Escape Analysis in V8

Tobias Tebbi
V8 Team, Chrome, Google
function double(x, array) {
    while (x >= 0) {
        array[x] *= 2;
        x--;
    }
}

Start:
x1 = Param[x]
array = Param[array]

Loop:
x2 = ϕ(x1, x3)
Branch(x2 >= 0)

IfTrue:
v1 = Load(array, x2)
v2 = v1 * 2
Store(array, x2, v2)
x3 = x2 - 1

IfFalse:
Return

SSA-CFG
Turbofan - A Sea of Nodes Compiler

SSA-CFG

Start:
\[ x_1 = \text{Param}[x] \]
\[ \text{array} = \text{Param[array]} \]

Loop:
\[ x_2 = \phi(x_1, x_3) \]
\[ \text{Branch}(x_2 \geq 0) \]

IfFalse:
Return

IfTrue:
\[ v_1 = \text{Load}(\text{array}, x_2) \]
\[ v_2 = v_1 \times 2 \]
\[ \text{Store}(\text{array}, x_2, v_2) \]
\[ x_3 = x_2 - 1 \]

Sea of Nodes

Start

Loop

Branch

IfFalse

IfTrue

Return
Turbofan - A Sea of Nodes Compiler

SSA-CFG

Start:
- $x_1 = \text{Param}[x]$
- $\text{array} = \text{Param}[\text{array}]$

Loop:
- $x_2 = \phi(x_1, x_3)$
- Branch($x_2 \geq 0$)

IfTrue:
- $v_1 = \text{Load}($array, $x_2$$)$
- $v_2 = v_1 * 2$
- Store($\text{array}$, $x_2$, $v_2$)
- $x_3 = x_2 - 1$

IfFalse:
- Return

Sea of Nodes

Start
- Loop
- Branch
- $\geq 0$

IfFalse
- Return

IfTrue
- $\phi$
- $-1$

Param[$x$]
Turbofan - A Sea of Nodes Compiler

SSA-CFG

Start:
- \( x_1 = \text{Param}[x] \)
- \( \text{array} = \text{Param}[\text{array}] \)

Loop:
- \( x_2 = \phi(x_1, x_3) \)
- Branch\( (x_2 \geq 0) \)

IfFalse:
- Return

IfTrue:
- \( v_1 = \text{Load}(\text{array}, x_2) \)
- \( v_2 = v_1 \times 2 \)
- Store\( (\text{array}, x_2, v_2) \)
- \( x_3 = x_2 - 1 \)

Sea of Nodes

Start
- Loop
- Branch

IfFalse
- Return

IfTrue
- \( v_1 = \text{Load}(\text{array}, x_2) \)
- \( v_2 = v_1 \times 2 \)
- Store\( (\text{array}, x_2, v_2) \)
- \( x_3 = x_2 - 1 \)

Param[x]
- \( \phi \)
- \( \geq 0 \)

Param[\text{array}]
Turbofan - A Sea of Nodes Compiler

SSA-CFG

Start:
  x1 = Param[x]
  array = Param[array]

Loop:
  x2 = φ(x1, x3)
  Branch(x2 >= 0)

IfTrue:
  v1 = Load(array, x2)
  v2 = v1 * 2
  Store(array, x2, v2)
  x3 = x2 - 1

IfFalse:
  Return

Sea of Nodes

Start
  Loop
    Branch
      >= 0
        IfFalse
          Return
        IfTrue
          v1 = Load(array, x2)
          v2 = v1 * 2
          Store(array, x2, v2)
          x3 = x2 - 1

Param[x]
  φ
  -1
  *2
  Load
  Store
  Param[array]
  φ
<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why Sea of Nodes?

Advantages
- All dependencies are explicit.

Disadvantages
- All dependencies have to be explicit.
Why Sea of Nodes?

Advantages
- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.

Disadvantages
- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.
Why Sea of Nodes?

Advantages

- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.
- Clever scheduler can improve node placement.

Disadvantages

- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.
- Scheduler can make it worse: register pressure, redundant computations.
Why Sea of Nodes?

Advantages
- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.
- Clever scheduler can improve node placement.
- Graph reductions only see reachable nodes.

Disadvantages
- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.
- Scheduler can make it worse: register pressure, redundant computations.
- Graph reductions easily happen in bad order.
Why Sea of Nodes?

**Advantages**
- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.
- Clever scheduler can improve node placement.
- Graph reductions only see reachable nodes.

