A WebAssembly-Based Runtime for Client-Side C/C++ Compilation and Interactive Debugging

Jacob Tristan Roberts-Baca   Yousef Anees AbuHashem

Abstract

Online programming environments traditionally rely on server-side compilation and execution, introducing infrastructure cost and complexity. We present a system for client-side compilation and interactive debugging of DWARF-compatible programs entirely within the browser, built on WebAssembly and DWARF debug information. Our approach instruments WebAssembly binaries at compile time to maintain a parallel debug stack, recording raw machine state (locals, globals, and operand stack values) needed to reconstruct source-level variable values at runtime. We describe our design for probe-point identification, debug frame layout, and variable reconstruction, and evaluate a proof-of-concept prototype on four C benchmarks. Instrumentation introduces modest build-time overhead (2–6%) but currently incurs substantial execution-time slowdown in debug mode (17–43×); we discuss targeted optimizations to address this. Our system demonstrates that source-level interactive debugging of WebAssembly is feasible entirely in-browser, without server infrastructure, VMs, or interpreter-level overhead.

1 Introduction

Despite WebAssembly's prevalence as a portable bytecode and compilation target, it remains difficult to compile and run arbitrary user code on the web that is not known in advance. Online IDEs traditionally rely on server side compilation and execution of code, which increases infrastructural complexity, cost, and maintenance. Beyond simply running arbitrary code, it is desirable to also be able to inspect and debug WebAssembly programs. A fully local execution environment seems, in theory, to allow for collecting data about code as it runs, such as function backtraces, declared variable types, and the contents of the stack and heap memory. This would allow the creation of interactive debuggers and program visualizers for education, all without leaving the browser.

Our project aims to showcase a design for a system that would allow debugging WebAssembly code within the browser via instrumentation of binaries compiled from a DWARF-capable compiler (e.g. Clang/LLVM, GCC). We have implemented a small proof-of-concept of this design, as well as an interactive demo of the system.

2 Background

WebAssembly is a low-level, portable bytecode designed to run safely and efficiently inside web browsers and other sandboxed environments. It serves as a compilation target for languages such as C, C++, and Rust, allowing programs written in these languages to execute on the web with near-native performance. WebAssembly defines a stack-based virtual machine with a compact binary format, structured control flow, and a strongly typed instruction set. Programs are compiled into native instructions on the host machine to allow execution at near-native performance, and effectively run inside a sandbox with explicit access to linear memory and host-provided imports (e.g., JavaScript functions).

We build on prior work [1, 2] to allow execution of user programs written in C/C++ within the browser by compiling the compiler toolchain into WebAssembly and running it in-browser to produce and execute new WebAssembly binaries.

3 Alternatives

Before beginning this project, we considered the following alternative approaches.

Compiling an existing debugger

Porting GDB or LLDB to WebAssembly is fundamentally infeasible due to the constraints of the WebAssembly execution model. Linux debuggers rely on `ptrace` to inspect child process memory and registers, and dynamically rewrite the code section to insert breakpoints as trap instructions. Neither mechanism is available in WebAssembly: the runtime state of a module — local variables, the operand stack, and other

program state — is not introspectable, and the module binary cannot be modified at runtime. The latter restriction is by design, as permitting runtime code mutation would violate WebAssembly's security and performance guarantees.

Running in a VM

An alternative approach is to boot a Linux VM via an x86 emulation layer such as copy.sh/v86 [3], WebVM [4], or Fabrice Bellard's QEMU [5], within which a conventional debugger can run unmodified. This approach is unattractive for two reasons. First, it imposes substantial startup latency: the Linux distribution image, debugger binary, and supporting environment must all be fetched and initialized before any code can execute. In practice, we measured startup times on the order of tens of seconds, which is unacceptable for what is intended to be a lightweight, low-friction debugging library. Second, x86 emulation introduces significant runtime overhead relative to native WebAssembly execution, which browsers compile to machine code via JIT; we expected the emulation penalty to be at least an order of magnitude, making interactive debugging performance impractical.

Running in an interpreter

Running WebAssembly inside an interpreter — whether custom-built or via an existing runtime such as wasm3 [6] — would make machine state introspection straightforward and eliminate startup overhead. However, doing so foregoes JIT compilation by the browser, which is the primary source of near-native WebAssembly performance. For a general-purpose debugger, imposing interpreter-level overhead on the debuggee is unacceptable.

Using existing browser tooling

Chrome's DevTools infrastructure supports source-level WebAssembly debugging by consuming DWARF debug information [7], implemented via a C++ plugin compiled to WebAssembly. This functionality, however, is exposed exclusively as developer tooling and not as an API accessible to library or application developers. Even if such an API were stabilized, a dependency on it would bind the debugger to Chromium, precluding cross-browser deployment.

