JWST—— A JavaScript-to-WebAssembly Static Translator

Prof. Shi Xiaohua
Beihang University, Beijing, China
Sep. 2023
Outlines

• Challenges of Statically Compiling JavaScript
  • “Dynamic types” & “Dynamic loading”

• JWST—— A JavaScript-to-WebAssembly Static Translator
  • Compiler architecture
  • Type profiling
  • Optimizations

• Performance Evaluation
  • ECMA TEST-262: JavaScript language compatibility testing
  • Performance tests: SunSpider
  • Practical cases: React Native Application
Demonstration of dynamic types

Example 1: Dynamically modify the type of a variable

```javascript
var foo = 123;
console.log(foo);
foo = false;
console.log(foo);
foo = "foo";
console.log(foo);
foo = function () {
    return "string";
};
console.log(foo());
```

Output:

123
false
foo
string
Demonstration of dynamic types

Example 2: Operations between variables of different types

```javascript
var foo = 123;
var bar = "456"
var t = new Boolean(true);
var f = false;
console.log(foo + bar + t + f);
```

Output:

```
123456truefalse
```
Demonstration of dynamic types

Example 3: Dynamically increase or decrease object attributes

```
var obj = {
  foo: 123
};
obj.bar = "456";
console.log(obj.foo + obj.bar);
delete obj.foo;
console.log(obj.foo + obj.bar);
```

Output:

```
123456
undefined456
```
Demonstration of dynamic types

Example 4: Arrays containing different types of elements

```
var arr = [1, "2", true, null, undefined, NaN, Infinity, -Infinity, {}, [1, 2, 3], function () { }];

for (var item of arr) {
    console.log(item);
}
```

Output:
```
1
2
true
null
undefined
NaN
Infinity
-Infinity
[object Object]
1,2,3
function () {
    [native code]
}
```
Demonstration of dynamic loading

Example 1:

```javascript
var x = 1, y = 2;

function f() {
    console.log("hello world!");
    return x;
}

console.log(eval("1 + 2"));
console.log(eval("x - y"));
console.log(eval("f()"));

eval("x = 10");
console.log(x);
```

Output:

```
3
-1
hello world!
1
10
```
Demonstration of dynamic loading

Example 2:

```javascript
var str = "var x = 1;
    + "function f() {
    + "var y = 2;
    + "console.log(eval("x"));"
    + "console.log(eval("y"));"
    + "console.log(eval("x + y")); }"
    + "f();"

eval(str);
```

Output:

```
1
2
3
```
Outlines

• Challenges of Statically Compiling JavaScript
  • “Dynamic types” & “Dynamic loading”

• JWST—— A JavaScript-to-WebAssembly Static Translator
  • Compiler architecture
  • Type profiling
  • Optimizations

• Performance Evaluation
  • ECMA TEST-262: JavaScript language compatibility testing
  • Performance tests: SunSpider
  • Practical cases: React Native Application
JWST—A JavaScript-to-WASM Static Translator

• JWST is a static compiler developed by Huawei Tech. & Beihang University for compiling JavaScript to WASM and native code
  • Uses QuickJS bytecode as input, and outputs LLVM IR.
  • Got the similar pass rate as QuickJS for ECMA TEST 262.
  • JWST is about 30% and 50% faster than node.js when running SunSpider on Intel i7 and Kirin 990E CPUs, respectively, for the first execution.
  • For a React Native application, the native code generated by JWST is about 30% faster than V8 for the first execution on Kirin 990E CPUs, in terms of Time-to-Interactive(TTI) .
### WASM Sample Code

