Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code

Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha, University of Massachusetts Amherst

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference. July 10–12, 2019 • Renton, WA, USA

Presented By: **Md Awsaf Alam [0421052071]**

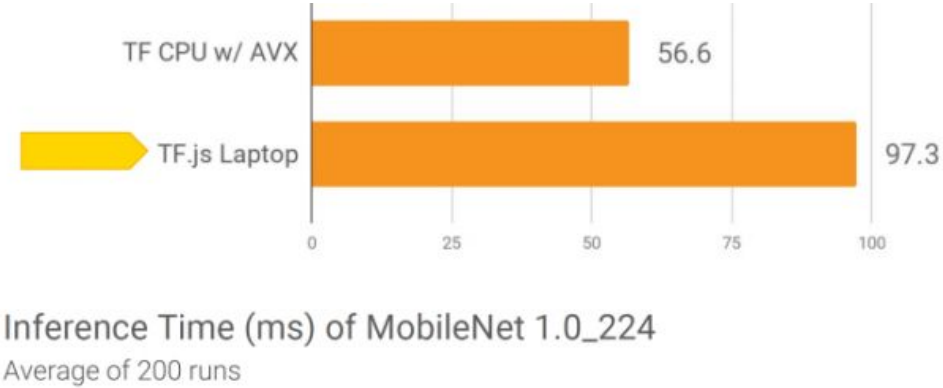
WebAssembly Background

JavaScript is an interpreted language that requires a runtime engine to execute, which can lead to performance overhead and slow startup times for web applications. On the other hand, web applications have grown more and more complex over time.

With the rise of web applications that require high-performance computing, such as games, simulations, and audio-video software, a solution was needed which provided both efficiency and security on the web.

Comparison of Inference Times

Tensorflow.js vs Native



Solutions prior to WebAssembly

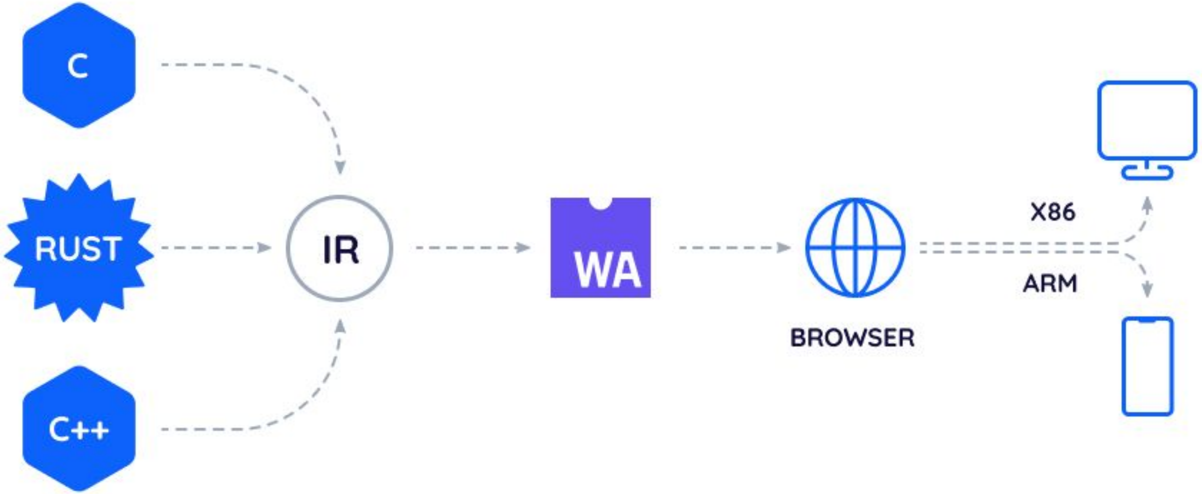
While these technologies have their own strengths and weaknesses, WebAssembly has emerged as a widely adopted solution for running performance-critical code on the web platform. It provides a portable, safe, and fast runtime environment for multiple languages and has broad support across major web browsers.

Technology	Compiled Format	Speed	Portability	Security
ActiveX	code-signed-binary	Fast	Not Portable	Not Secure
Native Client & Native Portable Client	NACl module	Fast	Not Portable (only supported in Chrome)	Safe
Emscripten*	asm.js (subset of js0	Faster than JS but Slower than Native	Portable	Safe

Emscripten* - does not support 64-bit integer, and has larger code size

Webassembly

WebAssembly was released in 2017. All major web browsers now support WebAssembly. It is a lowlevel bytecode intended to serve as a compilation target for code written in languages like C, C++, Rust & GO etc. It offers portability along with performance and security.