**Disadvantages**
- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.
- Scheduler can make it worse: register pressure, redundant computations.
- Graph reductions easily happen in bad order.
- Hard to replace pure nodes with local effect or control.
**Why Sea of Nodes?**

**Advantages**
- All dependencies are explicit.
- Simple node replacements without worrying about scheduling.
- Clever scheduler can improve node placement.
- Graph reductions only see reachable nodes.

**Disadvantages**
- All dependencies have to be explicit.
- If we need a schedule, we have to compute it.
- Scheduler can make it worse: register pressure, redundant computations.
- Graph reductions easily happen in bad order.
- Hard to replace pure nodes with local effect or control.
- Reductions operate on nodes instead of blocks.
Escape Analysis: Removing Temporary Objects

function diagonal(a) {
    return abs({x:a, y:a});
    return Math.sqrt(a*a + a*a);
}

function abs(v) {
    return Math.sqrt(v.x*v.x + v.y*v.y);
}

How do we remove this object allocation?
Step 1: Inlining

```javascript
function diagonal(a) {
    let v = {x:a, y:a};
    return abs(v);
}

function abs(v) {
    return Math.sqrt(v.x*v.x + v.y*v.y);
}
```
Step 2: Replace Field Accesses

```javascript
function diagonal(a) {
  let v = {x:a, y:a};
  return Math.sqrt(v.x*v.x + v.y*v.y);
}

function diagonal(a) {
  let v = {x:a, y:a};
  return Math.sqrt(a*a + a*a);
}
```
Step 3: Remove the unused allocation

```javascript
function diagonal(a) {
  let v = {x:a, y:a};
  return Math.sqrt(a*a + a*a);
}
```

```javascript
// Step 3: Remove the unused allocation

function diagonal(a) {
  return Math.sqrt(a*a + a*a);
}
```
It’s not always that easy...
function write_field(x) {
    let o = {a: 5};
    while (x > 0) o.a += x--;
    return o.a;
}

Replace object fields with a local variable.
Nested Objects

```javascript
function nested_objects(b) {
    let o = {a: {x: 5}};
    o.a = {x: 7}
    return o.a.x;
}
```

```javascript
function nested_objects(b) {
    return 7;
}
```
Limitations
Escaping Objects

```javascript
function escaping_object(foo) {
    let o = {};
    foo(o);
}
```

When we can’t inline `foo`, then we cannot dematerialize `o` because `foo` could do anything with it.
Index Access

```javascript
let l = [1, 2, 3];
let sum = 0;
for(let i = 0; i < 3; ++i) sum += l[i];
```

Index access requires linear memory. Thus we have to materialize `l`. 

""
Dynamic Object Identity

```javascript
function object_identity(b) {
  let o1 = {x: 5};
  let o2 = {x: 7};
  (b?o1:o2).x = 1;
  return o1.x;
}
```

This computation uses the object identity of `o1` and `o2`.

It’s impossible to map this to local variables: Local variables don’t have identity.

In principle, this could be solved with stack-allocation. (Java VMs do this.)
### The Magic of Deoptimization

```javascript
// function harmless(copy) {}

// function foo(x) {
//   let copy = {};
//   copy.a = x + 1;
//   harmless(copy);
// }

// function evil(copy) {
//   global = copy;
// }

foo({valueOf: () => harmless=evil});
```

While executing `foo`, the temporary object might suddenly escape.

Monkey patching can destroy any optimization, while the optimized code is running.
The Magic of Deoptimization

```javascript
function harmless(copy) {} 

function foo(x) {
    let copy = {};
    copy.a = x + 1;
    harmless(copy);
}

function foo(x) {
    if (typeof x !== 'number')
        %Deoptimize();
}
```

When deoptimizing, we have to re-create dematerialized objects.

Escape Analysis needs to store the state of dematerialized objects at each deoptimization point.
The Algorithm
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y;
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
  a.x.y = i;
}
foo();
return a.x.y;

Virtual Object 1
+0: var_a_shape
+8: …
+16: …
+24: var_a_x

Current Variable Value

<table>
<thead>
<tr>
<th>var_a_shape</th>
<th>shape of {x:?}</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_a_x</td>
<td>null</td>
</tr>
</tbody>
</table>

global for every effectful node
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y;
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y;
Escape Analysis in Turbofan

```javascript
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y;
```

Virtual Object 1

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_a_shape</td>
<td>+0</td>
</tr>
<tr>
<td>...</td>
<td>+8</td>
</tr>
<tr>
<td>var_a_x</td>
<td>+16</td>
</tr>
<tr>
<td>var_a_y</td>
<td>+24</td>
</tr>
</tbody>
</table>