4 Methodology

Our approach transforms a WebAssembly binary into an instrumented one that supports introspection, and provides a runtime interface for resolving source-level variable values from the state exposed by the instrumented binary.

```
  1 int add(int a, int b) {
→ 2     int c = a + b;
→ 3     return c;
  4 }
```

```
  1 (func $add
  2     (param i32 i32)
  3     (result i32)
→ 4     local.get 0
  5     local.get 1
→ 6     i32.add
  7     return)
```

Figure 1: Source and WebAssembly with probe points (arrows).

4.1 Probe Points

We identify *probe points*—locations in the binary where instrumentation may trigger a break—by correlating DWARF debug information with the WebAssembly code section. This approach limits the set of locations where we must generate and run instrumentation code at the expense of being able to introspect the program with instruction-level granularity. The compiler emits DWARF line information (e.g., `.debug_line`) that maps source file and line to byte offsets in the code segment. We parse this mapping to obtain a set of breakable locations: each is a (file, line, column) triple plus a code-section offset. The instrumenter then inserts a call to a host-provided breakpoint function immediately before the instruction at each such offset. When execution reaches that instruction, control is transferred to the runtime, which can consult a shared buffer [1] to determine whether any breakpoint is enabled for that location and, if so, pause and expose the current call stack and variable state. Figure 1 illustrates the correspondence between source lines and WebAssembly instructions; the arrows indicate program points where a probe may be placed.

4.2 Instrumentation

To support variable inspection on pause, the instrumented code maintains a separate *debug stack* in a dedicated linear memory (or a reserved region), distinct from the program's

[1] The use of a shared buffer was intended to prevent additional communication with the host environment to determine if a breakpoint was hit. In the common case, no breakpoint is set, and so nothing more needs to be done to handle a breakpoint. When a breakpoint is hit, the running thread is put to sleep and control is transferred to the main thread.

own stack and heap. A global holds the current debug stack pointer. On function entry, the instrumented prologue decrements this pointer by the size of that function's *debug frame* and writes a frame *tag* (e.g., the function index) at the new base. Each debug frame does not store source-level variables directly; instead, it stores a fixed layout of *WebAssembly-level* state: the values of locals, relevant globals, and (where needed) operand-stack slots at the points where breakpoints may fire. That is, the instrumentation records the raw machine state (locals, globals, operands) that the DWARF location expressions refer to, rather than evaluating those expressions inside the instrumented code. This keeps the instrumentation simple and avoids implementing the full DWARF location expression language in WebAssembly.

The layout of each stack frame is determined at instrumentation time. At each probe point, we store the raw machine state (locals, globals, and operands) that would be needed to determine the values of that probe point's extant variables during runtime (more specifically, we examine the DWARF expression bytecode to see which WebAssembly locations are implicated). The metadata for each stack frame slot also maintains information about the stored value type (e.g. `i32`, `f64`) [2]. This is necessary because the type of a location in the WebAssembly operand stack may change throughout the invocation of a function. For each stack slot, we track metadata about which probe points the slot was active in to (its *lifetime*) aid in selecting the slot with the correct type when pausing execution.

Note that some WebAssembly value types, particularly reference types (e.g. `externref`, `funcref`, etc.) are opaque and cannot be stored to linear memory. Where values of these types are implicated by the DWARF info, we ignore them and treat them as optimized out. In our experience, LLVM compiler toolchains do not implicate values with these types, so in practice this is not an issue.

Figure 2 shows the layout of a single debug stack frame. $f \in \mathbb{N}$ is a tag word identifying a function F that has executed. $v_0, \ldots, v_k$ hold information about machine state needed to reconstruct values of the variables of F – each could represent either a WebAssembly local, global, or operand with a specific index and type. Depending on the type, the value will be stored with a different bit width, e.g. `i32` will be stored with 4 bytes and `i64` with 8. p is the last probe point executed by this invocation of F, needed to determine which stack slots are currently active at this point in the function[3].

Frames are laid out in call order: the innermost (current) frame is at the lowest address, and the stack pointer advances downward. Each frame begins with a tag word identifying the function and probe point; the remainder is a contiguous block of slots whose layout is described in the debug metadata (which locals/globals/operands are stored and at which byte offsets, what type each slot holds, and for which probe points the slot is active). The runtime uses this layout, together with the saved stack pointer at pause time, to walk the stack and associate each frame with its function and stored state.

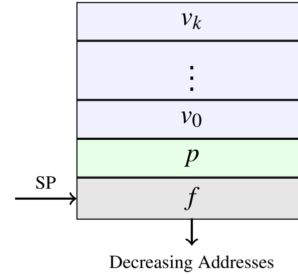


Figure 2: Debug stack frame layout. Each frame begins with a function tag and probe, followed by typed slots for globals, locals, and operand stack entries.