WebAssembly Features

- **Safety**: WebAssembly provides a sandboxed execution environment for running code in a web browser, which helps protect user data and system integrity by isolating the code from the rest of the system.
- **Performance**: Low-level code emitted by a C/C++ compiler is optimized ahead-of-time for full machine performance.
- **Portability**: Essential for code targeting the Web to run across all hardware and platform types.
- **Compact code**: Crucial for reducing load times, saving bandwidth, and improving responsiveness on the Web.

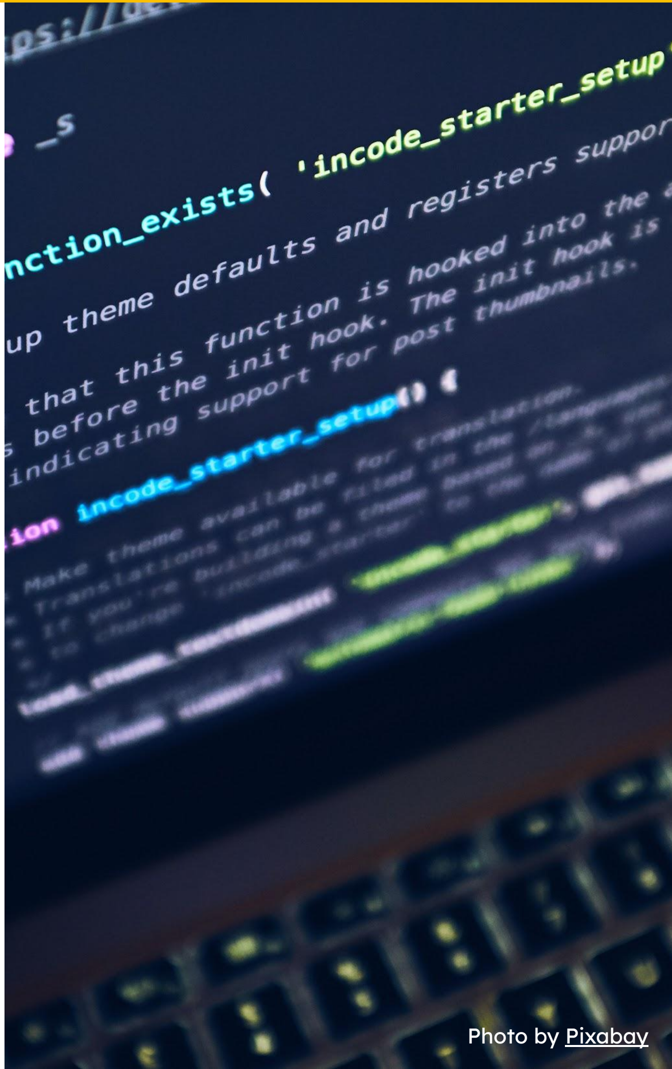


Photo by Pixabay

Is WebAssembly Fast?

According to the paper that Introduced WebAssembly[1] their evaluation on polybenchC benchmarks and found that:

WebAssembly is <u>only</u> 26% slower than Native Code.

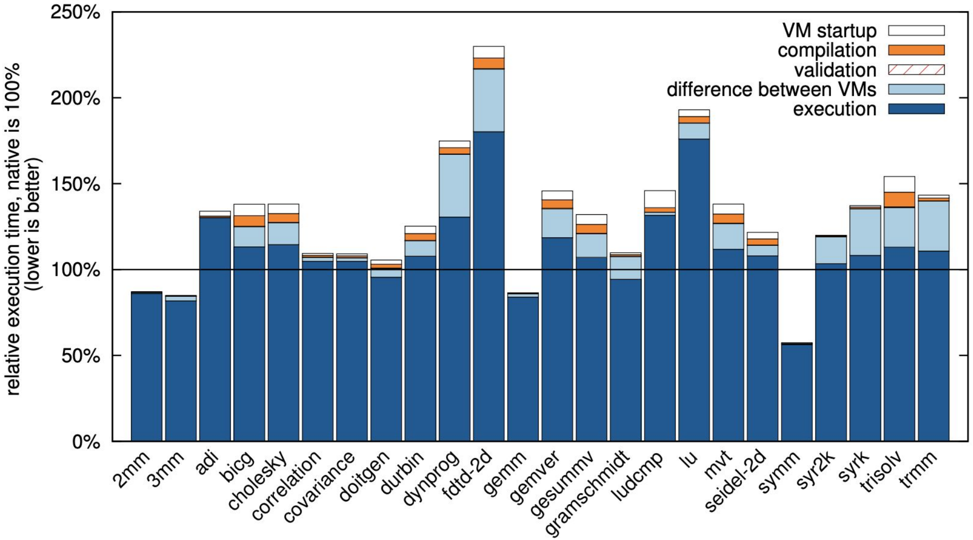


Fig: Relative execution time of the PolyBenchC benchmarks on WebAssembly normalized to native code

