Continuing Stack Switching in Wasmtime

Frank Emrich, Daniel Hillerström

The University of Edinburgh

WebAssembly Workshop 2025

Stack Switching in WebAssembly

Stack Switching subgroup working on non-local control flow for Wasm

- Enable various source language features:
- async/await, coroutines, lightweight threads, generators, first-class continuations, ...

Proposal

- Based on Plotkin and Pretnar's handlers for algebraic effects: **Asymmetric** stack switching
- OOPSLA 2023: "Continuing WebAssembly with Effect Handlers"
- Additional `switch` instruction to optimise performance of **symmetric** stack switching
- Advanced to stage 2 in August 2024

Implementations

- Reference interpreter
- Wasmtime (industrial strength, standalone Wasm engine), currently being upstreamed

Instruction Set

Module-level definitions

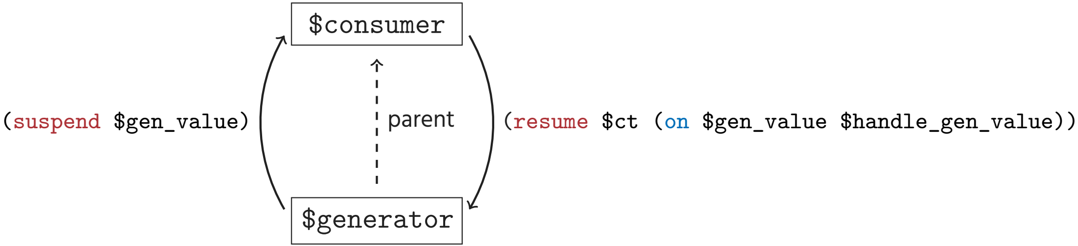
- Tags denote delimeters/effects
  `(tag $yield (param i32) (result i32))`
- New heap type/structural type for continuation
  `(type $ct (cont $ft)))`

Core instructions

- Create continuation from function reference
  `(cont.new $ct)`
- Perform effect/suspend to handler for given tag
  `(suspend $yield)`
- Resume a continuation, install handlers for tags/effects
  `(resume $ct (on $yield $handler_block))`
- Switch directly to a target continuation instead of suspending to parent
  `(switch $ct $yield)`

Asymmetric Switching: Example

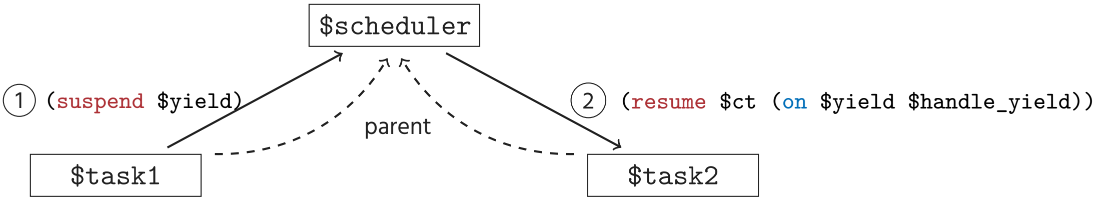
Simple use case for asymmetric switching: Two functions `$consumer` and `$generator`.



Symmetric Switching: Motivation

Use case: Switching between different tasks/coroutines/lightweight threads. Here: `$task1` and `$task2`.

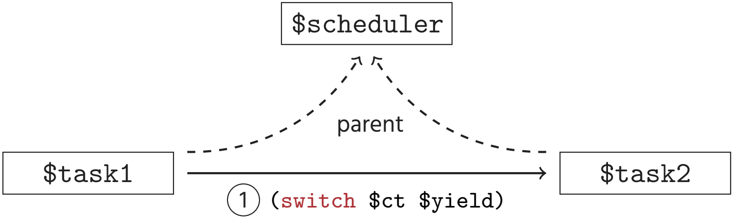
Asymmetric implementation



Observation: Going from `$task1` to `$task2` requires two stack switches

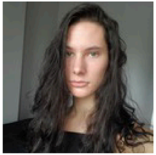
Symmetric Switching: Example

Symmetric implementation of previous example



The Challenge

How to implement a complex feature in an industrial-strength Wasm engine with limited resources?



