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

```
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

```
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:

```
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:

```
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

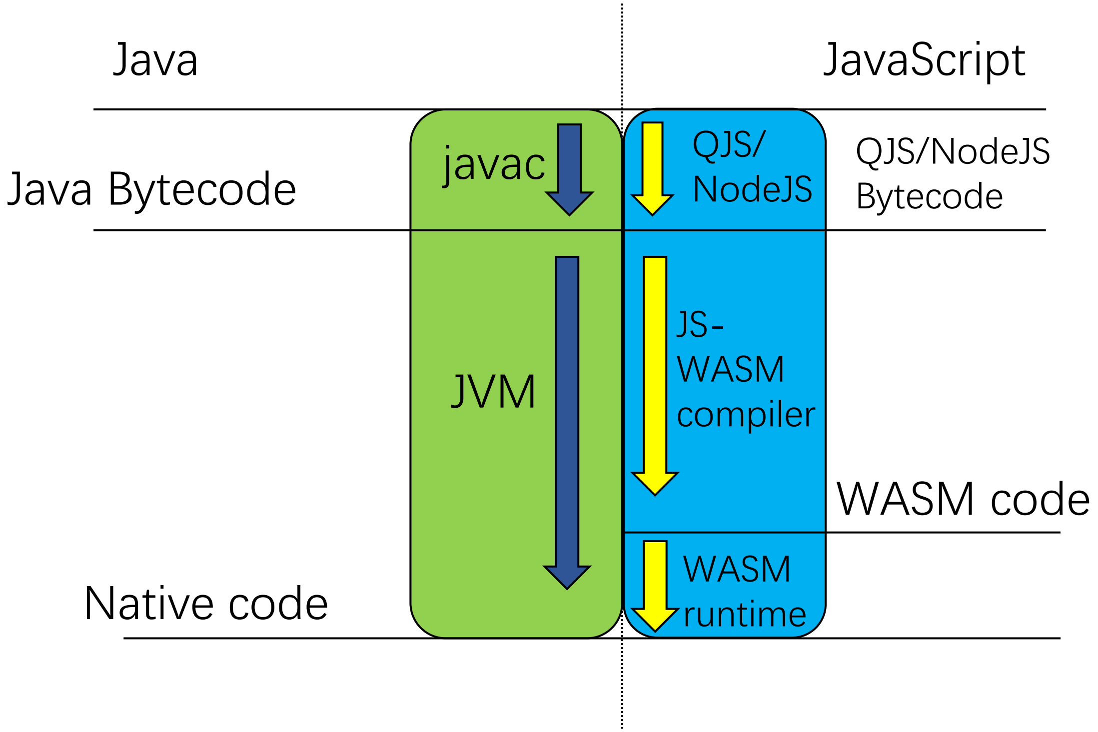
C++	WASM Binary	WASM Text

int factorial(int n) {
if (n == 0)
return 1;
else
return n * factorial(n-1);
}
	20 00	get_local 0

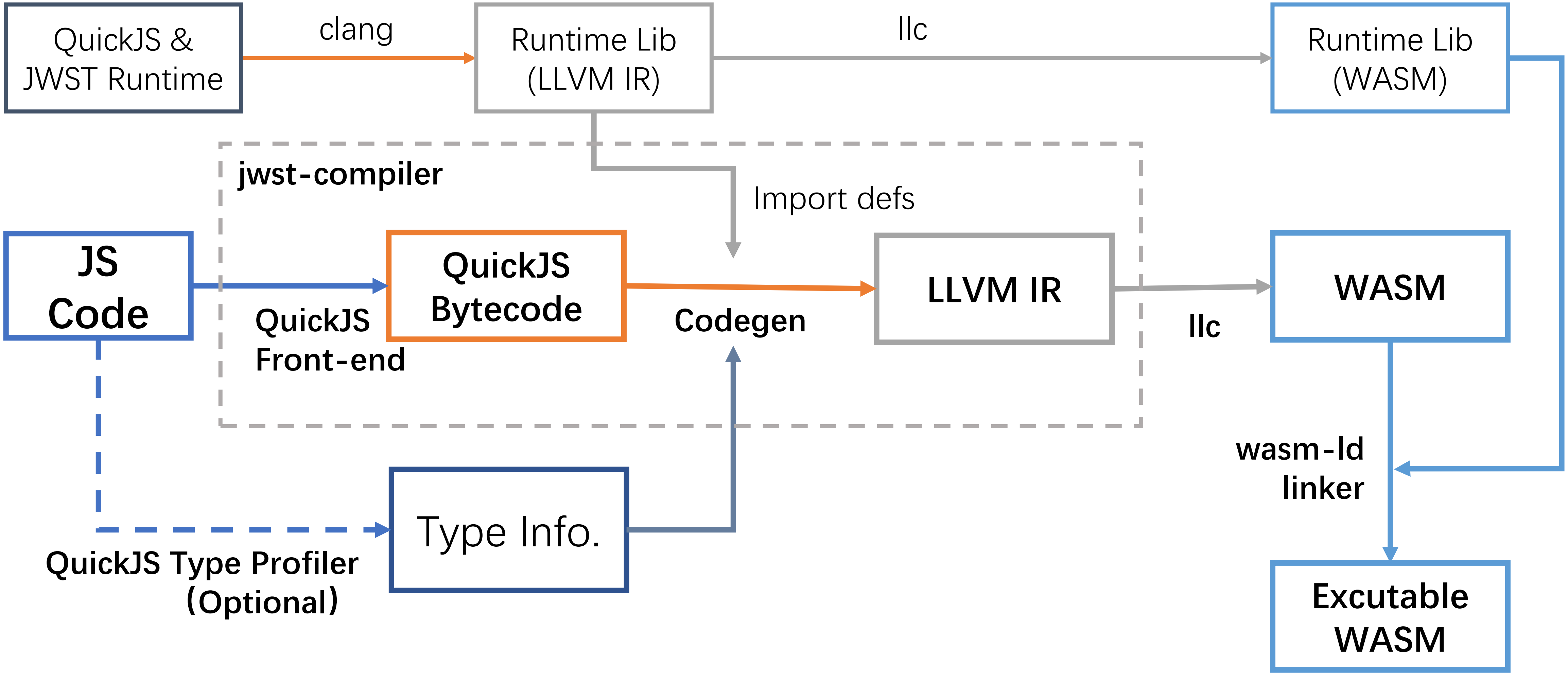
	42 00	i64.const 0
	51	i64.eq
	04 7e	if i64
	42 01	i64.const 1
	05	else
	20 00	get_local 0
	20 00	get_local 0
	42 01	i64.const 1
	7d	i64.sub
	10 00	call 0
	7e	i64.mul
	0b	end

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



Compilation process demonstration(1)

QuickJS Bytecode

JavaScript code:

```
function foo(a, b) {
  return a + b + 10086;
}

foo(1, 2);
foo(10.2, 44.56);
```

```
-- 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: 0x60400001d750, 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 @ 0x60f000000310
source ptr: 0x60400001d750, 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
```

Prolog

```
; Function Attrs: nounwind
define i64 @_JS_F3foo__root.0(
  i64 %_fn_obj_arg,          ; 函数对象
  i64 %_this_arg,            ; this 对象
  i64 %a_0_arg,              ; 参数
  i64 %b_0_arg
) local_unnamed_addr #9 {
entry:
; 分配函数调用空间
  %call_scratch22 = alloca [2 x i64], align 8
  %call_scratch22.sub = getelementptr …

; 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 --
```

Epilog

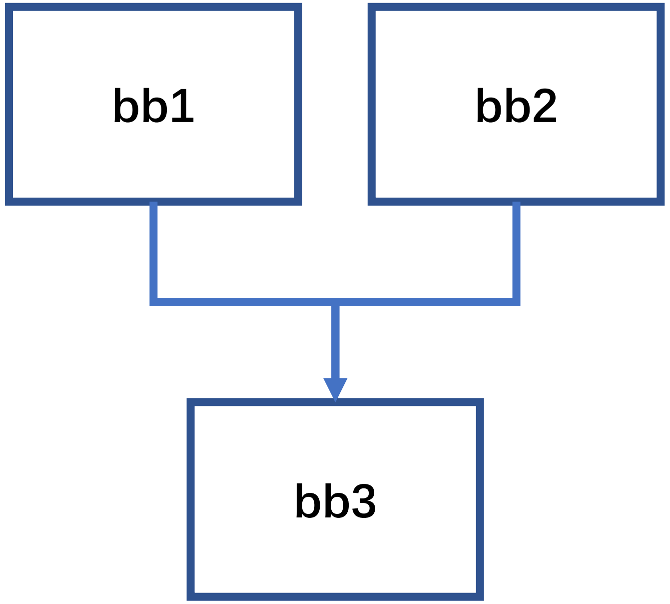
```
ret1:
  %46 = phi i64 …
  call void @jwst_close_var_refs(…)
  call void @JS_FreeValue(%struct.JSContext* %ctx, i64 %a_0_arg)
  call void @JS_FreeValue(%struct.JSContext* %ctx, i64 %b_0_arg)
  ret i64 %46
}
```

Compilation process demonstration(3)

Codegen by a Mimic Stack

	Mimic Stack	Generated Pseudo Code
`0: get_arg0`	`%a1`	`%a1 = load %a`
`1: get_arg1`	`%a1` `%b1`	`%b1 = load %b`
`2: add`	`%2`	`%2  = add_any(%a1, %b1)`
`3: push_i16 10086`	`%2` `i32 10086`	
`6: add`	`%6`	`%6  = add_any(%2, i32 10086)`
`7: return`		`store %6 to %ret` `br label %ret_bb`

