An Actionable Performance Profiler for Optimizing the Order of Evaluations

Marija Selakovic   Thomas Glaser   Michael Pradel

TECHNISCHE UNIVERSITÄT DARMSTADT

ISSTA'17, 11 July

Inefficient Order Of Evaluation

expensiveAndUnlikely() && cheapAndLikely()

Example

Checking whether the input is NaN:

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

Inputs:	Evaluations:
3.14	*(true, false)*
Number(3.14)	*(true, false)*
NaN	*(true, true)*

UNDERSCORE.JS

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

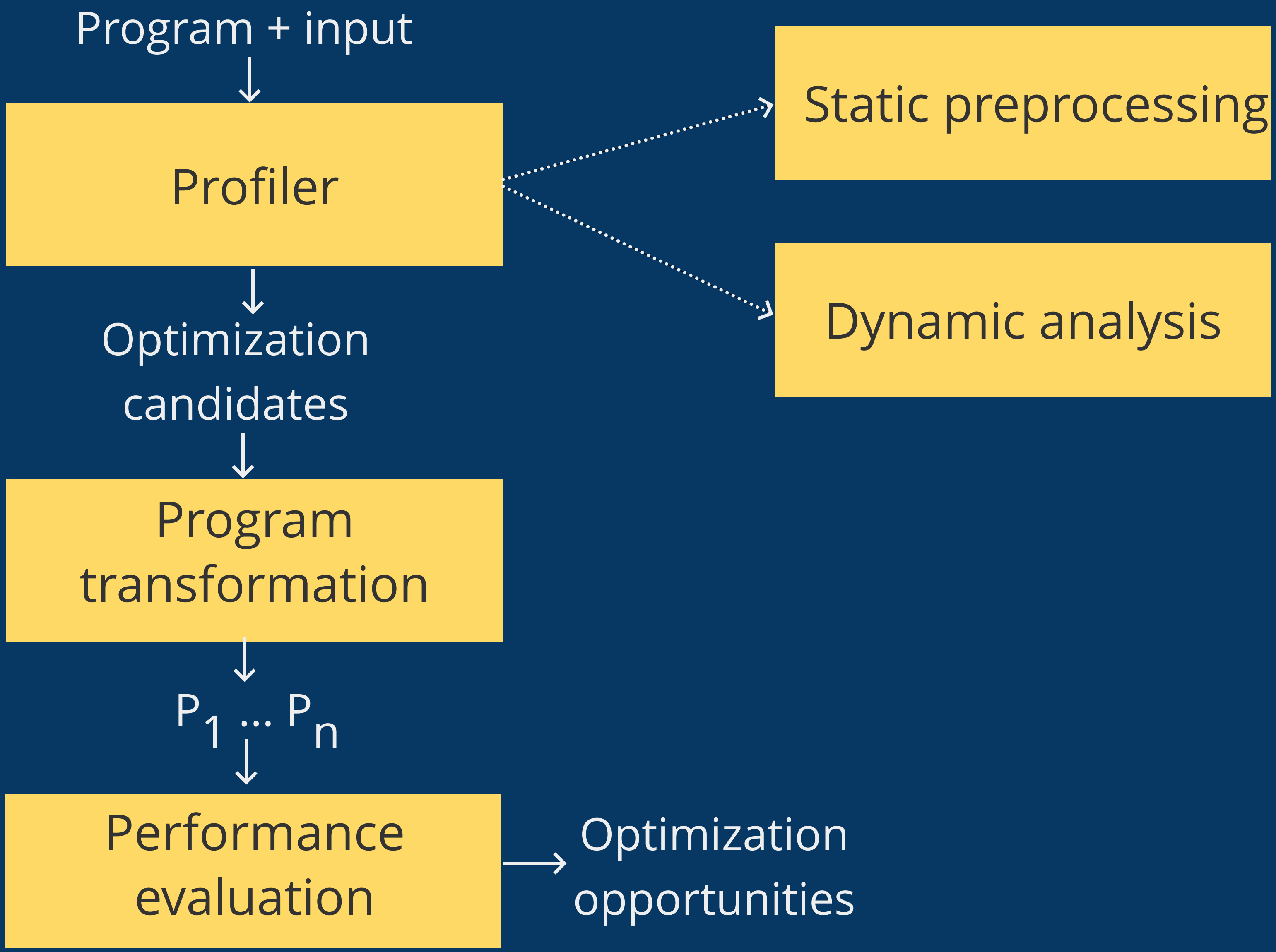
Traditional profiler

- Where time is spent, not where time is wasted

DecisionProf

- **Actionable** - suggests concrete optimizations
- **Guaranteed** performance improvement

DecisionProf: Overview



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

	Cost	Value
Execution 1	c_{a1}	v_{a1}
Execution 2	c_{a2}	v_{a2}
	….	
Execution n	c_{an}	v_{an}