<table>
<thead>
<tr>
<th>C++</th>
<th>WASM Binary</th>
<th>WASM Text</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int factorial(int n) {</code></td>
<td><code>20 00</code></td>
<td><code>get_local 0</code></td>
</tr>
<tr>
<td><code> if (n == 0) {</code></td>
<td><code>42 00</code></td>
<td><code>i64.const 0</code></td>
</tr>
<tr>
<td><code>  return 1;</code></td>
<td><code>51</code></td>
<td><code>i64.eq</code></td>
</tr>
<tr>
<td><code>} else</code></td>
<td><code>04 7e</code></td>
<td><code>if i64</code></td>
</tr>
<tr>
<td><code>  return n * factorial(n-1);</code></td>
<td><code>42 01</code></td>
<td><code>i64.const 1</code></td>
</tr>
<tr>
<td></td>
<td><code>05</code></td>
<td><code>else</code></td>
</tr>
<tr>
<td></td>
<td><code>20 00</code></td>
<td><code>get_local 0</code></td>
</tr>
<tr>
<td></td>
<td><code>20 00</code></td>
<td><code>get_local 0</code></td>
</tr>
<tr>
<td></td>
<td><code>42 01</code></td>
<td><code>i64.const 1</code></td>
</tr>
<tr>
<td></td>
<td><code>7d</code></td>
<td><code>i64.sub</code></td>
</tr>
<tr>
<td></td>
<td><code>10 00</code></td>
<td><code>call 0</code></td>
</tr>
<tr>
<td></td>
<td><code>7e</code></td>
<td><code>i64.mul</code></td>
</tr>
<tr>
<td></td>
<td><code>0b</code></td>
<td><code>end</code></td>
</tr>
</tbody>
</table>
WASM vs. Java Bytecode

- Java bytecode is designed for expressing Java programs
  - They have almost the same semantic expression capability

- WASM is designed for working with JavaScript, not expressing JavaScript programs
  - For instance, WASM is not a dynamically typed language
  - WASM does not support dynamic loading
  - WASM does not have a GC, etc.
JWST Compilation Process

JS Code

QuickJS & JWST Runtime

clang

Runtime Lib (LLVM IR)

jwst-compiler

Import defs

QuickJS Bytecode

Codegen

LLVM IR

llc

WASM

llc

wasm-ld linker

Executable WASM

QuickJS Type Info.

Type Info. (Optional)

QuickJS Front-end
Compilation process demonstration(1)

JavaScript code:

```javascript
function foo(a, b) {
    return a + b + 10086;
}
foo(1, 2);
foo(10.2, 44.56);
```