Mimic stack at Basic Block Merge Point



Mimic Stack

bb1:

var1
114
var3

Mimic Stack

bb2:

var2
var4
514

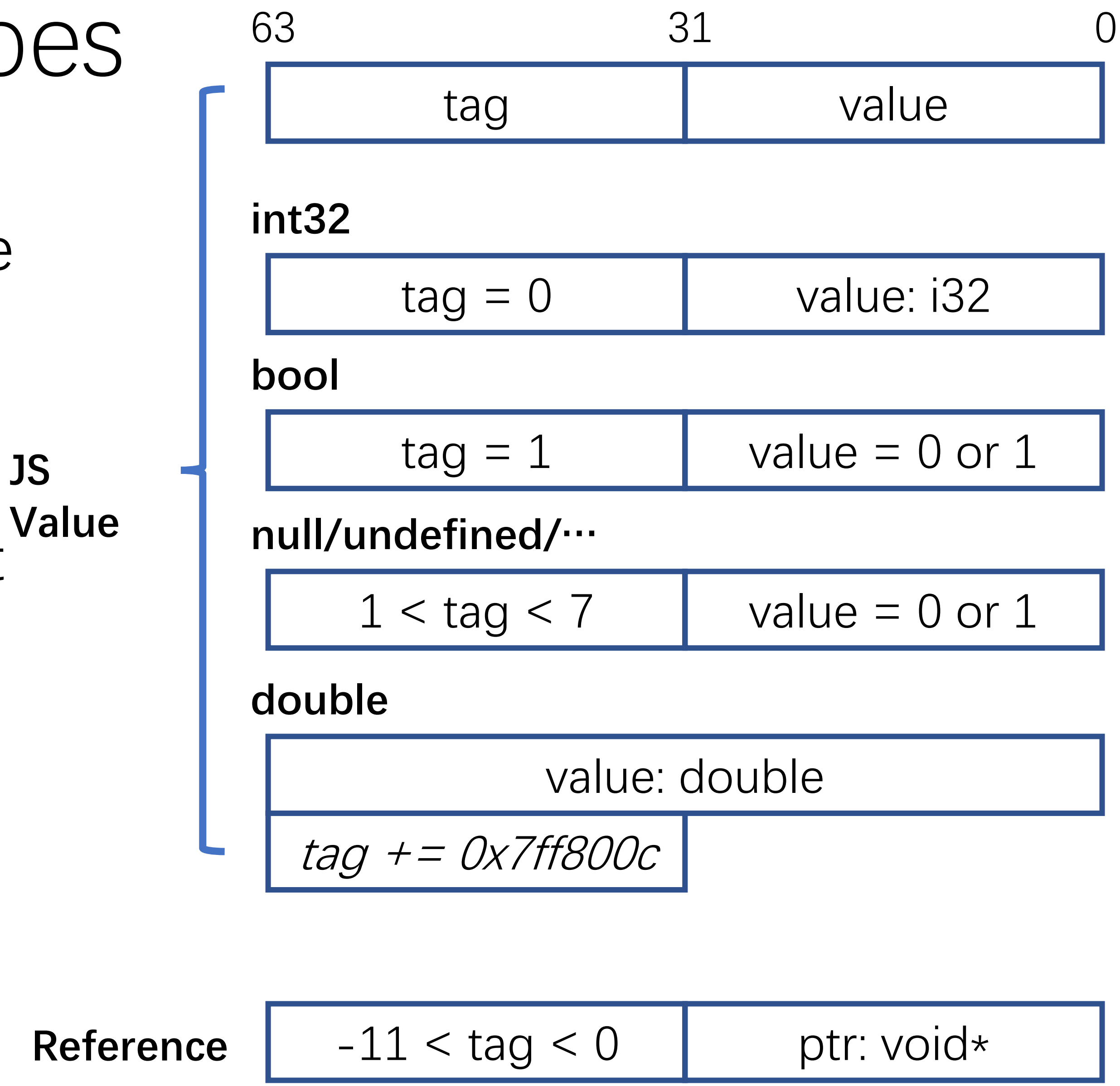
Mimic Stack

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



63 31 0

tag | value

JS Value

int32

tag = 0 | value: i32

bool

tag = 1 | value = 0 or 1

null/undefined/…

1 < tag < 7 | value = 0 or 1

double

value: double

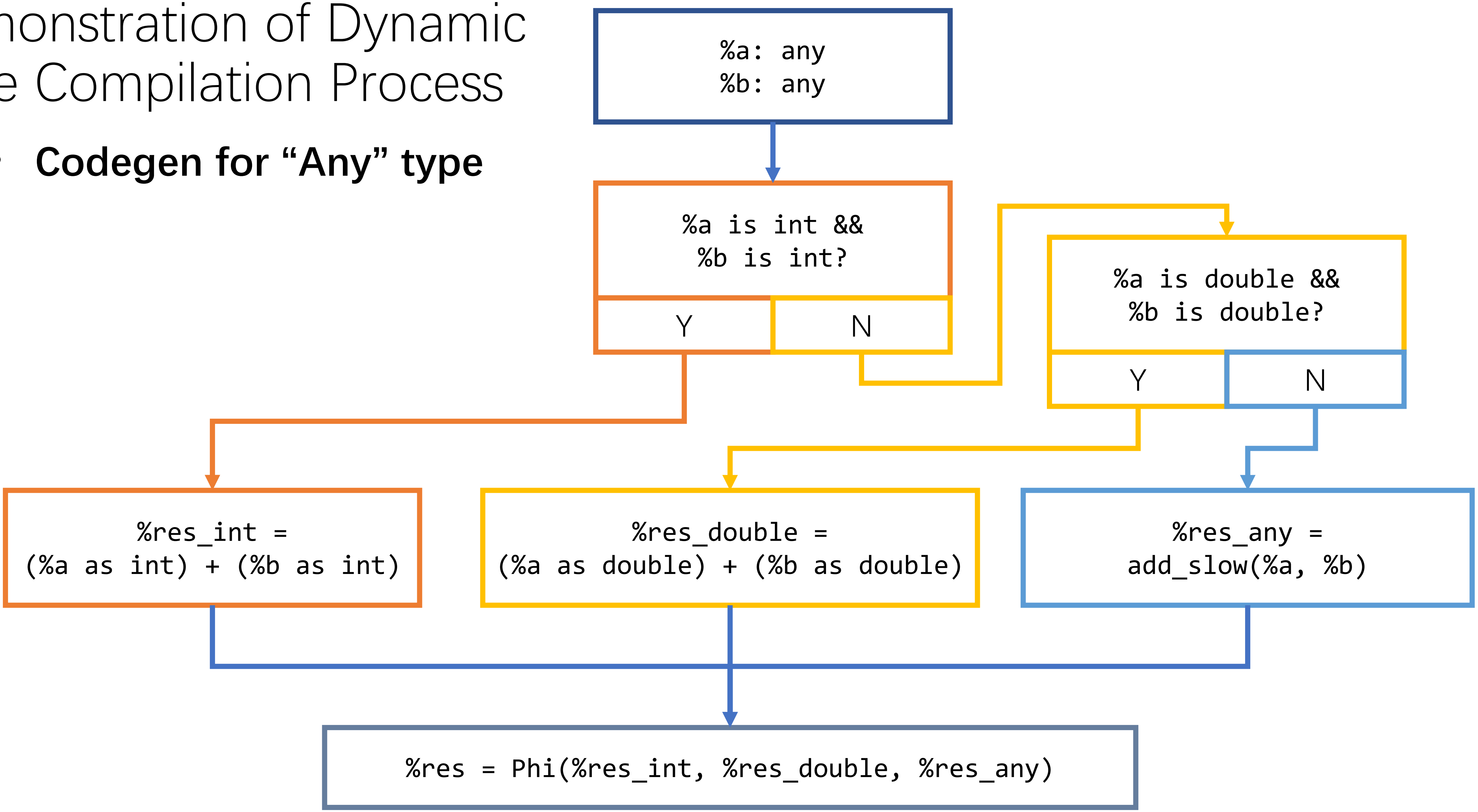
tag += 0x7ff800c

Reference

-11 < tag < 0 | ptr: void*

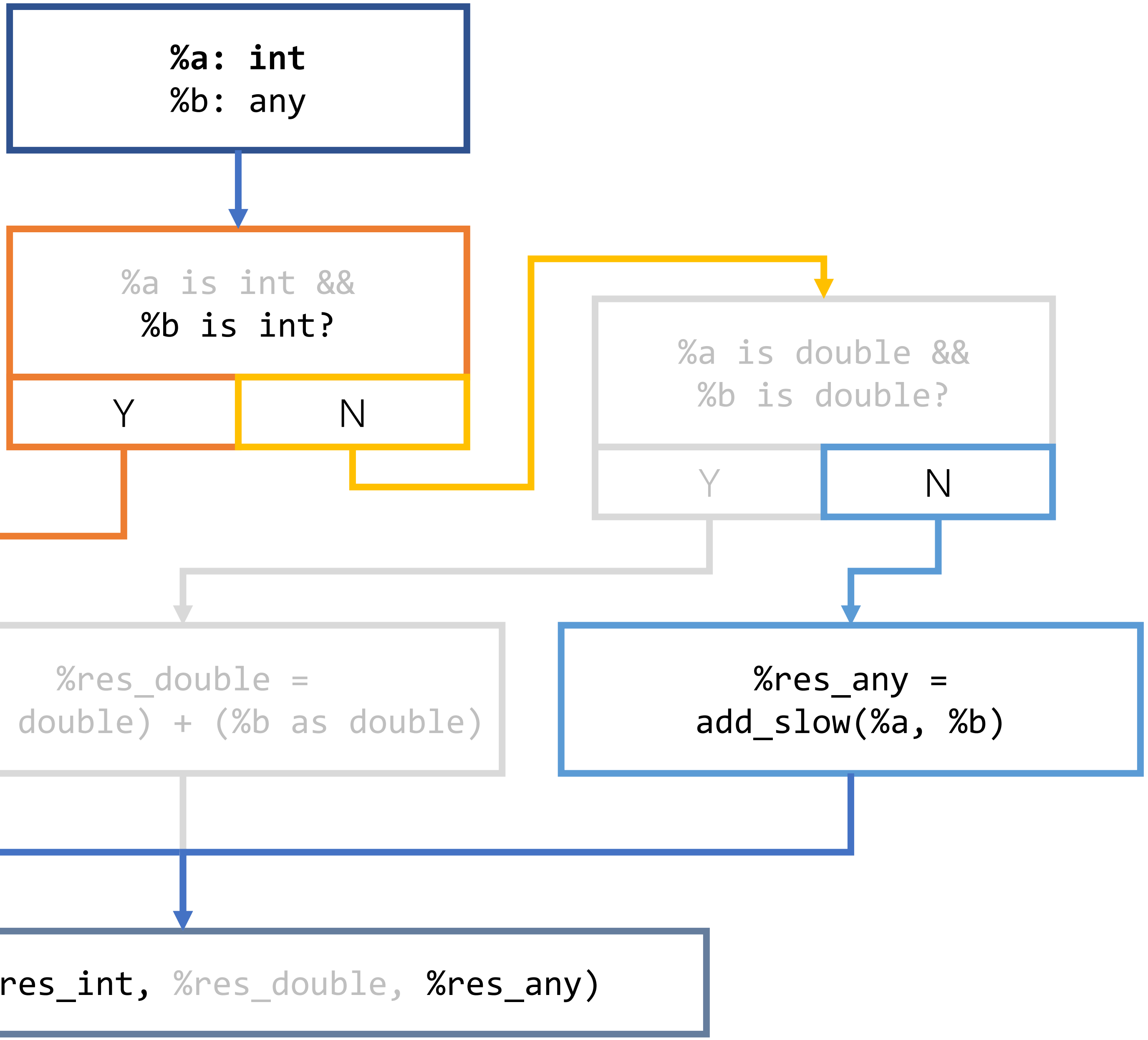
Demonstration of Dynamic Type Compilation Process