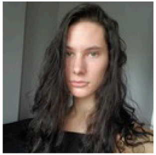
Luna Phipps-Costin

Daniel Hillerström

Frank Emrich

The Challenge

How to implement a complex feature in an industrial-strength Wasm engine with limited resources?



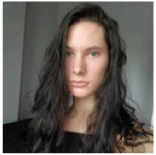
Luna Phipps-Costin

Daniel Hillerström

Frank Emrich

The Challenge

How to implement a complex feature in an industrial-strength Wasm engine with limited resources?



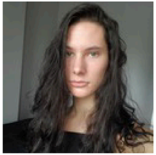
Luna Phipps-Costin

Daniel Hillerström

Frank Emrich

The Challenge

How to implement a complex feature in an industrial-strength Wasm engine with limited resources?



Luna Phipps-Costin

Daniel Hillerström

Frank Emrich

1. Create inefficient, but easy to implement prototype
2. Sketch design of optimised implementation
3. Incremental changes towards optimised implementation: No big bang
4. Arrive at optimised implementation!

Design of Prototype Implementation

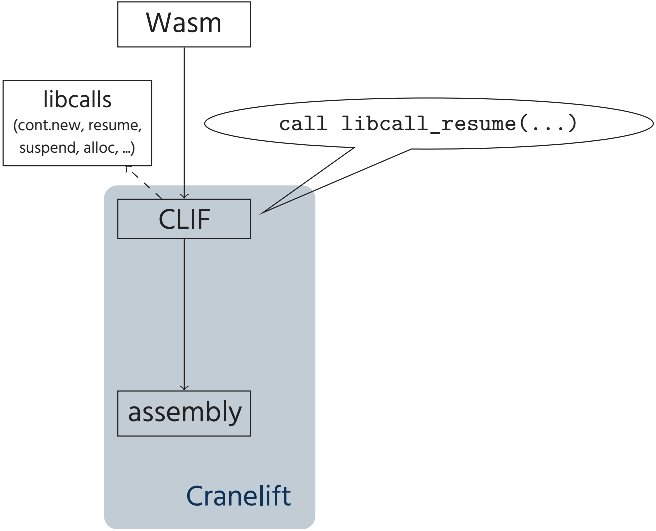
→ Prototype implemented at level of Wasm → Cranelift intermediate format (CLIF) translation

→ Cranelift remains unchanged

→ Escape hatch: Libcalls allow executing arbitrary Rust code

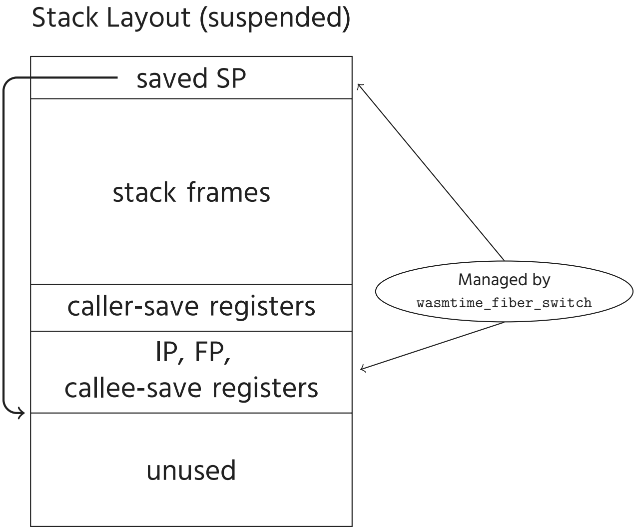
→ Relied on new libcalls to ...

