Status update: Stack switching in Wasmtime

Frank Emrich
University of Edinburgh

Stack Switching Subgroup meeting
4 December 2023

Overview

→ Working implementation of typed continuations/WasmFX approach in fork of Wasmtime

→ Current limitations

- Not implemented `resume.throw` yet (waiting for EH support in Wasmtime)
- Not implemented `barrier` yet (easy to implement once needed)
- No support for growing stacks or detecting stack overflow

→ Topics today:

- Feature work: Plans to overcome these limitations/refine some other aspects
- Optimisation work: Finished and planned performance optimisations

Feature work

Growing stacks/Preventing stack overflow

→ Current behaviour: Continuations created with fixed amount of stack space, exceeding causes unmitigated disaster ϟ

→ Plan: Investigate two different solutions

- Add stack checks to function preludes, trigger resize if needed
  - Infrastructure in place in Wasmtime
  - Downside: Affects code never performing stack switching
  - Resizing approach: Segmented stacks or copying stack to larger allocation (OCaml approach, need to ensure no pointers into stack)
- `mmap` large amounts of stack memory, committed only on first use, guard page at bottom of stack
  - Platform-specific implementations
  - Approach taken by `libmprompt`
  - Downside: Potentially makes allocation slower ⇒ Use stack pools?

Deallocation of continuations

```
(func $leak
  (cont.new $ct (ref.func $myfunc))
  (drop)
  ;; continuation object (stack memory, etc) leaked here
)
```

→ Should call `resume.throw` on continuations not run to completion

→ Current behaviour: Only deallocate continuation's memory when computation returns

→ One solution to avoid memory leakage: Use refcounting to determine when continuations become unreachable

→ Requires GC/refcounting infrastructure in Wasmtime

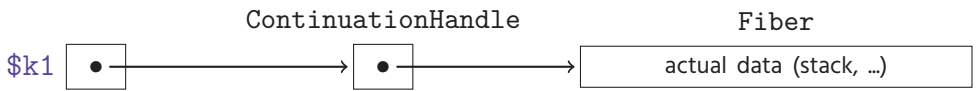
Detecting reuse of continuations: Current approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

Detecting reuse of continuations: Current approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

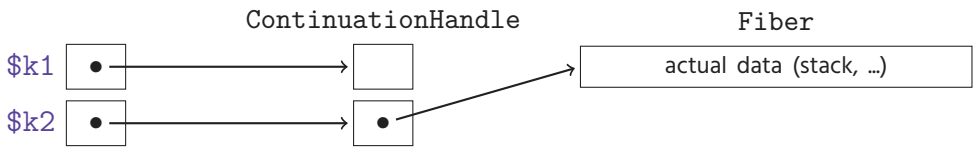
Continuation values are pointers to pointers, `null`-ed on use



Detecting reuse of continuations: Current approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

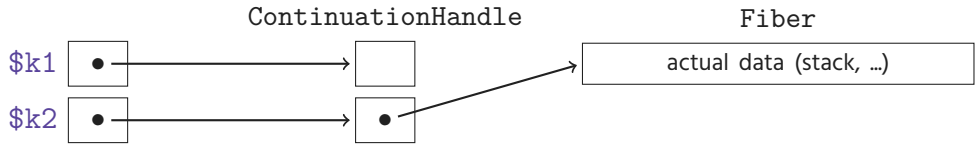
Continuation values are pointers to pointers, `null`-ed on use



Detecting reuse of continuations: Current approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

Continuation values are pointers to pointers, `null`-ed on use



Downsides:

- Additional allocations per operation returning continuation
- Now we also need to ensure deallocation of the `ContinuationHandle`!