QuickJS Bytecode

```javascript
-- JSFunction <eval> @ 0x611000008240
source ptr: 0x0, ln: 1
arg count: 0
stack size: 3
Funtion bytecode:
0: check_define_var "foo", 01000000
6: fclosure8 0
8: define_func "foo", 00000000
14: get_var "foo"
19: push_1
20: push_2
21: call2
22: put_loc0
// ...
var count: 1
<ret>: normal
Constant pool
length: 3
0: ptr 0x60f000000310 (function foo)
1: double 10.2
2: double 44.56

-- JSFunction foo @ 0x60f000000310
source ptr: 0x60f000001d750, ln: 1
arg count: 2
stack size: 2
Funtion bytecode:
0: get_arg0
1: get_arg1
2: add
3: push_i16 10086
6: add
7: return
var count: 0
Constant pool
length: 0
Compilation process demonstration(2)

Codegen/Function Prolog/Epilog

-- JSFunction foo @ 0x60f00000310
source ptr: 0x60400001d750, ln: 1
arg count: 2
stack size: 2
Function bytecode:
0: get_arg0
1: get_arg1
2: add
3: push_i16 10086
6: add
7: return
var count: 0
Constant pool
length: 0

; Function Attrs: nounwind
define i64 @_JS_F3foo__root.0(
  i64 %fn_obj_arg,
  i64 %this_arg,
  i64 %a_0_arg,
  i64 %b_0_arg
) local_unnamed_addr #9 {
entry:
;加载JSContext
%ctx = load %struct.JSContext*, %struct.JSContext** @js_global_ctx, align 4

;分配局部变量引用空间
%0 = alloca %struct.JWST_VarRefInfo, align 8
%1 = alloca [2 x i64]*, align 4
%sub = getelementptr ...
%2 = call %struct.JWST_VarRefInfo* @jwst_init_var_ref_info(...)  ;进行引用计数管理
%3 = call i64 @JS_DupValue(%struct.JSContext* %ctx, i64 %a_0_arg)
%4 = call i64 @JS_DupValue(%struct.JSContext* %ctx, i64 %a_0_arg)
%5 = call i64 @JS_DupValue(%struct.JSContext* %ctx, i64 %b_0_arg)
%6 = call i64 @JS_DupValue(%struct.JSContext* %ctx, i64 %b_0_arg)

; -- SNIP --
ret1:
%46 = phi i64 ...
call void @jwst_close_var_refs(...)
call void @JS_FreeValue(%struct.JSContext* %ctx, i64 %a_0_arg)
call void @JS_FreeValue(%struct.JSContext* %ctx, i64 %b_0_arg)
ret i64 %46
}
Compilation process demonstration(3)

**Codegen by a Mimic Stack**

0: get_arg0

1: get_arg1

2: add

3: push_i16 10086

6: add

7: return

**Mimic Stack**

- %a1
- %a1 %b1
- %2
- %2 i32 10086
- %6

**Generated Pseudo Code**

- %a1 = load %a
- %b1 = load %b
- %2 = add_any(%a1, %b1)
- %6 = add_any(%2, i32 10086)
- store %6 to %ret
- br label %ret_bb
Mimic stack at Basic Block Merge Point

bb1: 
- var1
- 114
- var3

bb2: 
- PHI(var1, var2)
- PHI(114, var4)
- PHI(var3, 514)

bb3: 
- PHI(var1, var2)
- PHI(114, var4)
- PHI(var3, 514)
**JSValue for Dynamic types**

- JWST reuses the JSValue data structure of QuickJS
  - All JavaScript values are represented as JSValues, including number, bool and object, etc.

- On a 32-bit u-arch, JSValue is a 64-bit vector
  - The higher 32 bits are tags, and the lower 32 bits are values
  - Double values use NaN boxing to avoid conflicts with other tags

- Before each operation, the type of JSValue must be determined

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int32</td>
</tr>
<tr>
<td>1</td>
<td>bool</td>
</tr>
<tr>
<td>2-6</td>
<td>null/undefined/…</td>
</tr>
<tr>
<td>7</td>
<td>double</td>
</tr>
<tr>
<td>8-9</td>
<td>ptr: void*</td>
</tr>
</tbody>
</table>

![Diagram of JSValue structure](image)
Demonstration of Dynamic Type Compilation Process

- Codegen for “Any” type

\[
\begin{align*}
%a &\text{ is int } \&\& \text{ %b is int?} \\
Y &\quad \text{N}
\end{align*}
\]

\[
%a\text{ is double } \&\& \text{ %b is double?} \\
Y &\quad \text{N}
\]

\[
%\text{res_int} = (%a \text{ as int}) + (%b \text{ as int})
\]

\[
%\text{res_double} = (%a \text{ as double}) + (%b \text{ as double})
\]

\[
%\text{res_any} = \text{add_slow}(%a, %b)
\]

\[
%\text{res} = \text{Phi}(%\text{res_int}, %\text{res_double}, %\text{res_any})
\]
Demonstration of Dynamic Type Compilation Process

- Codegen with types

```plaintext
%a: int  
%b: any

%a is int && %b is int?
Y  N

%a is double && %b is double?
Y  N

%res_int = (%a as int) + (%b as int)

%res_double = (%a as double) + (%b as double)

%res_any = add_slow(%a, %b)

%res = Phi(%res_int, %res_double, %res_any)
```
Demonstration of Dynamic Type Compilation Process

- Codegen with more types

\[
\begin{align*}
%a \; \text{is int} & \land \; %b \; \text{is int} \quad \Rightarrow \quad %\text{res_int} = (%a \; \text{as int}) + (%b \; \text{as int}) \\
%a \; \text{is double} & \land \; %b \; \text{is double} \quad \Rightarrow \quad %\text{res_double} = (%a \; \text{as double}) + (%b \; \text{as double}) \\
\text{add_slow}(%a, %b) & \quad \Rightarrow \quad %\text{res_any} = \text{add_slow}(%a, %b)
\end{align*}
\]

%\text{res} = \Phi(%\text{res_int}, %\text{res_double}, %\text{res_any})
Type Profiling: Type information fusion

• A single function may produce multiple versions of type profiling results
• Need to fuse type information from multiple versions

