Polyhedral Optimization For JavaScript: The Challenges

Manuel Selva, Julien Pagès, Philippe Clauss

INRIA CAMUS, ICube, CNRS, University of Strasbourg

January 23, 2018

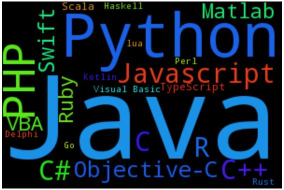
The JavaScript Language

Created in 1995 at Netscape

- To implement dynamism in web pages
- High level and dynamic
- ECMAScript standard

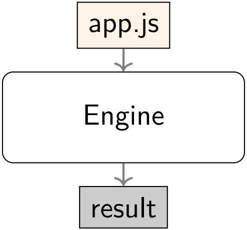
Now widely used both on client and server sides

- For complex applications
- Because of portability
- Because of performances



5th language - PYPL index

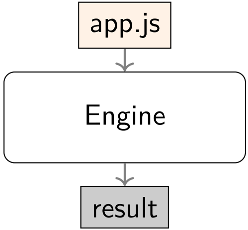
JavaScript Implementation



Widespread engines

- SpiderMonkey - *Mozilla*
- JavaScriptCore - *Apple*
- V8 - *Google*
- Chakra - *Microsoft*

JavaScript Implementation



Widespread engines

- SpiderMonkey - *Mozilla*
- JavaScriptCore - *Apple*
- V8 - *Google*
- Chakra - *Microsoft*

Browser wars

- Complex optimization
- Nevertheless, **no parallelism**

You Said Polyhedral Model And JavaScript Together

How to cope with dynamism?

- Static Control Parts (SCoPs) cannot be detected statically
- When and how detect polyhedral opportunities?

Is it worthwhile to use the polyhedral model at runtime?

- Gain versus overhead

Outline

JavaScript Is Dynamic → We Need An Engine

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

JavaScript Is Dynamic → We Need An Engine

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

- Types are dynamic

JavaScript Is Dynamic → We Need An Engine

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

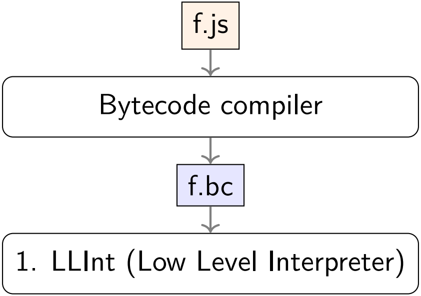
- Types are dynamic
- Arrays are dynamic