Detecting reuse of continuations: Unsafe approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

Detecting reuse of continuations: Unsafe approach

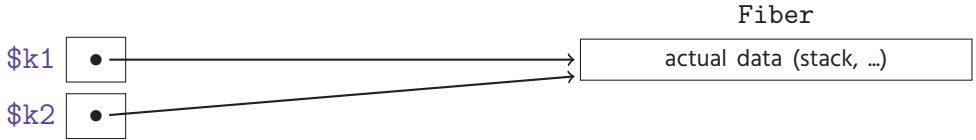
```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

Unsafe: Continuation values are pointers to `Fiber` objects

Detecting reuse of continuations: Unsafe approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

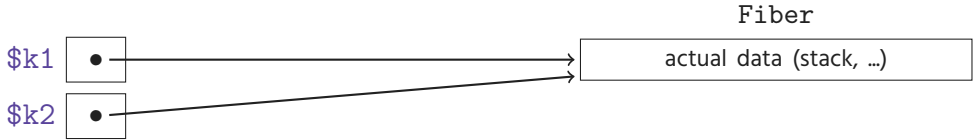
Unsafe: Continuation values are pointers to `Fiber` objects



Detecting reuse of continuations: Unsafe approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

Unsafe: Continuation values are pointers to `Fiber` objects



This behaviour can be enabled with flag

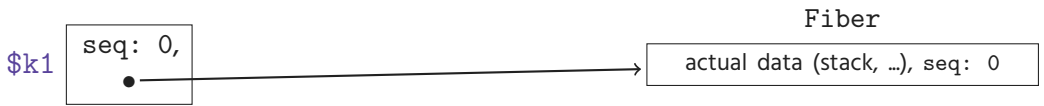
Detecting reuse of continuations: Planned approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

Detecting reuse of continuations: Planned approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

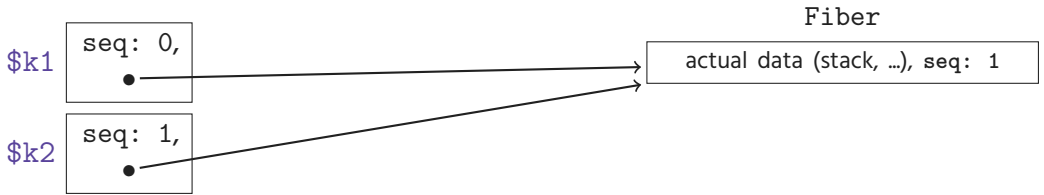
Continuation values are fat pointers: (sequence number, `*Fiber`)



Detecting reuse of continuations: Planned approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

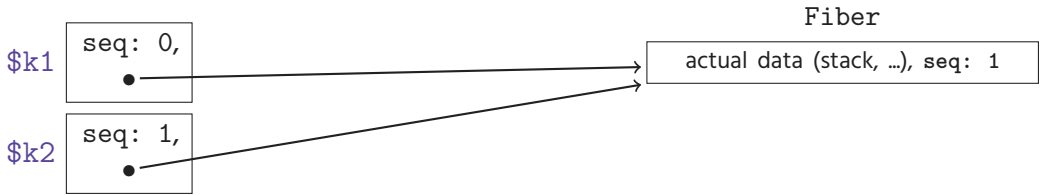
Continuation values are fat pointers: (sequence number, `*Fiber`)



Detecting reuse of continuations: Planned approach

