Performance of WebAssembly

Abdus Satter and Yuchen He

Video Link:

Presentation Link

https://www.youtube.com/watch?v=O-JuAI-6xNA&list=PLJ8tUiNKIo6u2844hWsrUbbW59dr-X7g9&index=2

Building Project and Demo Link

https://www.youtube.com/watch?v=-kuoLHFv6TY&list=PLJ8tUiNKIo6u2844hWsrUbbW59dr-X7g9&index=1

Presentation Slide Link:
https://docs.google.com/presentation/d/1TNskKg1WZrxt9bG8WS4N7skrVMBVW9WxKl9yIPbsJmM/edit?usp=sharing

Link to Github Repo:

https://github.com/Yuchen-He1/CS263Project

Steps for building the project

You can follow the steps below or go to the readme file of the project on github

1. **Environment:** Ubuntu/Linux
2. **Clone the project**
   https://github.com/Yuchen-He1/CS263Project
   cd CS263Project
3. **Requirements:**

```
sudo apt install build-essential
sudo apt install python3 python3-pip
sudo apt install nodejs npm
npm install pyodide
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
```

4. **step to run wasm c benchmark**

   ```
   cd WASM/C_Benchmark/
   chmod +x compile_and_run_wasm.sh
   ./compile_and_run_wasm.sh
   ```

   Output file: 'wasm_results_c.csv'

5. **step to run wasm c++ benchmark**

   ```
   cd WASM/CPP_Benchmark/
   chmod +x compile_and_run_wasm.sh
   ./compile_and_run_wasm.sh
   ```

   Output file: 'wasm_results_cpp.csv'

6. **step to run wasm python benchmark**

   ```
   cd WASM/Python_Benchmark/
   node run_python_wasm.mjs
   ```

   Output file: 'python_wasm_results.csv'

7. **step to run wasm Java benchmark**

   ```
   cd WASM/Java_Benchmark/java-wasm-app/target/java-wasm-app-1.0-SNAPSHOT
   python3 -m http.server 8080
   ```

   To see the output, open the link in the browser and then you will be able to see the output on the webpage.

Report

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 Wasm performs compared to native execution in languages like C, C++, Java, and Python. Although Wasm is faster than JavaScript, its performance compared to compiled languages is still being studied. This project aims to measure Wasm'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 Wasm. 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.

For the study, we used Intel(R) Core(TM) Ultra7 165U 1.70 GHz, 64GB RAM, Ubuntu 24.04 LTS.

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

Language	**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

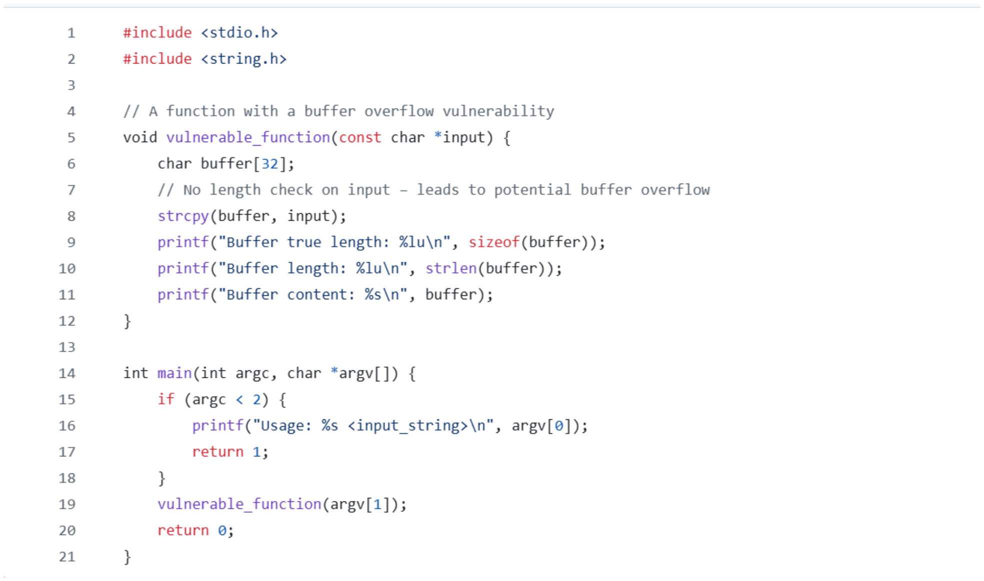
Table 6: Execution Time (ms) for Tower of Hanoi

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.



```c
#include <stdio.h>
#include <string.h>

// A function with a buffer overflow vulnerability
void vulnerable_function(const char *input) {
    char buffer[32];
    // No length check on input - leads to potential buffer overflow
    strcpy(buffer, input);
    printf("Buffer true length: %lu\n", sizeof(buffer));
    printf("Buffer length: %lu\n", strlen(buffer));
    printf("Buffer content: %s\n", buffer);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <input_string>\n", argv[0]);
        return 1;
    }
    vulnerable_function(argv[1]);
    return 0;
}
```

Figure 1: A Vulnerable C Program

As shown in **Error! Reference source not found.**, 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 Wasm. This is due to the linear unmanaged memory with full permission in Wasm. Since Wasm itself is memory safe, these vulnerabilities can only hurt the sandbox without crushing it.