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



wasmtime-fiber

- General purpose library for (asymmetric) stack switching, developed as part of Wasmtime
- Used to implement Wasmtime's `async` feature
- At its heart: Hand-written assembly function `wasmtime_fiber_switch` that stores registers, updates stack pointer, etc



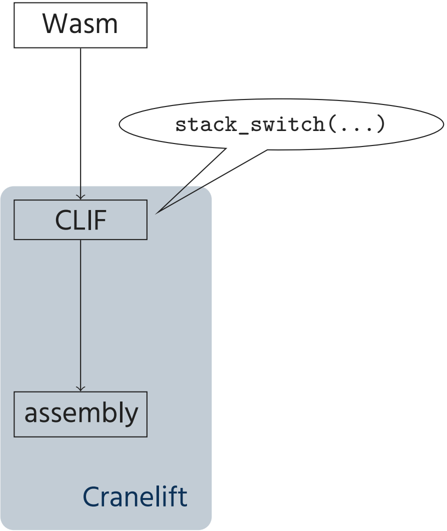
Design of Final Implementation

→ Goal: Perform actual stack switching using Cranelift-generated code

→ Only single new libcall left (`cont.new` needs support from runtime)

→ Solution: New CLIF instruction `stack_switch`

- Minimal addition to Cranelift: Only does what cannot be expressed already
- Platform-independent



stack_switch Instruction

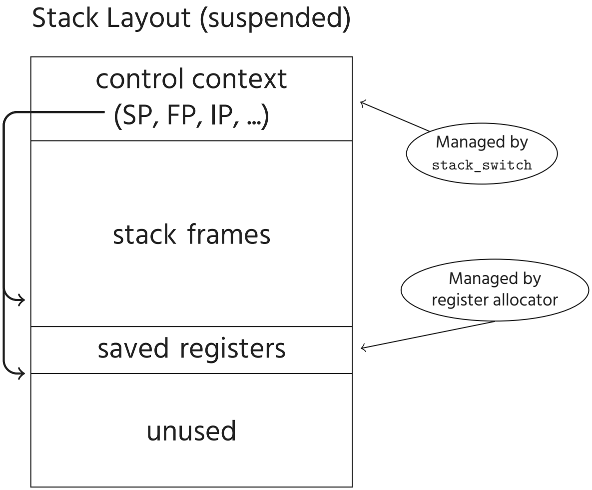
→ Instruction acts on pointers to **control contexts**

```
stack_switch(
  source_control_ctx,
  dest_control_ctx,
  payload
)
```

→ Layout and contents platform dependent

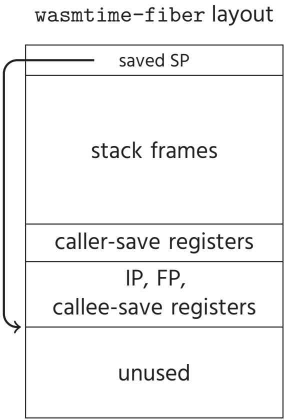
→ Provides **symmetric** switching!

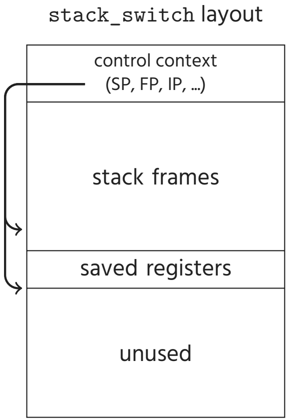
→ Similar to Dolan et al.'s SWAPSTACK (TACO 2013: "Compiler Support for Lightweight Context Switching")



Stack Layout Comparison

Layout similar to one used by `wasmtime-fiber` turned out to be natural fit





To ease transition: Introduced third, intermediate version of stack layout

Benchmark Results

Measuring performance change of single commit enabling native stack switching

- Platform: x64 Linux
- CPU: AMD Ryzen 3900X

Benchmark	Relative improvement
c10m	1.49
sieve	2.61
skynet	1.72
state	4.48
suspend_resume	5.97