```
...
(local.set $k1 (cont.new (ref.func $g)))
(block $handler (result (ref $ct2))
  (resume $ct1 (tag $mytag) (local.get $k1)))
  (return ...)
)
(local.set $k2) ;; $k2 : (ref $ct2) usable, $k1 : (ref $ct1) is not
```

Continuation values are fat pointers: (sequence number, `*Fiber`)



On continuation use: Compare fat pointer's `seq` with `Fiber`'s, increment latter

Optimisation work

Current implementation approach

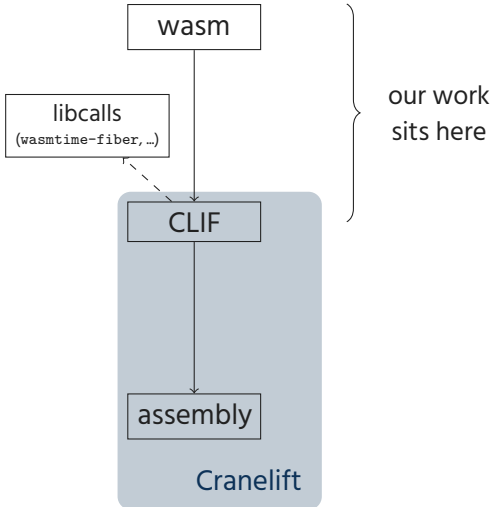
→ We act at level of wasm → Cranelift intermediate format (CLIF) translation

→ Cranelift remains unchanged

→ Escape hatch: Libcalls allow executing arbitrary Rust code

→ We added new libcalls to …

- perform actual stack switching using `wasmtime-fiber`
- perform allocation
- simplify implementation work



Current implementation approach

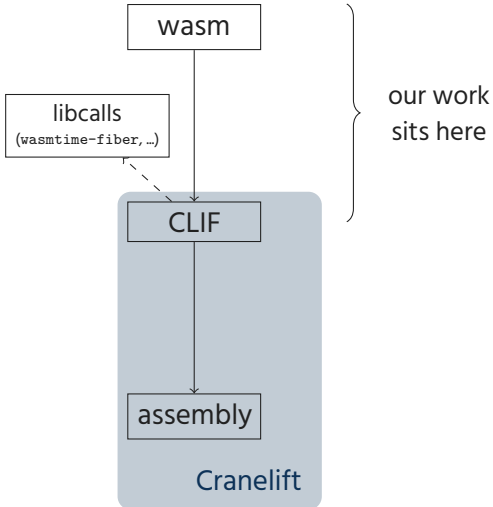
→ We act at level of wasm → Cranelift intermediate format (CLIF) translation

→ Cranelift remains unchanged

→ Escape hatch: Libcalls allow executing arbitrary Rust code

→ We added new libcalls to …

- perform actual stack switching using `wasmtime-fiber`
- perform allocation
- ~~simplify implementation work~~



Libcall infrastructure

Multiple layers of indirection for each libcall. Invoking `foo` libcalls involves:

→ From jitted code: Call `libcalls::trampolines::foo`
(4 instructions of macro-generated trampoline code, storing PC and FP)

→ From there: `jmp` to `libcalls::trampolines::impl_foo`

```
let result = std::panic::catch_unwind(
    std::panic::AssertUnwindSafe(|| {
        ... libcalls::foo(...) ...
}));
match result {
    Ok(ret) => LibcallResult::convert(ret),
    Err(panic) => crate::traphandlers::resume_panic(panic)
}
```

→ From there: Call `libcalls::foo` (actual implementation of `foo`)

wasmtime-fiber

→ Standalone Rust implementation of general-purpose stack switching

→ Developed as part of Wasmtime, but independent from it

→ Key part: Function `wasmtime_fiber_switch` (handwritten assembly) allows switching between stacks

- Push all callee-save registers on current stack
- Set current `SP` aside
- Obtain new stack pointer and set `SP` to it
- Restore callee-save registers from (now switched) stack
- Return

→ Start of execution: `wasmtime_fiber_switch` into new, carefully prepared stack, proceed into actual function to run through 2 trampolines

Payload handling

→ When starting, suspending, resuming continuations we can pass arbitrary payload data

```
...
(resume $ct (local.get $myarg) (cont.new (ref.func $f)))
...
(suspend $mytag (i32.const 123) (i32.const 456))
...
```

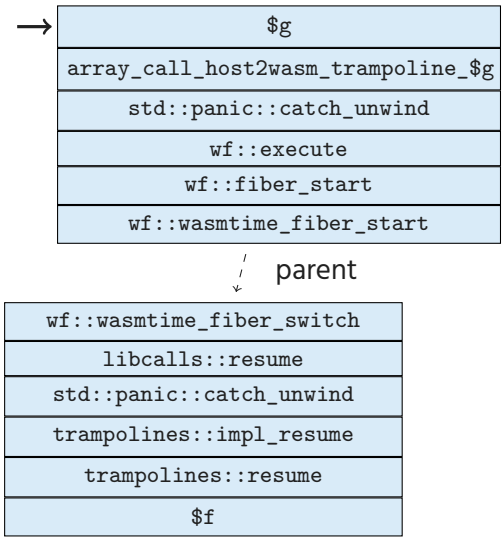
→ Current, naive approach: All payload data passed via heap-allocated buffers

→ For each wasm function `$f`: Wasmtime provides “array call” trampoline

- Consistent signature, independent from `$f`’s: Takes buffer and length
- Reads arguments from buffer and calls `$f`

Overhead galore

