WasmFX Stack Switching: Status and Future Plans

Frank Emrich
University of Edinburgh

Wasm CG Meeting
6 June 2024

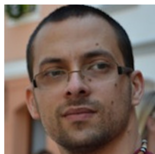
Collaborators

Sam Lindley

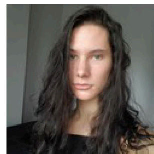
Andreas Rossberg

Daan Leijen

KC Sivaramakrishnan


Matija Pretnar

Daniel Hillerström


Luna Phipps-Costin

Arjun Guha

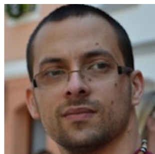
Collaborators

Sam Lindley

Andreas Rossberg

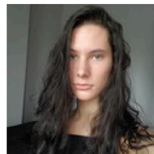
Daan Leijen

KC Sivaramakrishnan



Matija Pretnar

Daniel Hillerström



Luna Phipps-Costin

Arjun Guha

Overview

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, ...

Two approaches explored in parallel

- WasmFX (aka Typed Continuations):
  Parent-child relationships between continuations (= stacks)
- Bag-o-Stacks (BoS):
  No hierarchy between stacks

WasmFX

- OOPSLA 2023: “Continuing WebAssembly with Effect Handlers”
- Implemented in (up-to-date) fork of Wasmtime

WasmFX in a Nutshell: Hello Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ft))
  (tag $yield (param i32))

  (func $generator
    ...
  )


  (func $consumer
    ...
  )







)
```

WasmFX in a Nutshell: Hello Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1000))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i (i32.sub (local.get $i) (i32.const 1)))
      (br_if $l)
    )
  )


  (func $consumer
    ...
  )

)
```

WasmFX in a Nutshell: Hello Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator ... (suspend $yield (local.get $i)) ... )

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1000))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1000))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`

Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

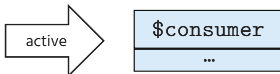
Parent-child relationships during `(call $consumer)`

Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`

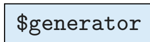


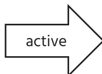
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`



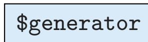


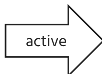
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`



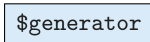


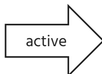
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`



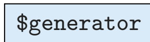


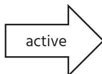
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`

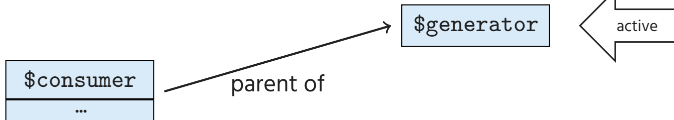




Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

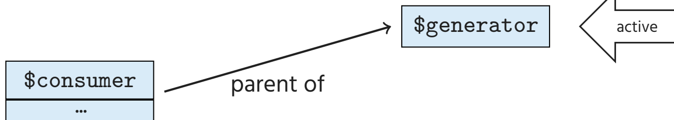
```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```



Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

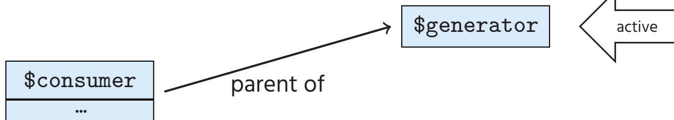
```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```



Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

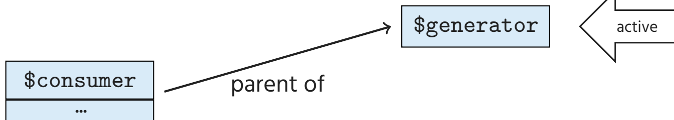
```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```



Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

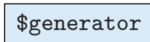


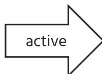
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`



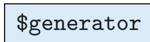


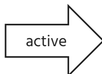
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`





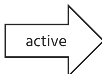
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`

$generator



$consumer

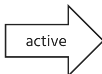
...

Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`



$generator

$consumer

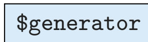
...

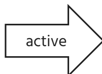
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`



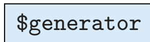


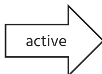
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`



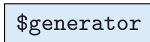


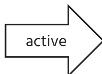
Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`

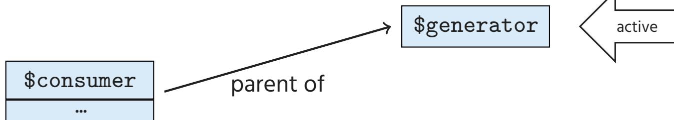




Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

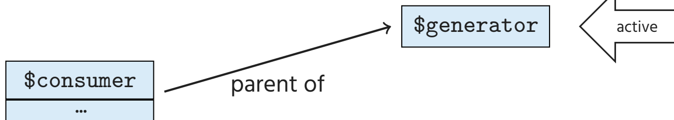
```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```



Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

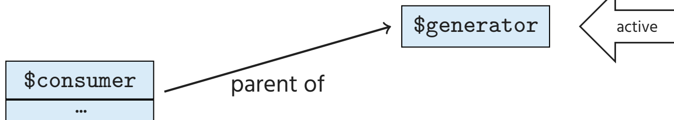
```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```



Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

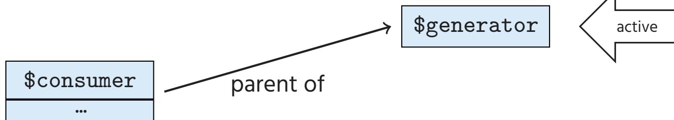
```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```



Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

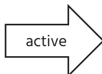


Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`

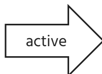


Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships during `(call $consumer)`



Unleashing the Generator

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))

  (func $generator
    (local $i i32)
    (local.set $i (i32.const 1))
    (loop $l
      (suspend $yield (local.get $i))
      (local.tee $i
        (i32.sub
          (local.get $i)
          (i32.const 1)))
      (br_if $l)
    )
  )

...
```

```
...

  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Implementation Status @ October 2023

WasmFX implemented in fork of Wasmtime

Reliable, but somewhat naive implementation

Previous limitations (as of last update, October 23)

Safety

- Inefficient checks that continuations only used once
- Unsafe stacks: Continuation stacks can overflow unnoticed
- No proper treatment of tags crossing module boundaries

Usability/Features

- No stack growing, need to allocate large stacks upfront
- No stack traces when using continuation stacks
- Cancellation of continuations not implemented

Implementation Status @ October 2023

WasmFX implemented in fork of Wasmtime

Reliable, but somewhat naive implementation

Previous limitations (as of last update, October 23)

Performance

- Actual stack switching implemented by calling into runtime
- All payloads passed by using dedicated buffers
- No pooling/reuse of continuation stacks

Benchmarking

- Micro benchmarks only, using handwritten .wat files
- No macro benchmarks

Safety

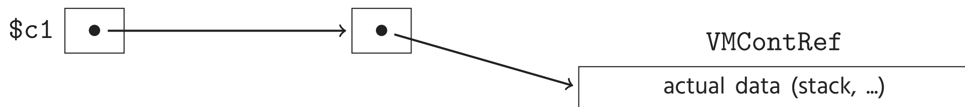
Linearity Checks (1)

Continuations are **one-shot**, can only be `resume`-d once

```
(local $c1 (ref $ct1)
(local $c2 (ref $ct2)

...

(block $handler (result (ref $ct2))
  (resume $ct1 (tag $some_tag $handler) (local.get $c1))
  ;; $c1 invalid now
)
;; $c1 invalid now, new continuation on stack
(local.set $c2)

...
```

Linearity Checks (2)

Previously Extra memory indirection ⇒ extra allocations need to be managed



Linearity Checks (2)