- **Codegen for "Any" type**



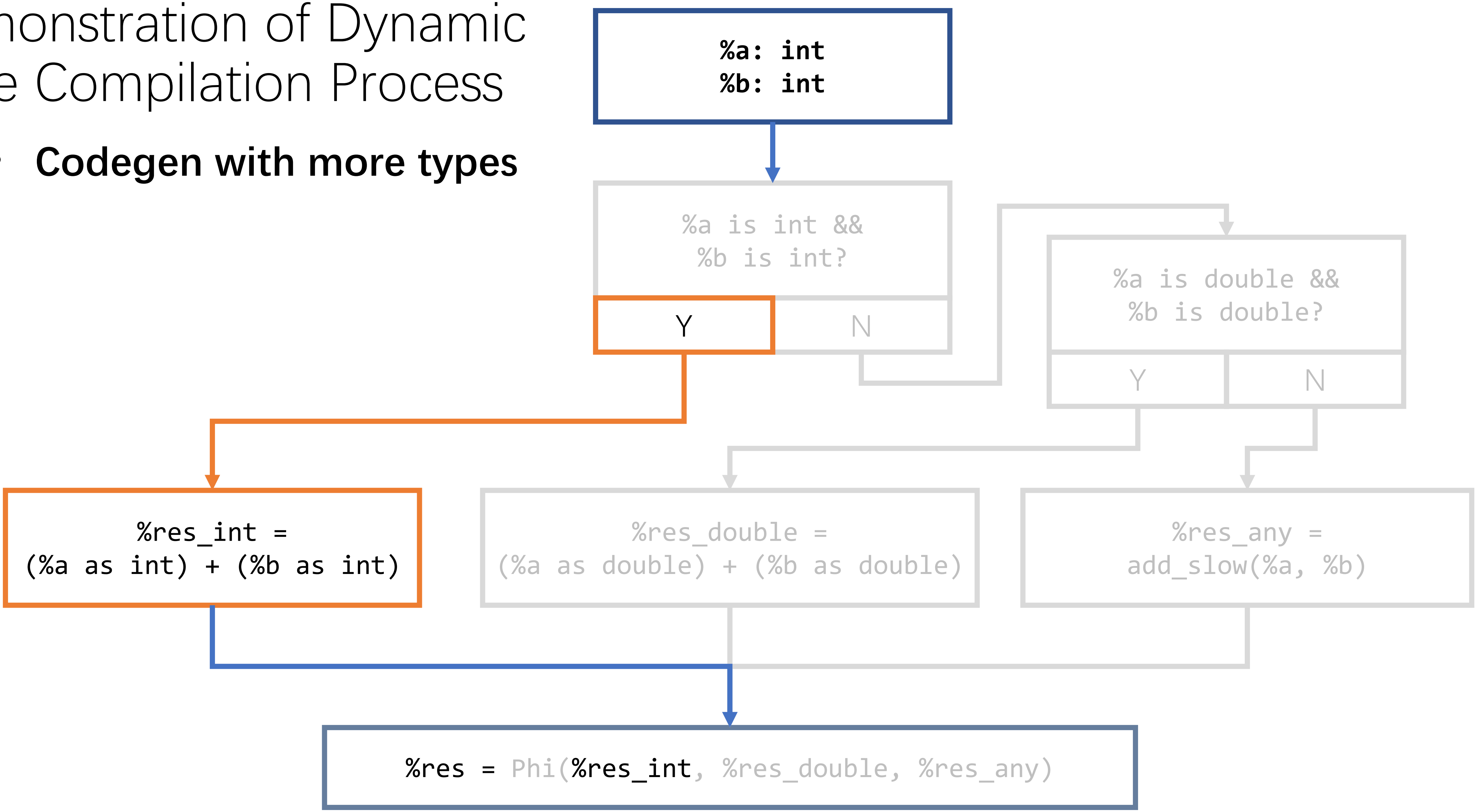
Demonstration of Dynamic Type Compilation Process

- **Codegen with types**



Demonstration of Dynamic Type Compilation Process

- **Codegen with more types**



%a: int
%b: int

%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)

Type Profiling: Type information fusion

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

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

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

bytecode	**type v1,hit 10**		**type v2,hit 10**		**type sum up,hit 20**	
	operands	**result**	**operands**	**result**	**operands**	**result**
get_arg0 0: n		INT		INT		INT
push_i8 10		INT		INT		INT
lt	INT, INT	BOOL	INT, INT	BOOL	INT, INT	BOOL
if_false8 9	BOOL		BOOL		BOOL	
push_const8 0: 1"1"				STRING		STRING
return			STRING		STRING	
9: push_0 0		INT				INT
return	INT				INT	

Type Profiling: Example

```
function fib(n) {
  if (n === 0 || n === 1) return 1;
  return fib(n - 1) + fib(n - 2);
}
fib("20");
```

```
fib:    0: get_arg0             14: get_var   'fib'
        1: push_0               19: get_arg0
        2: strict_eq            20: push_1
        3: dup                  21: sub
        4: if_true8  5          22: call1
        6: drop                 23: get_var   'fib'
        7: get_arg0             28: get_arg0
        8: push_1               29: push_2
        9: strict_eq            30: sub
        10: if_false8  3        31: call1
        12: push_1              32: add
        13: return              33: return
```

Testing source code and bytecode

Bytecode	v1 (hit = 1)		v2 (hit = 10944)		v3 (hit = 4181)		v4 (hit = 6765)	
	operand	result	operand	result	operand	result	operand	result
0		S		I		I		I
1		I		I		I		I
2	S I	B	I I	B	I I	B	I I	B
3	B	B B	B	B B	B	B B	B	B B
4	B		B		B		B	
6	B		B		▨	▨	B	
7		S		I	▨	▨		I
8		I		I	▨	▨		I
9	S I	B	I I	B	▨	▨	I I	B
10		B		B		B		B
12	▨	▨	▨	▨		I		I
13	▨	▨	▨	▨	I		I	
14		O		O	▨	▨	▨	▨
19		S		I	▨	▨	▨	▨
20		I		I	▨	▨	▨	▨
21	S I	I	I I	I	▨	▨	▨	▨
22	I	I	I	I	▨	▨	▨	▨
23		O		O	▨	▨	▨	▨
28		S		I	▨	▨	▨	▨
29		I		I	▨	▨	▨	▨
30	S I	I	I I	I	▨	▨	▨	▨
31	I	I	I	I	▨	▨	▨	▨
32	I I	I	I I	I	▨	▨	▨	▨
33	I		I		▨	▨	▨	▨

QuickJS Profiling resules

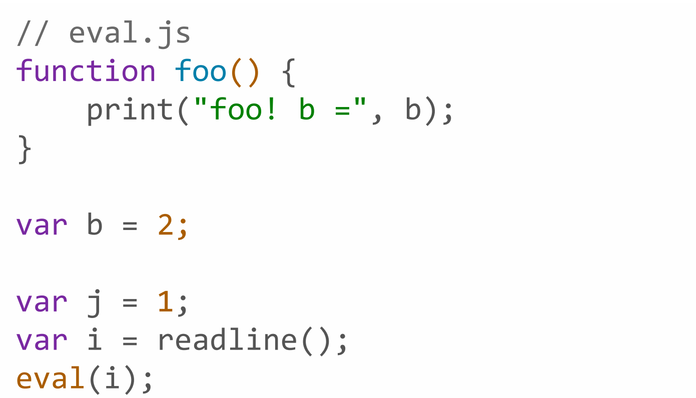
Note: S=string, I=int, B=bool, O=object, The gray mark indicates that the control flow has not passed through this bytecode.

```
(fn _JS_F3fib__703c
(sig (args undefined
    i32
    )
  i32)
  (locs
  )
  (bytecodes
    (0 i32)
    (1 i32)
    (2 i32)
    (3 bool)
    (5 i32)
    (6 any)
    (7 i32)
    (8 i32)
    (9 i32)
    (10 bool)
    (12 i32)
    (13 any)
    (14 object)
    (19 i32)
    (20 i32)
    (21 i32)
    (22 i32)
    (23 object)
    (28 i32)
    (29 i32)
    (30 i32)
    (31 i32)
    (32 i32)
    (33 any)
  )
)
```

Type Info Interface

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.



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

var b = 2;

var j = 1;
var i = readline();
eval(i);
```

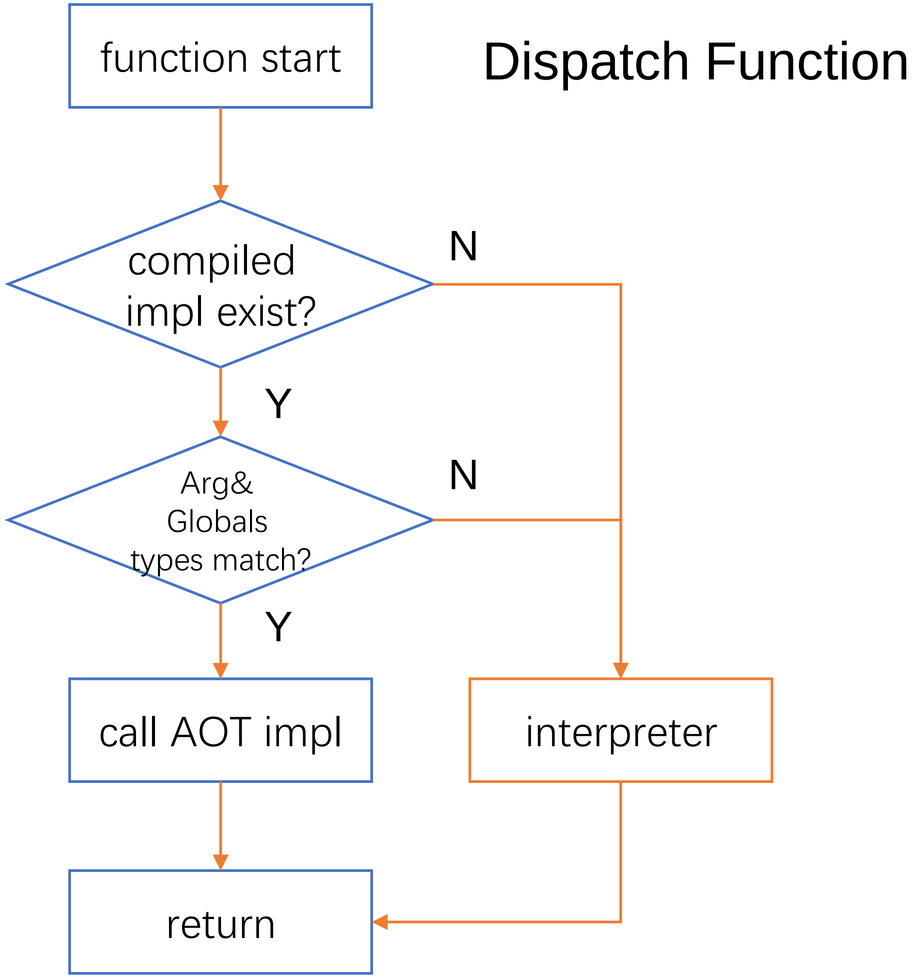
//readline is a temporary function added to demonstrate this feature

```
$ 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.