```
(func $f (resume $ct (cont.new (ref.func $g))))
```



→ Disclaimer: Logical, slightly edited view. Some of these are tail calls, don't actually occupy stack space, may be inlined, etc

→ This is not criticising Wasmtime at all: All these components (libcall infrastructure, `wasmtime-fiber`, array call mechanism) are well engineered!

→ Many of these components are more general than what we need

Optimisation roadmap

→ Starting position: Architectural decisions based on need to get research prototype built by small team

→ Short-term: Squeeze more performance out of current approach (use libcalls + `wasmtime-fiber`, but customise further)

→ Medium-term: Gradually switch towards internalising stack switching into Cranelift

→ Approach: Incremental improvement instead of big leap

Finished/In-progress optimisations

Done

→ Stop using libcalls for purposes other than stack switching or allocation
- Translated Rust code to CLIF

→ Optimised layout of data structures (inline data, remove unused fields)

→ Re-use allocated payload buffers when possible

In progress

→ Stop using mechanism provided by `wasmtime-fiber` to pass payloads altogether
- Currently only used to pass info about return vs suspend-with-tag from `suspend` to handler
- Transferred via heap indirection
- Now: passed through register argument of `wasmtime_fiber_switch`

Planned optimisations (short-term)

Libcalls

- Call into Rust code (`wasmtime-fiber`, allocation) more efficiently
- Ideally want to emit direct call to `wasmtime_fiber_switch`
- … let's see how that goes (panics, backtraces, etc)

wasmtime-fiber

- Specialise to our needs (no need to be able to invoke arbitrary closures, …)

Payload passing

- When possible, pass all payloads through arguments/return values of `wasmtime_fiber_switch`, otherwise fall back to using buffers

Memory management

- Where payload buffers still needed, stack-allocate whenever possible
- Pool stack memory allocations (meaningful impact in OCaml!)
- (De-)Allocate without needing libcalls

Benchmark results

Setup

- x64 Linux (AMD Ryzen 3900X)
- WASI SDK 20

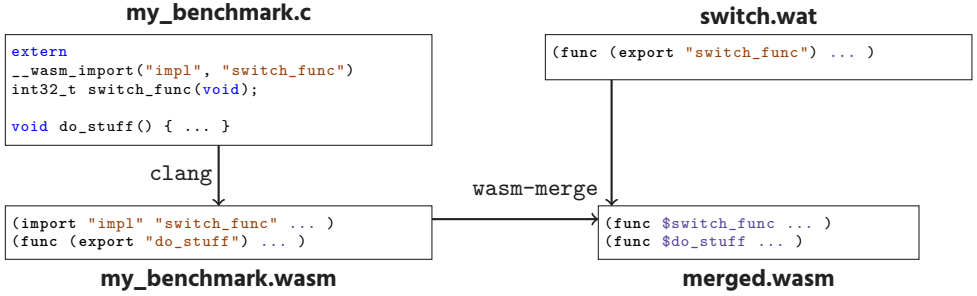
Relative performance improvement

	Benchmark			
	`c10m`	`sieve`	`skynet`	`state`
WasmFX @ September	1.00	1.00	1.00	1.00
WasmFX @ Now	0.81	0.64	0.82	0.64

Other ongoing efforts

Binaryen support

- Implemented basic support for WasmFX instructions in Binaryen
- Pleasantly accessible code base!
- Main motivation: `wasm-merge`. Link generated and hand-written wasm into single module for benchmarking



- No particular focus on `wasm-opt` optimisations for now
- Currently being upstreamed

Other ongoing efforts (cont'd)

Benchmarking

- Ongoing work to create additional benchmarks
- Using C + handwritten `.wat` approach using `wasm-merge`
- Notable example: Webserver

TinyGo

- Existing (subset of) Go $\Rightarrow$ wasm compiler
- Goroutines currently handled using `asyncify`
- Forked to emit WasmFX instructions instead

WasmFX resource list

- → 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)
- → OOPSLA'23 research paper (https://doi.org/10.48550/arXiv.2308.08347)

https://wasmfx.dev