Performance Evaluation of WebAssembly

Punnal Khan and Md Shafiuzzaman

WebAssembly

- "Assembly" like language for the web
- Compilation target for other languages
- Delivered as compiled binaries
- Execution environment: browser (Web APIs) or a virtual machine
- Memory model: loads and stores to untyped arrays of bytes
- Modular secure compilation

WebAssembly

C++

```cpp
int add(){
    int b = 9;
    int a = 1 + b;

    return a;
}
```

WebAssembly Text

```
(module
  (memory 1)
  (func $add (result i32)
    (local $var0 i32)
    i32.const 0
    i32.load offset=4
    i32.const 16
    i32.sub
    tee_local $var0
            .
            .
            .
```

WebAssembly Bytes

```
00000000: 0061 736d 0100
0000 0185 8080 8000 0160
00000010: 0001 7f03 8380
8080 0002 0000 0484 8080
00000020: 8000 0170 0000
0583 8080 8000 0100 0106
00000030: 8180 8080 0000
079b 8080 8000 0306 6d65
00000040: 6d6f 7279 0200
            .
            .
            .
```

Native code compilation in web

- Node.js C++ Addon: update requires for every new version of Node.js
- Asm.js: still a javascript

WebAssembly claims..

- Runs faster than JavaScript
- Near-native performance

Group, WebAssembly Community, and Andreas Rossberg. “WebAssembly Specification,” n.d., 185.

Is WebAssembly fast?

- A barcode scanner implemented in WebAssembly at eBay boosted the performance of the JavaScript implementation by 50 times [1]

- WebAssembly is slower than JavaScript on the Samsung Internet browser (v7.2.10.12) when performing multiplications on matrices of certain sizes [2]

[1] 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/

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

Is WebAssembly fast?

```
hira@hira-Inspiron-3443:~/WebAssembly/iswasmfast$ node benchmark.js
Levenstein Distance:
   Native x 212,945 ops/sec ±1.25% (88 runs sampled)
   Web Assembly x 190,769 ops/sec ±0.47% (92 runs sampled)
 Fastest is Native

Fibonacci:
   Native x 4,909,081 ops/sec ±0.45% (94 runs sampled)
   Web Assembly x 13,347,474 ops/sec ±2.62% (89 runs sampled)
 Fastest is Web Assembly

Fermat Primality Test:
   Native x 2,149,635 ops/sec ±0.71% (90 runs sampled)
   Web Assembly x 3,083,462 ops/sec ±0.96% (86 runs sampled)
 Fastest is Web Assembly

Simple Linear Regression:
   Native x 253,335 ops/sec ±1.25% (91 runs sampled)
   Web Assembly x 38,706 ops/sec ±0.71% (89 runs sampled)
   Web Assembly using TypedArrays x 47,491 ops/sec ±2.16% (88 runs sampled)
 Fastest is Native

SHA256:
   Native x 15,145 ops/sec ±21.00% (90 runs sampled)
   Web Assembly x 42,139 ops/sec ±0.76% (90 runs sampled)
 Fastest is Web Assembly
```

Courtesy: https://github.com/zandaqo/iswasmfast

Impact of Input Sizes

Benchmarks: 41 WebAssembly binaries and 41 JavaScript programs compiled from 41 widely-used C benchmarks (PolyBenchC)

Input Size	**XS**	**S**	**M**	**L**	**XL**
Input size in benchmarks	MINI_DATASET	SMALL_DATASET	MEDIUM_DATASET	LARGE_DATASET	EXTRALARGE_DATASET
CORRELATION	M=28 N =32	M=80 N=100	M=240 N=260	M=1200 N=1400	M=2600 N=3000
COVARIANCE	M=28 N =32	M=80 N=100	M=240 N=260	M=1200 N=1400	M=2600 N=3000
GEMM	NI=20 NJ=25 NK=30	NI=60 NJ=70 NK=80	NI=200 NJ=220 NK=240	NI=1000 NJ=1100 NK=1200	NI=2000 NJ=2300 NK=2600
GEMVER	N=40	N=120	N=400	N=2000	N=4000

