WebAssembly’s execution time and analyze its control flow and security properties.

## Background of the Problem

The performance of Wasm has been a topic of debate. It is well known that Wasm is much faster than JavaScript, but its efficiency compared to native code is unclear. Since Wasm runs in a secure, sandboxed environment, it might have some extra overheads that slow it down. Although it is designed to be memory-safe, it can still inherit security risks from languages like C and C++, which allow unsafe memory access.

This project aims to answer three key questions:

1. Execution Performance – How fast is Wasm compared to native execution?
2. Security Considerations – Can security flaws like buffer overflows and dangling pointers from native code affect Wasm execution?
3. Control Flow Efficiency – Does Wasm provide any advantages in branch prediction and execution speed?

By answering these questions, this study will help us understand when Wasm is a good alternative to native execution and where it may fall short.

## Methodology

To compare the performance of Wasm and native execution, we conducted a benchmarking study using four programming languages: C, C++, Java, and Python. Each program followed the same algorithmic logic, and execution times were measured separately for each workload.

We tested two types of workloads: memory-intensive and computation-intensive. Bubble Sort, Quick Sort, and Sieve of Eratosthenes were used to evaluate memory access and data movement in WebAssembly. On the other hand, Matrix Multiplication, Tower of Hanoi, and Fibonacci Sequence were chosen to test heavy computations, recursion, and arithmetic operations.

For implementation, C and C++ were compiled to Wasm using Emscripten, which converts LLVM-based code into Wasm. Java was compiled using TeaVM, a tool that translates Java bytecode into Wasm. Python was executed in a Wasm environment using Pyodide, a runtime that allows Python to run in the browser. These tools ensured a fair comparison between native execution and Wasm across different languages.

The benchmarking process was carefully structured. Firstly, all programs were implemented and tested in their native environments to verify correctness. Then, they were compiled into Wasm using the appropriate toolchains. Each benchmark was executed separately in both native and Wasm environments to ensure consistent results. The main performance metric recorded was execution time, which was measured directly for each workload without averaging multiple runs. By analyzing individual results, we obtained an accurate understanding of how well Wasm performs compared to native execution.

## Results Analysis

### Execution Time Analysis

We tested Bubble Sort, Fibonacci Sequence, Matrix Multiplication, Prime Number Sieve, Quick Sort, and Tower of Hanoi in C, C++, Java, and Python, running them both in native execution and WebAssembly (Wasm).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Language | Workloads | | | |
| 1000 | 5000 | 10000 | 50000 |
| C++\_Native | 1.87344 | 32.5418 | 174.319 | 4947.59 |
| C++\_WASM | 9.598442 | 134.3627 | 551.1654 | 14474.38 |
| C\_Native | 1.896093 | 30.47092 | 129.9138 | 4849.3 |
| C\_WASM | 5.644179 | 27.9066 | 114.1526 | 5632.069 |
| Java\_Native | 3.05 | 14.03 | 59.77 | 2930.5 |
| Java\_WASM | 2.1 | 59.5 | 238 | 5483.9 |
| Python\_Native | 32.08 | 783.86 | 3148.15 | 82304.33 |
| Python\_WASM | 58.87486 | 1623.345 | 6632.985 | 176013.6 |