JavaScript Is Dynamic → We Need An Engine

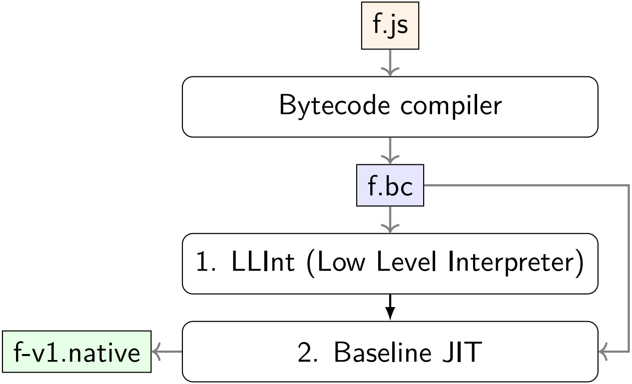
```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

- Types are dynamic
- Arrays are dynamic
- **Only** double precision floating point numbers

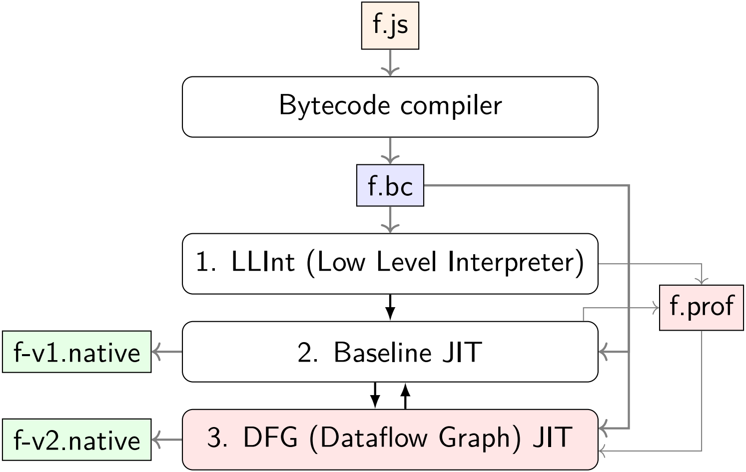
4 Layers



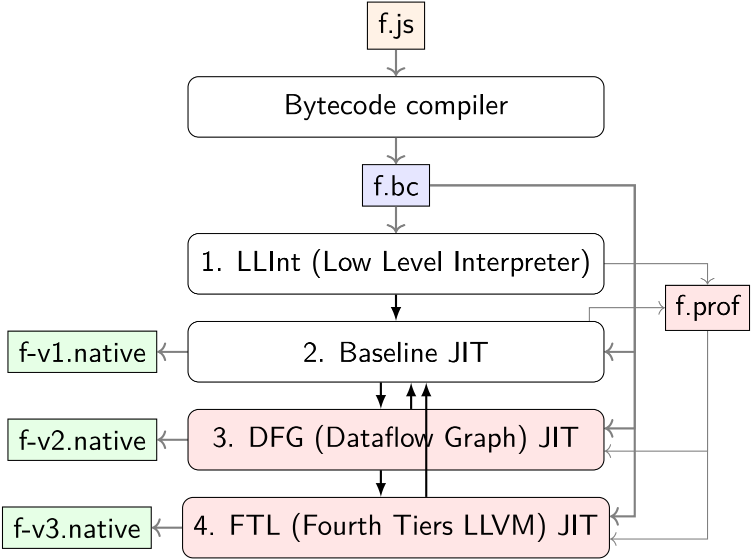
4 Layers


f.js
Bytecode compiler
f.bc
1. LLInt (Low Level Interpreter)
2. Baseline JIT
f-v1.native

4 Layers


f.js
Bytecode compiler
f.bc
1. LLInt (Low Level Interpreter)
f.prof
2. Baseline JIT
f-v1.native
3. DFG (Dataflow Graph) JIT
f-v2.native

4 Layers


f.js
Bytecode compiler
f.bc
1. LLInt (Low Level Interpreter)
f.prof
f-v1.native
2. Baseline JIT
f-v2.native
3. DFG (Dataflow Graph) JIT
f-v3.native
4. FTL (Fourth Tiers LLVM) JIT

1. LLInt

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

Source

1. LLInt

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

Source

→

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

Bytecode

1. LLInt

Source

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

→

Bytecode

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

Interpreter

```
while(i = next_instruction()) {
  switch(i->opcode) {
    case add:
      switch (type_pair(i->operand1->type(), i->operand2->type())):
        case number_number:
          i->dest = add(i->operand1, i->operand2);
        case object_number:
          i->dest = add(i->operand1->as_number(), i->operand2);
        ...
    case mul: ...
  }
}
```

2. Baseline JIT

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

Source

→

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

Bytecode

2. Baseline JIT

Source

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

Bytecode

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

Native code

```
...
switch (type_pair(v, 41)):
  case number_number:
    v = add(v, 41);
  case object_number:
    ...
switch (type_pair(v, 2)):
  case number_number:
    v = mul(v->as_number() * 2);
  case object_number:
    ...
...
```

op_add v 41 v;

op_mul v 2 v;

3. DFG JIT

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

Source

→

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

Bytecode

3. DFG JIT

Source

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

Bytecode

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

DFG IR - Typed

```
if (!is_32int_array(img)) { return to Baseline JIT; }
...
res = v->as_32int() + 41;
if (overflow(res)) { return to Baseline JIT; }
v = res;
res = v->as_32int() * 2;
if (overflow(res)) { return to Baseline JIT; }
v = res;
...
```

3. DFG JIT

Source

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

Bytecode

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

DFG IR - Typed

```
if (!is_32int_array(img)) { return to Baseline JIT; }
...
res = v->as_32int() + 41;
if (overflow(res)) { return to Baseline JIT; }
v = res;
res = v->as_32int() * 2;
if (overflow(res)) { return to Baseline JIT; }
v = res;
...
```

Homemade optim and backend

Native code

```
...
...
```

4. FTL JIT

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

Source

→

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

Bytecode

4. FTL JIT

Source

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

→

Bytecode

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

↓

LLVM IR - Typed