```plaintext
function foo(n) {
    if (n < 10) {
        return "1";
    } else {
        return 0;
    }
}

for (i = 0; i < 20; i++) {
    foo(i);
}
```

<table>
<thead>
<tr>
<th>bytecode</th>
<th>type v1, hit 10</th>
<th>type v2, hit 10</th>
<th>type sum up, hit 20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>operands</td>
<td>result</td>
<td>operands</td>
</tr>
<tr>
<td>get_arg0 0: n</td>
<td>INT</td>
<td></td>
<td>INT</td>
</tr>
<tr>
<td>push_i8 10</td>
<td>INT, INT</td>
<td>INT</td>
<td>INT, INT</td>
</tr>
<tr>
<td>lt</td>
<td>INT, INT</td>
<td>BOOL</td>
<td>INT, INT</td>
</tr>
<tr>
<td>if_false8 9</td>
<td>BOOL</td>
<td></td>
<td>BOOL</td>
</tr>
<tr>
<td>push_const8 0: 1&quot;1&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>9: push_0 0</td>
<td>INT</td>
<td></td>
<td>INT</td>
</tr>
<tr>
<td>return</td>
<td>INT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Type Profiling: Example

```javascript
function fib(n) {
    if (n === 0 || n === 1) return 1;
    return fib(n - 1) + fib(n - 2);
}

fib("20");
```

Testing source code and bytecode

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>v1 (hit = 1)</th>
<th>v2 (hit = 10944)</th>
<th>v3 (hit = 41819)</th>
<th>v4 (hit = 6765)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>operand</td>
<td>result</td>
<td>operand</td>
<td>result</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>2</td>
<td>S</td>
<td>B</td>
<td>I</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td></td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>S</td>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>9</td>
<td>S</td>
<td>B</td>
<td>I</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>B</td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>I</td>
</tr>
</tbody>
</table>

QuickJS Profiling results

Note: S=string, I=int, B=bool, O=object, The gray mark indicates that the control flow has not passed through this bytecode.
Dynamic loading and execution of JS programs

• JWST supports `eval()` and `new Function()` to run dynamically loaded code.

• Due to the limitations of WASM runtime, generating or linking statically compiled code at runtime is currently not supported.

```javascript
// eval.js
function foo() {
    print("foo! b =", b);
}

var b = 2;
var j = 1;
var i = readline();
eval(i);

//readline is a temporary function added to demonstrate this feature

$ wasmer eval.wasm
(输入) print("j =", j); foo(); b = 1000; foo();
j = 1
foo! b = 2
foo! b = 1000
```
Mixed running mode of compiled code and QuickJS Interpreter

- JWST statically compiled code and the embedded QJS interpreter could be called bi-directional through the dispatch function, supporting mixed running of static code and scripts.

- The dispatch function checks if the static compiled version exists and if the argument types match. It then calls if both matches.
Speculative Optimizations of Type Fixing

• On speculative optimization sites, the generated code checks if speculative optimization requirements match, and deoptimizes to the interpreter if not.

• The interpreter supports both regular calls and deoptimized calls.
GC Support

• JWST uses a reference-counting GC to manage memory
  • The GC of QuickJS is used in JWST by default

• JWST manipulates the reference counts for objects
  • In the generated code, JWST needs to fully support the modification of reference counting
try {
    offset(A)
}

// thrower
throw_exception(…)

} catch (...) {
    offset(A)
}

catchoffset

push catchoffset

catchoffset

create exception handling bb

// mimic stack
bottom of the stack

generate the instruction to jump to the bb containing the bytecode specified by catchoffset

// mimic stack for the target basic block

catchoffset

pop the mimic stack to find catchoffset
async/await Support

• When the program is executed to `await`, the function `foo()` pauses the execution.

• After the main function has been completed, the remaining code of the function `foo()` will be executed.