Table 1: Execution Time (ms) for Bubble Sort

For Bubble Sort (shown in Table 1), C and C++ Native were the fastest, sorting 1000 elements in around 1.87 ms and taking about 4947.59 ms at 50,000 elements. C++ Wasm was much slower, needing 9.59 ms for 1000 elements and 14,474.38 ms for 50,000 elements. Java Wasm worked better than C/C++ Wasm, while Python Wasm performed well for small inputs but slowed down a lot for larger sizes.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Language | Workloads | | | | | |
| 30 | 32 | 34 | 36 | 38 | 40 |
| C++\_Native | 1.47006 | 2.81934 | 6.67778 | 16.8523 | 51.1875 | 126.525 |
| C++\_WASM | 8.233715 | 20.212 | 54.4344 | 143.0021 | 370.4293 | 972.3543 |
| C\_Native | 0.851695 | 2.591264 | 7.621798 | 20.29935 | 51.73716 | 129.6603 |
| C\_WASM | 8.199221 | 21.5523 | 55.73264 | 148.6082 | 389.8939 | 951.3853 |
| Java\_Native | 3.1 | 6.98 | 18.41 | 50.07 | 132.09 | 346.07 |
| Java\_WASM | 8.399999 | 22.9 | 59 | 155.4 | 408 | 1064.2 |
| Python\_Native | 67.12 | 183.1 | 475.05 | 1270.61 | 3443.91 | 8853.32 |
| Python\_WASM | 195.6523 | 500.0233 | 1325.581 | 3585.429 | 9402.483 | 23222.12 |

Table 2: Exceution Time (ms) for Fibonacci

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| anguage | Workloads | | | | |
| 100 | 200 | 300 | 400 | 500 |
| C++\_Native | 0.617865 | 5.17659 | 13.6343 | 39.9919 | 89.8693 |
| C++\_WASM | 18.9101 | 100.6451 | 328.5828 | 783.129 | 1534.49 |
| C\_Native | 0.751025 | 4.238883 | 13.02448 | 34.24716 | 75.82411 |
| C\_WASM | 7.230259 | 16.68828 | 59.74207 | 145.0552 | 298.1021 |
| Java\_Native | 3.46 | 11.64 | 32.97 | 60.37 | 123.02 |
| Java\_WASM | 4.5 | 35.3 | 118.3 | 291.2 | 604.8 |
| Python\_Native | 45.36 | 364.54 | 1324.6 | 3257.03 | 6690.76 |
| Python\_WASM | 142.2765 | 1045.397 | 3601.135 | 8345.9 | 16209.56 |

Table 3: Execution Time (ms) for Matrix Multiplication

For Fibonacci Sequence (shown in Table 2), C++ Native calculated Fibonacci(40) in 126.52 ms, but C++ Wasm took 972.35 ms, showing that Wasm struggles with recursion. Java Wasm performed better than C/C++ Wasm, while Python Wasm was fast for small inputs but became very slow for large ones.

For Matrix Multiplication (shown in Table 3), C++ Native finished a 500×500 matrix in 89.87 ms, but C++ Wasm took 1534.49 ms, showing a big slowdown. Java Wasm worked better than C/C++ Wasm, and Python Wasm followed its usual pattern—fast at small sizes but much slower at large ones.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Languages | Workloads | | | |
| 100000 | 500000 | 1000000 | 5000000 |
| C++\_Native | 0.279497 | 1.72379 | 3.56816 | 11.7507 |
| C++\_WASM | 3.23563 | 9.910138 | 20.63128 | 110.395 |
| C\_Native | 0.135598 | 0.789461 | 1.675266 | 9.32448 |
| C\_WASM | 1.023874 | 2.5044 | 5.215022 | 28.78811 |
| Java\_Native | 1.11 | 3.13 | 6.24 | 16.32 |
| Java\_WASM | 0.5 | 1.7 | 3.5 | 21.8 |
| Python\_Native | 2.74 | 16.98 | 38.71 | 229.37 |
| Python\_WASM | 9.851856 | 47.5931 | 100.1485 | 505.9445 |