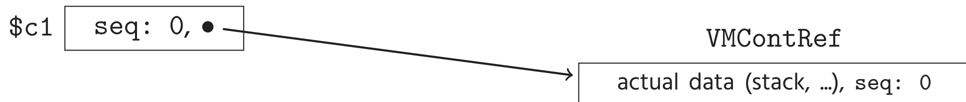
Previously Extra memory indirection ⇒ extra allocations need to be managed

Linearity Checks (2)

Previously Extra memory indirection ⇒ extra allocations need to be managed

Now

- Continuations are fat pointers: Pointer to a `VMContRef` + sequence counter
- `VMContRef` also stores a sequence counter
- On `resume`: Compare counters, increment the one inside `VMContRef`



Linearity Checks (2)

Previously Extra memory indirection ⇒ extra allocations need to be managed

Now

- Continuations are fat pointers: Pointer to a `VMContRef` + sequence counter
- `VMContRef` also stores a sequence counter
- On `resume`: Compare counters, increment the one inside `VMContRef`

Stack Pooling & Stack Safety

Previously

- `mmap`-ing stack on each `cont.new` is slow
- `malloc` very small stacks instead
- No guard pages, stacks can overflow unnoticed ⚡

Now

- Stack pooling available as option
- Entire pool `mmap`-ed on startup, with guard pages

Usability/Features

Backtrace Support (1)

Example at beginning: `$consumer` runs `$generator` inside continuation

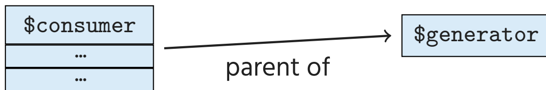
```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))
  (func $generator ... (suspend $yield (local.get $i)) ... )
  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new $ct (ref.func $generator)))
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Backtrace Support (1)

Example at beginning: `$consumer` runs `$generator` inside continuation

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))
  (func $generator ... (susp
  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships



Backtrace Support (1)

Example at beginning: `$consumer` runs `$generator` inside continuation

```
(module
  (type $ft (func))
  (type $ct (cont $ct))
  (tag $yield (param i32))
  (func $generator ... (susp
  (func $consumer
    (local $c (ref $ct))
    (local.set $c (cont.new
    (loop $loop
      (block $on_yield (result i32 (ref $ct))
        (resume $ct (tag $yield $on_yield) (local.get $c))
        (return) ;; generator returned: no more data
      ) ;; stack: [i32 (ref $ct)]
      (local.set $c)
      (drop) ;; would do something with generated value
      (br $loop)
    )
  )
)
```

Parent-child relationships

What if `$generator` traps?

Backtrace Support (2)

Reminder: All currently running stacks/continuations form a chain

Previously Backtraces disabled in Wasmtime when running continuations

Now If `$generator` traps while `resume`-d from `$consumer`:

```
Error: failed to run main module `generator.wat`
Caused by:
    0: failed to invoke command default
    1: error while executing at wasm backtrace:
           0:    0x3d - m!generator
           1:    0x50 - m!consumer
           2:    0x60 - m!start
    2: wasm trap: wasm `unreachable` instruction executed
```

Backtraces contain entire chain of arbitrarily many continuations

Backtrace Support (3)

Previous slide: Backtraces generated by engine itself (full access to runtime metadata about stacks)

What about external tools inspecting backtraces? (debuggers, profilers, ...)

Previously Custom DWARF info, does not work nicely with `perf` when switching stacks

Now

- Can also use standard frame pointer walking
- All active continuations form a single frame pointer chain
- ... at no additional runtime cost, just clever layout of data we store anyway

Performance

Architecture of Current Implementation

Stack switching

- Implemented using customized version of `wasmtime-fiber`
- On each WasmFX instruction: Call from generated code into runtime

Payload passing

- Write to dedicated buffer on sending side, read on receiving side
- Arguments for function running inside continuation:
  Accessed by using trampoline reading from buffer
- No payloads passed in registers

Work in progress

- Prototype implementation of generating stack switching code in Cranelift
- Once stabilized, tackle payload handling

Benchmarking & Toolchain Support

Binaryen Support

Working now

- Basic support for WasmFX instructions in Binaryen
- Our use case: `wasm-merge` for building benchmarks

Next steps

- Running `wasm-opt` on programs using WasmFX instruction
- Translations between WasmFX $\leftrightarrow$ BoS?

Fiber Library

Implemented simple library for general purpose stack switching in C: `fiber.h`

Two implementations of same interface: Using Asyncify and WasmFX