```
if (!is_32int_array(img)) { return to Baseline JIT; }
...
res = v->as_32int() + 41;
if (overflow(res)) { return to Baseline JIT; }
v = res;
res = v->as_32int() * 2;
if (overflow(res)) { return to Baseline JIT; }
v = res;
...
```

4. FTL JIT

Source

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; i++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

→

Bytecode

```
f(img, width, height) {
  ...
  ...
  op_add v 41 v;
  op_mul v 2 v;
  ...
  ...
}
```

↓

LLVM IR - Typed

```
if (!is_32int_array(img)) { return to Baseline JIT; }
...
res = v->as_32int() + 41;
if (overflow(res)) { return to Baseline JIT; }
v = res;
res = v->as_32int() * 2;
if (overflow(res)) { return to Baseline JIT; }
v = res;
...
```

↓ LLVM optim and backend

Native code

```
...
...
```

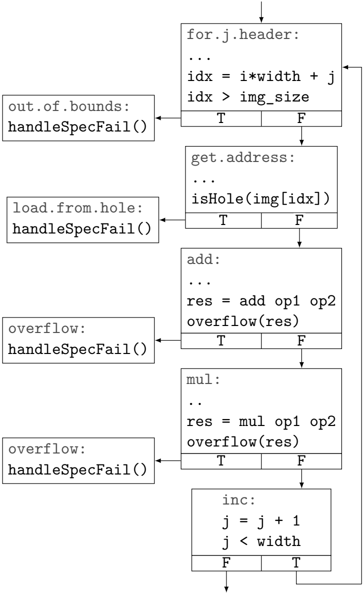
Proposal

Add polyhedral optimization in JavaScriptCore

- In the last layer - FTL
  - Dynamism has been removed
- Polly
  - Polyhedral optimizer for LLVM
  - Transform LLVM IR to optimized LLVM IR

Outline

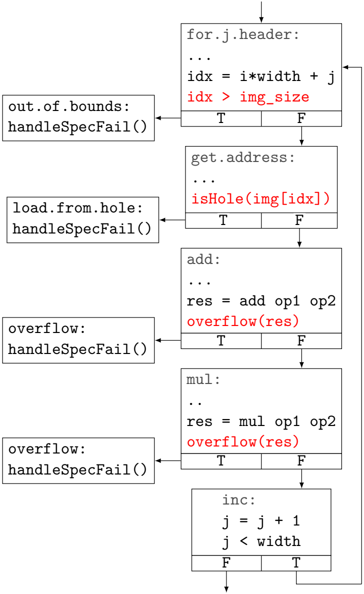
SESE Regions: Problem - *Apply To All Engines*




```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; j++) {
      var v = img[i*width + j]
      v = v + 41;
      v = v * 2;
      img[i*width + j] = v;
    }
  }
}
```

Source

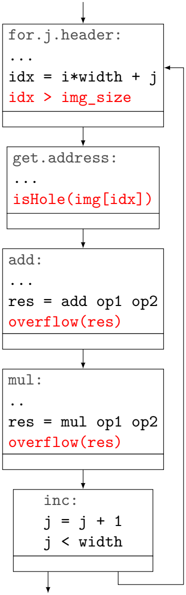
SESE Regions: Solution



Step 1 *(in our implementation)*

- Tag instructions branching to exit blocks

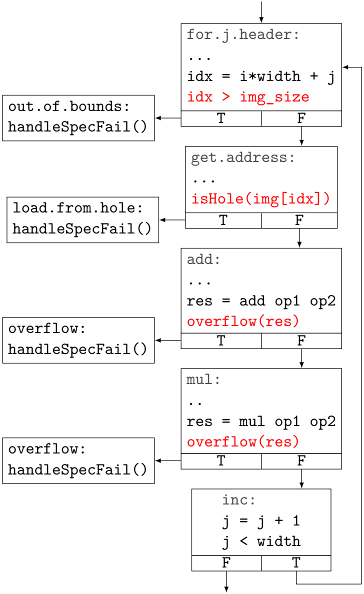
SESE Regions: Solution



Step 1 *(in our implementation)*

- Tag instructions branching to exit blocks
- Remove exit blocks
- Perform polyhedral optimization

SESE Regions: Solution



Step 1 *(in our implementation)*

- Tag instructions branching to exit blocks
- Remove exit blocks
- Perform polyhedral optimization

Step 2 *(not yet completed)*

- Add back exit blocks from tagged instructions

Two Dimensional Arrays And Arrays Of Objects

Problem - *Apply to all engines*