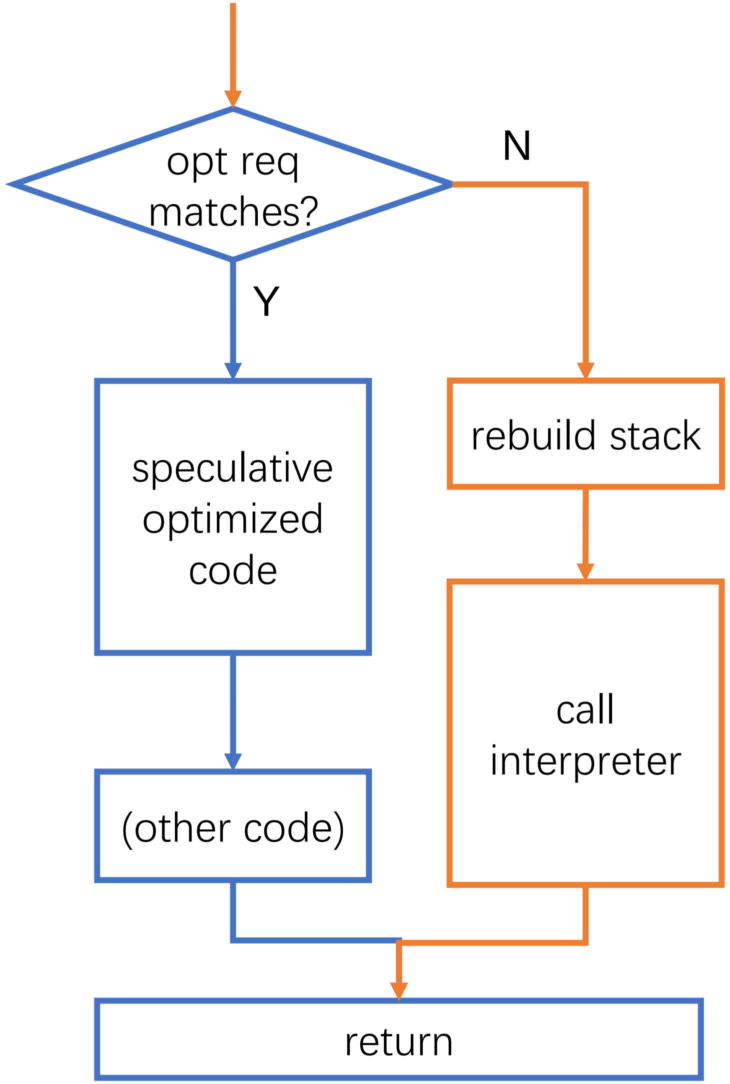


Dispatch Function

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.



1. Allocate interpreter state in stack

%stack =	**alloca**(stack_size)
%locs =	**alloca**(local_size)

2. Store current expected interpreter state

store	%stack[0]	%x
store	%stack[1]	%a
store	%locs[0]	%a

3. Call interpreter using saved state

%result =	**call**	interpreter	
	%stack	%locs	ip=100

return	%result

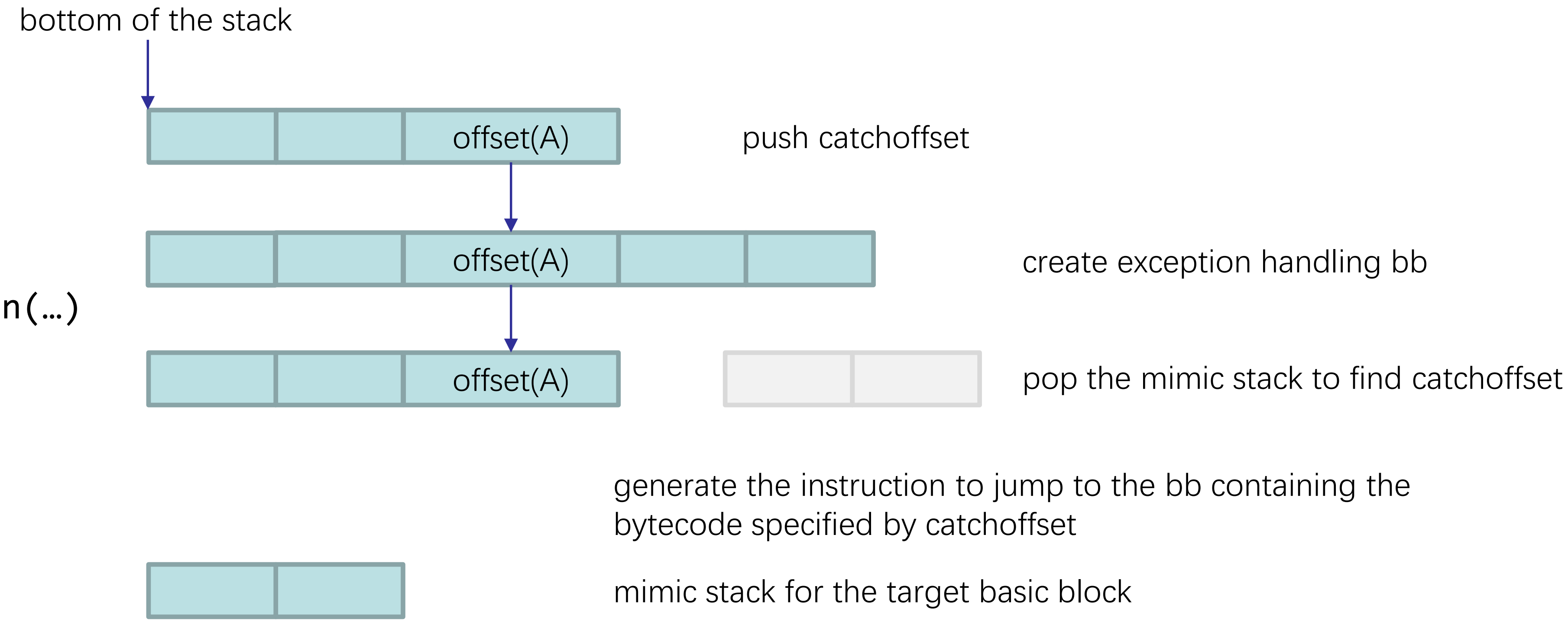
GC Support

- 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

Exception Support

```
try {


  // thrower
  throw_exception(…)

} catch (…) {



}
```



push catchoffset

create exception handling bb

pop the mimic stack to find catchoffset

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

mimic stack for the target basic block

async/await Support

- When the program is executed to ***await***, the function ***foo()*** pauses the execution.

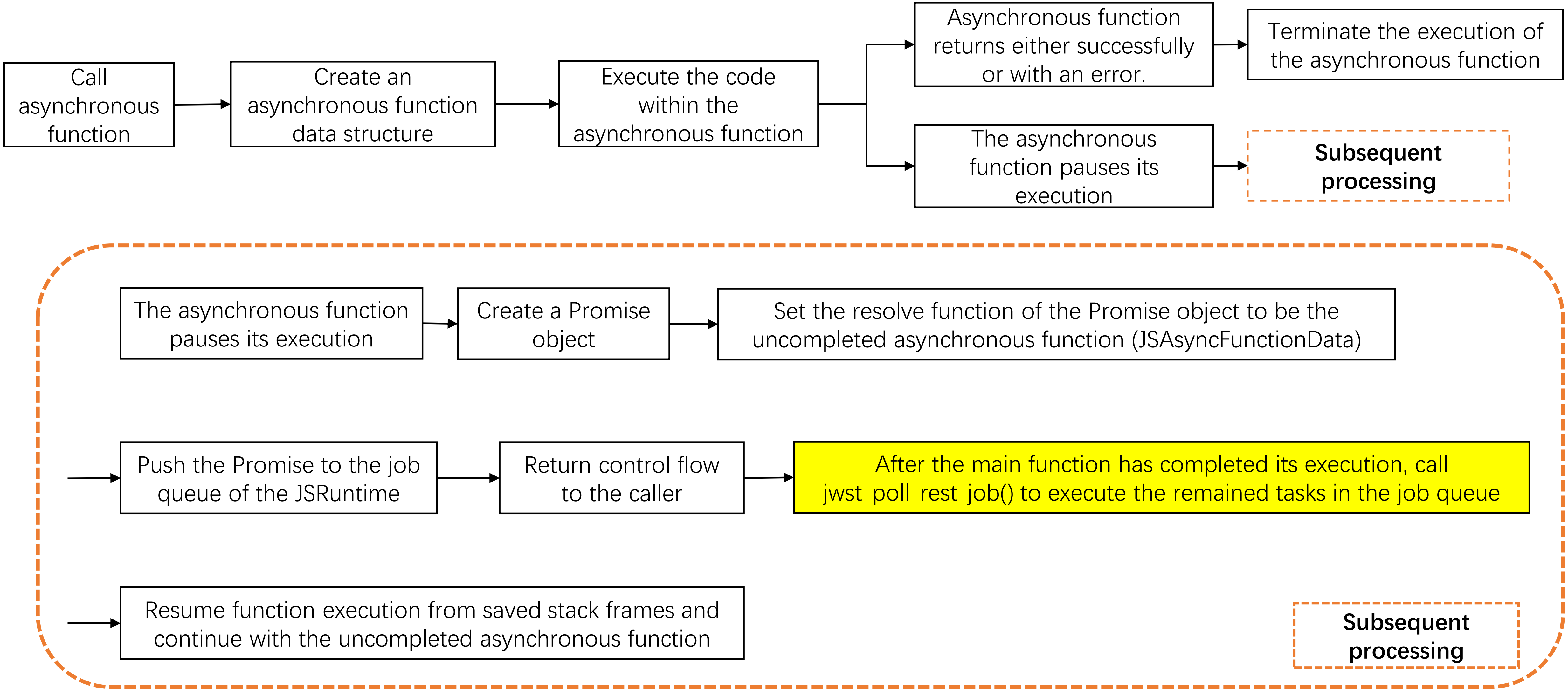
- After the main function has been completed, the remaining code of the function ***foo()*** will be executed.

```
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
main end
1
foo end
```

Asynchronous in mixed running mode



Call asynchronous function

Create an asynchronous function data structure

Execute the code within the asynchronous function

Asynchronous function returns either successfully or with an error.