[Bringing the Web up to Speed with WebAssembly[1]](#)

Is WebAssembly Fast?

There have been continuous improvements in webassembly implementation, and we have **15 benchmarks within 10% of native performance.**

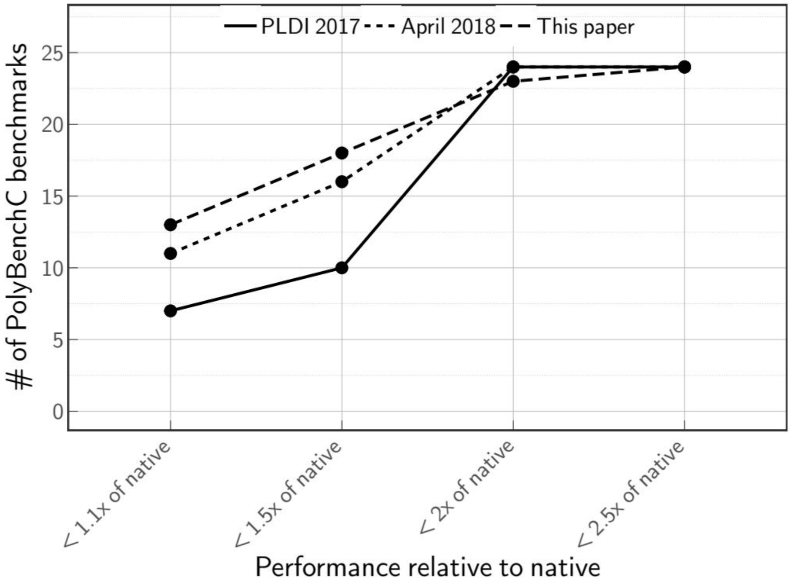


Fig: In 2017 [2], seven benchmarks performed within 1.1× of native.
In April 2018, 11 performed within 1.1× of native.
In May 2019, 13 performed with 1.1× of native

[Bringing the Web up to Speed with WebAssembly[1]]

But, polybenchC Benchmarks are not very practical!

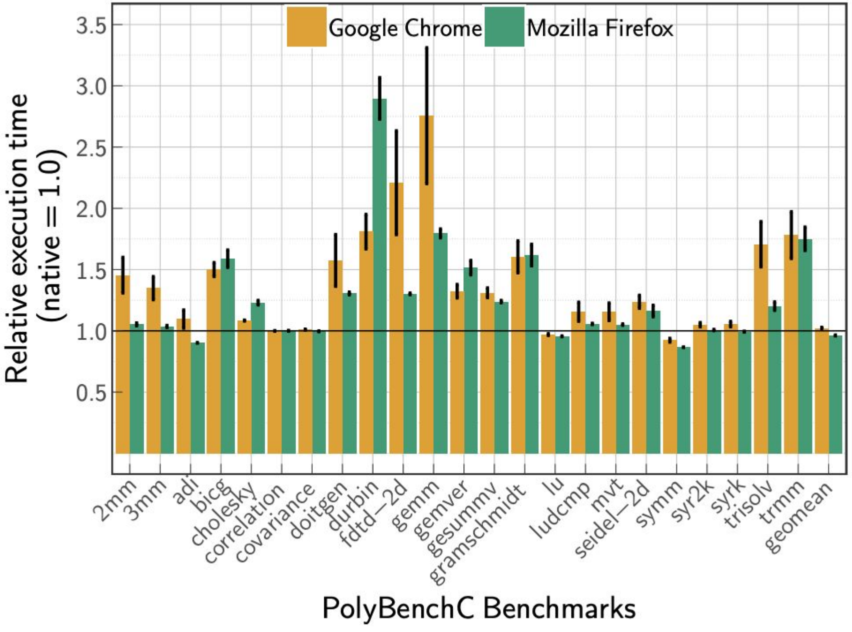
The authors tried using SPEC CPU suite of benchmarks - applications compiled to WebAssembly run slower by an average of 45-55%

PolybenchC only includes small scientific computing kernels rather than full applications (e.g., matrix multiplication and LU Decomposition); each is roughly 100 LOC.

