A Little on V8 and WebAssembly

An V8 Engine Perspective

Ben L. Titzer
WebAssembly Runtime TLM

Background

- A bit about me
- A bit about V8 and JavaScript
- A bit about virtual machines

Some history

- JavaScript ⇒ asm.js (2013)
- asm.js ⇒ wasm prototypes (2014-2015)
- prototypes ⇒ production (2015-2017)
- production ⇒ maturity (2017-)
- maturity ⇒ future (2019-)

This talk mostly

WebAssembly in a nutshell

- Low-level bytecode designed to be fast to verify and compile
  - Explicit non-goal: fast to interpret
- Static types, argument counts, direct/indirect calls, no overloaded operations
- Unit of code is a module
  - Globals, data initialization, functions
  - Imports, exports

WebAssembly module example

```
header: 8 magic bytes
types: TypeDecl[]
imports: ImportDecl[]
funcdecl: FuncDecl[]
tables: TableDecl[]
memories: MemoryDecl[]
globals: GlobalVar[]
exports: ExportDecl[]
code: FunctionBody[]
data: Data[]
```

- Binary format
- Type declarations
- Imports:
  - Types
  - Functions
  - Globals
  - Memory
  - Tables
- Tables, memories
- Global variables
- Exports
- Function bodies (bytecode)

WebAssembly bytecode example

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```

- Typed
- Stack machine
- Structured control flow
- One large flat memory
- Low-level memory operations
- Low-level arithmetic

Anatomy of a Wasm engine

- Load and validate wasm bytecode
- Allocate internal data structures
- Execute: compile or interpret wasm bytecode
- JavaScript API integration
- Memory management
- Debugging facilities
- Garbage collection

Compilers are awesome...

- WebAssembly performance goals:
  - Predictable: no lengthy warmup phase, no performance cliffs
  - Peak performance approaching native code (within ~20%)
- All major engine implementations reuse their respective JITs
  - V8: TurboFan AOT compile full module
  - SpiderMonkey: Ion AOT baseline compiler full module + background Ion JIT
  - JSC: B3 compile on instantiate, full module
  - Edge: lazy compile to internal bytecode and dynamic tier-up with Chakra

Compilation pauses aren't...

- WebAssembly performance goals:
  - Need **fast startup**: multi-second jank **not OK**
- V8 TurboFan compiles WASM at about 1-1.3MB/s
  - Builds full sea-of-nodes IR
  - Graph scheduling
  - Instruction selection
  - Register allocation
  - Code generation

10MB module ⇒ 8 seconds compiling :(

Can’t you just...make the compiler faster?

- Smaller graphs with TrapIf: ~30% total improvement
- TurboFan compilation time breakdown
  - TF Graph building: 10%
  - Optimization: 5%
  - TF Scheduling: 10-20%
  - Instruction selection: 10-15%
  - Register allocation: 20-40%
  - Code generation: 10-20%

Estimated additional max improvement ~30% :(

Can’t you just...make the compiler ~~faster~~ *parallel*?

- Parallel compilation of WebAssembly in TurboFan May 2016
- Not all phases of compilation can be done in parallel due to V8 limitations
- 5x-6x speedup on 8 core machine

AngryBots 8s ⇒ 1.5s

Can't you just...make the compiler ~~faster~~ *asynchronous*?

- WebAssembly JavaScript provides for async calls for compilation: `WebAssembly.compile(bytes)`
- API implemented in 2016
- Async implementation April 2017
- Lots of tricky race conditions with shutdown, shipped Nov 2017

AngryBots 1.5s ⇒ 0.15s initial + max 0.003s

Can't you just...make the compiler ~~faster~~ *streaming*?

- Why not compile while downloading?
- `WebAssembly.compileStreaming()` added in 2017
- Streaming compilation shipped in Dec 2017

8 threads can keep up with 50MBit/s

Can’t you just...make the compiler faster?

- Or how about a new compiler?
- Liftoff!