Terminate the execution of the asynchronous function

The asynchronous function pauses its execution

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

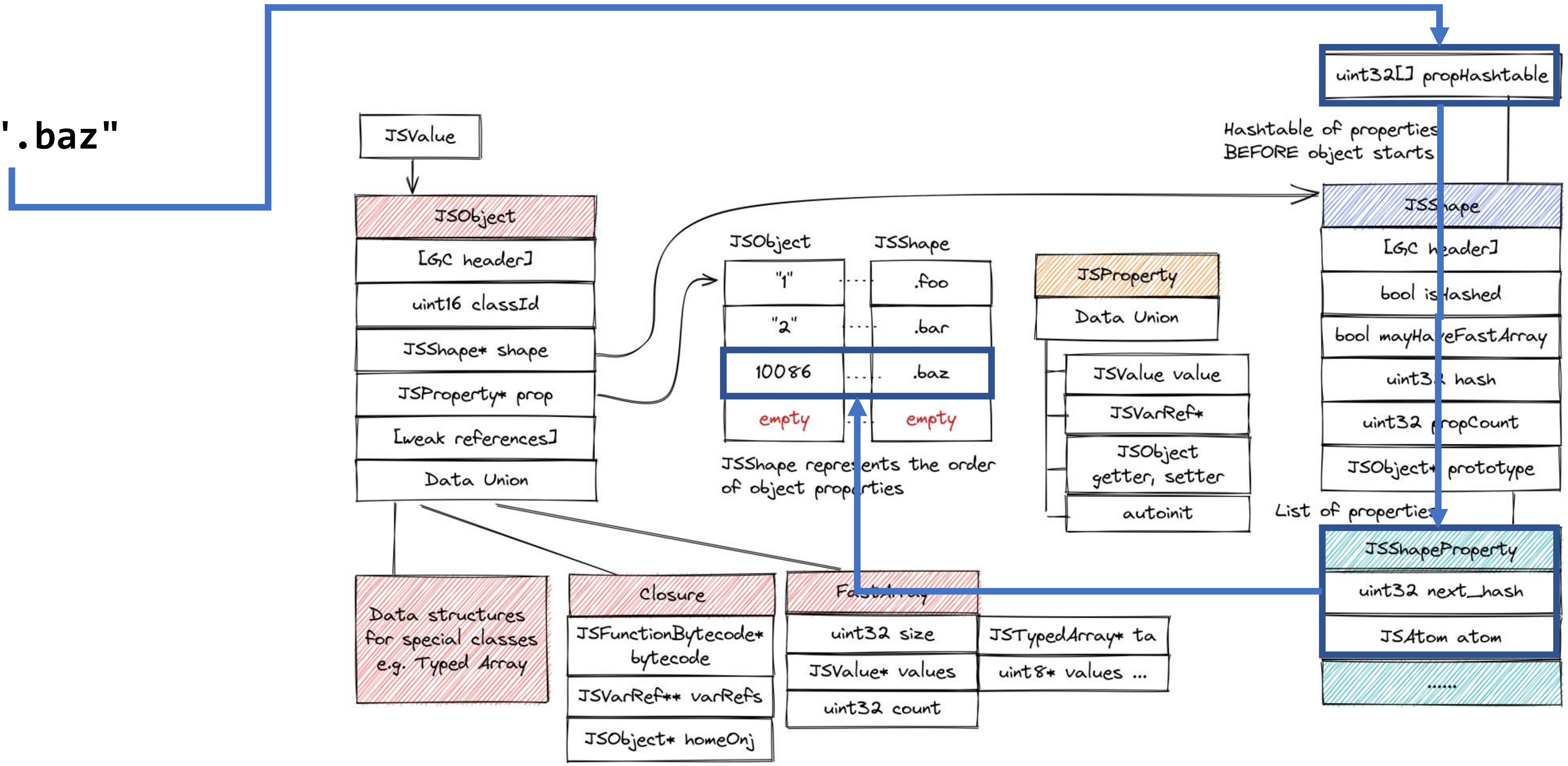
Subsequent processing

Typical Optimizations of JWST

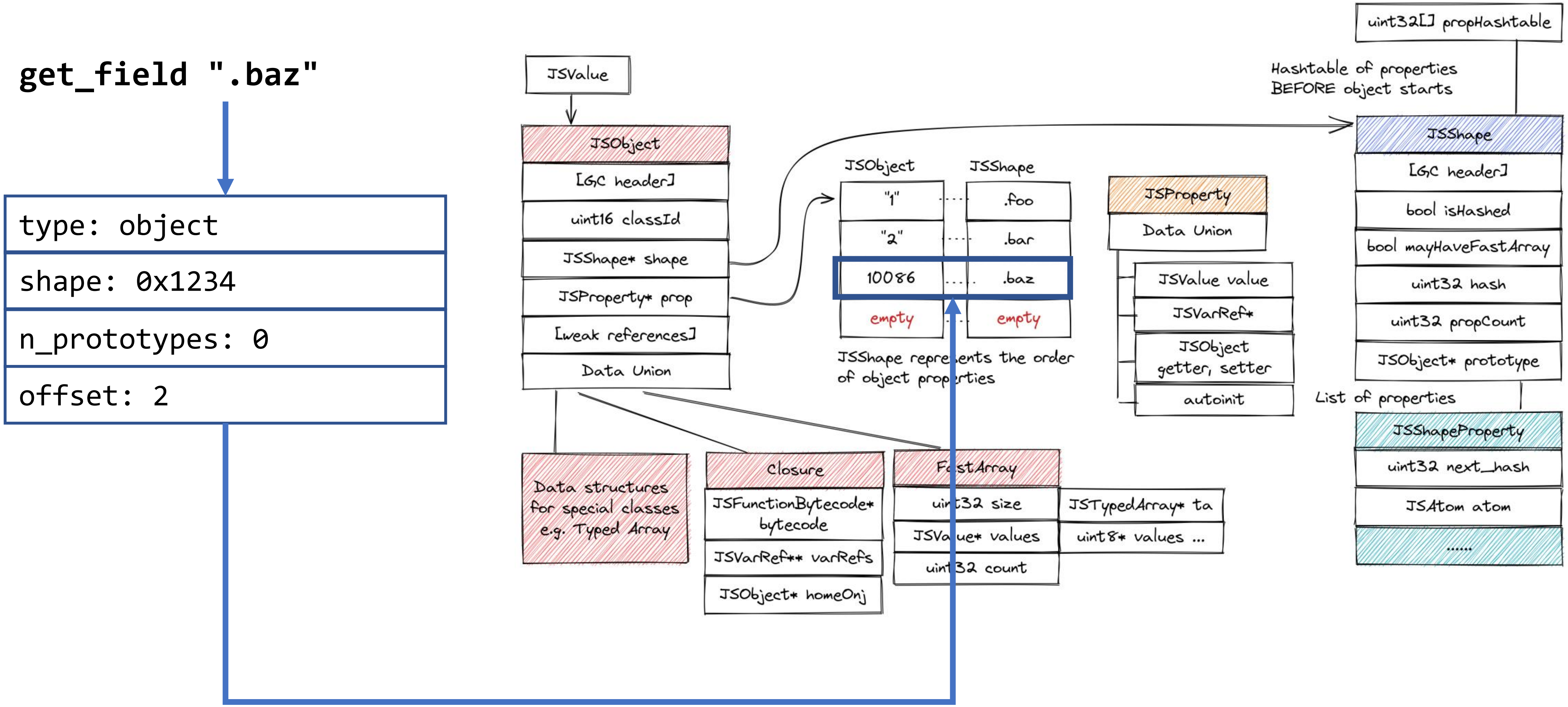
- Type fixing
  - already introduced in the previous section
- Inline Cache
  - Caching field offsets for accessing objects
- Array layout optimization
  - For instance, memory can be allocated during the construction of a new Array(10000)
- Global Function Caching
  - Directly calls function without dispatch on functions that can be determined at compile time

Inline Cache

get_field ".baz"



Inline Cache



get_field ".baz"

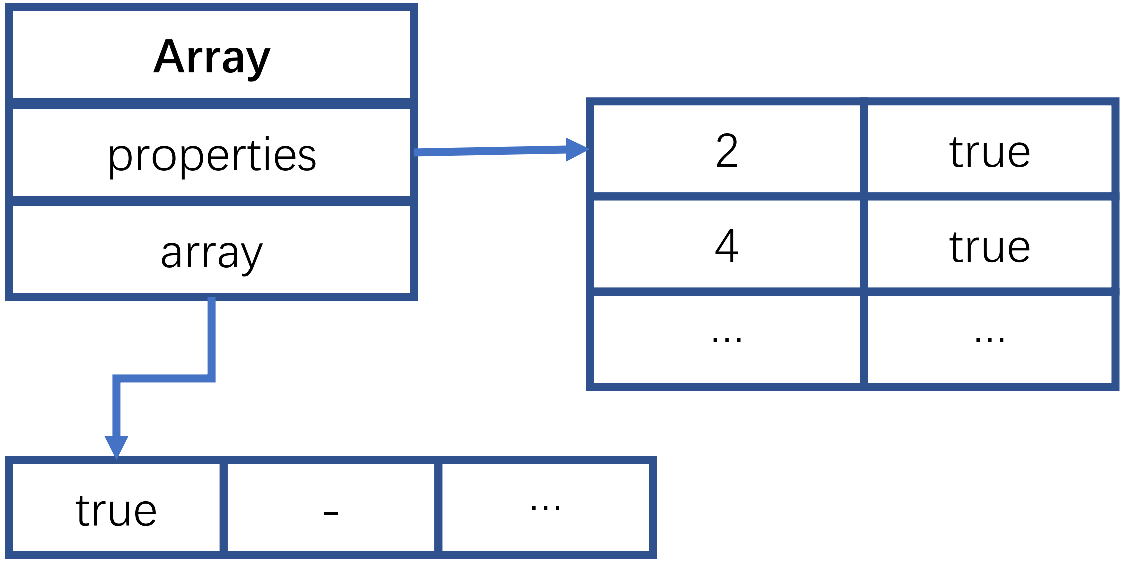
```
type: object
shape: 0x1234
n_prototypes: 0
offset: 2
```

Array layout optimization

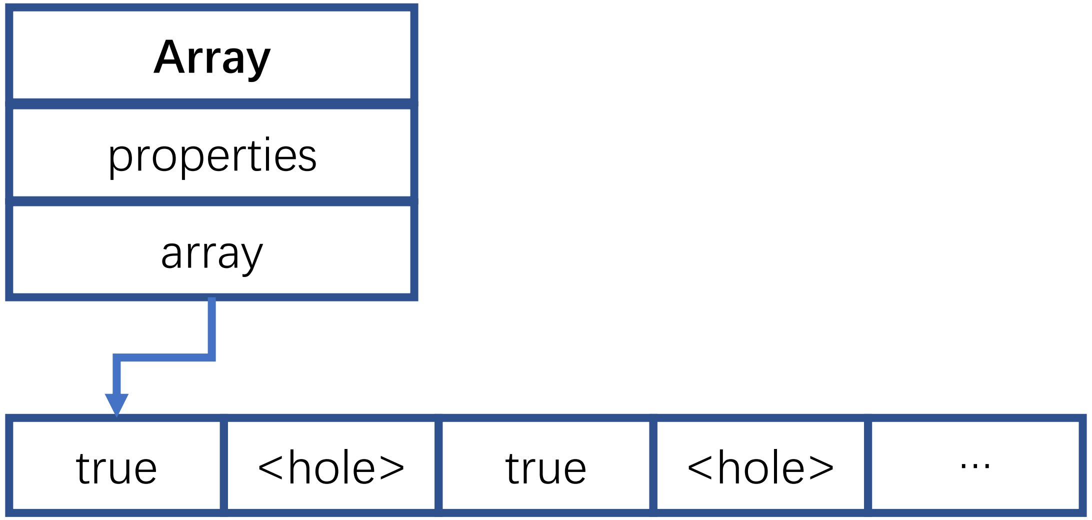
- Allow for early allocation of memory and insertion of elements in any order at the cost of slightly slowing down regular array operations
- Avoid array degradation to Hash table