```javascript
async function foo() {
  console.log("foo start");
  console.log(await 1);
  console.log("foo end");
}

console.log("main start");
foo();
console.log("main end");
```

Output:

```
main start
foo start
await 1
foo end
```
Asynchronous in mixed running mode

1. Call asynchronous function
2. Create an asynchronous function data structure
3. Execute the code within the asynchronous function
   - Asynchronous function returns either successfully or with an error.
   - The asynchronous function pauses its execution
4. Terminate the execution of the asynchronous function

Subsequent processing

- The asynchronous function pauses its execution
- Create a Promise object
- Set the resolve function of the Promise object to be the uncompleted asynchronous function (JSAsyncFunctionData)
- Push the Promise to the job queue of the JSRuntime
- Return control flow to the caller
- After the main function has completed its execution, call jwst_poll_rest_job() to execute the remained tasks in the job queue
- Resume function execution from saved stack frames and continue with the uncompleted asynchronous function

Subsequent processing
Typical Optimizations of JWST

• Type fixing
  • already introduced in the previous section

• Inline Cache
  • Caching field offsets for accessing objects

• Array layout optimization
  • For instance, memory can be allocated during the construction of a new Array(10000)

• Global Function Caching
  • Directly calls function without dispatch on functions that can be determined at compile time
Inline Cache

get_field ".baz"
Inline Cache

get_field ".baz"

type: object
shape: 0x1234
n_prototypes: 0
offset: 2
Array layout optimization

- Allow for early allocation of memory and insertion of elements in any order at the cost of slightly slowing down regular array operations
- Avoid array degradation to Hash table

```javascript
arr = new Array(100)
for (let i = 0; i < 100; i += 2) {
    arr[i] = true
}
```
Global Function Caching

• **Most** JS functions do not change their names after definitions.

• The compiler resolves the definitions at compile time, so the generated code calls into the function directly without going through dispatch() or other wrappers.
Outlines

• Challenges of Statically Compiling JavaScript
  • “Dynamic types” & “Dynamic loading”

• JWST——JavaScript to WebAssembly Static Translator
  • Compiler architecture
  • Optimizations
  • Type profiling

• Performance Evaluation
  • ECMA TEST-262： JavaScript language compatibility testing
  • Performance tests: SunSpider
  • Practical cases: React Native Application
## TEST262: language  \((22322/22322/22797,\ JWST/QJS/Total)\)

<table>
<thead>
<tr>
<th>Test sets</th>
<th>Passed (jwst/qjs/all)</th>
<th>Test sets</th>
<th>Passed (jwst/qjs/all)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arguments-object</td>
<td>263/263/263</td>
<td>identifiers</td>
<td>236/236/244</td>
</tr>
<tr>
<td>asi</td>
<td>102/102/102</td>
<td>import</td>
<td>6/6/16</td>
</tr>
<tr>
<td>block-scope</td>
<td>145/145/145</td>
<td>keywords</td>
<td>25/25/25</td>
</tr>
<tr>
<td>comments</td>
<td>46/46/52</td>
<td>line-terminators</td>
<td>41/41/41</td>
</tr>
<tr>
<td>computed-property-names</td>
<td>48/48/48</td>
<td>literals</td>
<td>448/448/448</td>
</tr>
<tr>
<td>destructuring</td>
<td>17/17/17</td>
<td>module-code</td>
<td>329/329/580</td>
</tr>
<tr>
<td>eval-code</td>
<td>347/347/347</td>
<td>reserved-words</td>
<td>27/27/27</td>
</tr>
<tr>
<td>export</td>
<td>3/3/3</td>
<td>rest-parameters</td>
<td>11/11/11</td>
</tr>
<tr>
<td>expressions</td>
<td>10500/10500/10636</td>
<td>source-text</td>
<td>1/1/1</td>
</tr>
<tr>
<td>function-code</td>
<td>217/217/217</td>
<td>statementList</td>
<td>80/80/80</td>
</tr>
<tr>
<td>future-reserved-words</td>
<td>55/55/55</td>
<td>statements</td>
<td>9068/9068/9131</td>
</tr>
<tr>
<td>global-code</td>
<td>40/40/41</td>
<td>types</td>
<td>113/113/113</td>
</tr>
<tr>
<td>identifier-resolution</td>
<td>14/14/14</td>
<td>white-space</td>
<td>67/67/67</td>
</tr>
</tbody>
</table>
TEST262: Built-ins (15905/16110/23373, JWST/QJS/Total)

- 204 test cases related to **Atomic** require multithreading support, while WASM runtime does not support multithreading currently.

- 1 test cases related to floating-point precision
  - The floating-point rounding modes provided by WASM are less than native mode.

- JWST native and QJS floating-point precisions are consistent
Floating-Point Rounding modes

• Unsupported Rounding Modes of WASM
  • WebAssembly only supports round-to-nearest.
  • WASI-libc only defines FE_TONEAREST.

<table>
<thead>
<tr>
<th>Rounding Modes</th>
<th>WASM</th>
<th>WASI-LIBC</th>
<th>Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE_TONEAREST</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FE_DOWNWARD</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>FE_UPWARD</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>FE_TOWARDZERO</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