Prototype gain ~10-20x

Liftoff, a baseline compiler for WASM

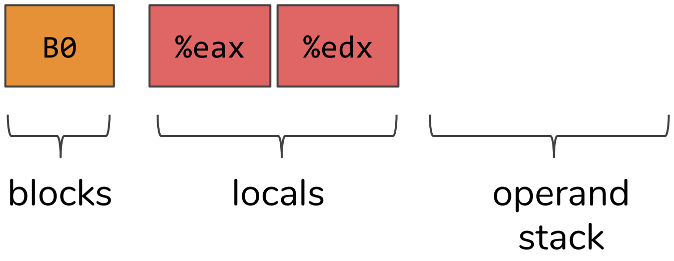
```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```

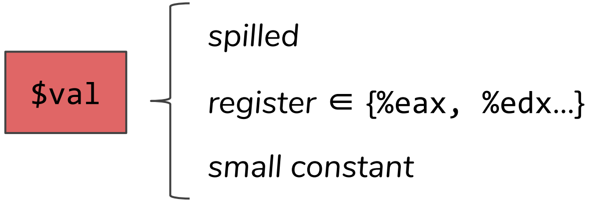
- Fast, single-pass compiler
- Templatized bytecode decoder/verifier
- On-the-fly register allocation
- Portable interface between decode logic, control flow and abstract stack tracking, and low-level codegen

A closer look at Liftoff

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```

Abstract state

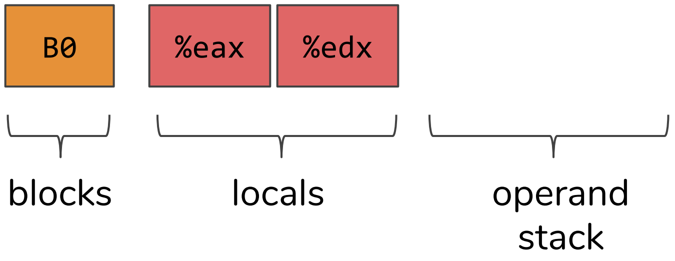




A closer look at Liftoff

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```