4.3 Variable Reconstruction

When execution is paused at a probe, the runtime must present the user with source-level variable names and values. DWARF describes, for each variable in a subprogram, a *location expression*: a small program that, when evaluated, yields where the variable's value may be found (e.g., "in local 2," "at `frame_base` + 8," or "in global 3"). We do not evaluate these expressions inside the instrumented WebAssembly; instead, the instrumented code records the underlying WebAssembly state (locals, globals, and operand stack) in the debug stack. At pause time, the runtime walks the debug stack to obtain the call chain and, for each frame, the saved values. For each variable declared in the corresponding DWARF subprogram, it evaluates the variable's location expression *in the runtime*: if the expression refers to a local, global, or operand stack value, the value is read from the corresponding slot in the debug frame; only active slots are considered, where an active slot is one whose lifetime includes the probe point p listed in its debug stack frame. If it refers to memory (e.g., `frame_base` + offset), the value is read from the program's linear memory. Type and name information come from DWARF, so the runtime can decode bytes into integers, floats, pointers, etc. and format them appropriately for display.

5 Results

We implemented a proof-of-concept of the above design. Due to time and scoping constraints, it differs from the above design in a number of important aspects. Notably,

[2]To determine the types of operand stack values, we must statically track which types have been pushed to the stack. For example, after an `i32.const 0` instruction, the top of the operand stack holds an `i32`. While tracking operand types, care must be taken to respect the block-scoping and inference rules of WebAssembly.

[3]To be precise, the value of p is stored by the caller before making a function call, and any time a breakpoint is hit before control is transferred to the debugger.

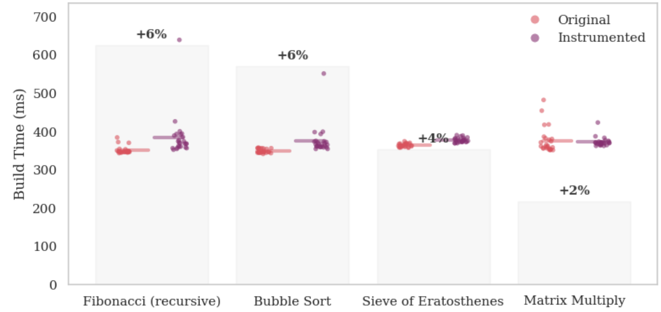
we faced a significant setback in recovering variables at runtime, as the Clang/LLVM compiler we used [9] predates the introduction of WebAssembly support in DWARF [10]. This made determining which WebAssembly locals, globals and operands were implicated by a variable quite challenging, as the DWARF location expressions for these gave limited information. This is a problem that will be solved by re-compiling the compiler toolchain to a more recent version.

In the meantime, we made some simplifying assumptions during instrumentation and runtime to achieve a barebones debugger prototype. At each breakpoint, rather than expose the minimal set of machine state for the given probe point, the instrumentation code stores **all** WebAssembly locals and globals into the debug stack. As shown in our evaluation figures, this design introduces only a small build-time overhead (on the order of a few percent across our benchmarks), but the execution-time overhead in debug mode is currently substantial, with instrumented binaries running over an order of magnitude slower in some cases. We believe this overhead can be significantly reduced by emitting more targeted instrumentation that exposes only the information actually needed at each probe site, and by optimizing the representation and transfer of debug state between the instrumented code and the runtime.
We evaluated our prototype on four small C benchmark programs compiled to WebAssembly (recursive Fibonacci, bubble sort, Sieve of Eratosthenes, and matrix multiplication; see Appendix for full source). For each program we measured both build time and end-to-end execution time for the uninstrumented (original) and instrumented (debug) variants, observing modest build-time overhead but substantial slowdowns when running in debug mode. Results in Figure 3 contain 25 samples from a MacBook M4 Pro on Google Chrome 145.0.

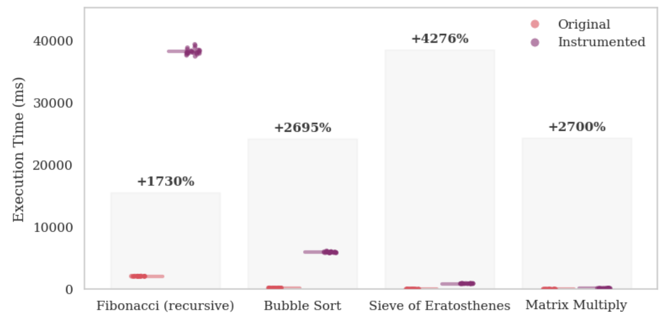
6 Future Work

General-purpose library. The instrumentation and runtime components should be packaged as a portable, compiler-agnostic WebAssembly debugging library, usable by any toolchain that emits DWARF.

