

Supporting Stack Switching in Wasm

Daniel Hillerström

Computing Systems Laboratory
Zurich Research Center
Huawei Technologies, Switzerland
and
The University of Edinburgh, UK

October 13, 2023

Wasm Research Day 2023

Collaborators



Sam Lindley



Andreas Rossberg



Daan Leijen



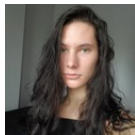
KC Sivaramakrishnan



Matija Pretnar



Frank Emrich



Luna Phipps-Costin



Arjun Guha

<https://wasmfx.dev>

Collaborators



Sam Lindley



Andreas Rossberg



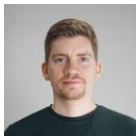
Daan Leijen



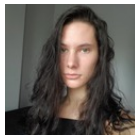
KC Sivaramakrishnan



Matija Pretnar



Frank Emrich



Luna Phipps-Costin



Arjun Guha

<https://wasmfx.dev>

This is a continuation of my talk last year

Continuing WebAssembly with Effect Handlers

LUNA PHIPPS-COSTIN, Northeastern University, USA

ANDREAS ROSSBERG, Unaffiliated, Germany

ARJUN GUHA, Northeastern University, USA

DAAN LEIJEN, Microsoft Research, USA

DANIEL HILLERSTRÖM, Huawei Zurich Research Center, Switzerland

KC SIVARAMAKRISHNAN, Tarides and IIT Madras, India

MATIJA PRETNAR, Ljubljana University, Slovenia

SAM LINDLEY, The University of Edinburgh, United Kingdom

WebAssembly (Wasm) is a low-level portable code format offering near native performance. It is intended as a compilation target for a wide variety of source languages. However, Wasm provides no direct support for non-local control flow features such as `async/await`, `generators/iterators`, `lightweight threads`, `first-class continuations`, etc. This means that compilers for source languages with such features must ceremoniously transform whole source programs in order to target Wasm.

We present *WasmFX*, an extension to Wasm which provides a universal target for non-local control features via *effect handlers*, enabling compilers to translate such features directly into Wasm. Our extension is minimal and only adds three main instructions for creating, suspending, and resuming continuations. Moreover, our primitive instructions are type-safe providing typed continuations which are well-aligned with the design principles of Wasm whose stacks are typed.

Non-local control is a staple ingredient of many programming languages



OCaml



...

The problem

How do I compile non-local control flow abstractions to Wasm?

The solution

A bespoke instruction set

The WasmFX instruction set extension

Types

- **cont** $\$ft$

Tags

- **tag** $\$tag$ (**param** σ^*) (**result** τ^*)

Core instructions

- **cont.new** $\$ct$
- **resume** $\$ct$ ($tag \$t \h)*
- **suspend** $\$tag$

Other instructions

- **cont.bind** $\$ct \ct'
- **resume_throw** $\$ct \tag ($tag \$t \h)*
- **barrier**




We call this instruction set extension **WasmFX**.

The WasmFX instruction set extension










Types

- **cont** $\$ft$   








Tags

- **tag** $\$tag$ (**param** σ^*) (**result** τ^*)   




Core instructions

- **cont.new** $\$ct$   
- **resume** $\$ct$ ($tag \$t \h)*   
- **suspend** $\$tag$   

Other instructions

- **cont.bind** $\$ct \ct'   
- **resume_throw** $\$ct \tag ($tag \$t \h)*  
- **barrier**  

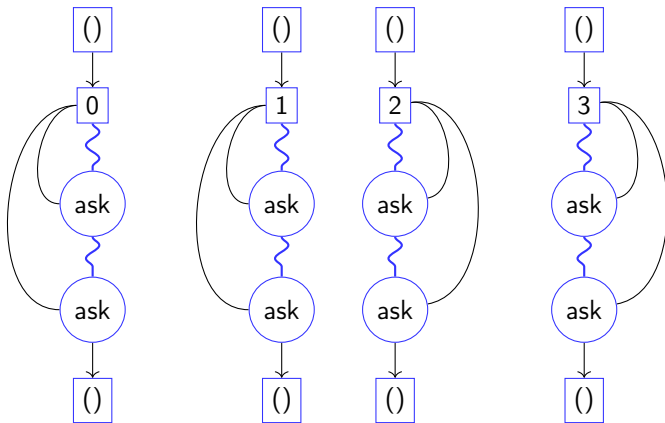
Legend

-  Spec'ed
-  Reference impl.
-  Wasmtime impl.

We call this instruction set extension **WasmFX**.

