The JavaScriptCore Virtual Machine

Filip Pizlo
Apple Inc.
3 Pizlo Keynotes / Week

ICCV’17

“Symmetry as the fundamental prior in human 3D vision”

Zygmunt Pizlo
webkit.org

https://svn.webkit.org/repository/webkit/trunk
What JSC Supports

• ECMAScript 2016+

• WebAssembly
What JSC Supports

- ECMAScript 2016+
- WebAssembly
Architecture
Architecture

• Interpreters and JITs
• Object Model
• Type Inference
• Garbage Collector
Interpreters and JITs
Four Tiers

- LLInt (interpreter)
- Baseline (template JIT)
- DFG (less optimizing JIT)
- FTL (optimizing JIT)

Latency vs. Throughput
• Four tiers for JavaScript
• Two tiers for WebAssembly
• Two tiers for regular expressions
• Four tiers for JavaScript
• Two tiers for WebAssembly
• Two tiers for regular expressions
Four Tiers

• How we tier up
• How the tiers work
• How we OSR exit
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
    let o = {f: i};
    result += o.f;
}

print(result);
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
    let o = {f: i};
    result += o.f;
}

print(result);
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
    let o = {f: i};
    result += o.f;
}

print(result);
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
    let o = {f: i};
    result += o.f;
}

print(result);
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
  let o = {f: i};
  result += o.f;
}

print(result);
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
    let o = {f: i};
    result += o.f;
}

print(result);
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
    let o = {f: i};
    result += o.f;
}

print(result);
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
    let o = {f: i};
    result += o.f;
}

print(result);
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
    let o = {f: i};
    result += o.f;
}

print(result);
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
  let o = {f: i};
  result += o.f;
}

print(result);
How We Tier Up

• Counting trigger
• Concurrent JITs
• Parallel JITs
• OSR
Profiling

- LLInt (interpreter)
- Baseline (template JIT)
- DFG (less optimizing JIT)
- FTL (optimizing JIT)
Speculation and OSR

- LLInt (interpreter)
- Baseline (template JIT)
- DFG (less optimizing JIT)
- FTL (optimizing JIT)
function foo(a, b)
{
    return a + b;
}

[  0] enter
[  1] get_scope  loc3
[  3] mov  loc4, loc3
[  6] check_traps
[  7] add  loc6, arg1, arg2
[ 12] ret  loc6
[  0] enter
[  1] get_scope     loc3
[  3] mov          loc4, loc3
[  6] check_traps
[  7] add         loc6, arg1, arg2
[ 12] ret         loc6
23: GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
24: GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
25: ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
26: MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
28: Return(Untyped:@25, W:SideState, Exits, bc#12)
[   7] add               loc6, arg1, arg2

23: GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
24: GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
25: ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
26: MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
23: GetLocal(Untyped:@1, arg1(B<int32>/FlushedInt32), R:Stack(6), bc#7)
24: GetLocal(Untyped:@2, arg2(C<boolInt32>/FlushedInt32), R:Stack(7), bc#7)
25: ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
26: MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
add loc6, arg1, arg2

23: GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
24: GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
25: ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
26: MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
23: GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
24: GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
25: ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
26: MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
23: GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
24: GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
25: ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
26: MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
23: GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
24: GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
25: ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
26: MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
add loc6, arg1, arg2

23: GetLocal( Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
24: GetLocal( Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
25: ArithAdd( Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
26: MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
add loc6, arg1, arg2

23: GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
24: GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
25: ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
26: MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
Add

25:

arg1

arg2

Return

loc0
loc1
loc3
loc4
loc5
loc6

MovHint

loc6 := @25
• OSR exit
Deoptimization

- OSR exit
- Invalidation
- Jettison
Int32 @37 = Trunc(@27, DFG:@25)
Int32 @38 = Trunc(@22, DFG:@25)
Int32 @39 = CheckAdd(@37:WarmAny, @38:WarmAny, generator = 0x109ec5b90,
    earlyClobbered = [], lateClobbered = [], usedRegisters = [],
    ExitsSideways|Reads:Top, DFG:@25)
Int64 @40 = ZExt32(@39, DFG:@28)
Int64 @41 = Add(@40, $-281474976710656(@13), DFG:@28)
Void @42 = Return(@41, Terminal, DFG:@28)
Patch &BranchAdd32, Overflow, %tmp4, %tmp5, %tmp3, @39
Move32 %tmp3, %tmp1, @40
Add64 %tmp1, %tmp2, %tmp0, @41
Move %tmp0, %rax, @42
Ret64 %rax, @42
Patch &Branch
Add32, Overflow, %rcx, %rdx, %rdx, @39
Add64 %rdx, %rax, %rax, @41
Ret64 %rax, @42
add %ecx, %edx
jo 0x267160c025ed
add %rdx, %rax
Optimizations

• Generatorification
• Inlining
• Strength Reduction
• CSE (local and global)
• LICM
• Type/Bounds/Overflow Check Removal
• Object Allocation Sinking
• Arguments/Varargs Elimination
• Sparse Conditional Constant Propagation
• Barrier Placement

• Strength Reduction
• Tail Duplication
• Switch Inference
• Float Inference
• DCE
• Register Allocation
  • Linear Scan
  • Briggs
  • Iterated Register Coalescing
• Stack Allocation
Interpreters and JITs

- Optimized for breadth
- Four tiers
- Many optimizations in many IRs
- Speculative
Object Model
structure

prototype

global object

\{x, y\}

\{1, 2\}

\{-5, 7\}

\{42, 3\}
poly proto just landed last Thursday
@saambarati and I have been working on it for ~2 months
Structures