Abstract state



Initialize first control block

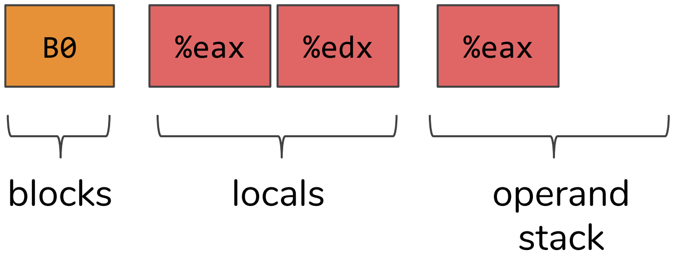
Initialize locals for parameters

`uint32_t LiftoffAssembler::PrepareStackFrame()`

A closer look at Liftoff

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```

Abstract state



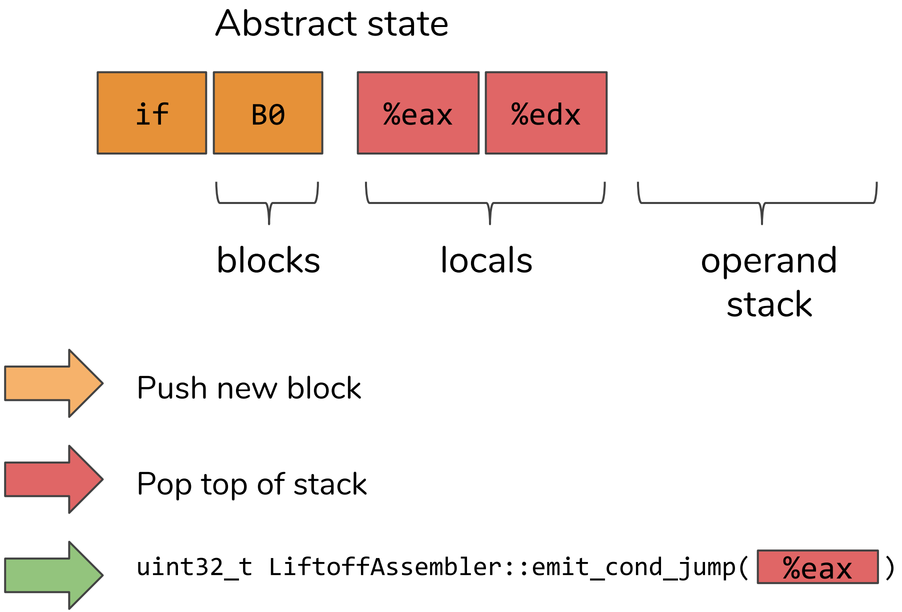
∅

Copy local[0] register onto abstract stack

∅

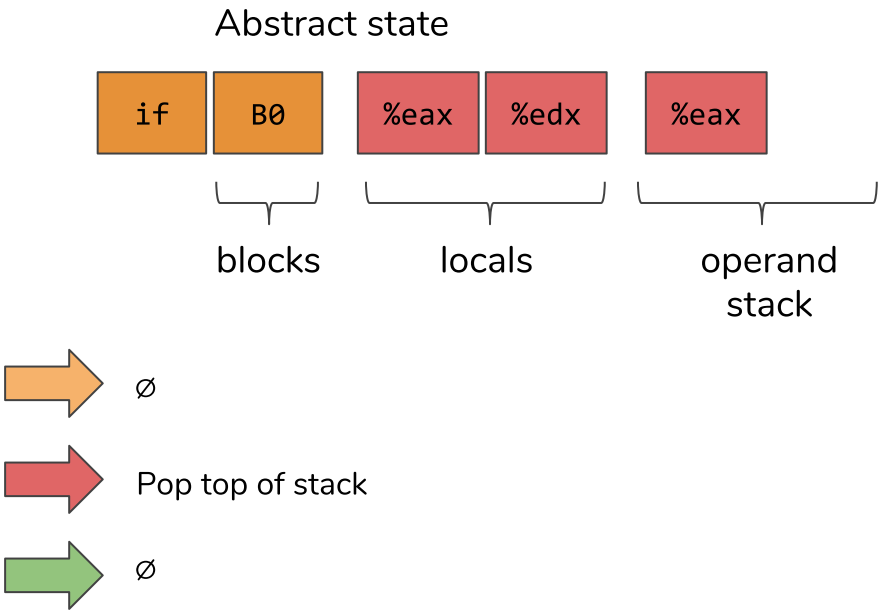
A closer look at Liftoff

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```



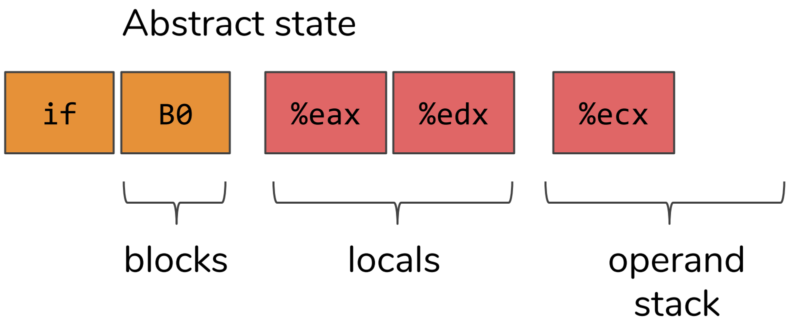
A closer look at Liftoff

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```



A closer look at Liftoff