	Cost	Value
Execution 1	c_{b1}	v_{b1}
Execution 2	c_{b2}	v_{b2}
	….	
Execution n	c_{bn}	v_{bn}

Cost = number of executed branching points

Dynamic Analysis: Example

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

	Cost	Value	Cost	Value
Execution 1	3	true	1	false
Execution 2	3	true	1	false
Execution 3	3	true	1	true

Overall cost = 12

Dynamic Analysis: Example

- Estimate execution times of different orders

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

	Cost	Value	Cost	Value
Execution 1	1	false	3	true
Execution 2	1	false	3	true
Execution 3	1	true	3	true

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

[1] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. (OOPSLA 07)

Analysis Of All Checks: Challenges

Static preprocessing - hoists all checks outside the conditional

```
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*

```
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

var x;

a() && b()

- **Static**: known patterns

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

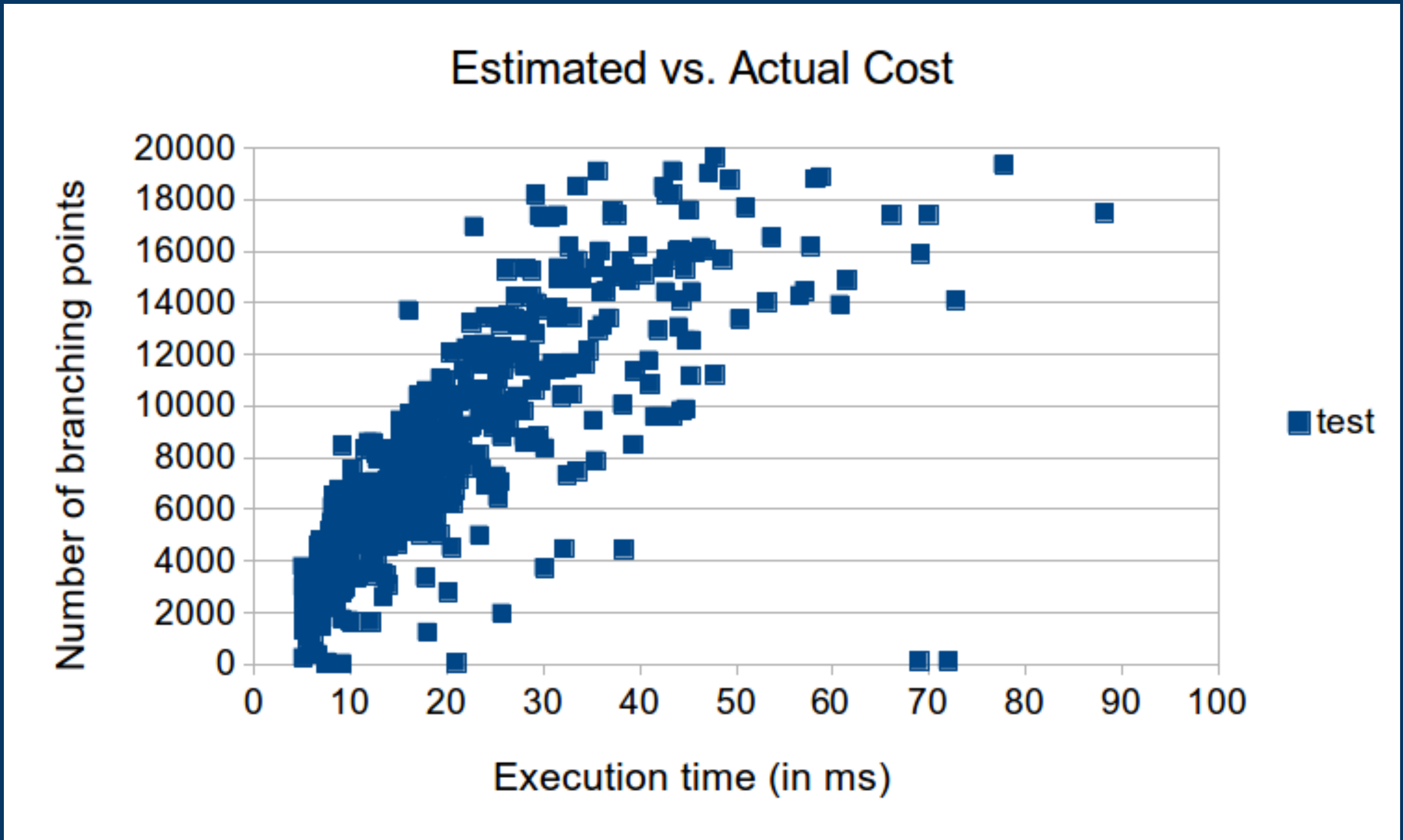
- $N_{VM} = 5$, $N_{warmUp} = 5$, $N_{measure} = 10$

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)
```

Performance improvements

unit tests:
26%, 34%

Gbemu benchmark:

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

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

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

expensiveAndUnlikely() && cheapAndLikely()