Yan, Yutian, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. “Understanding the Performance of Webassembly Applications.” In Proceedings of the 21st ACM Internet Measurement Conference, 533–49. Virtual Event: ACM, 2021. https://doi.org/10.1145/3487552.3487827

Impact of Input Sizes

Execution time statistics

(a)

Input Size	isWasmFaster	isJsFaster
XS	40	1
S	39	2
M	23	18
L	25	16
XL	23	18

Average memory usages (in KB)

(b)

Input Size	WASM	JS
XS	2,001.54	879.41
S	2,077.27	878.73
M	2,985.78	880.54
L	26,991.05	883.10
XL	100,943.88	889.20

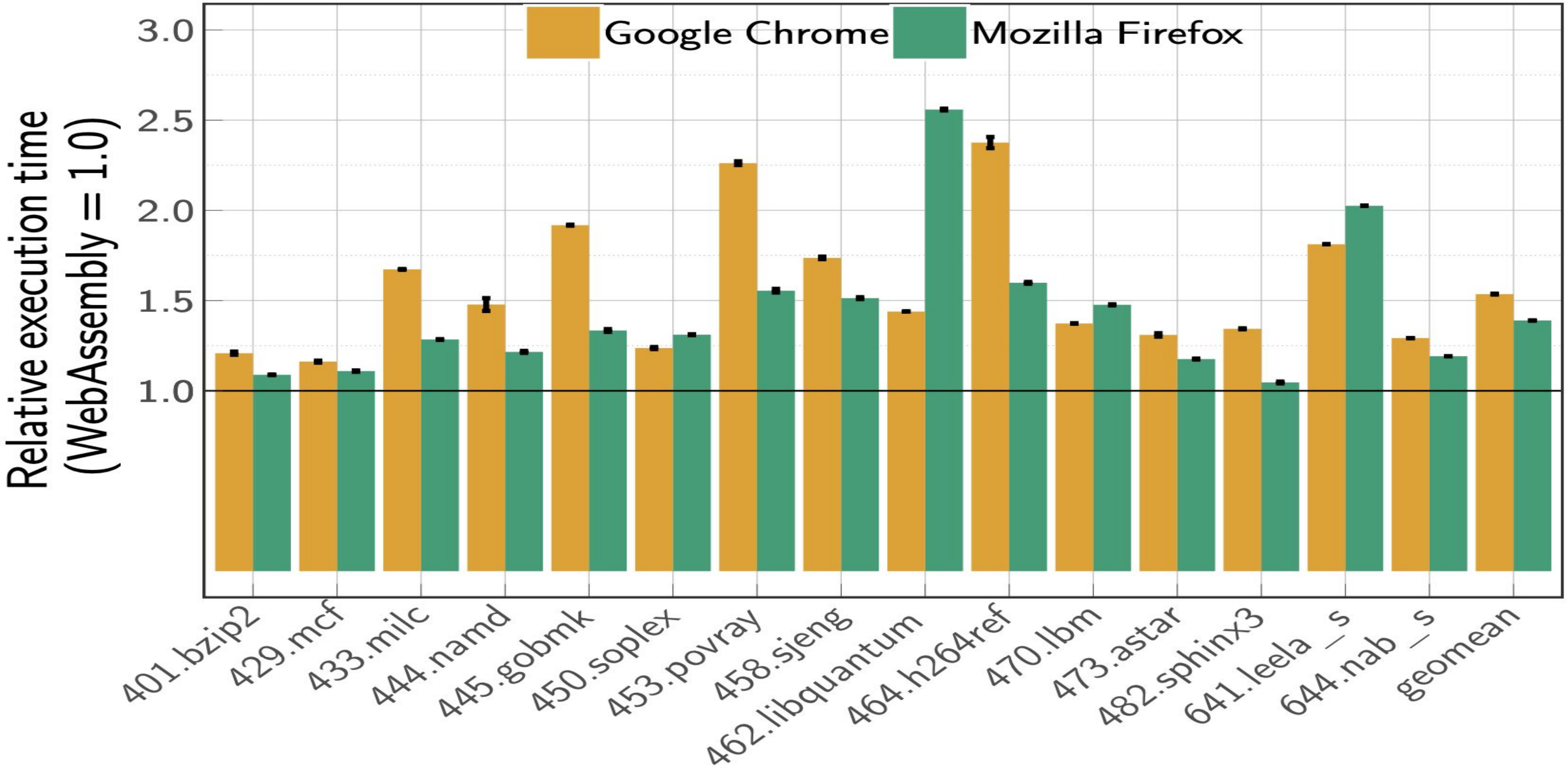
WebAssembly vs. Native Code

- A key goal of WebAssembly to achieve near-native performance

- WebAssembly spec reports WebAssembly runs on average **10% slower** than native code

Group, WebAssembly Community, and Andreas Rossberg. “WebAssembly Specification,” n.d., 185.

WebAssembly vs. Native Code



Jangda, Abhinav, Bobby Powers, Emery D Berger, and Arjun Guha. “Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code,” n.d., 15.

Root cause analysis

```
void matmul (int C[NI][NJ],
             int A[NI][NK],
             int B[NK][NJ]) {
  for (int i = 0; i < NI; i++) {
    for (int k = 0; k < NK; k++) {
      for (int j = 0; k < NJ; j++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Root cause analysis

```
xor   r8d, r8d              #i <- 0
L1:                         #start first loop
  mov   r10, rdx
  xor   r9d, r9d            #k <- 0
  L2:                       #start second loop
    imul   rax, 4*NK, r8
    add    rax, rsi
    lea    r11, [rax + r9*4]
    mov    rcx, -NJ      #j <- -NJ
    L3:                     #start third loop
      mov    eax, [r11]
      mov    ebx, [r10 + rcx*4 + 4400]
      imul   ebx, eax
      add    [rdi + rcx*4 + 4*NJ], ebx
      add    rcx, 1         #j <- j + 1
    jne L3                  #end third loop

    add    r9,  1           #k <- k + 1
    add    r10, 4*NK
    cmp    r9,  NK
  jne L2                    #end second loop

  add   r8,   1             #i <- i + 1
  add   rdi,  4*NJ
  cmp   r8,   NI
jne L1                      #end first loop
pop   rbx
ret
```