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```

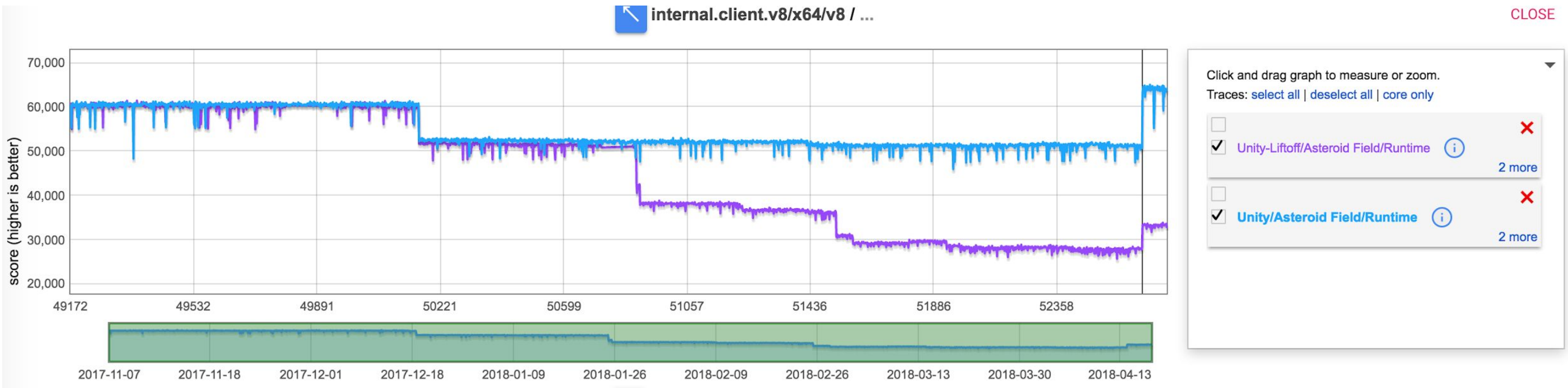


∅

Pop index off top of stack, allocate result

`uint32_t LiftoffAssembler::Load( %ecx , %eax , offset=8)`

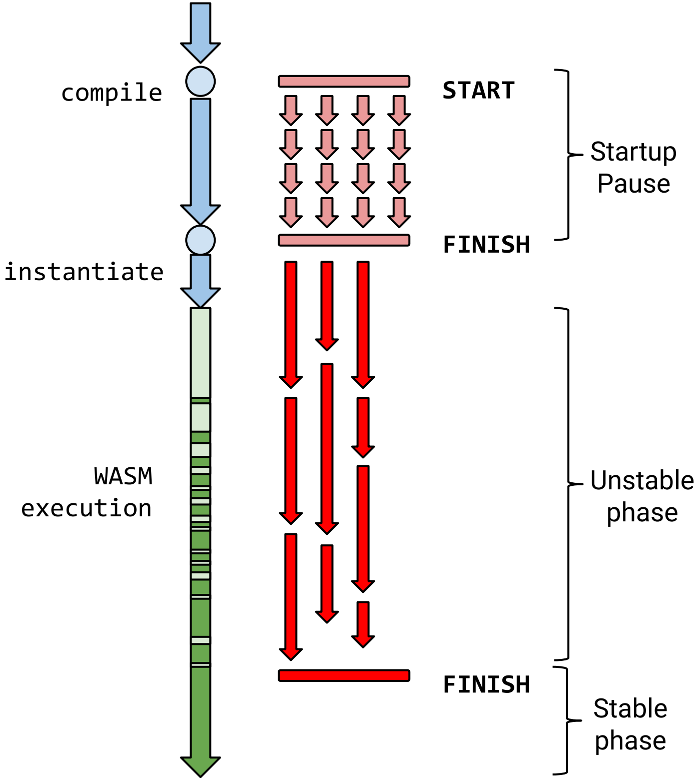
5x faster startup, 50% lower throughput

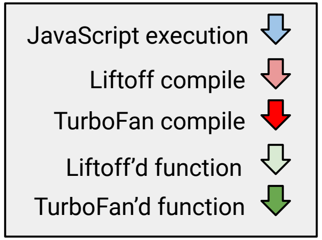


A zoo of tiers

- Tiering: balance compilation speed versus throughput
  - Liftoff is ~5x faster to compile, 1.5x slower to execute
  - Best startup requires Liftoff, peak performance requires TurboFan
  - (C++ interpreter is non-production, debugging only)
- Identified 4 different tiering strategies