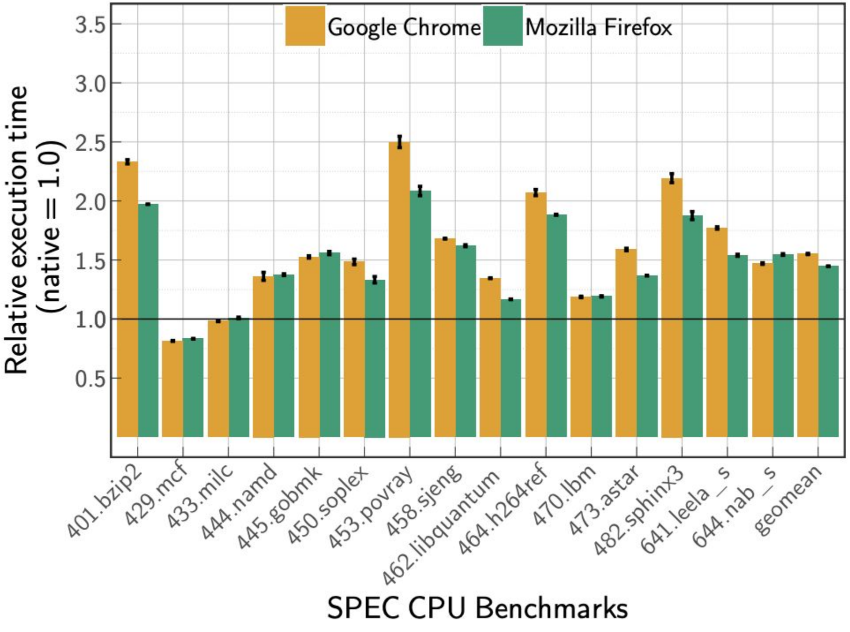
Browser	Average Slowdown	Peak Slowdown
Firefox	45%	2.08×
Chrome	55%	2.5×

Results

According to the benchmark results, there is a significant speed difference between WebAssembly and native code.



(a)



(b)

SPEC-CPU Benchmark

- The WebAssembly documentation lists a number of targeted use cases, including simulations, programming language interpreters, virtual machines, POSIX programs, image editing, video editing, image recognition, and image editing.

- The high performance of WebAssembly on the scientific kernels in PolybenchC does not, therefore, suggest that it will perform well given a different sort of application.

Problem:

- Not possible to compile a sophisticated native program to WebAssembly. (system call & file systems not supported in browser)
- Modifying the benchmark code would be a threat to validity of the experiments.

Solution: **BROWSIX-WASM**

KEY CONTRIBUTIONS

BROWSIX-WASM:

Extension to Browsix [provides system-calls for web apps in JS] to run unmodified WebAssembly-compiled Unix applications directly inside the browser.

BROWSIX-SPEC:

A harness that extends BROWSIX-WASM to allow automated collection of detailed timing and hardware on-chip performance counter information in order to perform detailed measurements of application performance.

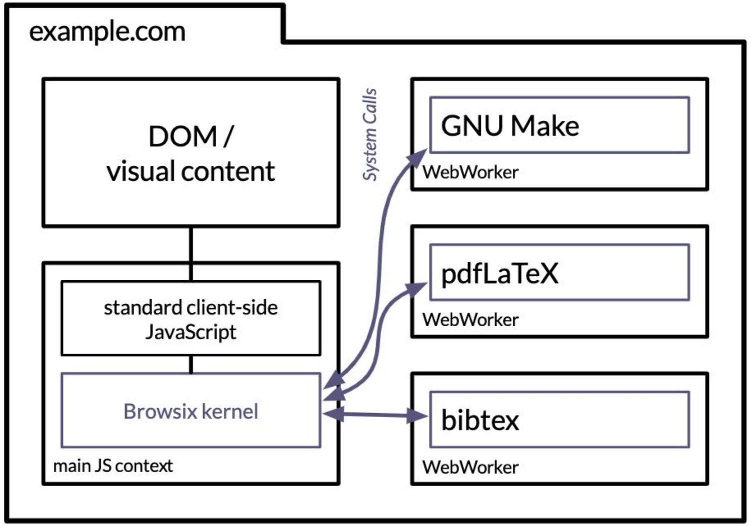
Browsix

Browsix bridges the gap between conventional operating systems and the browser, enabling programs expecting a Unix-like environment to **run directly in the browser**.