Table 4: Execution Time (ms) for Prime Sieve

For Prime Number Sieve (shown in Table 4), C Native found 5,000,000 primes in 9.32 ms, while C++ Wasm needed 110.39 ms, showing that Wasm is slower at memory-related tasks. Java Wasm worked better than C/C++ Wasm, while Python Wasm struggled with large inputs.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Languages | Workloads | | | | |
| 1000 | 5000 | 10000 | 50000 | 100000 |
| C++\_Native | 0.033168 | 0.228525 | 0.467884 | 2.80371 | 5.90974 |
| C++\_WASM | 1.137995 | 2.116613 | 3.138066 | 11.75135 | 23.1096 |
| C\_Native | 0.057978 | 0.335669 | 0.888841 | 4.65647 | 8.457704 |
| C\_WASM | 0.959236 | 1.587933 | 3.616432 | 18.93647 | 46.64778 |
| Java\_Native | 0.4 | 1.51 | 3.39 | 13.04 | 23.36 |
| Java\_WASM | 0.099999 | 0.599999 | 2 | 7.099999 | 14.1 |
| Python\_Native | 0.07 | 0.61 | 0.99 | 6.27 | 12.32 |
| Python\_WASM | 1.953635 | 7.212384 | 14.61795 | 67.38306 | 131.5044 |

Table 5: Execution Time (ms) for Quick Sort

For Quick Sort (shown in Table 5), C++ Native sorted 100,000 elements in 5.91 ms, while C++ Wasm took 23.11 ms. Java Wasm was more stable than C/C++ Wasm, and Python Wasm was fast for small inputs but slowed down a lot for big ones.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Language | Workloads | | | | | |
| 10 | 15 | 20 | 22 | 24 | 25 |
| C++\_Native | 0.000379 | 0.004851 | 0.145428 | 3.91706 | 4.4665 | 4.4073 |
| C++\_WASM | 0.128118 | 0.408267 | 3.773976 | 15.09481 | 58.5395 | 128.6963 |
| C\_Native | 0.00184 | 0.005631 | 0.140876 | 4.363013 | 4.431838 | 4.3622 |
| C\_WASM | 0.135154 | 0.384469 | 8.325989 | 32.23789 | 137.6945 | 281.2903 |
| Java\_Native | 0.02 | 0.3 | 1.49 | 3.85 | 16.16 | 24.78 |
| Java\_WASM | 0.1 | 0 | 3.800001 | 14.2 | 56.8 | 116.6 |
| Python\_Native | 0.04 | 0.98 | 30.67 | 123.36 | 496.78 | 982.14 |
| Python\_WASM | 0.343445 | 3.584972 | 84.20718 | 313.4282 | 1263.992 | 2679.514 |

Table 6: Execution Time (ms) for Tower of Hanoi

For Tower of Hanoi (shown in Table 6), C++ Native solved 25 disks in 4.40 ms, but C++ Wasm needed 128.69 ms. Java Wasm handled recursion better than C/C++ Wasm, while Python Wasm became very slow for large inputs.

Overall, Wasm is much slower than native execution for C and C++, performs better for Java, and gives mixed results for Python—fast for small inputs but very slow for large ones. This means Wasm is not a perfect replacement for native execution, especially in tasks that use a lot of memory or recursion.

### Program Equivalence Analysis

Normally, a program compiled to Wasm should behave the same as its native version. However, we found a case where the behavior is not exactly the same.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 1: A Vulnerable C Program

As shown in Figure 1, the C program has a buffer overflow issue. The function vulnerable\_function copies user input into a fixed-size buffer (char buffer[32]) using strcpy, but it does not check the input size. In native execution, if the input is too large, the program crashes because memory gets overwritten. However, in Wasm, the program does not crash. This is because Wasm has built-in memory safety checks, which stop out-of-bounds writes from happening. As a result, the Wasm version behaves differently from the native version. Even though they come from the same source code, their execution is not the same. This shows that Wasm’s memory safety rules change how the program runs and prevent errors that would normally happen in a native environment.

### Vulnerabilities in Webassembly

Unfortunately, vulnerability from memory unsafe language can still propagate to the Webassembly. This is due to the linear unmanaged memory with full permission in Webassembly . Since Webassembly itself is memory safe, these vulnerabilities can only hurt the sandbox without crushing it.