Buffer Overflow

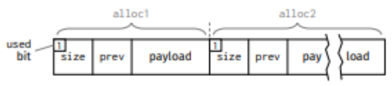
Due to the lack of built-in bounds checking, buffer overflows can overwrite adjacent data, enabling attacks that compromise data integrity and application logic. Figure 2 shows C program prone to overflow because line 9 actually fails to perform proper bounds checking.

```c
1  void parent() {
2    char parent_frame[8] = "BBBBBBBB"; // Also overwritten
3    vulnerable(readline());
4    // Dangerous if parent_frame is passed, e.g., to exec
5  }
6  void vulnerable(char* input) {
7    char same_frame[8] = "AAAAAAAA"; // Can be overwritten
8    char buffer[8];
9    strcpy(buffer, input); // Buffer overflow on the stack
10 }
```

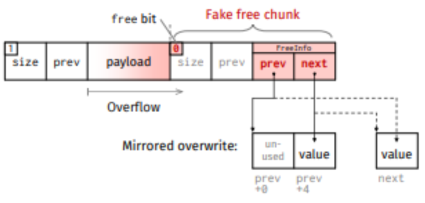
Figure 2: Buffer Overflow.

Heap Metadata Corruption

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



(a) Heap layout before the overflow: two adjacent chunks.



(b) Heap layout after an overflow of alloc1: manipulated metadata causes mirrored write to a chosen location on free.

```c
struct FreeInfo { FreeInfo* prev; FreeInfo* next; };
struct Chunk {
  size_t used : 1; size_t size : 31;
  Chunk* prev;
  union {  // Depending on whether the chunk is free or not.
    char payload[];
    FreeInfo freeInfo;
  };
};
// Called on alloc2, before merging it into alloc1.
void removeFromFreeList(Chunk* chunk) {
  FreeInfo* freeInfo = chunk->freeInfo;
  freeInfo->prev->next = freeInfo->next; // mirrored
  freeInfo->next->prev = freeInfo->prev; // write
}
```

(c) Excerpt from the emmalloc allocator (edited for clarity).

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 a request. So, at the very beginning of the execution of program, two allocation requests will yield two chunks adjacent to each other in Figure 3 (a). Then, we perform a write overflow for alloc1 in Figure 3 (b), overwrite the free bit of alloc2, after this free alloc1. Finally, we can achieve heap metadata corruption.

Segmentation Fault

Because Webassembly's memory doesn'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 Wasm.

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

Overall, due to its well-organized control flow, Wasm exhibits a lower branch miss rate than the native environment. In some benchmarks, such as quicksort, Wasmtime shows a higher miss rate than native code because its JIT adds extra branches on critical paths for boundary and security checks, making them harder to predict. Furthermore, since Wasm3 only interprets WebAssembly bytecode without adding extra checks, its branch miss rate remains lower across all benchmarks.

(we use perf command to extract branch miss rate information)

Benchmark	**Branch miss rate Wasmtime**	**Branch miss rate Native**	**Branch miss rate Wasm3**
Fibonacci	0.31%	0.25%	0.14%
Bubble Sort	8.19%	12.54%	1.45%
Quick Sort	12.64%	9.32%	2.33%
Sieve	0.16%	0.03%	0.02%
Hanoi	0.58%	0.50%	0.24%
Matrix Multiplication	0.22%	0.10%	0.04%

Table 7: Branch miss rate in different benchmarks (C)

Benchmark	**Branch miss rate Wasmtime**	**Branch miss rate Native**	**Branch miss rate Wasm3**
Fibonacci	0.38%	0.26%	0.13%
Bubble Sort	8.07%	11.91%	1.48%
Quick Sort	11.24%	9.16%	2.79%
Sieve	0.28%	0.06%	0.02%
Hanoi	0.57%	0.60%	0.19%
Matrix Multiplication	0.22%	0.10%	0.03%

Table 8: Branch miss rate in different benchmarks(CPP)

As shown in Table 7 and Table 8, the branch miss rate varies across different benchmarks. Wasm3 has the lowest branch miss rate in most cases, especially in tasks like Fibonacci, Sieve, and Matrix Multiplication. Native execution is generally better than Wasmtime, except for Bubble Sort. Wasm3 has a better prediction than both Wasmtime and Native. In recursive tasks like Tower of Hanoi, Native and Wasmtime have similar prediction accuracy, but Wasm3 still performs better. Overall, Wasm3 has the best branch prediction, especially in sorting and iterative tasks.

Conclusion

This study compares Wasm with native execution in C, C++, Java, and Python. The results show that Wasm is much faster than JavaScript but still slower than native execution, especially for memory-intensive and recursive tasks. C and C++ Native are the fastest, while Java Wasm performs better than C/C++ Wasm. Python Wasm works well for small inputs but slows down a lot for larger ones.

We also looked at branch miss rates and found that Wasm3 predicts branches better than Wasmtime and Native execution. This means Wasm3 is more efficient in handling branch-heavy tasks, while Wasmtime has higher miss rates due to extra security checks.

From a security standpoint, Wasm prevents buffer overflows from crashing programs, but it can still inherit some security issues from C and C++. Also, heap metadata corruption is a risk because Wasm uses a simple memory allocator.

Overall, Wasm is useful for secure and portable execution, but it is not a perfect replacement for native code. It needs improvements in speed and memory management to compete with native execution in high-performance tasks.