By mapping current browser APIs, such as Web Workers and postMessage, onto low-level Unix primitives, such as processes and system calls, Browsix does this.

- Browsix only supports JS, not WASM

- Browsiz uses SharedArrayBuffer for process-kernel communication which WASM does not support.



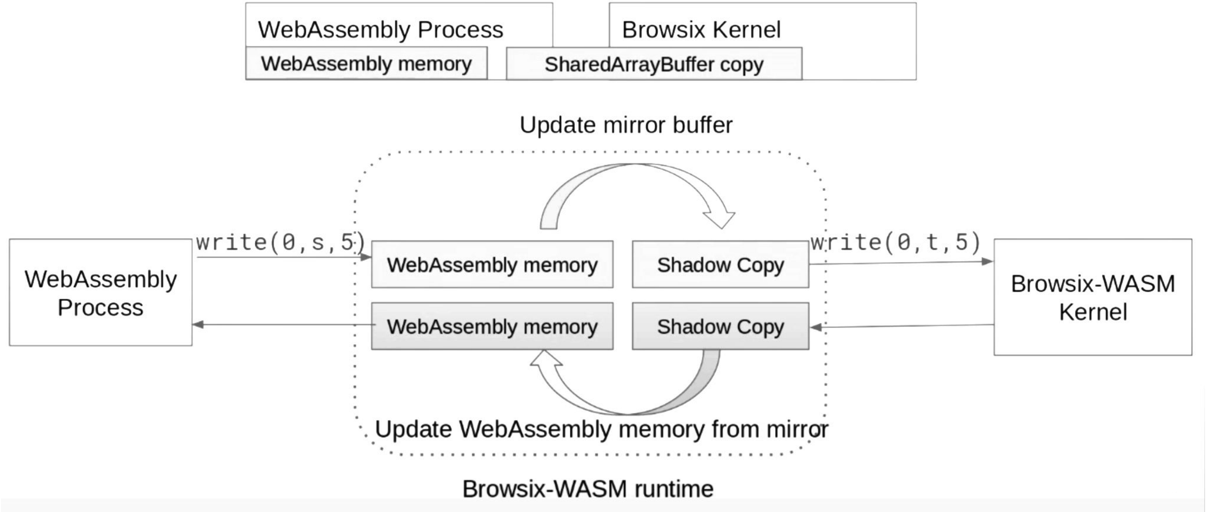
Compilation of Browsix-wasm

Generates WebAssembly binary embedded in a JS module with Browsix-WASM runtime.

Browsix-WASM Runtime provides:

- Libmusl C library
- Communication with Browsix-WASM kernel

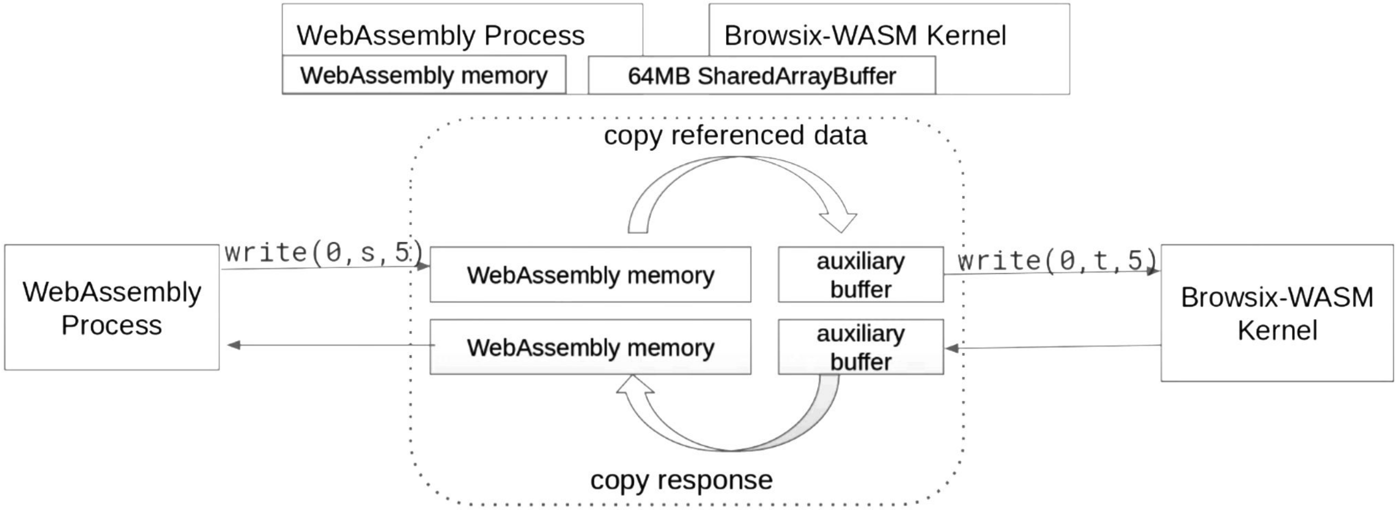
Shadow Copy of WebAssembly memory



For every systemcall, the buffer is updated with the latest version of WASM Memory

Unfortunately this has high copying overhead and 2x memory usage.

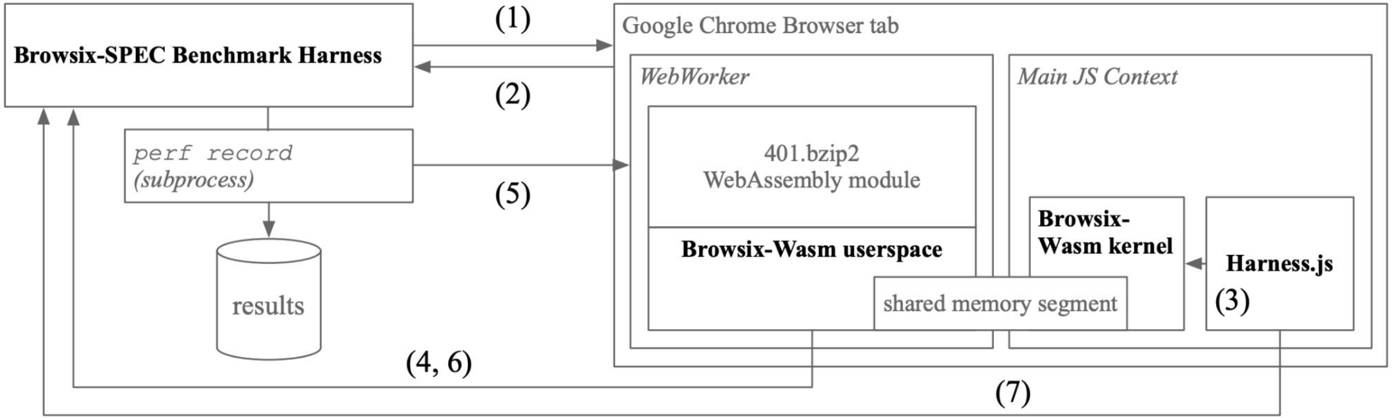
Auxiliary buffer for process-kernel communication



Browsix-WASM runtime copies arguments and response

- Only the referenced data is copied.
- For more than 64Mb, a single systemcall is split into several messages.
- This has minimum execution and memory overhead.

Browsix-SPEC workflow



1. Launch new browser instance using WebBrowser automation tool (selenium)
2. Load page's HTML, harness JS, and BROWSIX-WASM kernel JS over HTTP
3. Initialize BROWSIX-WASM kernel and start new BROWSIX-WASM process executing the runspec shell script

4. XHR request to BROWSIX-SPEC to begin recording performance counter stats

5. Attach perf to Chrome thread corresponding to Web Worker process 401.bzip2

6. Final XHR to benchmark harness to end perf record process

7. POST tar archive of SPEC results directory to BROWSIX-SPEC after runspec program exits, and validate output.

Challenges of JIT Compiler

- Fast but Poor register allocation
- Fast but Poor instruction selection
- Extra branches
- Does not use all x86 addressing modes
- Stack overflow checks (for safety)
- Indirect function call check (for safety)