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

Before optimization



After optimization



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.

```
function foo() {
  // …
}

function bar() {
  foo()
  // ^ Speculates that
  // The `foo()` function
  // is the one defined above
}
```

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）

Test sets	Passed (jwst/qjs/all)	Test sets	Passed (jwst/qjs/all)
arguments-object	263/263/263	identifiers	236/236/244
asi	102/102/102	import	6/6/16
block-scope	145/145/145	keywords	25/25/25
comments	46/46/52	line-terminators	41/41/41
computed-property-names	48/48/48	literals	448/448/448
destructuring	17/17/17	module-code	329/329/580
directive-prologue	62/62/62	punctuators	11/11/11
eval-code	347/347/347	reserved-words	27/27/27
export	3/3/3	rest-parameters	11/11/11
expressions	10500/10500/10636	source-text	1/1/1
function-code	217/217/217	statementList	80/80/80
future-reserved-words	55/55/55	statements	9068/9068/9131
global-code	40/40/41	types	113/113/113
identifier-resolution	14/14/14	white-space	67/67/67

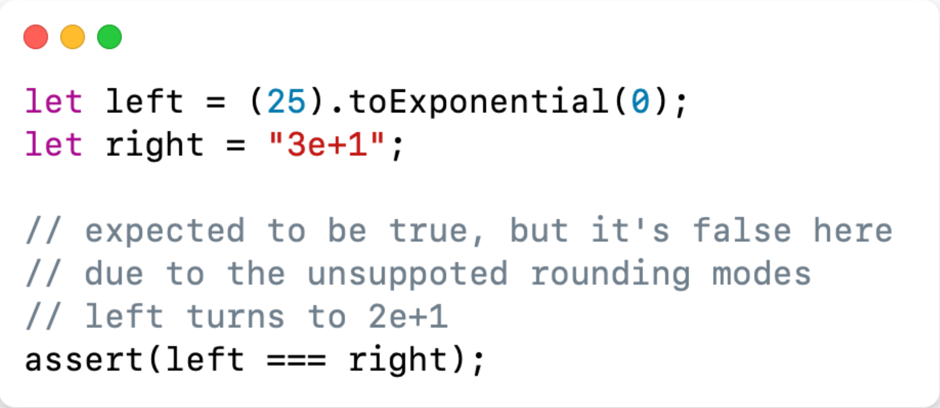
TEST262: Built-ins (15905/16110/23373, JWST/QJS/Total)

- 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.

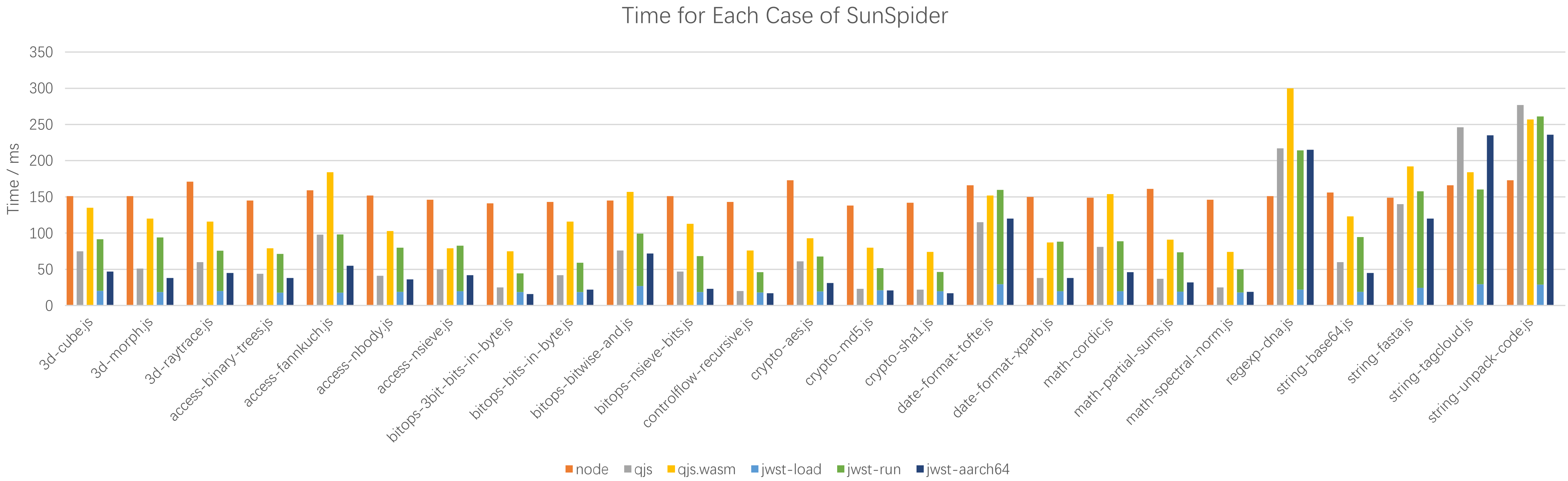
Rounding Modes	WASM	WASI-LIBC	Native
FE_TONEAREST	√	√	√
FE_DOWNWARD	×	×	√
FE_UPWARD	×	×	√
FE_TOWARDZERO	×	×	√



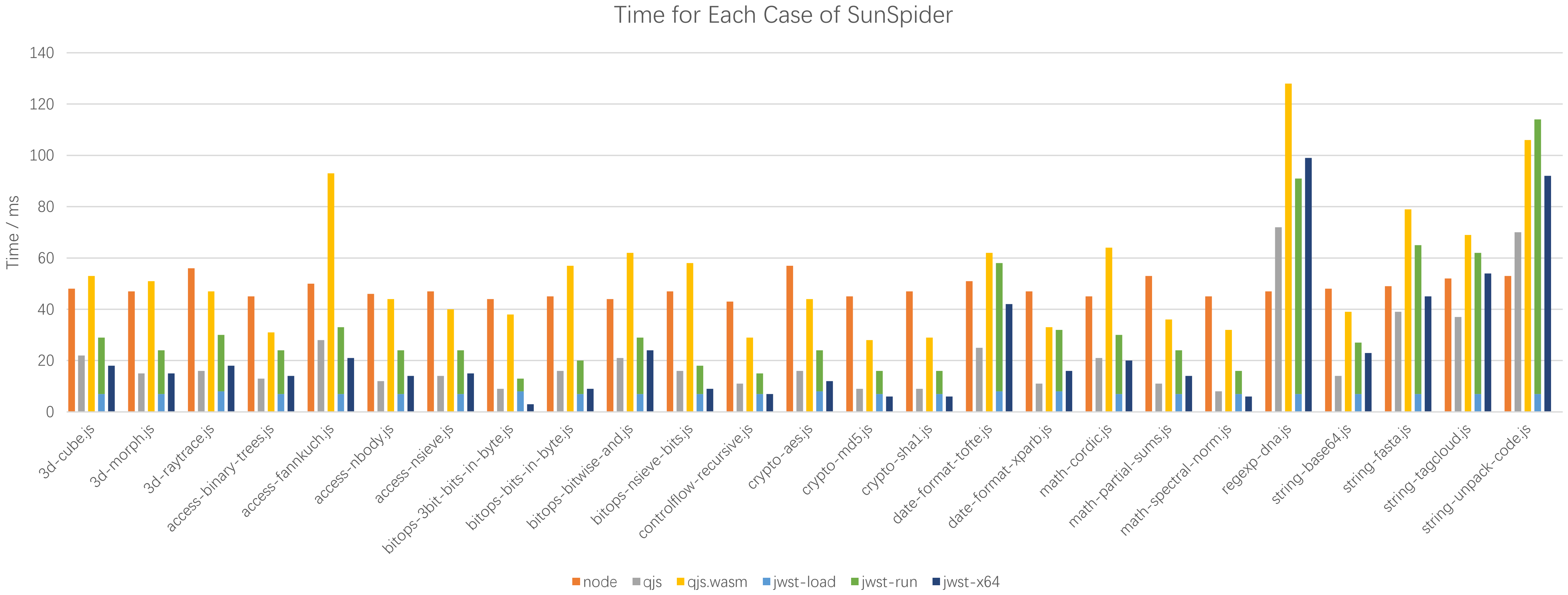
```
let left = (25).toExponential(0);
let right = "3e+1";

// expected to be true, but it's false here
// due to the unsuppoted rounding modes
// left turns to 2e+1
assert(left === right);
```

SunSpider@**wasmer, Kirin 990E**

- 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**

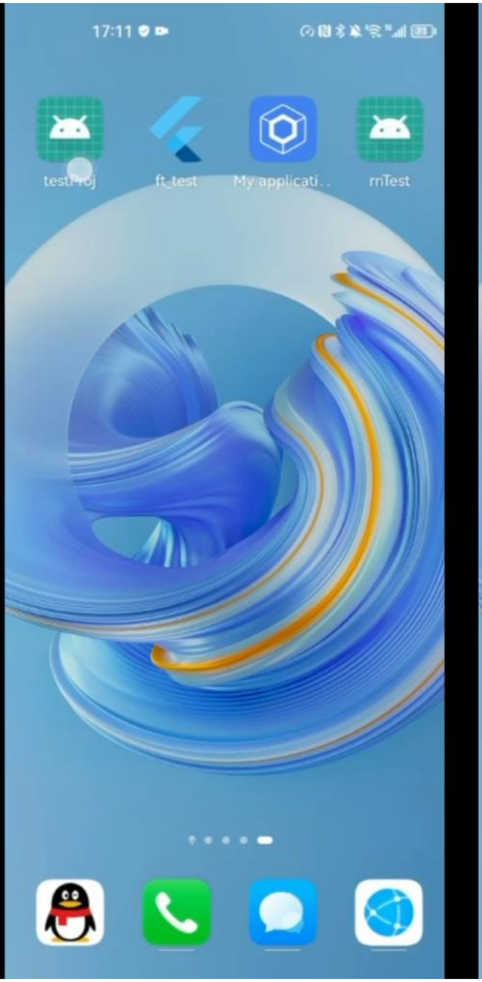


- JWST is ~30% faster than Node.js on i7-12700K, for the first execution.