Benchmark Results

Measuring performance change of single commit enabling native stack switching

- Platform: x64 Linux
- CPU: AMD Ryzen 3900X

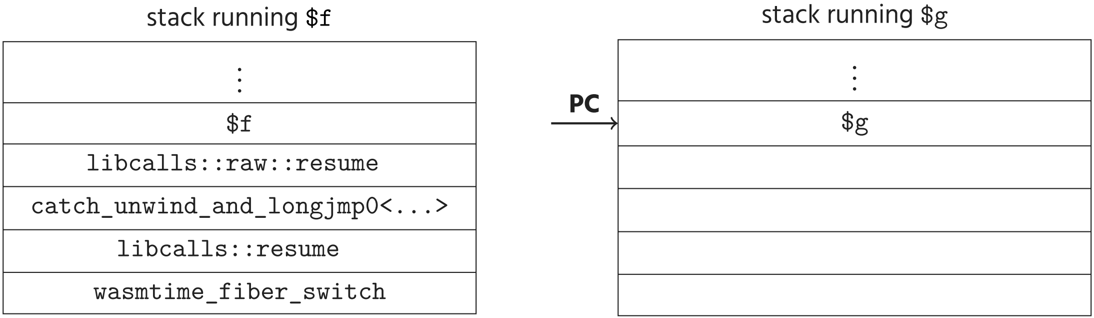
Benchmark	Relative improvement
c10m	1.49
sieve	2.61
skynet	1.72
state	4.48
suspend_resume	5.97

surprisingly good?

Benchmark Analysis

Prototype implementation executing following situation

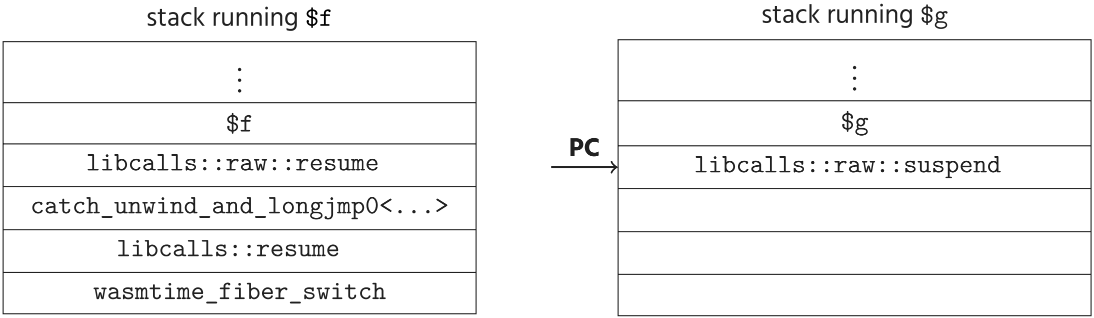
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

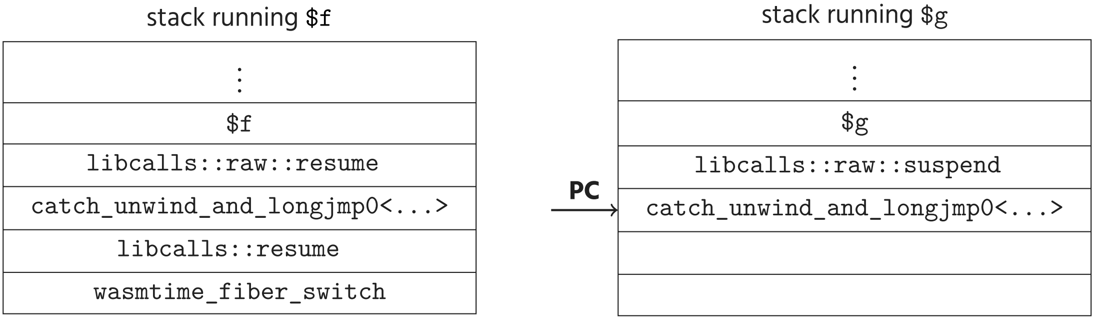
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