- Fast property access
- Property type inference
- Immutable property inference
- Prototype optimizations
JSC Object Model

structure ID
indexing
type
flags
cell state
butterfly pointer
inline slot 0
inline slot 1
out of line slot 0
public length
vector length
array slot 0
...
JSC Object Model

- Structure
- ID
- Indexing
- Type
- Flags
- Cell State
- Butterfly Pointer
- Inline Slot 0
- Inline Slot 1
- Out of Line Slot 0
- Public Length
- Vector Length
- Array Slot 0

64 bits
JSC Object Model

- structure
- ID
- indexing
- type
- flags
- cell state
- butterfly pointer
- inline slot 0
- inline slot 1
- out of line slot 0
- public length
- vector length
- array slot 0

[Diagram showing the JSC Object Model with various slots and data types]
JSC Object Model

- structure
- ID
- indexing
- type
- flags
- cell state
- butterfly pointer
- inline slot 0
- inline slot 1
- out of line slot 0
- public length
- vector length
- array slot 0

statically configurable
JSC Object Model

- Structure
  - ID
  - Indexing
  - Type
  - Flags
  - Cell state
- Butterfly pointer
- Inline slot 0
- Inline slot 1
- Out of line slot 0
- Public length
- Vector length
- Array slot 0

- Statically configurable
- Dynamically configurable
Empty JSObject

| structure | ID | indexing | type | flags | cell state | null |
Fast JSObject

var o = {f: 5, g: 6};
var o = {f: 5, g: 6};
o.h = 7;
JSArray with room for 3 array elements

```
var a = [];  
0   3   <hole> <hole> <hole>
```
Object with fast properties and array elements

```javascript
var o = {f: 5, g: 6};
o[0] = 7;
```
Object with fast and dynamic properties and array elements

```javascript
var o = {f: 5, g: 6};
o[0] = 7;
o.h = 8;
```
Exotic object with dynamic properties and array elements

```javascript
var o = new Date();
o[0] = 7;
o.h = 8;
```
Object Model

- Structures
- Cells
- Butterflies
Type Inference
Type Inference

- Watchpoints
- Value Profiles
- Polymorphic Inline Caches
Type Inference

- Watchpoints
- Value Profiles
- Polymorphic Inline Caches
Watchpoints
class Watchpoint {
public:
    virtual void fire() = 0;
};
numberToStringWatchpoint
numberToStringWatchpoint

1. Compiler wants to optimize `42.toString()` to “42”

2. Check if already invalidated
   - If invalid, don’t do the optimization.
   - If valid, register watchpoint and do the optimization.
Many watchpoints

• haveABadTime
• Structure transition
• InferredValue
• InferredType
• many others
Garbage Collector
Garbage Collector

- No copying
- Conservative on the stack
Garbage Collector

- Constraint-based
- Generational
- Concurrent
- Parallel
Garbage Collector

- Constraint-based
- Generational
- Concurrent
- Parallel
Constraint-Based Marking

• Transitive reachability is not always enough

• Common examples:
  • Soft references
  • Weak map
Constraint-Based Marking

- Transitive reachability is not always enough
- WebKit examples:
  - Type inference
  - Weak map
  - DOM
  - Native code
Constraint-Based Marking

- Transitive reachability is not always enough
- WebKit examples:
  - Type inference
  - Weak map
  - DOM
  - Native code
Type Inference
Structure

Objects

{1, 2}

{x, y}

{-5, 7}

{42, 3}

prototype

global object
Is this a weak reference?

- JIT code
- {1, 2}
- {-5, 7}
- {42, 3}
- prototype
- global object

{x, y}
JIT code references a structure

- Strong reference?
- Weak reference?
- Marking constraint?
Strong reference?

JIT code

\{x, y\}

\{1, 2\}

\{-5, 7\}

\{42, 3\}

prototype

global object
Strong reference?

JIT code → {x, y} → prototype

{y} → global object
Strong reference?

- JIT code
- \{x, y\}
  - prototype
  - global object
Strong reference?

so many leaks
Weak reference?

- JIT code
- \{x, y\}
- 1, 2
- -5, 7
- 42, 3
- prototype
- global object
Weak reference?

- JIT code
- \{x, y\}
- prototype
- global object
Weak reference?

JIT code

{x, y}

prototype

global object
Weak reference?

recomp storm
Marking Constraint

- JIT code
- prototype
- global object

structure
Marking Constraint

- JIT code references the structure weakly.
if (isMarked(structure->globalObject())
    && isMarked(structure->storedPrototype()))
mark(structure);

• JIT code references the structure weakly.

• JIT code also registers the above marking constraint.
Marking Constraint!

- JIT code
- prototype
- global object
- \{x, y\}
- \{1, 2\}
- \{-5, 7\}
- \{42, 3\}
Marking Constraint!

- JIT code
- {x, y}
- prototype
- global object
Marking Constraint!

It’s cool - the prototype and global object are long-lived.
Marking Constraint!

- JIT code
- \{x, y\}
Marking Constraint!

JIT code

\{x, y\}
We want the JIT code to die in this case.
Marking Constraint!

- If the objects that use the structure die, then:
  - Keep structure alive if the user objects it points to are alive anyway.
  - Kill the structure (and the JIT code) if keeping it alive would not be safe-for-space.
Marking Constraints

- Constraints can query which objects are marked.
- Constraints can mark objects.
- GC executes constraints to fixpoint.
Garbage Collector

- Constraint-based
- Generational
- Concurrent
- Parallel
Conclusion

• JavaScriptCore Architecture:
  • Interpreters and Multiple JITs
  • Cells, Structures, and Butterflies
  • Watchpoints, Value Profiles, and Inline Caches
  • Constraint-Based GC