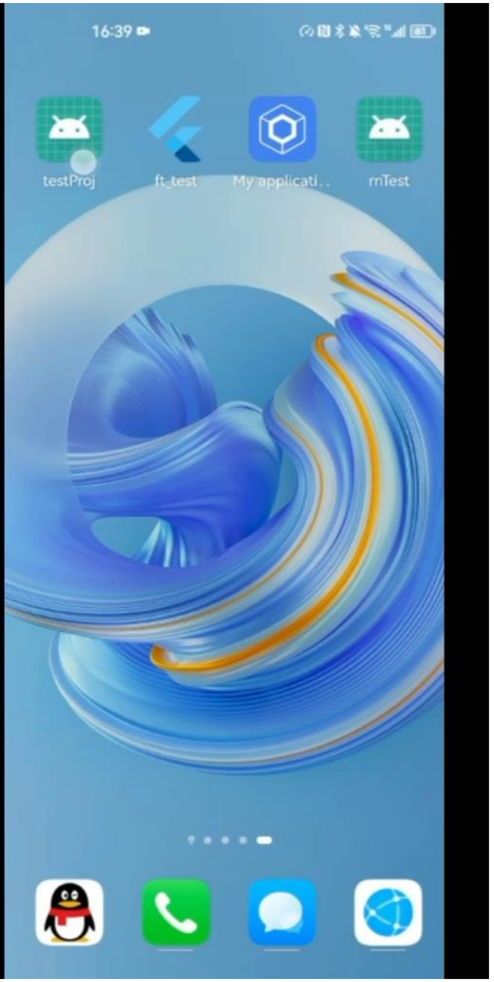
React Native Samples (TTI, 0.1x play)

V8-Cache

JWST Native

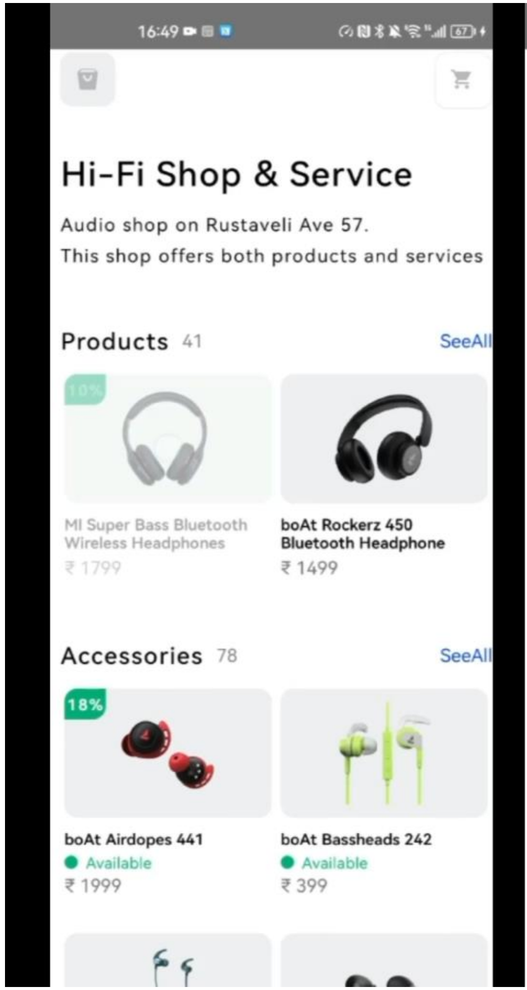


V8-No-Cache



JWST (WASM)

React Native Samples (0.1x play)



16:49

Hi-Fi Shop & Service

Audio shop on Rustaveli Ave 57.
This shop offers both products and services

Products 41 SeeAll

10%

MI Super Bass Bluetooth Wireless Headphones
₹ 1799

boAt Rockerz 450 Bluetooth Headphone
₹ 1499

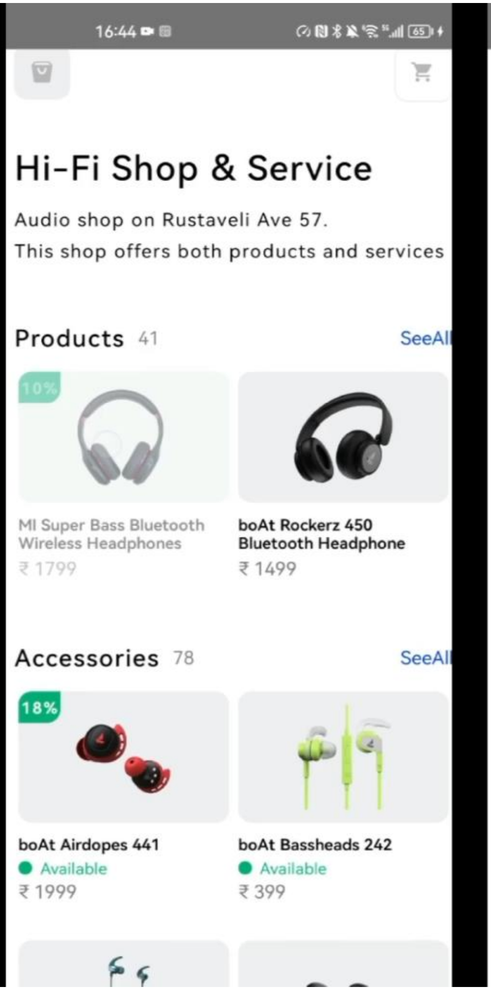
Accessories 78 SeeAll

18%

boAt Airdopes 441
Available
₹ 1999

boAt Bassheads 242
Available
₹ 399

JWST (WASM)