```
typedef struct fiber* fiber_t;
typedef void* (*fiber_entry_point_t)(void*);
typedef enum {
  FIBER_OK = 0,
  FIBER_YIELD = 1,
  FIBER_ERROR = 2
} fiber_result_t;

fiber_t fiber_alloc(fiber_entry_point_t entry);
void fiber_free(fiber_t fiber);

void* fiber_yield(void* arg);
void* fiber_resume(fiber_t fiber, void* arg, fiber_result_t* status);
```

Shadow Stacks vs Stack Switching

Shadow Stack

- Area of linear memory managed by Clang/LLVM
- `$__stack_pointer` `global` updated on function entry/exit
- Example: C locals whose address taken stored on shadow stack

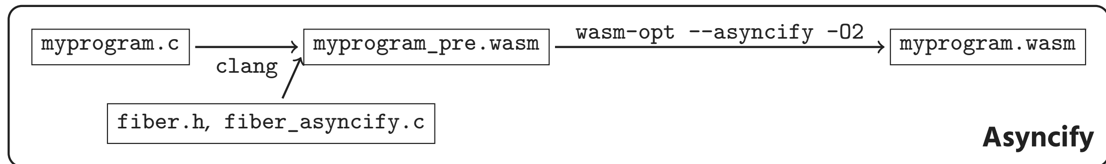
Challenge for WasmFX implementation

- Must switch shadow stacks, too!
- Shadow stacks of fibers must be independent from parent/caller

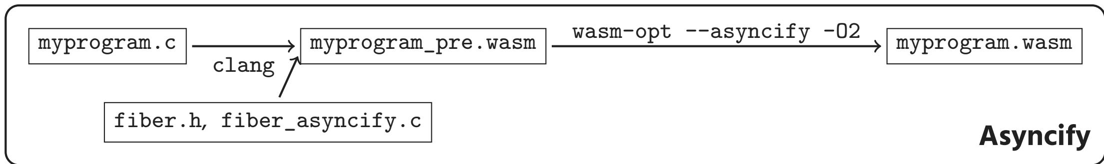
Solution

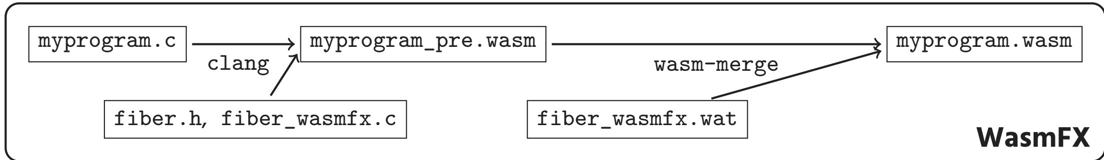
- Allocate dedicated shadow stack per fiber
- On `fiber_yield` & `fiber_resume`: Save and update shadow stack pointer

Fiber Library: Compilation & Benchmarking

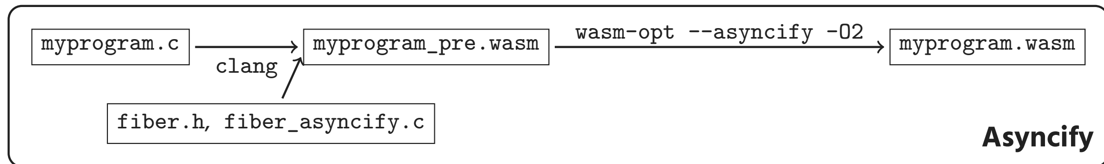


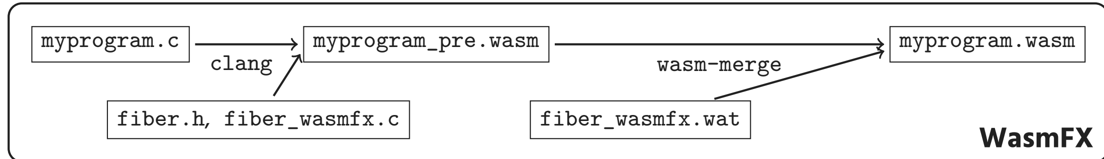
Fiber Library: Compilation & Benchmarking





Fiber Library: Compilation & Benchmarking





Previously Handwritten benchmarks using WasmFX instructions

Now

- Benchmarks written in C against `fiber.h`
- Automatically get Asyncify and WasmFX version of each benchmark
- Library enables writing arbitrary programs using fibers in C and compile to Wasm(FX)

HTTP Server

Benchmarking previously

- Only micro benchmarks
- Either focusing on rapid stack switching or stack creation
- Performance generally lagging behind Asyncify

New macro benchmark Minimal HTTP server benchmark

- Implemented in C against `fiber.h`
- Requests served asynchronously
- Benchmarked using standard load generator
- Performance matches Asyncify

Summary & Future Plans

Implementation Status Now

Safety

- Inefficient checks that continuations only used once
- Unsafe stacks: Continuation stacks can overflow unnoticed
- No proper treatment of tags crossing module boundaries