Demo: dynamic binding

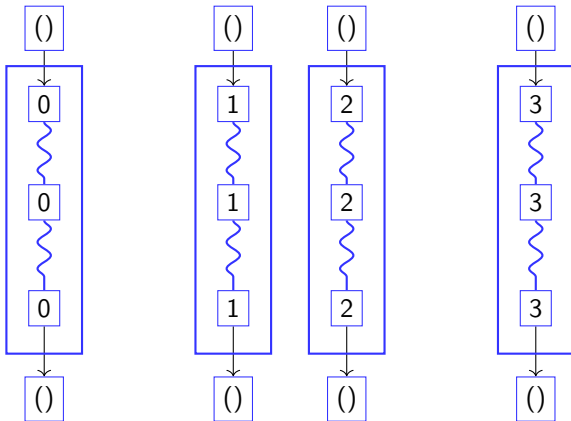
The environment handler provides a context-dependent variable.



Relevant code: `dynbind.wast`

Demo: lightweight threads

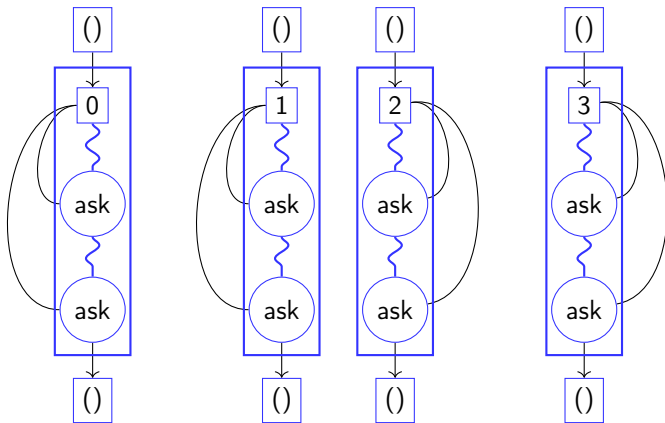
The scheduler handler reifies each task computation as a first-class object.



Relevant code: `lwt.wast`

Demo: task-local state

We obtain task-local state by composing the scheduler and environment handlers.



Relevant code: `lwt-dynbind.wast`

The alternative: Asyncify

Asyncify

- A whole-program approach
- Source-to-source transformation
- Transforms the program into a state machine

A 'small' example

```
(func $doSomething (param $arg i32) (result i32)
  (call $foo
    (call $bar (local.get $arg))))
```

The alternative: Asyncify

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))           ;; test rewind state
    (then (local.set $arg                                           ;; store local $arg
      (i32.load offset=4 (global.get $asyncify_heap_ptr))
      (local.set $call_idx                                         ;; continuation point
        (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
            (then (br $call_bar))                                     ;; restore $call_bar
            (else (br $restore_foo))))
          (else (br $call_bar)))                                     ;; regular $call_bar
        (local.set $ret (call $bar (local.get 0)))
        (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32) ;; test unwind state
          (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
            (i32.store offset=8 (global.get $asyncify_heap_ptr) (i32.const 0))
            (return (i32.const 0))) ...))))))
```

The alternative: Asyncify

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))
    (then (local.set $arg
      (i32.load offset=4 (global.get $asyncify_heap_ptr))
      (local.set $call_idx
        (i32.load offset=8 (global.get $asyncify_heap_ptr))
      (else))
    (block $call_foo (result i32)
      (block $restore_foo (result i32)
        (block $call_bar (result i32)
          (local.get $arg)
          (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
            (then (if (i32.eq (local.get $call_idx) (i32.const 0))
              (then (br $call_bar))           ;; restore $call_bar
              (else (br $restore_foo))))
            (else (br $call_bar)))           ;; regular $call_bar
          (local.set $ret (call $bar (local.get 0)))
          (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32) ;; test unwind state
            (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
              (i32.store offset=8 (global.get $asyncify_heap_ptr) (i32.const 0))
              (return (i32.const 0))) ...))))))
```