16:44

Hi-Fi Shop & Service

Audio shop on Rustaveli Ave 57.
This shop offers both products and services

Products 41 SeeAll

10%

MI Super Bass Bluetooth Wireless Headphones
₹ 1799

boAt Rockerz 450 Bluetooth Headphone
₹ 1499

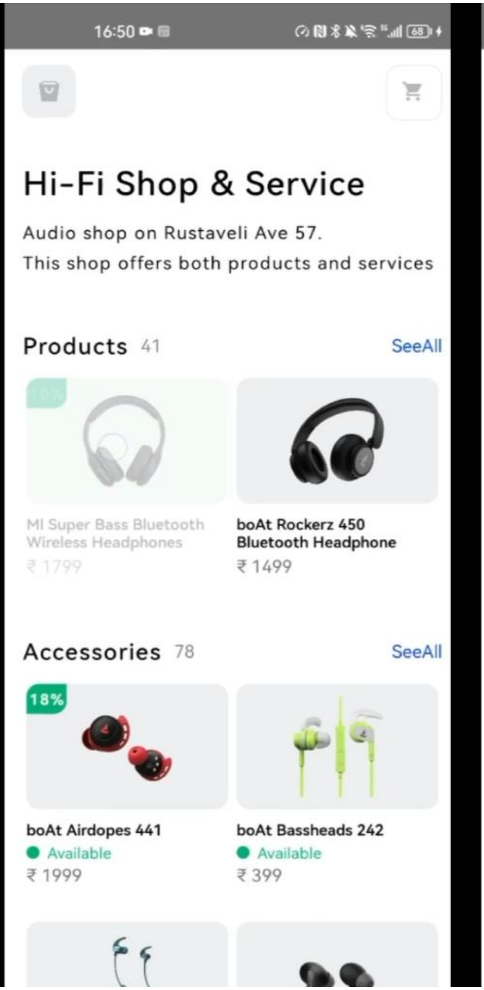
Accessories 78 SeeAll

18%

boAt Airdopes 441
Available
₹ 1999

boAt Bassheads 242
Available
₹ 399

JWST Native



16:50

Hi-Fi Shop & Service

Audio shop on Rustaveli Ave 57.
This shop offers both products and services

Products 41 SeeAll

10%

MI Super Bass Bluetooth Wireless Headphones
₹ 1799

boAt Rockerz 450 Bluetooth Headphone
₹ 1499

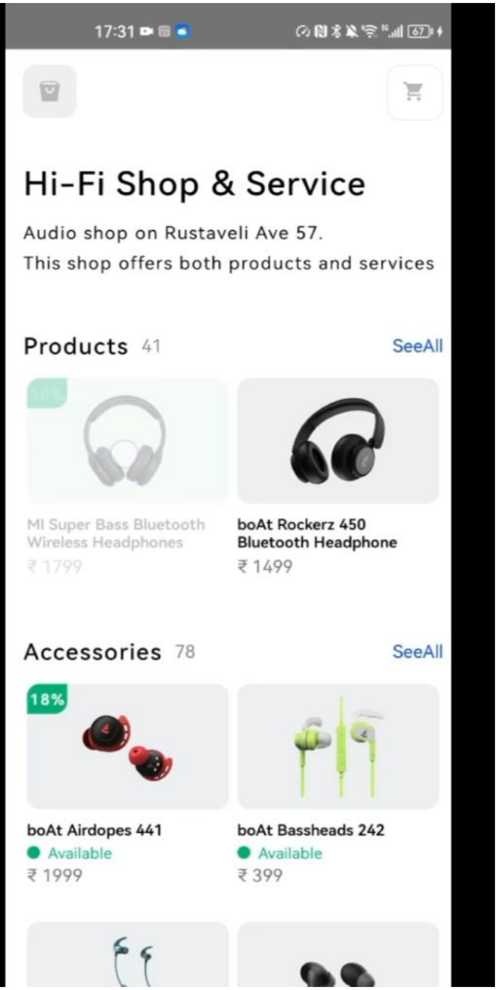
Accessories 78 SeeAll

18%

boAt Airdopes 441
Available
₹ 1999

boAt Bassheads 242
Available
₹ 399

V8-Cache



17:31

Hi-Fi Shop & Service

Audio shop on Rustaveli Ave 57.
This shop offers both products and services

Products 41 SeeAll

10%

MI Super Bass Bluetooth Wireless Headphones
₹ 1799

boAt Rockerz 450 Bluetooth Headphone
₹ 1499

Accessories 78 SeeAll

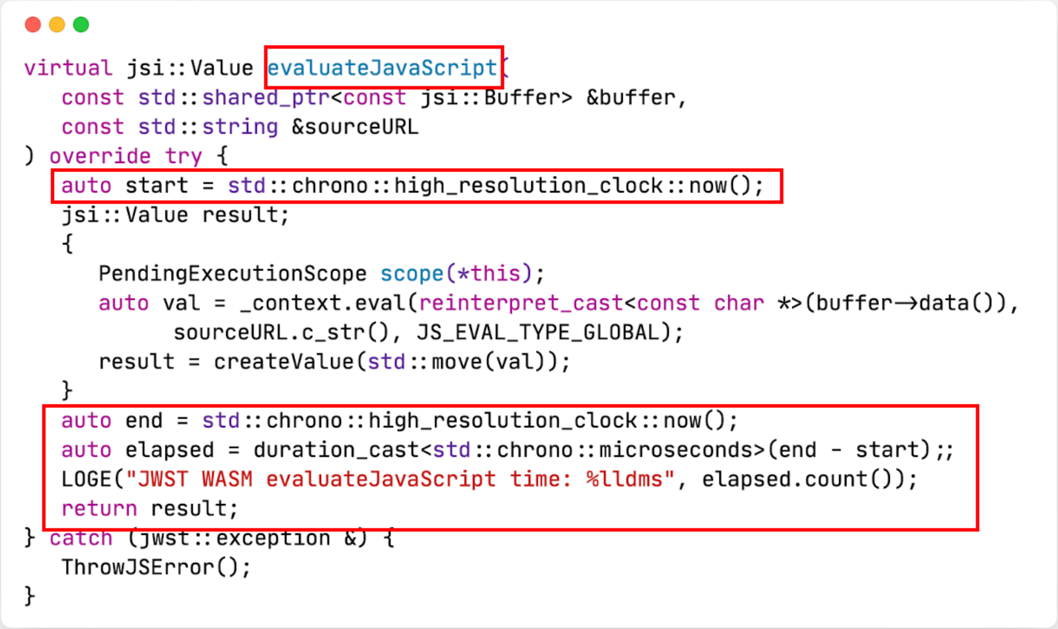
18%

boAt Airdopes 441
Available
₹ 1999

boAt Bassheads 242
Available
₹ 399

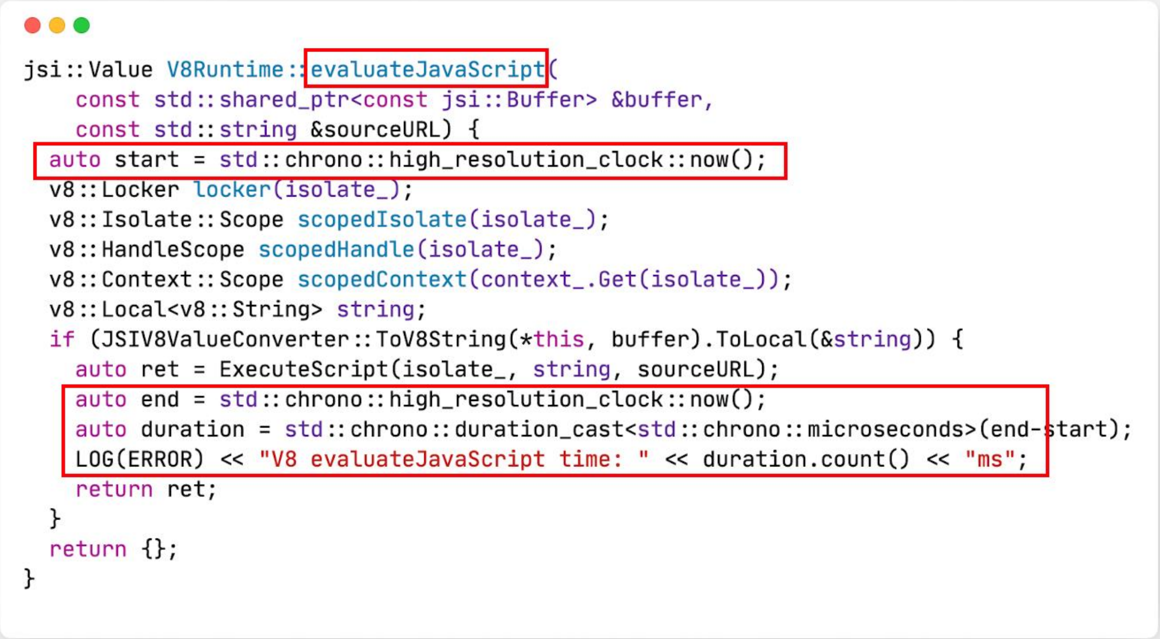
V8-No-Cache

Evaluate the Start-up JS Scripts of a RN APP



```cpp
virtual jsi::Value evaluateJavaScript(
   const std::shared_ptr<const jsi::Buffer> &buffer,
   const std::string &sourceURL
) override try {
   auto start = std::chrono::high_resolution_clock::now();
   jsi::Value result;
   {
      PendingExecutionScope scope(*this);
      auto val = _context.eval(reinterpret_cast<const char *>(buffer->data()),
            sourceURL.c_str(), JS_EVAL_TYPE_GLOBAL);
      result = createValue(std::move(val));
   }
   auto end = std::chrono::high_resolution_clock::now();
   auto elapsed = duration_cast<std::chrono::microseconds>(end - start);;
   LOGE("JWST WASM evaluateJavaScript time: %lldms", elapsed.count());
   return result;
} catch (jwst::exception &) {
   ThrowJSError();
}
```

JWST-WASM/JWST-Native

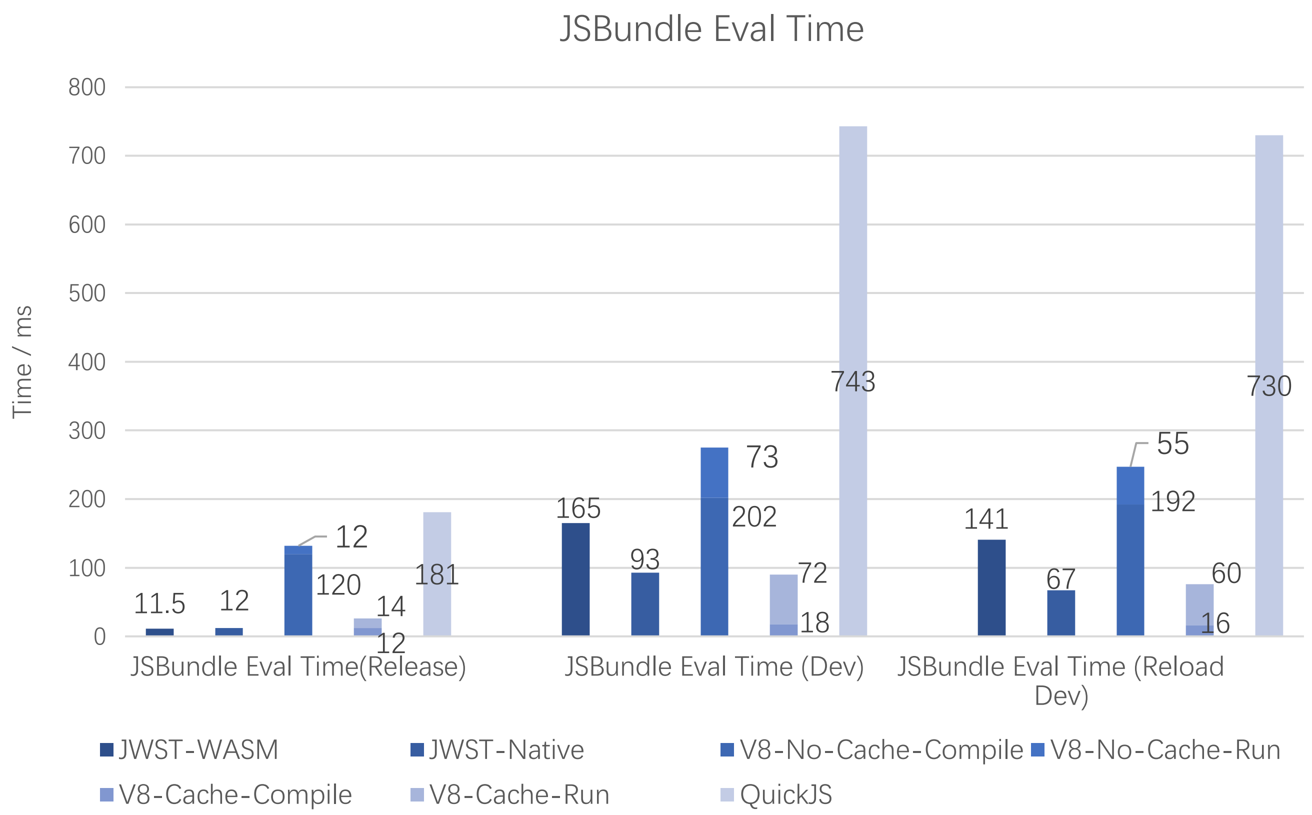


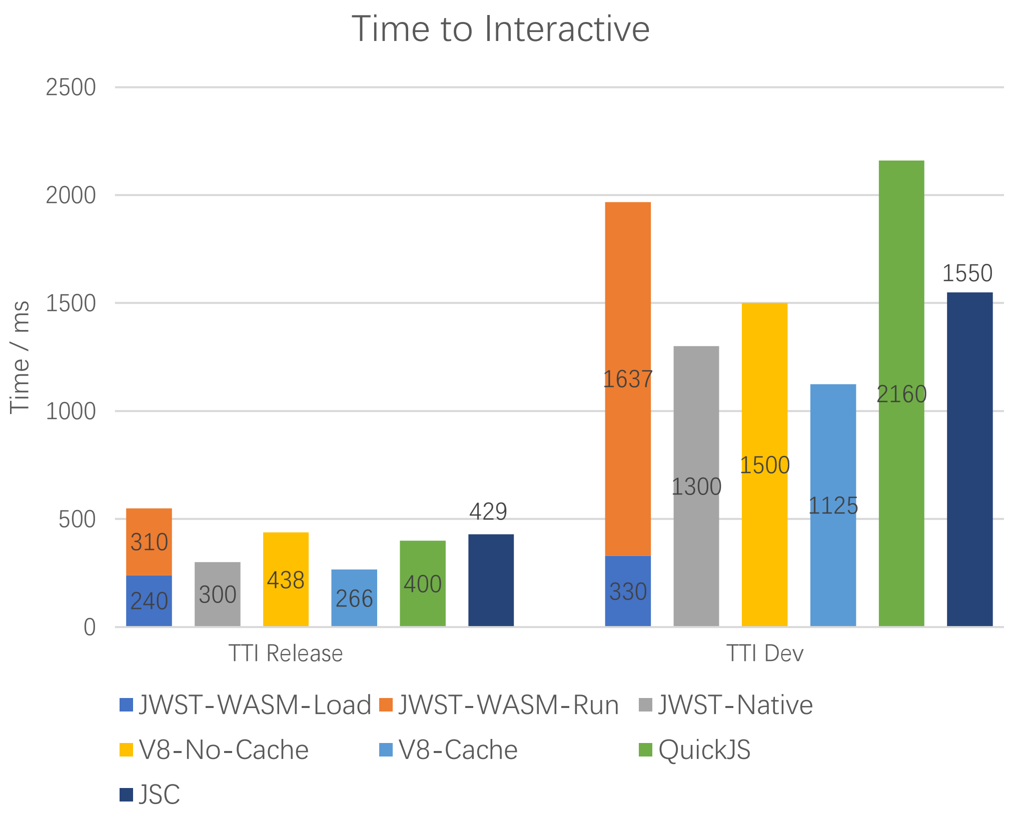
```cpp
jsi::Value V8Runtime::evaluateJavaScript(
    const std::shared_ptr<const jsi::Buffer> &buffer,
    const std::string &sourceURL) {
  auto start = std::chrono::high_resolution_clock::now();
  v8::Locker locker(isolate_);
  v8::Isolate::Scope scopedIsolate(isolate_);
  v8::HandleScope scopedHandle(isolate_);
  v8::Context::Scope scopedContext(context_.Get(isolate_));
  v8::Local<v8::String> string;
  if (JSIV8ValueConverter::ToV8String(*this, buffer).ToLocal(&string)) {
    auto ret = ExecuteScript(isolate_, string, sourceURL);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end-start);
    LOG(ERROR) << "V8 evaluateJavaScript time: " << duration.count() << "ms";
    return ret;
  }
  return {};
}
```

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

Size	JSBundle	WASM(no-link)	JWST Runtime	WASM(linked)	llvm-strip WASM(linked)
Dev	4.9MB	46MB	5.1MB	45MB	41MB
Release	1.3MB	27MB	5.1MB	28MB	25MB

Compilation Time	JWST (JSBundle => LLVM IR)	llc (LLVM IR => WASM)	wasm-ld (linking)
Dev	1600s (26min40s)	490s (8min10s)	0.13s
Release	640s (10min40s)	200s (3min20s)	0.08s

- 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!