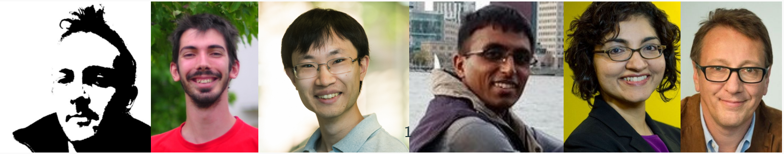
Correctness of Speculative Optimizations with Dynamic Deoptimization

Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel,
Amal Ahmed, Jan Vitek

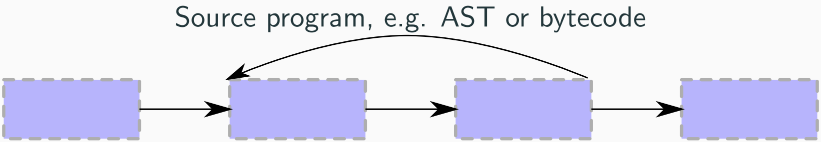
November 27, 2017

Northeastern University, Boston, USA

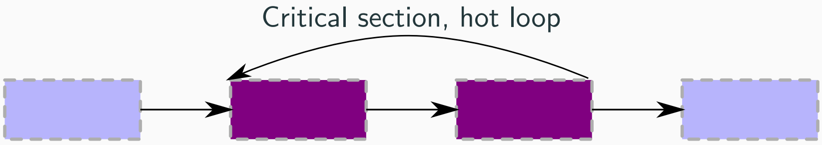


Context

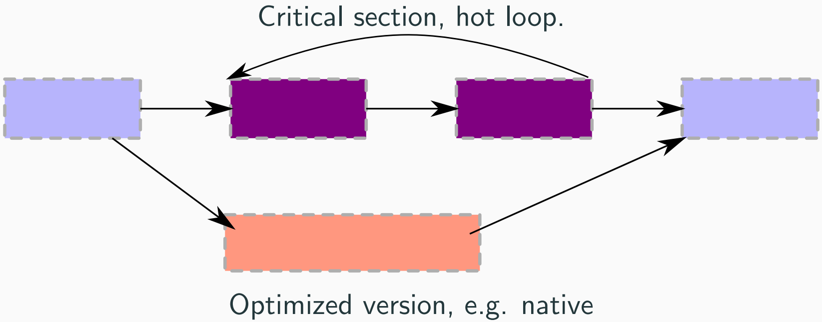
Just-in-time compilation: Deoptimization



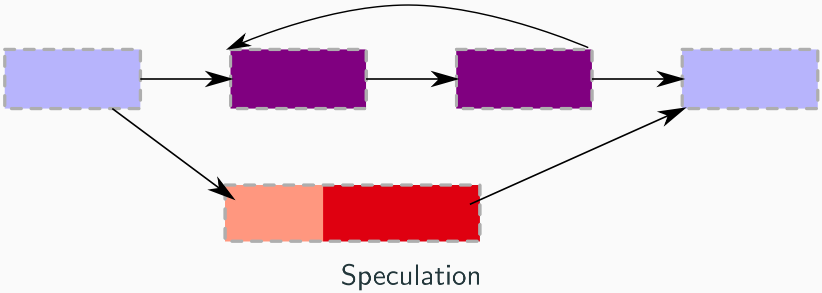
Just-in-time compilation: Deoptimization



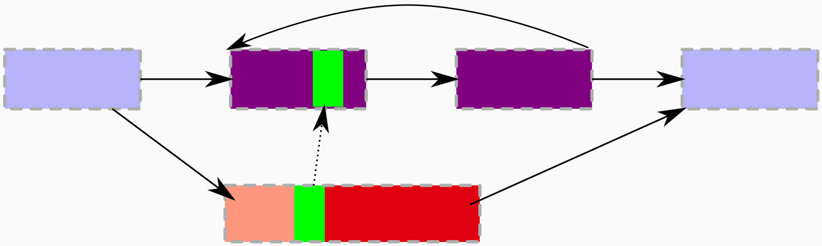
Just-in-time compilation: Deoptimization



Just-in-time compilation: Deoptimization



Just-in-time compilation: Deoptimization



Deoptimization/OSR point

Outline

Case study: V8 and speculation

Sourir: modeling deoptimization

Optimizations in sourir

Formalization

yield

Example: JS Array Representation in V8

```
// Considers only the first element
function eq(x) {
    return x[0] === 42
}
```

Example: JS Array Representation in V8

```
// Considers only the first element
function eq(x) {
    return x[0] === 42
}

// Array of length 3
var x = [42, 1, .2]
```

Example: JS Array Representation in V8

```
// Considers only the first element
function eq(x) {
    return x[0] === 42
}

// Array of length 3
var x = [42, 1, .2]

// Sparse array of length 3 with element 1 undefined
var x = [42];    x[2] = .2
```

yield

Compiler Correctness?