  - Liftoff AOT, TurboFan background full compile
  - Liftoff AOT, dynamic tier-up
  - Liftoff lazy compile, dynamic tier-up
  - Liftoff background compile, dynamic tier-up

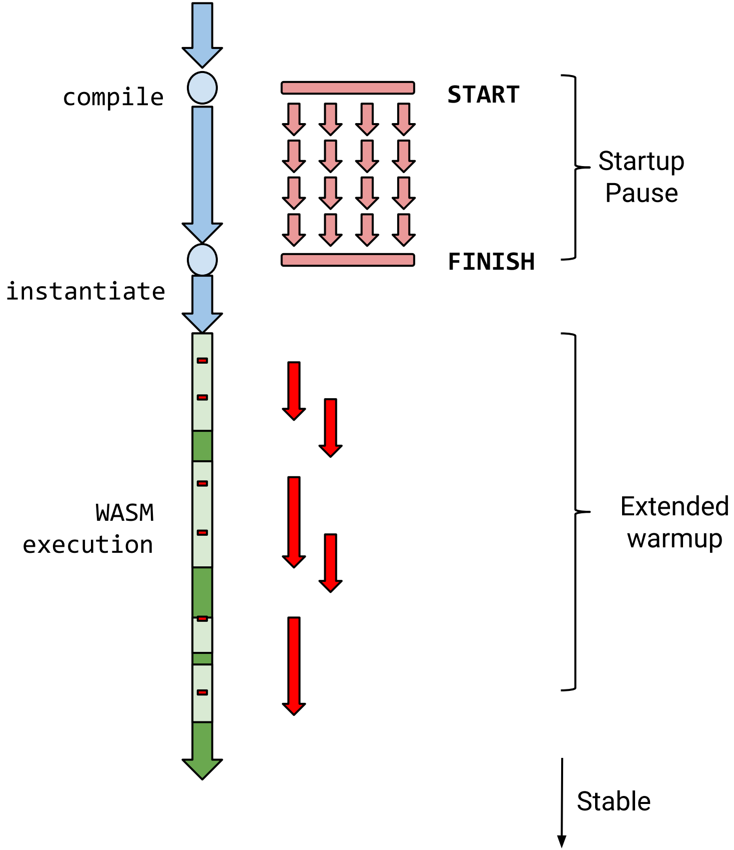
Liftoff AOT + TurboFan background

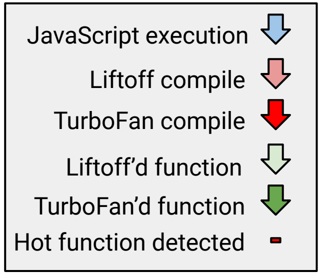




- **Advantages:**
  - Short startup pause
  - Smooth warmup: no jank
- **Disadvantages:**
  - Memory consumption
  - Double compile of everything

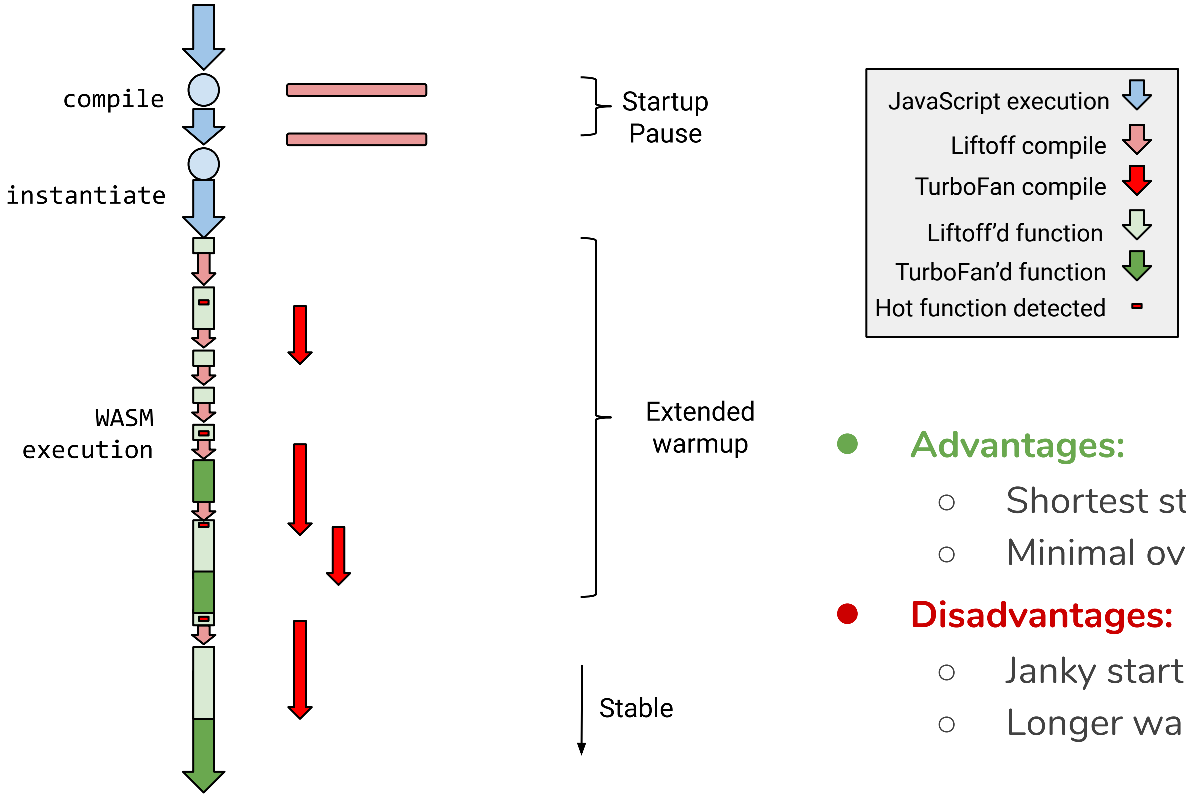
Liftoff AOT + dynamic tier-up





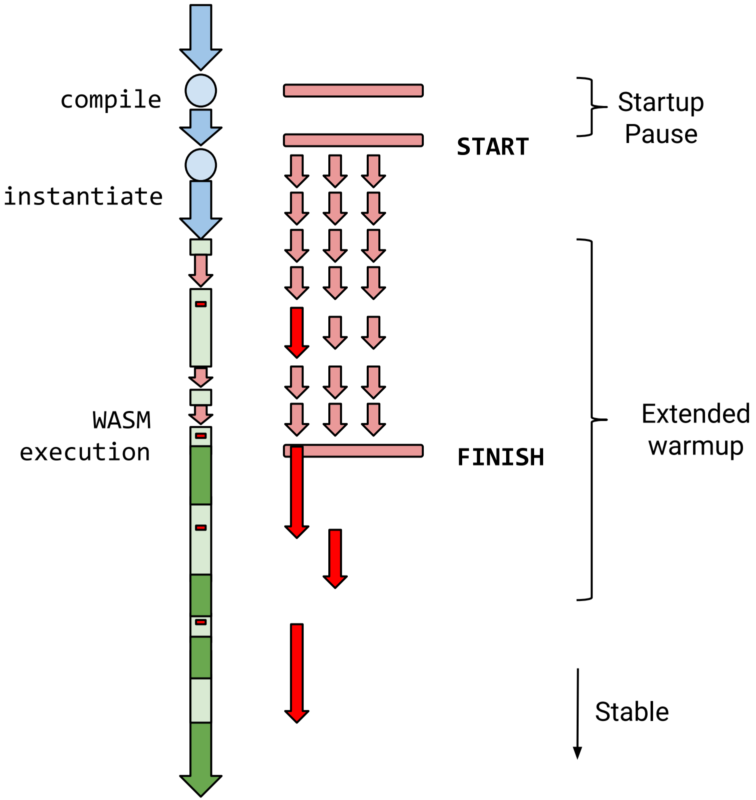
- **Advantages:**
  - Short startup pause
  - Smooth warmup: no jank
  - Less overall compile work