```
t[i].foo = 17;
t[i][j]  = 17;
```

→ 2 memory accesses

Two Dimensional Arrays And Arrays Of Objects

Problem - *Apply to all engines*

```
t[i].foo = 17;
t[i][j]  = 17;
```

→ 2 memory accesses

Solutions

- Do not handle them *(in our implementation)*
- Inspector / executor
- Modify language and object allocator

Handling `inttoptr` And `sext` Instructions

Problem - *Specific to JavaScriptCore and Polly*

- Polly is designed for LLVM IR coming from frontends for static languages
- `intoptr` used by the runtime for known addresses
- `sext` used by the runtime for values representation (NaN boxing)
- `inttoptr` and `sext` instructions make SCoP detection fail

Solution *(in our implementation)*

- Modify Polly to support these instructions

Parallel Speculation

Problem - *Apply to all engines*

- Speculation failure in one thread
- Other threads may have performed wrong computation

Solutions

- Ignore speculation failure *(in our implementation)* - **Wrong!**
- Only optimize idempotent regions
- Save and rollback

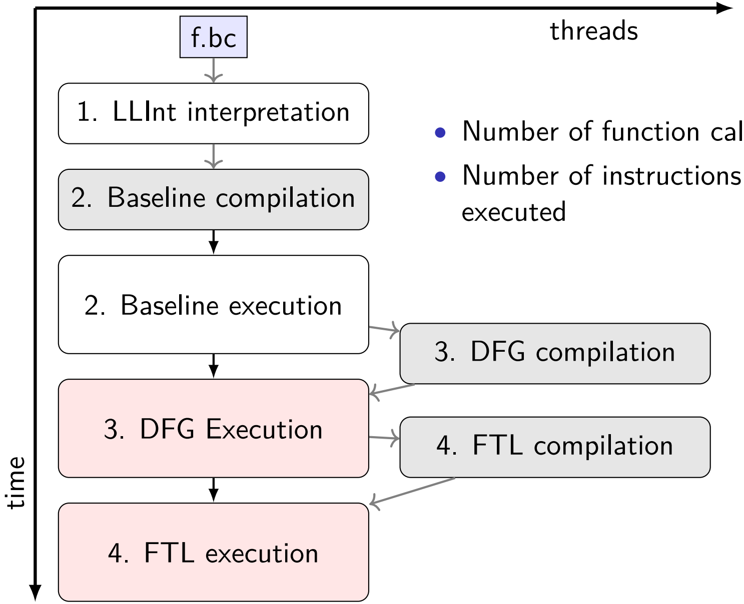
Gain vs Overhead - Problem - *Apply To All Engines*

Execution time of optimized version + Polly optimization time

<

Execution time of original version

Solution - JavaScriptCore Cost Model + Parallel Compil.



- Number of function calls
- Number of instructions executed

Outline

Matrix Multiply - Transformations

```
matmul(left, right, res, left_nblines, left_nbcols, right_nbcols) {
  for (var i = 0; i < left_nblines; i++) {
    for (var j = 0; j < left_nbcols; j++) {
      var idx_left = i * left_nbcols + j;
      for (var k = 0; k < right_nbcols; k++) {
        var idx_res = i*right_nbcols + k;
        var idx_right = j*right_nbcols + k;
        res[idx_res] = res[idx_res] + left[idx_left]*right[idx_right];
      }
    }
  }
}
```

Matrix Multiply - Transformations

```
matmul(left, right, res, left_nblines, left_nbcols, right_nbcols) {
  for (var i = 0; i < left_nblines; i++) {
    for (var j = 0; j < left_nbcols; j++) {
      var idx_left = i * left_nbcols + j;
      for (var k = 0; k < right_nbcols; k++) {
        var idx_res = i*right_nbcols + k;
        var idx_right = j*right_nbcols + k;
        res[idx_res] = res[idx_res] + left[idx_left]*right[idx_right];
      }
    }
  }
}
```