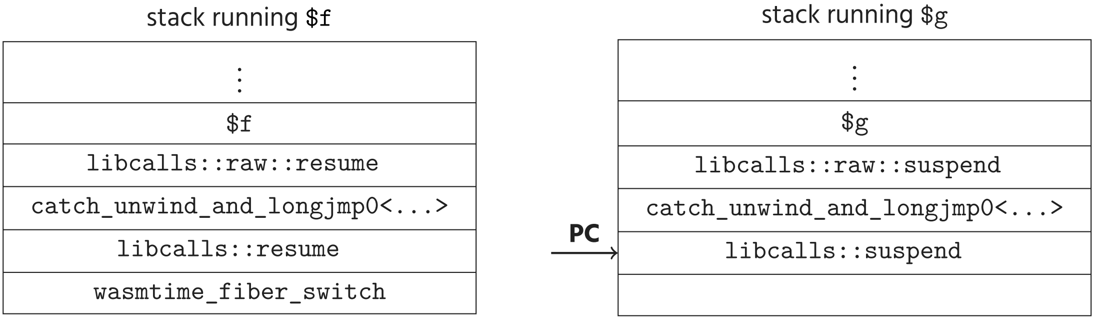
- At some point: $f resumed continuation now running $g
- Now: Continuation running $g suspends itself back to stack running $f



Benchmark Analysis

Prototype implementation executing following situation

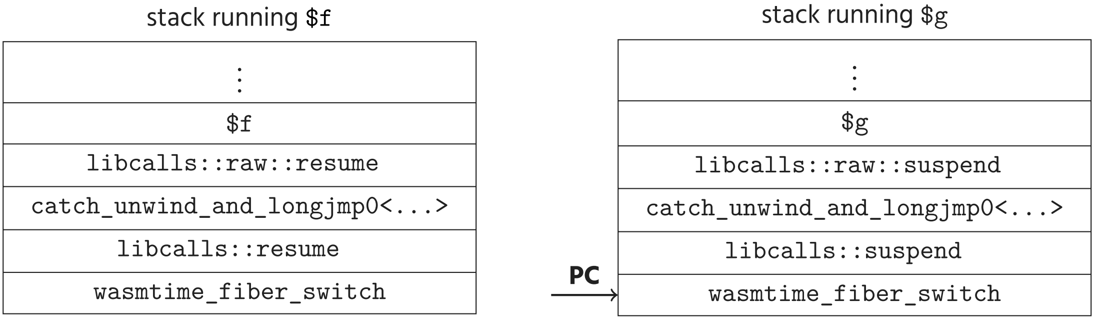
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

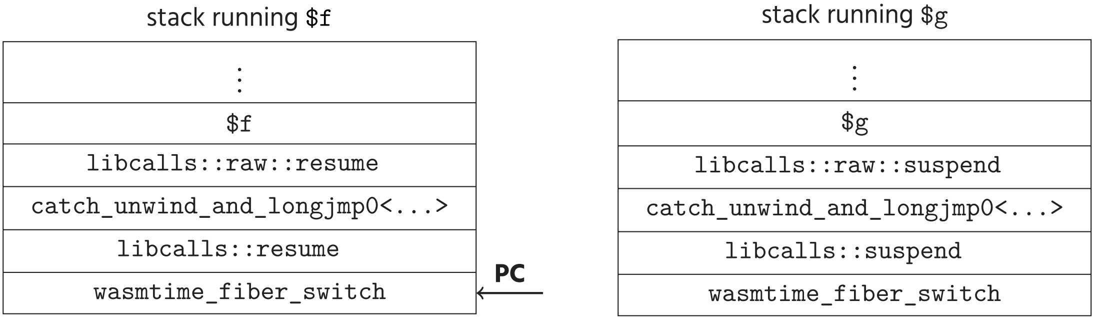
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`

stack running `$f`
⋮
`$f`
`libcalls::raw::resume`
`catch_unwind_and_longjmp0<...>`
`libcalls::resume`
`wasmtime_fiber_switch`

