Continuing Stack Switching in Wasmtime

WebAssembly Workshop Talk Proposal

FRANK EMRICH, The University of Edinburgh, United Kingdom
DANIEL HILLERSTRÖM, The University of Edinburgh, United Kingdom

1 INTRODUCTION

The WebAssembly (Wasm) stack switching proposal [7] introduces a bespoke instruction set extension that enables a single Wasm instance to efficiently handle multiple execution stacks. Previously producers had to simulate multiple stacks on the Wasm execution stack by performing a whole-program transform (e.g. by using Binaryen's asyncify [9]). The stack switching instruction set is well-suited for single-threaded asynchronous and cooperative concurrent workloads. The design and motivation for the instruction set has been described by Phipps-Costin et al. [8].

Our work continues where Phipps-Costin et al. left off: we have implemented the stack switching proposal in the production-grade off-the-web engine Wasmtime [1]. The objectives of the proposed talk is to 1) report on the experience of retrofitting a production-grade engine with stack switching capabilities; 2) describe our implementation approach, i.e. how we evolved a reasonably simple prototype, which used host facilities to switch stacks, into an implementation where the stack switching is fully native; and 3) share our insights on the technical intricacies of maintaining compatibility with standard tools (e.g. gdb and perf) as well as producing useful backtraces. In the remainder of this note we briefly elaborate on some of these objectives.

2 PIGGYBACKING ON WASMTIME FIBER

Wasmtime already had two key facilities that enables us to quickly prototype an implementation of the proposal: 1) Wasmtime Fiber [2], and 2) a library call mechanism. The former facility is an application programming interface (API) which allows Wasm functions to be run on separate execution stacks, thus providing some form of stack switching already. However, Wasmtime Fiber is only available to embedders (i.e. programs which deeply integrates the Wasmtime engine in their source), thus a core Wasm program cannot take advantage of it. The latter facility enables compiled Wasm code to perform a library call (libcall) into the host environment, allowing arbitrary computation to be performed by the host before returning back to the Wasm code. By composing these two facilities we were able to build a prototype by interpreting each instruction in the stack switching proposal as a libcall to a corresponding API call to Wasmtime Fiber.

Enabled us to build a fiber library in C, which can be compiled to the stack switching instruction set or transformed into a giant state machine via Asyncify [5]. On top of this library we have built various benchmarks and workloads, including a webserver [6].

3 SYMMETRIC ASYMMETRY

While the libcall-approach gave us a quick way to experiment with compiling and evaluating workloads on the stack switching proposal, it was clear that to satisfactorily demonstrate the implementability of the proposal, we eventually had to go straight to native code.

Wasmtime utilises Cranelift, a native code generation framework, as its backend compiler. Cranelift had no stack switching facilities, thus we first had to extend the Cranelift intermediate format (CLIF) with a dedicated instruction for stack switching. We opted for a design based on the

Authors' addresses: Frank Emrich, The University of Edinburgh, United Kingdom, frank.emrich@ed.ac.uk; Daniel Hillerström, The University of Edinburgh, United Kingdom, daniel.hillerstrom@ed.ac.uk.

`SWAPSTACKS` primitive by Dolan et al. [4]. It is a minimal machine-independent extension to CLIF, which is lowerable to machine-specific instructions.

An interesting facet of this design is that the primitive stack switching instruction provides *symmetric* switching semantics, meaning there is no caller-callee relationship between stacks [3]. In contrast, the stack switching proposal offers *asymmetric* switching, where resuming a stack establishes a caller-callee relationship between the resumer and resumee.

We were able to smoothly transition our libcall-implementation to a native implementation by stripping down the implementation of the libcalls to only perform the actual stack switching, shifting all administrative logic to generated code (e.g. maintenance of the caller-callee relationships). During this process, we also gradually adapted the libcall-implementation to use a memory layout for stack metadata as close as possible to the one expected by the CLIF instruction.

The transition to native stack switching ensures that execution stays within the realm of Wasm — we do not repeatedly enter and leave the host environment in order to perform stack switching. We observed up to 6x performance improvements in micro-benchmarks. We analysed this performance improvement, and found that it was due to a peculiar interaction between stack switching, function calls, and the branch predictor of the central processing unit (CPU) on function returns. CPUs expect that a return instruction following a call instruction will return to the address of this call instruction. However, our libcall-based implementation made the branch predictor constantly mispredict, essentially because stack switching via a libcall involves a sequence of nested function calls, where the innermost function performs the actual switch. Consequently, the branch predictor would mispredict whenever code returned from the libcall-related frames on the resumee stack.

ACKNOWLEDGMENTS

This work was supported by UKRI Future Leaders Fellowship “Effect Handler Oriented Programming” (reference number MR/T043830/1).

REFERENCES

[1] Bytecode Alliance. 2024. Wasmtime: A fast and secure runtime for WebAssembly. https://wasmtime.dev/. Accessed 2024-10-30.

[2] Alex Crichton. 2021. Wasmtime Fiber API. https://docs.wasmtime.dev/api/wasmtime_fiber/index.html. Accessed 2023-04-14.

[3] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2 (2009), 6:1–6:31. https://doi.org/10.1145/1462166.1462167

[4] Stephen Dolan, Servesh Muralidharan, and David Gregg. 2013. Compiler support for lightweight context switching. *ACM Trans. Archit. Code Optim.* 9, 4 (2013), 36:1–36:25. https://doi.org/10.1145/2400682.2400695

[5] Frank Emrich and Daniel Hillerström. 2024. A C fiber library compatible with Wasm. https://github.com/wasmfx/fiber-c Accessed 2024-10-24.

[6] Frank Emrich and Daniel Hillerström. 2024. Waeio: WebAssembly Effect-based I/O. https://github.com/wasmfx/waeio Accessed 2024-10-24.

[7] Frank Emrich, Daniel Hillerström, Sam Lindley, Thomas Lively, Francis McCabe, Andreas Rossberg, and Conrad Watt. [n. d.]. Stack Switching Proposal for WebAssembly. https://github.com/WebAssembly/stack-switching Accessed 2024-10-30.

[8] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 460–485. https://doi.org/10.1145/3622814

[9] Alon Zakai. 2019. Pause and Resume WebAssembly with Binaryen’s Asyncify. https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html Accessed 2022-10-27.