#### Buffer Overflow

Due to the lack of built-in bounds checking, buffer overflows can overwrite adjacent data, enabling attacks that compromise data integrity and application logic. Figure 2 shows C cpde prone to overflow because line 9 fails to perform bounds checking.

手机屏幕截图

AI 生成的内容可能不正确。

Figure 2: Buffer Overflow.

#### Heap Metadata Corruption

Memory accessed after deallocation can lead to unpredictable behavior, allowing attackers to corrupt data or hijack application control flows.

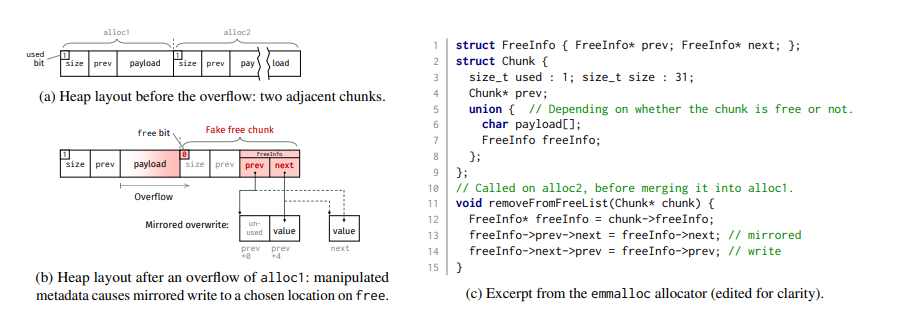


Figure 3 Heap Metadata Corruption

Because Emscripten compiler(compile C/CPP code into webassembly byte code) uses a simplified allocator emmalloc, typically, simplified allocators are vulnerable to metadata corruption attacks, so do emmalloc. Figure 3 shows the detail of how to achieve it on C code. Emmalloc is a first-fit allocator, it will return the first chunk in the free list large enough to satisfy an request. So at very beginning of the execution of program, two allocation requests will yield two chunks adjacent to each other in Figure3 (a). Then, we perform a write overflow for alloc1 in Figure3 (b), overwrite the free bit of alloc2, after this free alloc1. Finally, we can achieve heap metadata corruption.

#### Segmentation Fault

Because Webassembly’s memory don’t have any guard position to prevent read or write from user. The operation can trigger segmentation fault in native environment will still work in Webassembly

### Control Flow in Webassembly

In WebAssembly, although the total number of branch opcodes is higher than in native code, its structured control flow design leads to a lower branch misprediction rate. This design relies on three fundamental constructs:

• Block: Defines a code region with a single entry and exit point, similar to a traditional code block.

• Loop: Implements iterative control by using branch instructions that jump back to the beginning, enabling repetition.

• If/Else: Facilitates conditional execution by selecting between two mutually exclusive code blocks that share a common exit.

These constructs enforce regular and predictable branching patterns—particularly in scenarios such as safety checks (e.g., bounds and stack validations)—which allow CPU branch predictors to quickly learn and accurately anticipate outcomes. In contrast, native code may have fewer branches overall but often includes more complex, data-dependent conditions that are harder to predict, resulting in a higher misprediction rate.

#### Experiment Result For Comparing Branch Miss Rate

For computation intensive tasks, Because it don’t have too much branch in source code, so in this case, Webassembly will have higher branch miss rate than native environments due to it will add more branch for checking. But for other tasks have many branches in source code, Webassembly will outperform native environment.

|  |  |  |
| --- | --- | --- |
| Benchmark | Branch miss rate  Wasm | Branch miss rate  Native |
| Fibonacci | 5.71% | 8.94% |
| Bubble Sort | 7.8% | 12.45% |
| Quick Sort | 12.55% | 9.21% |
| Sieve | 0.18% | 0.02% |
| Hanoi | 0.45% | 0.34% |
| Matrix Multiplication | 0.21% | 0.10% |

Table 7: Branch miss rate in different benchmarks