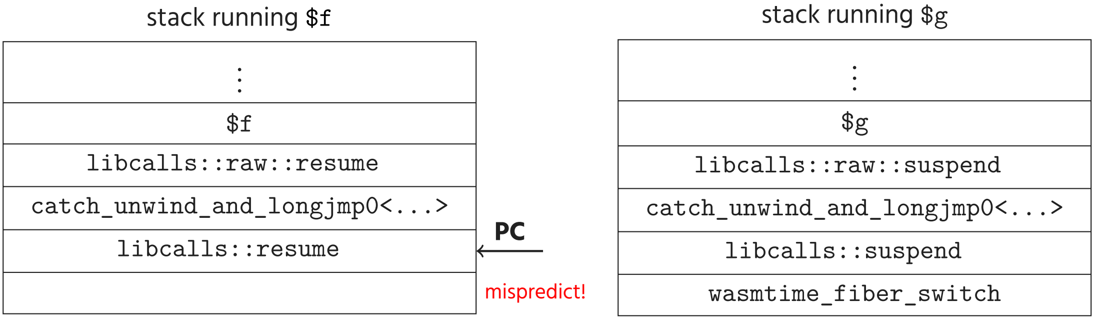
mispredict!

stack running `$g`
⋮
`$g`
`libcalls::raw::suspend`
`catch_unwind_and_longjmp0<...>`
`libcalls::suspend`
`wasmtime_fiber_switch`

Benchmark Analysis

Prototype implementation executing following situation

- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

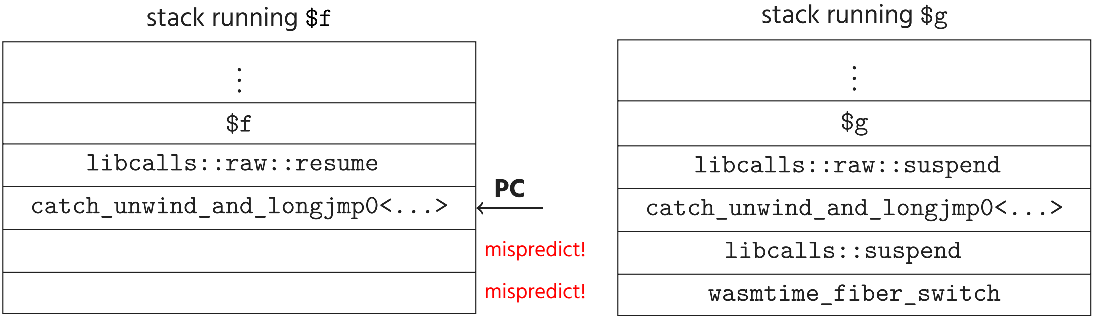
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`

stack running `$f`		stack running `$g`
⋮		⋮
`$f`		`$g`
`libcalls::raw::resume`		`libcalls::raw::suspend`
`catch_unwind_and_longjmp0<...>`		`catch_unwind_and_longjmp0<...>`
`libcalls::resume`	mispredict!	`libcalls::suspend`
	mispredict!	`wasmtime_fiber_switch`

Benchmark Analysis

Prototype implementation executing following situation

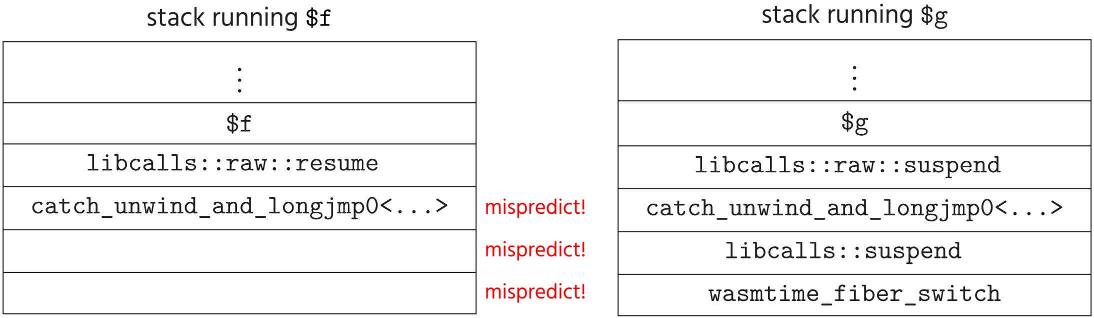
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