```
if(alias test ok){
 #pragma omp parallel for
 for(c0 = 0; c0 <= floord(p2-1, 32); c0 += 1)
  for(c1 = 0; c1 <= floord(p1-1, 32); c1 += 1)
   for(c2 = 0; c2 <= floord(p0-1, 32); c2 += 1) {
    for(c3 = 0; c3 <= min(31, p2-32*c0-1); c3 += 1)
     for(c4 = 0; c4 <= min(31, p1-32*c1-1); c4 += 1)
      for(c5 = 0; c5 <= min(31, p0-32*c2-1); c5 += 1)
       Stmt_68(32*c0+c3, 32*c2+c5, 32*c1+c4);
} else { original code version }
```

Matrix Multiply - Performances

Setup

- Intel Xeon W3520 with 4 cores running Linux 4.4.0
- LLVM and Polly 4.0.0 with `--parallel`
- `matmul` function called twice
- Right matrix size is 3000x300

Results

Size of left matrix	Execution time without Polly (s)	Execution time with Polly (s)	Speedup
50x3000	0.08	0.06	1.33
500x3000	0.85	0.22	3.86
2000x3000	3.3	0.87	3.79

Conclusion

Polyhedral optimization for JavaScript is almost there

- Speedups shown on a matrix multiply kernel

Ongoing work

- Complete implementation
- Evaluate speedups on real applications
- Study how to enrich JavaScriptCore profiling to help polyhedral optimizer

An Other Conclusion

JavaScript is a dynamic language. To efficiently execute it, dynamic features must not be used too often. Engineers at Google writing the V8 engine even recommend[1] to:

"Write code that looks like statically typed"

[1] Franziska Hinkelmann at JS Conf EU 2017

An Other Conclusion

JavaScript is a dynamic language. To efficiently execute it, dynamic features must not be used too often. Engineers at Google writing the V8 engine even recommend[1] to:

"Write code that looks like statically typed"

↓

WebAssembly

[1] Franziska Hinkelmann at JS Conf EU 2017

BACKUP

Tiers-Up In JavaScriptCore

LLInt $\rightarrow$ Baseline

- Same program state (variables values) representation
- OSR entry = jump (nothing to do)
- At any bytecode instruction

Baseline $\rightarrow$ DFG

- Different program state representations
- OSR entry = jump + copy of the state
- Only at function entry and loop header

DFG $\rightarrow$ FTL

- Different program state representations
- OSR entry = jump + copy of the state
- Only at function entry and loop header
    - Two code versions

NaN-Boxing In JavaScriptCore

	Tag	Payload	
IEE-754 doubles	0 0 0 0 . . . . F F F 8	0 0 0 0 | 0 0 0 0 | 0 0 0 0 . . . . F F F F | F F F F | F F F F	JavaScriptCore doubles
IEE-754 NaN	F F F 9 . . . . F F F D	0 0 0 0 | 0 0 0 0 | 0 0 0 0 . . . . F F F F | F F F F | F F F F	JavaScriptCore NaN
	F F F E . . . . F F F E	0 0 0 0 | 0 0 0 0 | 0 0 0 0 . . . . 0 0 0 0 | F F F F | F F F F	JavaScriptCore 32-bits integers
	. . . .	. . . .	Not used
	F F F F . . . . F F F F	0 0 0 0 | 0 0 0 0 | 0 0 0 0 . . . . F F F F | F F F F | F F F F	JavaScriptCore pointers

- **Direct** manipulation of doubles
- **Indirect** manipulation (masking) of pointers
- **Indirecte** manipulation (masking) 32 bits integers

NaN-Boxing In JavaScriptCore

	Tag	Payload	
IEE-754 doubles	0 0 0 0 … F F F 8	0 0 0 0 | 0 0 0 0 | 0 0 0 0 … F F F F | F F F F | F F F F	JavaScriptCore doubles
IE… N…	F F F 9 … F F F D	0 0 0 0 | 0 0 0 0 | 0 0 0 0 … F F F F | F F F F | F F F F	JavaScriptCore NaN
			…ptCore
			…d
	F F F F … F F F F	0 0 0 0 | 0 0 0 0 | 0 0 0 0 … F F F F | F F F F | F F F F	JavaScriptCore pointers

Favor pointers

- $+2^{48} = 0001|0000|0000|0000 = 281474976710656$

- **Direct** manipulation of doubles
- **Indirect** manipulation (masking) of pointers
- **Indirecte** manipulation (masking) 32 bits integers

NaN-Boxing In JavaScriptCore

	Tag	Payload	
IEE-754 doubles	0 0 0 1	0 0 0 0 | 0 0 0 0 | 0 0 0 0	JavaScriptCore doubles
	. . . .	. . . .	
	F F F 9	F F F F | F F F F | F F F F	
IEE-754 NaN	F F F A	0 0 0 0 | 0 0 0 0 | 0 0 0 0	JavaScriptCore NaN
	. . . .	. . . .	
	F F F E	F F F F | F F F F | F F F F	
	F F F F	0 0 0 0 | 0 0 0 0 | 0 0 0 0	JavaScriptCore 32-bits integers
	. . . .	. . . .	
	F F F F	0 0 0 0 | F F F F | F F F F	
	. . . .	. . . .	Not used
	0 0 0 0	0 0 0 0 | 0 0 0 0 | 0 0 0 0	JavaScriptCore pointers
	. . . .	. . . .	
	0 0 0 0	F F F F | F F F F | F F F F	

- **Direct** manipulation of pointers
- **Indirect** manipulation (substraction) of doubles
- **Indirect** manipulation (masking) of 32 bits integers

NaN-Boxing In JavaScriptCore

- `DoubleEncodeOffset`
  - Favor pointers
  - $0001|0000|0000|0000 = +281474976710656 = +2^{48}$
- `TagTypeNumber`
  - If all bits in the mask are set, this indicates an integer
  - If any but not all are set this value is a double.
  - $FFFF|0000|0000|0000 = -281474976710656$
- `TagMask = TagTypeNumber | TagBitTypeOther`
  - Used to check for all types of immediate values
  - Either number or other immediate (bool, null, undefined)
  - $FFFF|0000|0000|0002 = -281474976710654$
- add `DoubleEncodeOffset` $\equiv$ sub `TagTypeNumber`

JavaScriptCore History

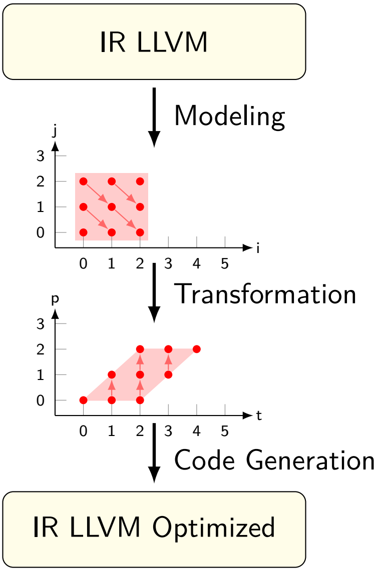
Developed by Apple

- Included in WebKit, LGPL license
- Before 2008 - SquirrelFish interpreter
- 2008 - 2014 - Nitro JIT
- 2014 - FTL JIT based on LLVM
- Mid 2016 - FTL replaced by B3, home made JIT and backend

Big project