Virtual Object 2

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_b_shape</td>
<td>+0</td>
</tr>
<tr>
<td>...</td>
<td>+8</td>
</tr>
<tr>
<td>var_b_y</td>
<td>+16</td>
</tr>
<tr>
<td>var_b_x</td>
<td>+24</td>
</tr>
</tbody>
</table>

Current Variable Value

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_a_shape</td>
<td>shape of {x:?}</td>
</tr>
<tr>
<td>var_a_x</td>
<td>b</td>
</tr>
<tr>
<td>var_b_shape</td>
<td>shape of {y:?}</td>
</tr>
<tr>
<td>var_b_y</td>
<td>5</td>
</tr>
</tbody>
</table>
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
  a.x.y = i;
}
foo();
return a.x.y;
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y;
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y;

<table>
<thead>
<tr>
<th>Var</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_a_shape</td>
<td>shape of {x:?}</td>
<td></td>
</tr>
<tr>
<td>var_a_x</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>var_b_shape</td>
<td>shape of {y:?}</td>
<td></td>
</tr>
<tr>
<td>var_b_y</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

### Virtual Objects

**Virtual Object 1**
- +0: var_a_shape
- +8: ...
- +16: ...
- +24: var_a_x

**Virtual Object 2**
- +0: var_b_shape
- +8: ...
- +16: ...
- +24: var_b_y

---

**Escape Analysis in Turbofan**
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y;

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_a_shape</td>
<td>shape of {x:?}</td>
</tr>
<tr>
<td>var_a_x</td>
<td>b</td>
</tr>
<tr>
<td>var_b_shape</td>
<td>shape of {y:?}</td>
</tr>
<tr>
<td>var_b_y</td>
<td>i</td>
</tr>
</tbody>
</table>
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y;

Virtual Object 1
+0: var_a_shape
+8: ...
+16: ...
+24: var_a_x

Virtual Object 2
+0: var_b_shape
+8: ...
+16: ...
+24: var_b_y

Current Variable Value

<table>
<thead>
<tr>
<th>var_a_shape</th>
<th>shape of {x:?}</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_a_x</td>
<td>b</td>
</tr>
<tr>
<td>var_b_shape</td>
<td>shape of {y:?}</td>
</tr>
<tr>
<td>var_b_y</td>
<td>Φ(5, i)</td>
</tr>
</tbody>
</table>
Escape Analysis in Turbofan

let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y;

---

Current Variable Value

<table>
<thead>
<tr>
<th>var_a_shape</th>
<th>shape of {x:?}</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_a_x</td>
<td>b</td>
</tr>
<tr>
<td>var_b_shape</td>
<td>shape of {y:?}</td>
</tr>
<tr>
<td>var_b_y</td>
<td>$\Phi(5,i)$</td>
</tr>
</tbody>
</table>

---

Virtual Object 1
+0: var_a_shape
+8: ...
+16: ...
+24: var_a_x

Virtual Object 2
+0: var_b_shape
+8: ...
+16: ...
+24: var_b_y

---

Remember Deoptimization Data

+0: shape of {x:?}
+8: ...
+16: ...
+24: $\Phi(5,i)$
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}
foo();
return a.x.y; Φ(5,i)
let a = {x: null};
let b = {y: 5};
a.x = b;
if (i > 10) {
a.x.y = i;
}

foo();
return a.x;

The inner object escapes!
Repeat analysis from its allocation point.
Challenges
Repeat When Escaping

- At any point, we might notice an object escapes.
- This invalidates all analysis steps using this object.
- But how to restore the previous state?

Solution:

- Separate analysis from graph mutation.
- Only do graph mutation once the analysis reached a fixed point.
- Track node replacements while analyzing.
How to Store the Variable State

- In a CFG: One map per basic block, updated imperatively when traversing the block
- In an unscheduled graph: One map per effectful node.

This is expensive! Solution: A purely functional map:

- Copy: $O(1)$
- Update/Access: $O(\log n)$

This can be achieved with any tree-based map datastructure. We chose a hash-tree.
Summary

- Escape analysis avoids allocating temporary object.
- V8 can dematerialize objects despite deoptimization.
- Limits of escape analysis are: escaping uses, index access, and using the object identity dynamically.
- Implementing escape analysis on an unscheduled graph is more challenging: All object fields need to be tracked for all effectful nodes.
- Purely functional maps can solve this issue without increased complexity.