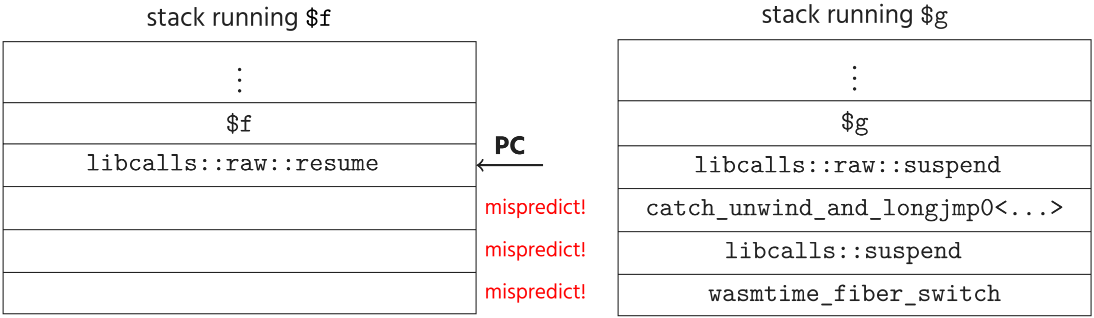
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

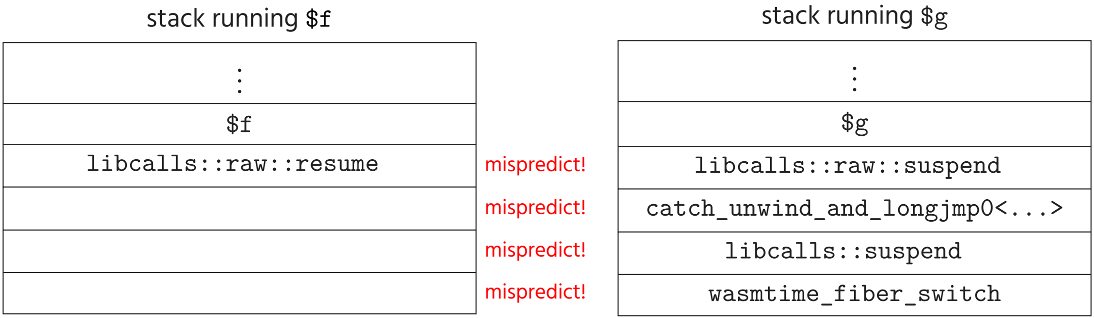
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

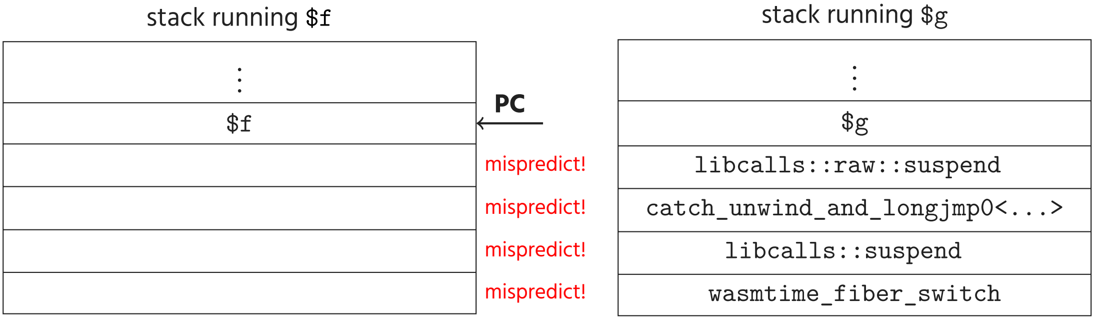
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