LLM code verification. We are interested in exploring the use of this runtime as a local execution sandbox for LLM-generated code. Rather than relying on remote execution environments, a browser-local verifier could allow a model to iteratively test and correct its output, potentially improving correctness at lower infrastructure cost. This direction connects to recent work on LLM self-debugging [11].



(a) Build times comparison between (uninstrumented) and instrumented binaries. Points show individual build runs, short horizontal segments indicate the mean build time for each variant, and the reported mean values together with the gray bands and +X% labels indicate the build-time overhead of instrumentation relative to the original.



(b) Execution times comparison for original (uninstrumented) and instrumented binaries. Data representation is the same as the previous graph

Figure 3: Performance results for build and execution times.

References

[1] T. Jefferson, C. Gregg, and C. Piech. *PyodideU: Unlocking Python Entirely in a Browser for CS1*. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*, ACM, New York, NY, USA, pp. 583–589, 2024. https://doi.org/10.1145/3626252.3630913

[2] J. Roberts-Baca, J. Delgadillo, and C. Piech. *Rooms of Their Own: Structured Small-Group Learning in a Realtime Browser-Based IDE*. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2026)*, Vol. 1, ACM, New York, NY, USA, pp. 943–949, 2026. https://doi.org/10.1145/3770762.3772664

[3] copy.sh. *v86: x86 Emulator in WebAssembly*. https://copy.sh/v86/

[4] Leaning Technologies. *WebVM 2.0: A Complete Linux Desktop Environment in the Browser via WebAssem-*

bly. Leaning Technologies Blog, 2024. https://labs.leaningtech.com/blog/webvm-20

[5] F. Bellard. *QEMU, a Fast and Portable Dynamic Translator*. In *Proceedings of the USENIX Annual Technical Conference (ATEC '05)*, USENIX Association, pp. 41–46, 2005.

[6] V. Shymanskyy. *wasm3: A high performance WebAssembly interpreter*. https://github.com/wasm3/wasm3

[7] I. Stepanyan. *Debugging WebAssembly with modern tools*. Google Chrome Developers Blog, 2020. https://developer.chrome.com/blog/wasm-debugging-2020

[8] A. Rossberg et al. *WebAssembly Specification*, Release 2.0. W3C, 2022. https://webassembly.github.io/spec/

[9] B. Smith, *wasm-clang: Running Clang/LLD in WebAssembly Demo*, GitHub repository, 2019. https://github.com/binji/wasm-clang

[10] Y. Delendik, *DWARF for WebAssembly*, Living Document, 31 August 2020. https://yurydelendik.github.io/webassembly-dwarf/

[11] X. Chen, M. Lin, N. Schärli, and D. Zhou. *Teaching Large Language Models to Self-Debug*. arXiv:2304.05128 [cs.CL], 2023. https://arxiv.org/abs/2304.05128

A Benchmark Programs

We benchmark four small C programs compiled to WebAssembly: recursive Fibonacci, bubble sort, Sieve of Eratosthenes, and dense matrix multiplication.

A.1 Recursive Fibonacci

Listing 1: Recursive Fibonacci benchmark

```
#include <stdio.h>

int fib(int n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}

int main() {
  printf("%d\n", fib(38));
  return 0;
}
```

A.2 Bubble Sort

Listing 2: Bubble sort benchmark

```
#include <stdio.h>
#define N 6000

int main() {
  int a[N];
  for (int i = 0; i < N; i++) a[i] = N - i;
  for (int i = 0; i < N - 1; i++)
    for (int j = 0; j < N - 1 - i; j++)
      if (a[j] > a[j + 1]) {
        int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t
            ;
      }
  printf("%d\n", a[N - 1]);
  return 0;
}
```

A.3 Sieve of Eratosthenes

Listing 3: Sieve of Eratosthenes benchmark

```
#include <stdio.h>
#include <string.h>
#define N 2000000

int main() {
  static char s[N + 1];
  memset(s, 1, sizeof(s));
  s[0] = s[1] = 0;
  for (int i = 2; (long long)i * i <= N; i++)
    if (s[i])
      for (int j = i * i; j <= N; j += i)
        s[j] = 0;
  int count = 0;
  for (int i = 2; i <= N; i++) if (s[i]) count++;
  printf("%d\n", count);
  return 0;
}
```

A.4 Matrix Multiplication

Listing 4: Matrix multiplication benchmark

```
#include <stdio.h>
#define N 80

static double a[N][N], b[N][N], c[N][N];

int main() {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
      a[i][j] = i + j + 1;
      b[i][j] = (i == j) ? 1.0 : 0.0;
    }
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
      c[i][j] = 0.0;
      for (int k = 0; k < N; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
  printf("%.1f\n", c[N - 1][N - 1]);
  return 0;
}
```