```
LOC per language:
cpp:          289 435  (89.38%)
ansic:         11 254  (3.48%)
ruby:           9 925  (3.06%)
python:         6 195  (1.91%)
asm:            4 982  (1.54%)
perl:           2 013  (0.62%)
sh:                24  (0.01%)
LOC Total: 323 828
```

Polly: Polyhedral Optimization for LLVM



Detection Of Affine Accesses To Arrays

Problem - *Specific to JavaScriptCore and Polly*

```
t[index] = 17;
```

```
%offset = shl i64 %index, 3
%cell_as_int = add i64 %base_as_int, %offset
%cell_ptr = inttoptr i64 %cell_as_int to i64*
store i64 %boxed_17, i64* %cell_ptr
```

Detection Of Affine Accesses To Arrays

Problem - *Specific to JavaScriptCore and Polly*

```
t[index] = 17;
```

```
%offset = shl i64 %index, 3
%cell_as_int = add i64 %base_as_int, %offset
%cell_ptr = inttoptr i64 %cell_as_int to i64*
store i64 %boxed_17, i64* %cell_ptr
```

Solution *(in our implementation)*

```
%base_ptr = inttoptr i64 %base_as_int
            to [1000 x i64]*
%cell_ptr = getelementptr [1000 x i64],
              [1000 x i64]* %base_ptr,
              i32 0, i32 %index
store i64 %boxed_17, i64* %cell_ptr
```