- **Disadvantages:**
  - Longer warmup

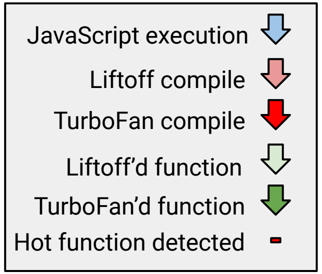
Liftoff lazy + dynamic tier-up



- **Advantages:**
  - Shortest startup pause
  - Minimal overall compile work
- **Disadvantages:**
  - Janky startup
  - Longer warmup

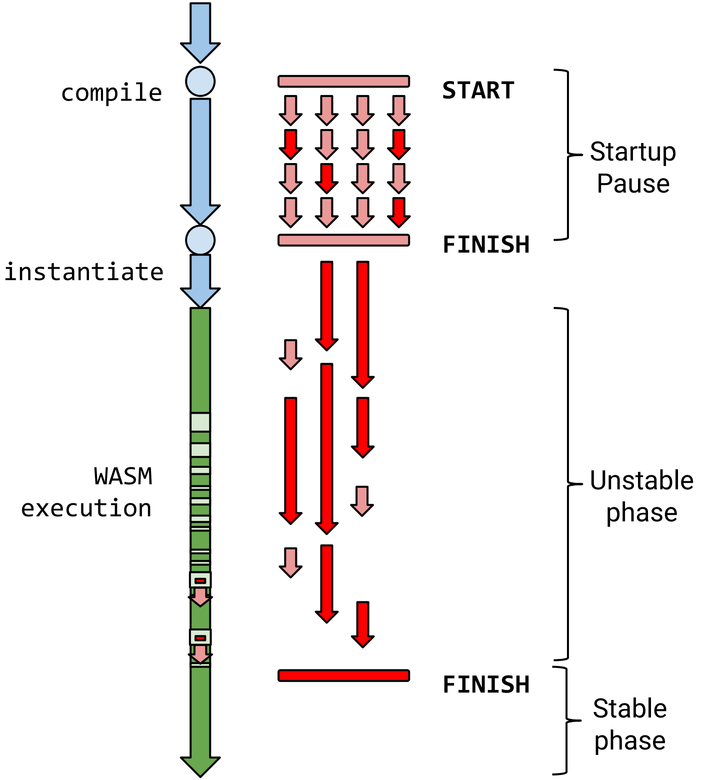
Liftoff background + dynamic tier-up

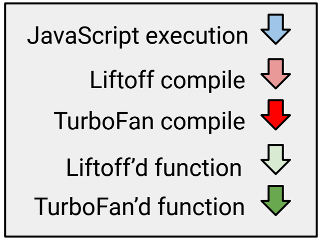




- **Advantages:**
  - Short startup pause
  - Smooth(er) warmup
  - Less overall compile work
- **Disadvantages:**
  - Longer warmup
  - Limited startup jank

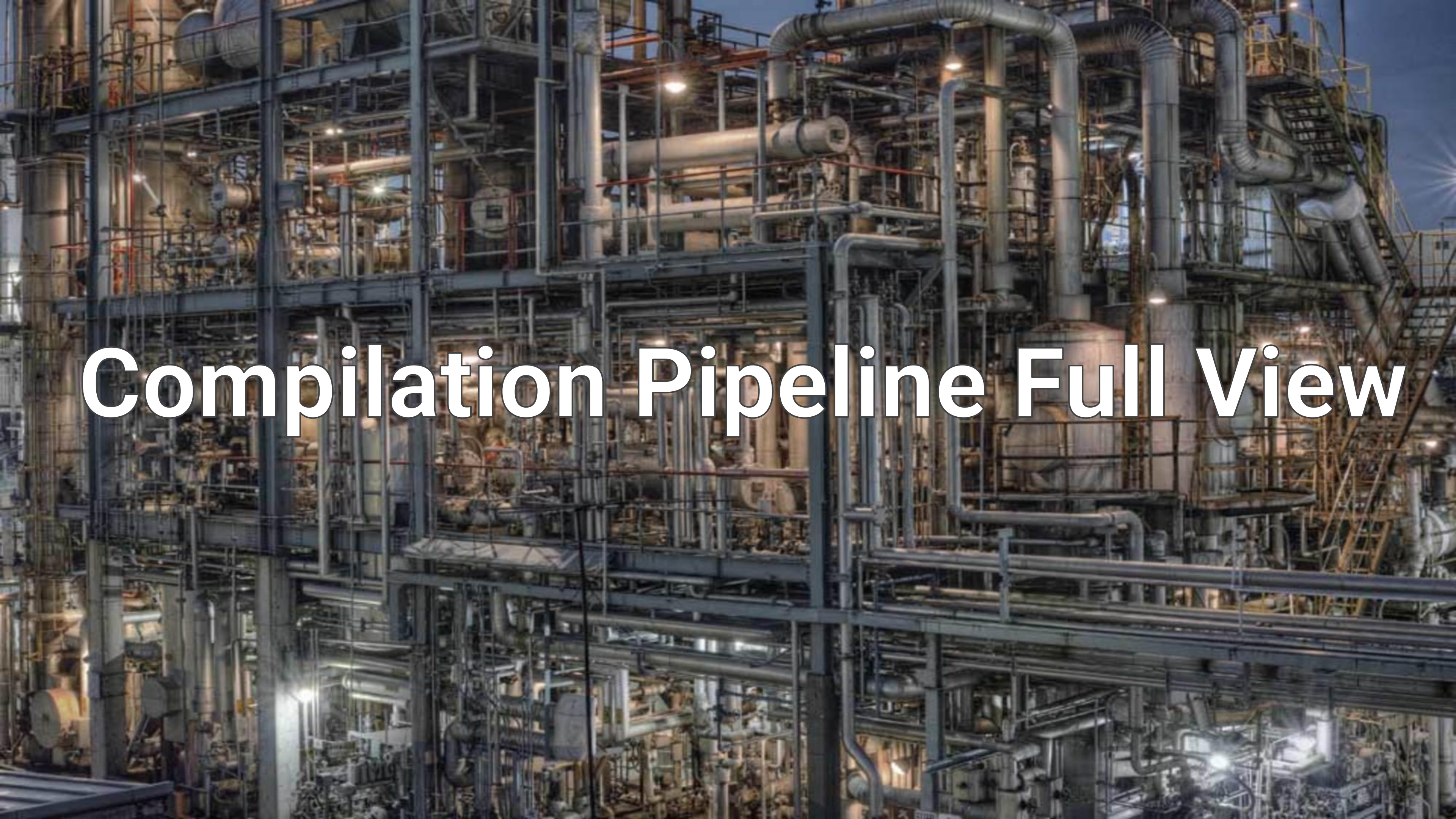
Hint AOT + background + lazy





- **Advantages:**