Usability/Features

- No stack growing, need to allocate large stacks upfront
- No stack traces when using continuation stacks
- Cancellation of continuations not implemented

Implementation Status Now

Safety

- ~~Inefficient checks that continuations only used once~~ ✓
- ~~Unsafe stacks: Continuation stacks can overflow unnoticed~~ ✓
- No proper treatment of tags crossing module boundaries **(WIP)**

Usability/Features

- No stack growing, need to allocate large stacks upfront **(TODO)**
- ~~No stack traces when using continuation stacks~~ ✓
- Cancellation of continuations not implemented **(Waiting for EH)**

Implementation Status Now

Performance

- Actual stack switching implemented by calling into runtime
- All payloads passed by using dedicated buffers
- No pooling/reuse of continuation stacks

Benchmarking

- Micro benchmarks only, using handwritten .wat files
- No macro benchmarks

Implementation Status Now

Performance

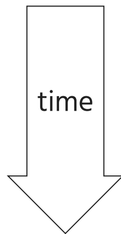
- Actual stack switching implemented by calling into runtime **(WIP)**
- All payloads passed by using dedicated buffers **(TODO)**
- ~~No pooling/reuse of continuation stacks~~ ✓

Benchmarking

- ~~Micro benchmarks only, using handwritten .wat files~~ ✓
- ~~No macro benchmarks~~ ✓

Future Plans

- Implement tags safely crossing module boundaries
- Experiment with different stack allocation & growing techniques
- More benchmarks, measuring use cases people care about
- Get `wasm-opt` to work on WasmFX
- Stabilize codegen for stack switching
- Implement payload passing on top of it
- Target WasmFX from other source language



WasmFX Resource List

- → Formal specification (https://github.com/WebAssembly/stack-switching/blob/main/proposals/continuations/Overview.md)
- → Informal explainer document (https://github.com/WebAssembly/stack-switching/blob/main/proposals/continuations/Explainer.md)
- → Reference implementation (https://github.com/WebAssembly/stack-switching/tree/wasmfx)
- → 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

https://wasmfx.dev