Benchmark	Native	Google Chrome	Mozilla Firefox
401.bzip2	370 ± 0.6	864 ± 6.4	730 ± 1.3
429.mcf	221 ± 0.1	180 ± 0.9	184 ± 0.6
433.milc	375 ± 2.6	369 ± 0.5	378 ± 0.6
444.namd	271 ± 0.8	369 ± 9.1	373 ± 1.8
445.gobmk	352 ± 2.1	537 ± 0.8	549 ± 3.3
450.soplex	179 ± 3.7	265 ± 1.2	238 ± 0.5
453.povray	110 ± 1.9	275 ± 1.3	229 ± 1.5
458.sjeng	358 ± 1.4	602 ± 2.5	580 ± 2.0
462.libquantum	330 ± 0.8	444 ± 0.2	385 ± 0.8
464.h264ref	389 ± 0.7	807 ± 11.0	733 ± 2.4
470.lbm	209 ± 1.1	248 ± 0.3	249 ± 0.5
473.astar	299 ± 0.5	474 ± 3.5	408 ± 1.0
482.sphinx3	381 ± 7.1	834 ± 1.8	713 ± 3.6
641.leela_s	466 ± 2.7	825 ± 4.6	717 ± 1.2
644.nab_s	2476 ± 11	3639 ± 5.6	3829 ± 6.7
Slowdown: geomean	–	**1.55**×	**1.45**×
Slowdown: median	–	**1.53**×	**1.54**×

Matrix Multiplication

- Chrome generated code has 2x more instructions
- Not using all x86 addressing modes
- Chrome uses 3 more registers than clang.
- Extra jumps for chrome

```
add [rdi + rcx*4 + 4*NJ], ebx
```

Figure 6. Addition in Clang

```
add ecx,r15d
mov [rbx+rdx*1],ecx
```

Figure 7. Addition in WebAssembly

```c
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];
      }
    }
  }
}
```

(a) matmul source code in C.