```javascript
let left = (25).toExponential(0);
let right = "3e+1";

// expected to be true, but it's false here
// due to the unsupported rounding modes
// left turns to 2e+1
assert(left === right);
```
The running time of Node.JS (18.16.0) includes all initialization and reads, and enabling all caching functions.

- **CPU**: Kirin 990E, 2×Cortex-A76 2.86GHz+2×Cortex-A76 2.36GHz+4×Cortex-A55 1.95GHz
- **WASM code** generated by JWST running on wasmer2.3.0( LLVM JIT).
- **21 of 26 test cases**, JWST is faster than Node.js, while its total performance is ~50% faster than Node.js, for the first execution.
SunSpider@wasmer, Intel i7

- JWST is ~30% faster than Node.js on i7-12700K, for the first execution.
React Native Samples (TTI, 0.1x play)
React Native Samples (0.1x play)
Evaluate the Start-up JS Scripts of a RN APP

JWST-WASM/JWST-Native

V8
Practical cases: React Native App (v0.68.2)

- For the start-up script of a release-mode RN app, JWST-WASM is ~10.5x and ~126% faster than V8 without code cache and with code cache, respectively, on Kirin990E CPUs.
- In terms of the Time-to-Interactive (TTI) of a release-mode RN app, JWST-WASM is slower than V8. However, JWST-native is ~30% faster than V8 for the first execution.
## React Native Samples: Compilation Time and Code Size

<table>
<thead>
<tr>
<th>Size</th>
<th>JSBundle</th>
<th>WASM(no-link)</th>
<th>JWST Runtime</th>
<th>WASM(linked)</th>
<th>llvm-strip WASM(linked)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev</td>
<td>4.9MB</td>
<td>46MB</td>
<td>5.1MB</td>
<td>45MB</td>
<td>41MB</td>
</tr>
<tr>
<td>Release</td>
<td>1.3MB</td>
<td>27MB</td>
<td>5.1MB</td>
<td>28MB</td>
<td>25MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compilation Time</th>
<th>JWST (JSBundle =&gt; LLVM IR)</th>
<th>llc (LLVM IR =&gt; WASM)</th>
<th>wasm-ld (linking)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev</td>
<td>1600s (26min40s)</td>
<td>490s (8min10s)</td>
<td>0.13s</td>
</tr>
<tr>
<td>Release</td>
<td>640s (10min40s)</td>
<td>200s (3min20s)</td>
<td>0.08s</td>
</tr>
</tbody>
</table>

- CPU: Intel Core i9-12900KF 3.20GHz
- Memory: 32G
Expectations for WASM

• 1. DLL Support
  • If WASM has a dynamic loading mechanism similar to DLL, which could significantly reduce the pre-compilation and loading time of WASM runtimes.

• 2. GC Support
  • Supporting GC at the WASM bytecode level can reduce the size of RC instructions in the WASM code generated by the compiler.

• 3. DOM Support
  • When could we directly call DOM APIs from WASM?

• 4. Furthermore, is it possible to use WASM as the fundamental language/bytecode for web applications?
  • Running JavaScript and all other languages on WASM runtimes
• Thanks!