  - Shorter startup pause
  - Smooth warmup: no jank
  - Reach peak perf fast
- **Disadvantages:**
  - Imprecise static heuristic

Compilation Pipeline Full View



WebAssembly Compilation Roadmap

- Parallel - **done** Q1 2016
- Asynchronous - **done** Q3 2016
- Streaming - **done** Q2 2017
- Baseline tier - **done** Q2 2018
- Two-tier JIT strategy (#1) - **done** Q2 2018
- Compilation hints section (#5) - **prototype now**
- Free dead baseline code - work ongoing
- Out-of-process JIT - work starting now

Wasm Runtime System Refactoring

- Remove code specialization (WasmInstanceObject)
- Trap handler for out-of-bounds memory accesses
- Proper lifetime management of all runtime data structures
- Move compiled code off heap and share
- Multi-threaded, shared engine
- Proper lifetime management of backing stores

New WebAssembly language features

- **Done** or **mostly done**: **atomics**, **tail call**, **exceptions**, **bulk-memory**, **multi-value**, **reference types**
- Most changes require changes to execution tiers
- Other are mostly reorganization of runtime data structures
- Smaller surface area helps!
- Relying on V8 infrastructure such as TurboFan has pluses and minuses
- On the horizon: WebIDL bindings, first-class functions, managed data (GC), SIMD, bigger memory