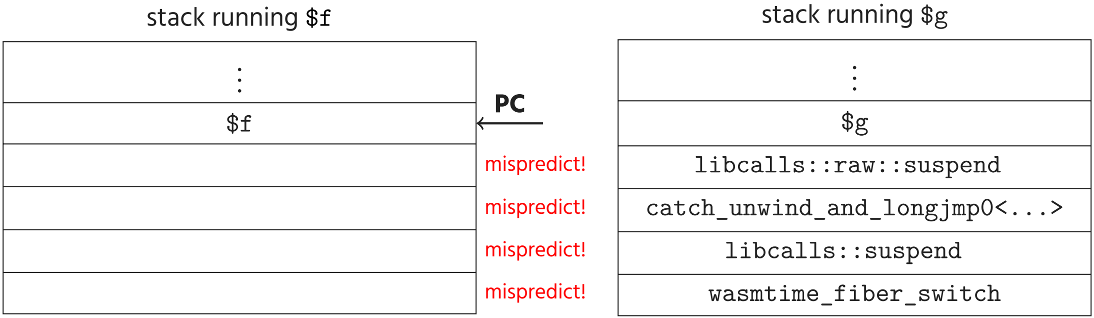
- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



Benchmark Analysis

Prototype implementation executing following situation

- At some point: `$f` resumed continuation now running `$g`
- Now: Continuation running `$g` suspends itself back to stack running `$f`



- Stack switching confuses CPU's Return Address Prediction unit:
  4 guaranteed mispredictions per Wasm stack switching operation

Summary

1. **Create inefficient, but easy to implement prototype**

2. **Sketch design of optimised implementation**

3. **Incremental changes towards optimised implementation: No big bang**

4. **Arrive at optimised implementation!**

Summary

1. **Create inefficient, but easy to implement prototype**
   - Built on `wasmtime-fiber`, libcall mechanism
   - No changes to Cranelift at this stage
   - For any complicated logic: Use libcall written in Rust

2. **Sketch design of optimised implementation**

3. **Incremental changes towards optimised implementation: No big bang**

4. **Arrive at optimised implementation!**

Summary

1. **Create inefficient, but easy to implement prototype**
   - Built on `wasmtime-fiber`, libcall mechanism
   - No changes to Cranelift at this stage
   - For any complicated logic: Use libcall written in Rust
2. **Sketch design of optimised implementation**
   - Early experimentation work with emitting stack switching in Cranelift
3. **Incremental changes towards optimised implementation: No big bang**
4. **Arrive at optimised implementation!**

Summary

1. **Create inefficient, but easy to implement prototype**
   - Built on `wasmtime-fiber`, libcall mechanism
   - No changes to Cranelift at this stage
   - For any complicated logic: Use libcall written in Rust
2. **Sketch design of optimised implementation**
   - Early experimentation work with emitting stack switching in Cranelift
3. **Incremental changes towards optimised implementation: No big bang**
   - Custom copy of `wasmtime-fiber`, adapted over time
   - Step-wise transition of stack layout used by (our) `wasmtime-fiber` vs `stack_switch` instruction
   - Not mentioned today: Many other small optimisations
4. **Arrive at optimised implementation!**

Summary

1. **Create inefficient, but easy to implement prototype**
   - Built on `wasmtime-fiber`, libcall mechanism
   - No changes to Cranelift at this stage
   - For any complicated logic: Use libcall written in Rust
2. **Sketch design of optimised implementation**
   - Early experimentation work with emitting stack switching in Cranelift
3. **Incremental changes towards optimised implementation: No big bang**
   - Custom copy of `wasmtime-fiber`, adapted over time
   - Step-wise transition of stack layout used by (our) `wasmtime-fiber` vs `stack_switch` instruction
   - Not mentioned today: Many other small optimisations
4. **Arrive at optimised implementation!**
   - Currently being upstreamed

WasmFX Resource List

- → Proposal repository: Informal overview, Reference interpreter (https://github.com/WebAssembly/stack-switching)
- → Wasmtime implementation (https://github.com/wasmfx/wasmfxtime)
- → Fiber library (https://github.com/wasmfx/fiber-c)
- → Benchmark suite (https://github.com/wasmfx/benchfx)
- → OOPSLA'23 research paper (https://doi.org/10.48550/arXiv.2308.08347)

https://github.com/WebAssembly/stack-switching

Bonus slides