```
mov  [rbp-0x28],rax
mov  [rbp-0x20],rdx
mov  [rbp-0x18],rcx
xor edi,edi                          #i <- 0
jmp L1'
L1:                                  #start first loop
   mov ecx,[rbp-0x18]
   mov edx,[rbp-0x20]
   mov eax,[rbp-0x28]
   L1':
   imul r8d,edi,0x1130
   add r8d,eax
   imul r9d,edi,0x12c0
   add r9d,edx
   xor r11d,r11d                     #k <- 0
   jmp L2'
   L2:                               #start second loop
      mov ecx,[rbp-0x18]
      L2':
      imul r12d,r11d,0x1130
      lea r14d,[r9+r11*4]
      add r12d,ecx
      xor esi,esi                    #j <- 0
      mov r15d,esi
      jmp L3'
      L3:                            #start third loop
         mov r15d,eax
         L3':
         lea eax,[r15+0x1]           #j <- j + 1
         lea edx,[r8+r15*4]
         lea r15d,[r12+r15*4]
         mov esi,[rbx+r14*1]
         mov r15d,[rbx+r15*1]
         imul r15d,esi
         mov ecx,[rbx+rdx*1]
         add ecx,r15d
         mov  [rbx+rdx*1],ecx
         cmp eax,NJ                  #j < NJ
      jnz L3                         #end third loop
   add r11,0x1                       #k++