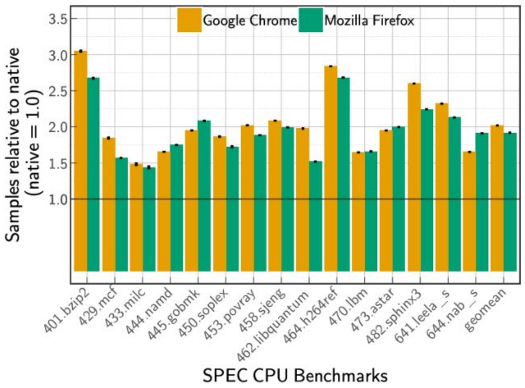
```
xor  r8d, r8d            #i <- 0
L1:                      #start first loop
 mov  r10, rdx
 xor  r9d, r9d           #k <- 0
 L2:                     #start second loop
  imul  rax, 4*NK, r8
  add   rax, rsi
  lea   r11, [rax + r9*4]
  mov   rcx, -NJ       #j <- -NJ
  L3:                    #start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

(b) Native x86-64 code for matmul generated by Clang.

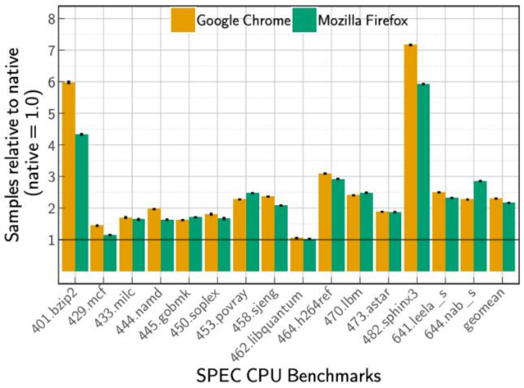
```
mov [rbp-0x28],rax
mov [rbp-0x20],rdx
mov [rbp-0x18],rcx
xor edi,edi                  #i <- 0
jmp L1'
L1:                          #start first loop
 mov ecx,[rbp-0x18]
 mov edx,[rbp-0x20]
 mov eax,[rbp-0x28]
 L1':
 imul r8d,edi,0x1130
 add r8d,eax
 imul r9d,edi,0x12c0
 add r9d,edx
 xor r11d,r11d               #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
```

(c) x86-64 code JITed by Chrome from WebAssembly matmul.

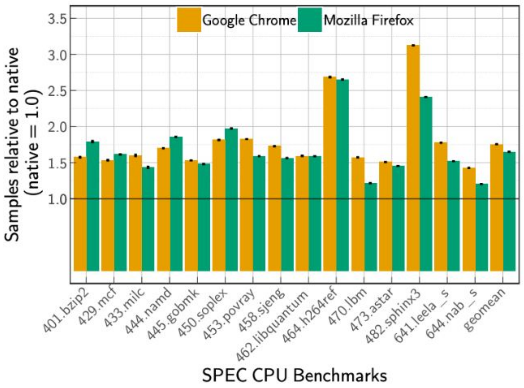
Performance Analysis



(a) `all-loads-retired`



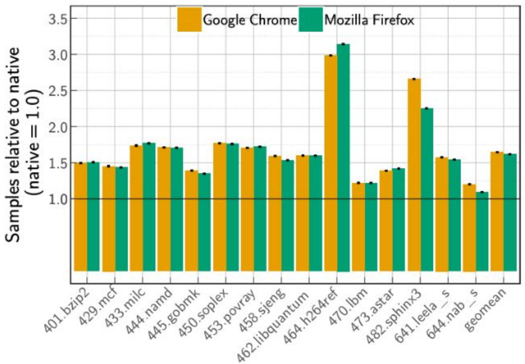
(b) `all-stores-retired`



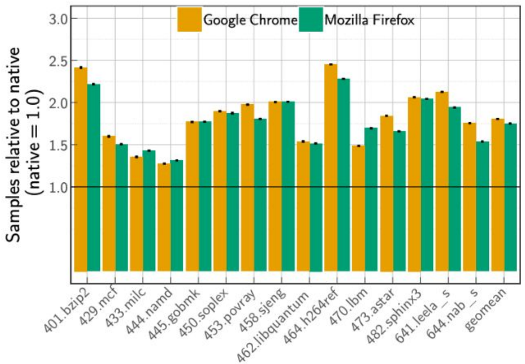
(c) `branch-instructions-retired`

Code generated by Chrome has 2.02× more load instructions retired and 2.30× more store instructions retired than native code. WebAssembly compiled SPEC CPU benchmarks suffer from increased register pressure and thus increased memory references.

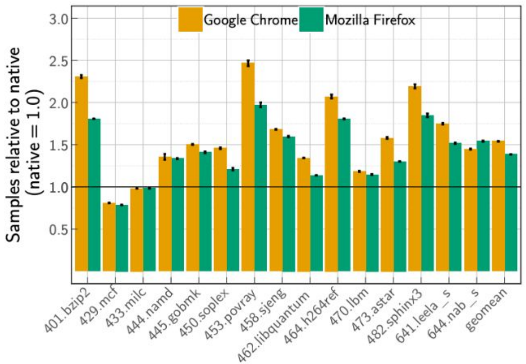
Performance Analysis



(d) conditional-branches



(e) instructions-retired



(f) cpu-cycles

In Chrome 1.75× and 1.65× more unconditional and conditional branch instructions retired respectively. So, all the SPEC-CPU benchmarks incur extra branches,

Conclusion

The authors developed BROWSIX-WASM, a significant extension of BROWSIX, and BROWSIX-SPEC, a harness that enables detailed **performance analysis**, to let them run the **SPEC CPU2006** and **CPU2017** benchmarks as WebAssembly in Chrome and Firefox. [Better alternative to **PolybenchC**

They found that the **mean slowdown** of WebAssembly vs. native across SPEC benchmarks is 1.55× for Chrome and 1.45× for Firefox, with **peak slowdowns** of 2.5× in Chrome and 2.08× in Firefox.

They provided actionable insights on how this performance could be improved.

Future Work & Criticisms

- Authors have not considered any other instruction set other than intel. [e.g ARM]

- More time could be provided to the optimizer - trade off between JIT compilation and Static compilation.
- Variations of output code for different source languages was not observed.

References

1. Bringing the web up to speed with WebAssembly - PLDI 2017: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. June 2017. Pages 185–200 https://doi.org/10.1145/3062341.3062363
2. Browsix - https://browsix.org/
3. Khan, Faiz and Foley-Bourgon, Vincent and Kathrotia, Sujay and Lavoie, Erick. [n.d.]. Ostrich Benchmark Suite. https://github.com/Sable/Ostrich
4. Lei Lopez. 2015. Halophile: Comparing PNacl to Other Web Technologies. (2015).
5. Bobby Powers, John Vilk, and Emery D Berger. 2017. Browsix: Bridging the gap between Unix and the browser. ACM SIGOPS Operating Systems Review 51, 2 (2017), 253–266.
6. David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. 2010. Adapting software fault isolation to contemporary CPU architectures. (2010).
7. Christian Wimmer and Michael Franz. 2010. Linear scan register allocation on SSA form. In Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization. ACM, 170–179.
8. Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009.
9. Native client: A sandbox for portable, untrusted x86 native code. In 2009 30th IEEE Symposium on Security and Privacy. IEEE, 79–93.

Thank you. Please feel free to ask any questions. 😄