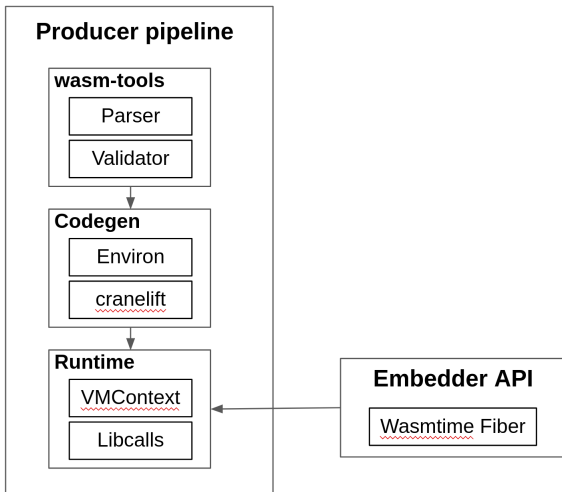
The alternative: Asyncify

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))           ;; test rewind state
    (then (local.set $arg                                           ;; store local $arg
      (i32.load offset=4 (global.get $asyncify_heap_ptr))
      (local.set $call_idx                                          ;; continuation point
        (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
            (then (br $call_bar))                                     ;; restore $call_bar
            (else (br $restore_foo))))
          (else (br $call_bar))))                                     ;; regular $call_bar
        (local.set $ret (call $bar (local.get 0)))
        (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32) ;; test unwind state
          (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
            (i32.store offset=8 (global.get $asyncify_heap_ptr) (i32.const 0))
            (return (i32.const 0))) ...))))))
```

The alternative: Asyncify

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))           ;; test rewind state
    (then (local.set $arg                                           ;; store local $arg
      (i32.load offset=4 (global.get $asyncify_heap_ptr))
      (local.set $call_idx                                          ;; continuation point
        (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
            (then (br $call_bar))                                     ;; restore $call_bar
            (else (br $restore_foo))))
          (else (br $call_bar))))                                     ;; regular $call_bar
        (local.set $ret (call $bar (local.get 0)))
        (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32) ;; test unwind state
          (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
            (i32.store offset=8 (global.get $asyncify_heap_ptr) (i32.const 0))
            (return (i32.const 0))) ...))))))
```


Wasmtime overview



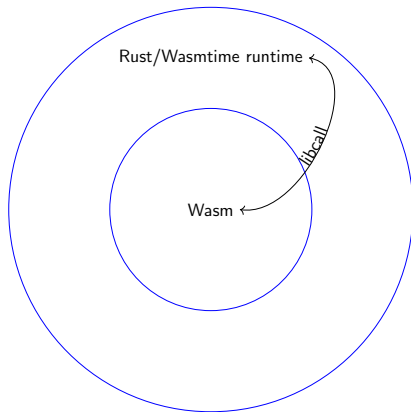
WasmFXtime, a baseline implementation of WasmFX

Prototype implementation in Wasmtime

- Naïve baseline implementation
- Enables running “real” programs
- Stack switching on top of Wasmtime Fiber

Key naïve implementation decisions

- Stack switching is non-Wasm native
- Use u128 as the universal type
- Reallocate argument buffers on each context switch



Experiments setup

Setup overview

- Common fiber interface in C; instantiated with either Asyncify or WasmFX
- A hand-written bespoke implementation

Tools

- WASI SDK version 20.0
- Binaryen version 116

Apples & oranges

- Bespoke and Asyncify implementations are optimised
 - `clang -O3`
 - `wasm-opt -O2 --asyncify --pass-arg=asyncify-ignore-imports`
- WasmFX implementation is unoptimised and linked by hand
- Different storage
 - Asyncify-backed fibers in linear memory
 - WasmFX-backed fibers in tables
- Tools do not understand function references

Experiments setup: Fiber interface in C

```
/** The signature of a fiber entry point. */
typedef void* (*fiber_entry_point_t)(void*);
/** The abstract type of a fiber object. */
typedef struct fiber_t;

/** Allocates a new fiber with the default stack size. */
fiber_t fiber_alloc(fiber_entry_point_t entry);
/** Reclaims the memory occupied by a fiber object. */
void fiber_free(fiber_t fiber);

/** Yields control to its parent context. This function must be called  
    from within a fiber context. */
void* fiber_yield(void *arg);

/** Possible status codes for 'fiber_resume'. */
typedef enum { FIBER_OK, FIBER_YIELD, FIBER_ERROR } fiber_result_t;

/** Resumes a given 'fiber' with argument 'arg'. */
void* fiber_resume(fiber_t fiber, void *arg, fiber_result_t *result);
```

Microbenchmark: C10m

Description

- HTTP server workload simulation.
- 10 million coroutines in total.
- Sliding window: 10000 coroutines run concurrently, each yielding once.
- Shallow call stack depth

No I/O

	Run-time ratio	Memory footprint ratio	Binary size ratio
Asyncify	1.00	1.00 (54mb)	1.00 (9.1kb)
WasmFX	4.00	1.02 (55mb)	0.09 (844b)

(lower is better)

Microbenchmark: C10m

Description

- HTTP server workload simulation.
- 10 million coroutines in total.
- Sliding window: 10000 coroutines run concurrently, each yielding once.
- Shallow call stack depth

One I/O call

	Run-time ratio	Memory footprint ratio	Binary size ratio
Asyncify	1.00	1.00 (54mb)	1.00 (9.2kb)
WasmFX	1.90	1.02 (55mb)	0.10 (921b)

(lower is better)

Microbenchmark: C10m

Description

- HTTP server workload simulation.
- 10 million coroutines in total.
- Sliding window: 10000 coroutines run concurrently, each yielding once.
- Shallow call stack depth

Repeated I/O calls

	Run-time ratio	Memory footprint ratio	Binary size ratio
Asyncify	1.00	1.00 (54mb)	1.00 (9.2kb)
WasmFX	1.14	1.02 (55mb)	0.10 (921b)