   cmp r11d,NK                       #k < NK
   jnz L2                            #end second loop
add edi,0x1                          #i++
cmp edi,NI                           #i < NI
jnz L1                               #end first loop
retl
```

Increased Register Pressure

- Native code uses 10 registers while WebAssembly uses 13 registers
- WebAssembly reserves r13 to point to an array of GC roots at all times
- WebASsembly uses r10 and xmm13 as dedicated scratch registers

Extra Branch Instructions

- Extra jumps to avoid memory loads
- Stack overflow checks per function call
- Function table indexing checks

Takeaway

- Register issues can be ameliorated by improved implementations

- Stack overflow checks, indirect call checks, and reserved registers checks are necessary for WebAssembly’s safety guarantees

Languages compiled to WebAssembly

Performance comparison between C and Rust compiled to WebAssembly

1) Matrix multiplication: 100X100 arrays with values 1-10,000
2) Insertion sort: int values between 0-999 in an array of size 1000
3) Add: A function that took two values and added them together

Medin, Magnus, and Ola Ringdahl. "Bachelor's Degree in Computer Science," n.d.,16.

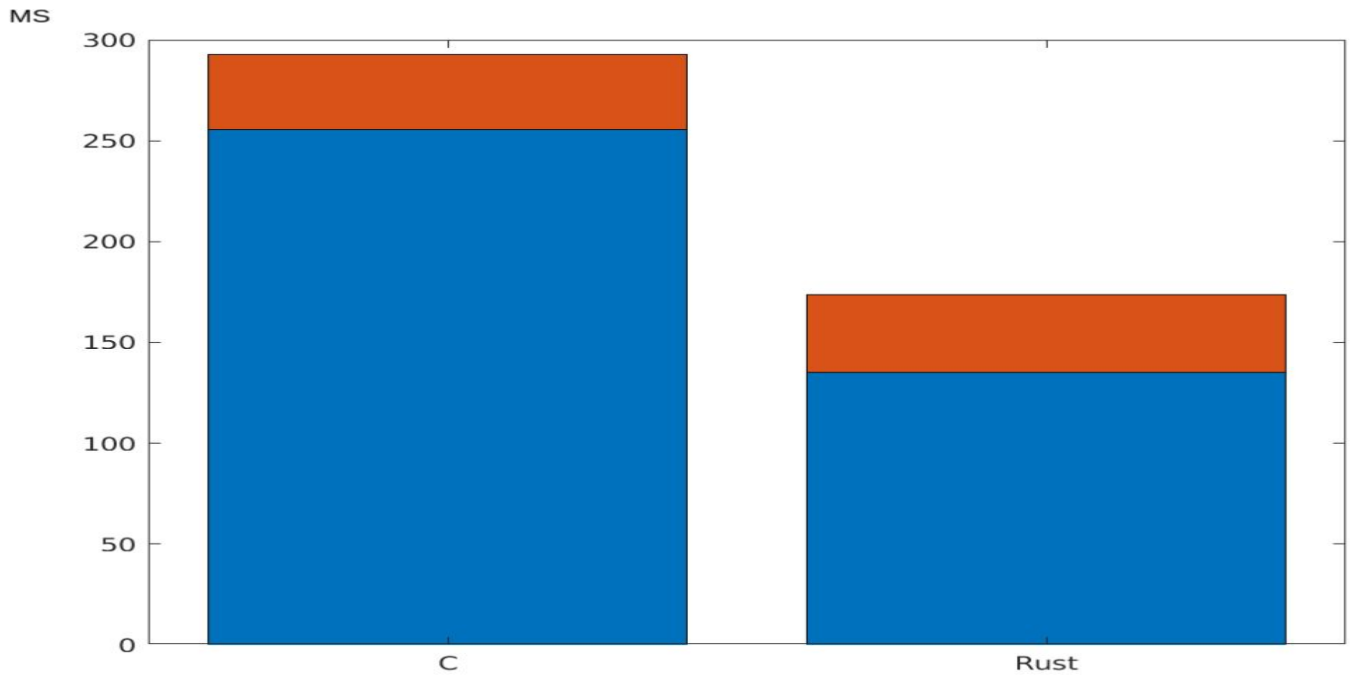
Languages compiled to WebAssembly

Number of instructions for each test:

Algorithm	get	set	add	store	load	offset	memory	tee
Rust MM	59	34	40	3	2	0	0	0
C MM	270	176	23	31	43	69	0	0
Rust Sort	21	11	12	3	2	0	0	8
C Sort	140	105	12	13	22	23	0	0
Rust ADD	2	0	3	0	0	0	3	0
C ADD	12	6	3	0	0	4	0	0

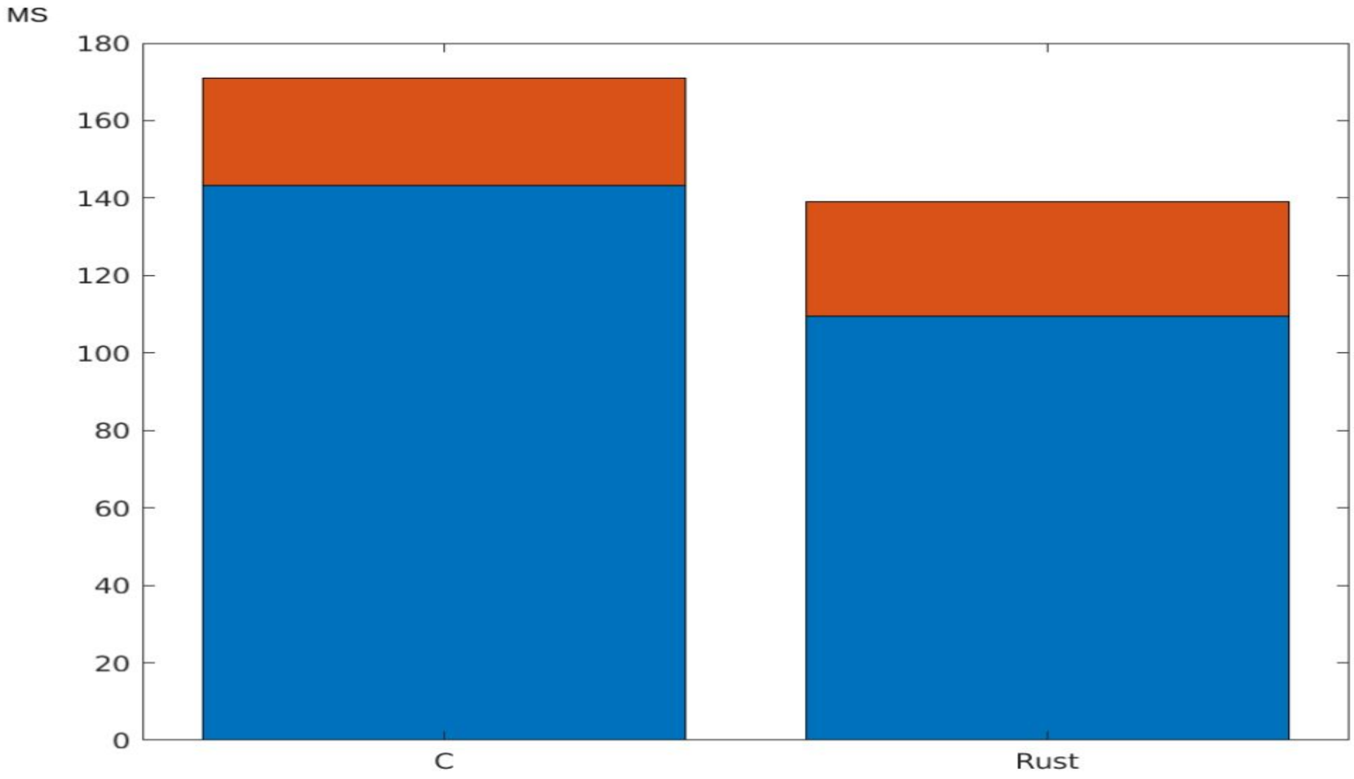
Languages compiled to WebAssembly

Matrix multiplication: Red bar - fetch time, Blue bar- execution time



Languages compiled to WebAssembly

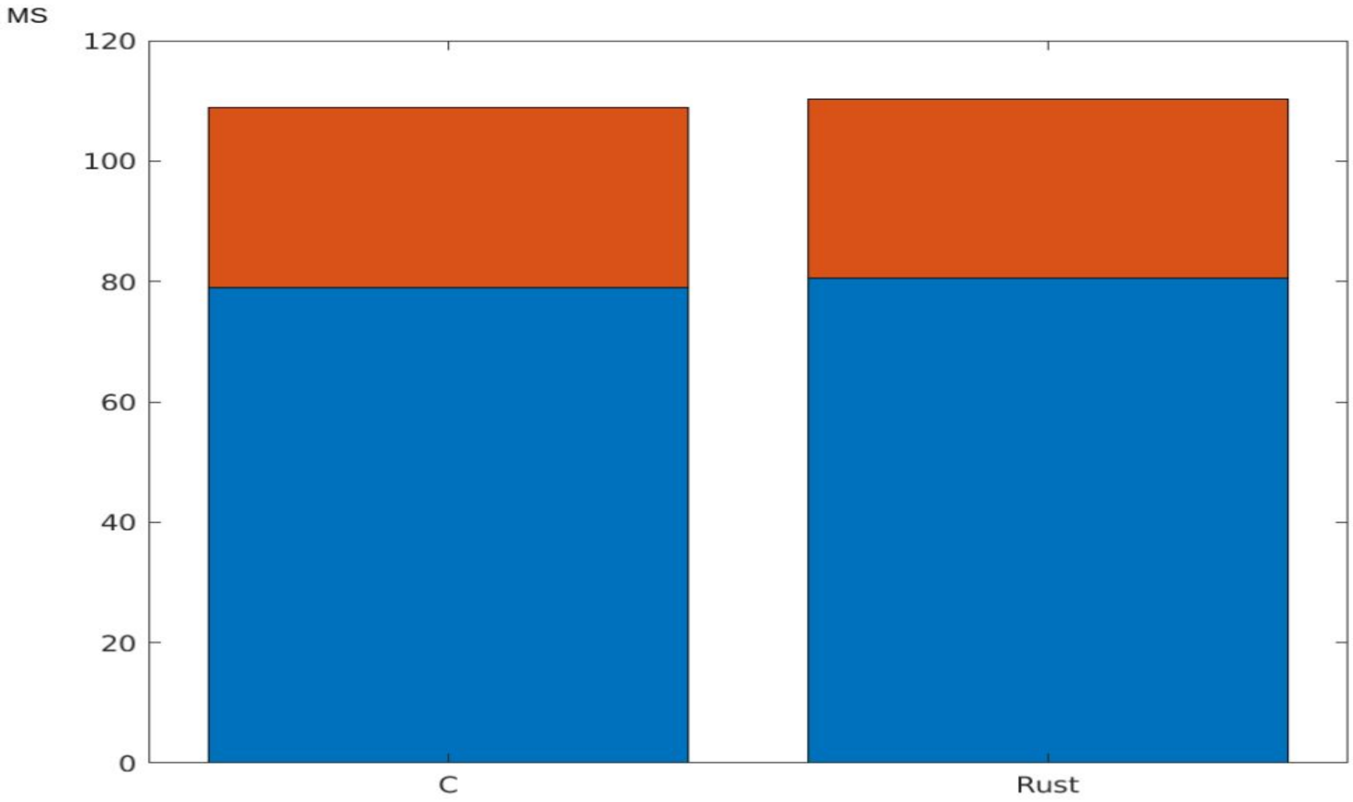
Insertion sort: Red bar - fetch time, Blue bar- execution time



Languages compiled to WebAssembly

UCSB

ADD: Red bar - fetch time, Blue bar- execution time



Languages compiled to WebAssembly

Takeaways:

1) WebAssembly performs differently when it compiles from different languages
2) Rust is preferable for more demanding calculations
3) For shorter and simpler tasks it is not as clear which language to choose
4) Performance of WebAssembly depends on the execution in browser engine rather than file transfer from the server