An Actionable Performance Profiler for Optimizing the Order of Evaluations

Marija Selakovic  Thomas Glaser  Michael Pradel

ISSTA'17, 11 July
Inefficient Order Of Evaluation

expensiveAndUnlikely() && cheapAndLikely()
Example

Checking whether the input is NaN:

```javascript
_.isNaN = function(obj) {
    return _.isNumber(obj) && isNaN(obj);
};
```

Inputs:

- 3.14
- Number(3.14)
- NaN

Evaluations:

- (true, false)
- (true, false)
- (true, true)

See pull request #2496 of Underscore.js
Optimizing the Order of Evaluations

Goal: To find the most cost effective order of checks in a conditional

Challenges:

• Analysis of all checks in a conditional
• Assessment of the computational cost
• Safe to apply and beneficial optimizations
This Talk: DecisionProf

Profiler to find reordering opportunities

Traditional profiler

- Where time is spent, not where time is wasted

DecisionProf

- **Actionable** - suggests concrete optimizations
- **Guaranteed** performance improvement
DecisionProf: Overview

Program + input → Profiler → Optimization candidates → Program transformation → \( P_1 \ldots P_n \) → Performance evaluation → Optimization opportunities

Static preprocessing → Dynamic analysis
Commutative Checks

**Check**: Condition in a logical expression or switch statement

**Non-commutative checks**: changes program's semantics

  e.g. `a && a.x`

**Goal**: Optimizing commutative checks
Dynamic Analysis

\[ a \&\& b \]

<table>
<thead>
<tr>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_{a1} )</td>
<td>( v_{a1} )</td>
</tr>
<tr>
<td>( c_{a2} )</td>
<td>( v_{a2} )</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td>( c_{an} )</td>
<td>( v_{an} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution 1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_{b1} )</td>
<td>( v_{b1} )</td>
</tr>
<tr>
<td>( c_{b2} )</td>
<td>( v_{b2} )</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td>( c_{bn} )</td>
<td>( v_{bn} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution 2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Execution n</th>
</tr>
</thead>
</table>

\[ \text{Cost} = \text{number of executed branching points} \]
Dynamic Analysis: Example

_.isNumber(input) && isNaN(input)

<table>
<thead>
<tr>
<th>Execution 1</th>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution 2</th>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution 3</th>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

Cost: 12

Overall cost = 12
Dynamic Analysis: Example

- Estimate execution times of different orders

```javascript
isNaN(input) && _.isNumber(input)
```

<table>
<thead>
<tr>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>true</td>
</tr>
</tbody>
</table>

Overall cost = 6
Performance Evaluation

- Program transformation for each optimization candidate
- Methodology by Georges et al.[1]

Original program + input

Optimized program + input

VM instances

Warm up

Measure

t-test

Static preprocessing - hoists all checks outside the conditional

```javascript
var x = 0;

function a () {
    x++;
    var y=1;
    .......
}

startCheck: a();
startCheck: b();

if (a () && b()) ...
```

*write to x affects program state*
Safe Check Evaluation

_Idea:_ Collect and undo all writes to variables and object properties that may affect code after check evaluation

```javascript
var x = 0;

function a () {
    x++;
    var y=1;
    .......
}

startCheck: a();
startCheck: b();

//reset all side effects
if (a () && b()) ...
```

(write to x affects program state)
(program state is changed outside normal execution)
(dynamically execute $x = 0$;)

Pruning Non-Commutative Checks

- **Dynamic**: accesses the same variable/object property

- **Static**: known patterns

```javascript
var x;
var x;
a() && b()
a && a.x
y = x || "z"
```
Evaluation

Subject Programs and Inputs

- 9 popular JavaScript libraries and their test suites
- 34 benchmarks from JetStream suite

Performance Measurements

- $N_{VM} = 5$, $N_{warmUp} = 5$, $N_{measure} = 10$
Results

Reordering Opportunities

- 23 optimizations across 9 libraries
- 29 optimizations across benchmarks
- Performance improvements: 2.5% - 59% (function level), 2.5% - 6.5% (application level)
- Reported 7 optimizations (3 already accepted)
Estimated vs. Actual Cost

- Correlation = 0.92 for unit tests
- Correlation = 0.98 for benchmarks
Examples

Cheerio library:

//code before
isTag (elem) && elems.indexOf(elem) === -1

//code after
elems.indexOf(elem) === -1 && isTag (elem)

Gbemu benchmark:

//code before
numberType != "float32" && GameBoyWindow.opera
   && this.checkForOperaMathBug ()

//code after
GameBoyWindow.opera && numberType != "float32"
   && this.checkForOperaMathBug ()

Performance improvements

unit tests: 26%, 34%

application: 5.8%
Limitations

- Input sensitivity
- Side effects of native calls
- Correctness guarantees
Conclusions

Profiler to detect reordering opportunities

- Easy to exploit class of optimizations
- Suggests concrete refactorings
- Performance improvement guarantees

\[ \text{expensiveAndUnlikely()} \&\& \text{cheapAndLikely()} \]