(lower is better)

Microbenchmark: C10m

Description

- HTTP server workload simulation.
- 10 million coroutines in total.
- Sliding window: 10000 coroutines run concurrently, each yielding once.
- Shallow call stack depth

Repeated I/O calls

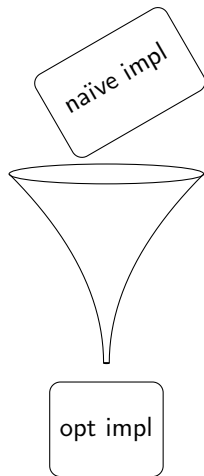
	Run-time ratio	Memory footprint ratio	Binary size ratio
Bespoke	1.00	1.00 (13mb)	1.00 (886b)
Asyncify	1.07	0.24 (54mb)	10.16 (9.0kb)
WasmFX	1.23	0.24 (55mb)	0.89 (789b)

(lower is better)

Distilling the implementation

Laundry list

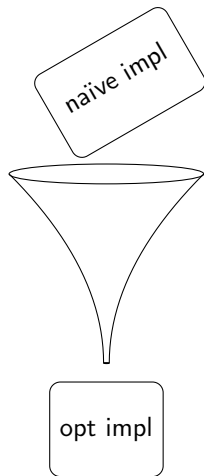
- Eliminate redundant libcalls
- Eliminate redundant allocations
- Improve data representation of continuations
- Utilise the types to generate tightly typed code
- Inline the Wasmtime Fiber logic
- Delayed continuation allocation
- Continuation pooling



Distilling the implementation

Laundry list

- Eliminate redundant libcalls (10%-ish perf)
- Eliminate redundant allocations
- Improve data representation of continuations
- Utilise the types to generate tightly typed code
- Inline the Wasmtime Fiber logic
- Delayed continuation allocation
- Continuation pooling



Continuation data representation (1)

```
#[repr(C)]
pub struct ContinuationObject {
    parent: *mut ContinuationObject, // 8 bytes
    fiber: *mut ContinuationFiber,    // 8 bytes
    args: Payloads,                   // 24 bytes
    suspend_args: Payloads,           // 24 bytes
    state: State,                     // 4 bytes
}                                     // Total: 68 bytes

struct Payloads {
    length: usize,
    capacity: usize,
    data: *mut u128, // null iff length == capacity == 0
}
```

Opportunities for denser packing

- The state field is redundant.
- Size of args is statically known.
- With other optimisations having both args and suspend_args is redundant.

Continuation data representation (2)

```
type ContinuationFiber = Fiber<'static, (), u32, ()>;

pub struct Fiber<'a, Resume, Yield, Return> {
    stack: FiberStack,
    inner: imp::Fiber,
    done: Cell<bool>,
    _phantom: PhantomData<&'a (Resume, Yield, Return)>,
}

pub struct FiberStack {
    top: *mut u8,
    len: usize,
    mmap: bool,
}
```

More opportunities

- Point directly to the FiberStack.
- Better yet, inline FiberStack directly in the ContinuationObject.
- More wants more: trim the Fiber API down to the bare minimum.

Internalise the logic of Wasmtime Fiber

```
// Save callee registers
push rbp
push rbx
push r12
push r13
push r14
push r15
// Load resume point and save our current sp
mov rax, -0x10[rdi]
mov -0x10[rdi], rsp
// Swap stacks and restore saved registers
mov rsp, rax
pop r15
pop r14
pop r13
pop r12
pop rbx
pop rbp
ret
```

Opportunities

- Emit the stack switching logic directly.
- Cranelift-native stack switching mechanism.
- Transfer payloads in registers.

Memory optimisations

Delayed continuation stack allocation

- Defer allocation until first resume
- Hypothesised significant memory savings in certain scenarios

Continuation stack pooling

- Amortise allocation and deallocation
- Hypothesised significant runtime performance boost in tight loops

Growable stacks

- Reduce memory footprint
- Experiment with different strategies (split stacks, gstacks, simple realloc)

WasmFX resource list

Resources

- Formal specification
(<https://github.com/wasmfx/specfx/blob/main/proposals/continuations/Overview.md>)
- Informal explainer document
(<https://github.com/wasmfx/specfx/blob/main/proposals/continuations/Explainer.md>)
- Reference implementation (<https://github.com/wasmfx/specfx>)
- Research prototype implementation in Wasmtime (<https://github.com/wasmfx/wasmfxtime>)
- Toolchain support (<https://github.com/wasmfx/binaryenfx>)
- OOPSLA'23 research paper (<https://doi.org/10.48550/arXiv.2308.08347>)

<https://wasmfx.dev>

References

Phipps-Costin, Luna et al. (2023). “Continuing WebAssembly with Effect Handlers”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2. To appear.