Multiple **versions** need to be considered.

Speculation requires keeping **deoptimization metadata**.

Difficulty: **intra-version** optimizations in the presence of **inter-version** controlflow.

Research Question: **Interaction** between deoptimization points and compiler optimizations.

yield

Sourir

Nothing left to remove

What does a JIT entail?

- High- and low-level representations
- Dynamic code generation
- Deoptimization metadata and supporting optimizations

Nothing left to remove

What does a JIT entail?

- ~~High- and low-level representations~~ One single language
- Dynamic code generation
- Deoptimization metadata and supporting optimizations

Nothing left to remove

What does a JIT entail?

- ~~High- and low-level representations~~ One single language
- ~~Dynamic code generation~~ One unrolled multi-version program
- Deoptimization metadata and supporting optimizations

Nothing left to remove

What does a JIT entail?

- ~~High- and low-level representations~~ One single language
- ~~Dynamic code generation~~ One unrolled multi-version program
- Deoptimization metadata and supporting optimizations ✓

HelloWorld

```
fun(c)
  Vluck
  |     assume c = 41 else fun.Vtough.L1 [c = c, o = 1]
  |     print 42
  Vtough
  |     var o = 1
  | L1  print c + o
```

Assume

```
fun(c)
    Vluck
    | L0   assume c = 41 else fun.Vtough.L1 [c = c, o = 1]
    | L1   print 42
    Vtough ...
```

$$\textbf{assume}\ e^*\ \textbf{else}\ \text{fun.V}_{\text{ver}}\text{.L}\ [x_1 = e_1, \ldots, x_n = e_n]$$

Predicates: list of boolean conditions e^*

Metadata:

where fun.Vver.L (unique location)

how $[x_1 = e_1, \ldots, x_n = e_n]$ (frame at bailout target)

Optimization: Constant Propagation

1)

$$\begin{aligned}&\textbf{var}\ \mathsf{o} = 1\\&\textbf{assume}\ \mathsf{c} = 41\ \textbf{else}\ F.V.L\ [\mathsf{c} = \mathsf{c}, \mathsf{o} = \mathsf{o}]\\&\textbf{print}\, \mathsf{c} + \mathsf{o}\end{aligned}$$

Optimization: Constant Propagation

1)

$$\begin{array}{l} \textbf{var } \mathsf{o} = 1 \\ \textbf{assume } \mathsf{c} = 41 \textbf{ else } F.V.L\ [\mathsf{c} = \mathsf{c}, \mathsf{o} = \mathsf{o}] \\ \textbf{print}\, \mathsf{c} + \mathsf{o} \end{array}$$

2)

$$\begin{array}{l} \textbf{assume } \mathsf{c} = 41 \textbf{ else } F.V.L\ [\mathsf{c} = \mathsf{c}, \mathsf{o} = 1] \\ \textbf{print}\, \mathsf{c} + 1 \end{array}$$

Optimization: Constant Propagation

1)

$$\begin{aligned}&\textbf{var } \mathsf{o} = 1\\&\textbf{assume } \mathsf{c} = 41 \textbf{ else } F.V.L\ [\mathsf{c} = \mathsf{c}, \mathsf{o} = \mathsf{o}]\\&\textbf{print}\, \mathsf{c} + \mathsf{o}\end{aligned}$$

2)

$$\begin{aligned}&\textbf{assume } \mathsf{c} = 41 \textbf{ else } F.V.L\ [\mathsf{c} = \mathsf{c}, \mathsf{o} = 1]\\&\textbf{print}\, \mathsf{c} + 1\end{aligned}$$

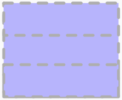
3)

$$\begin{aligned}&\textbf{assume } \mathsf{c} = 41 \textbf{ else } F.V.L\ [\mathsf{c} = \mathsf{c}, \mathsf{o} = 1]\\&\textbf{print}\, 42\end{aligned}$$

yield

Speculation Pipeline

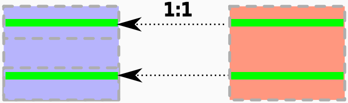
Baseline Version



Speculation Pipeline

Establish Invariants

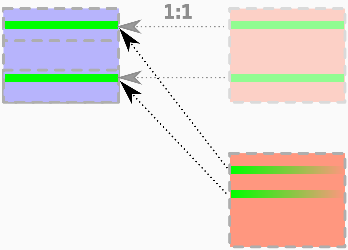
Copy Version: Assumes are trivial



Speculation Pipeline

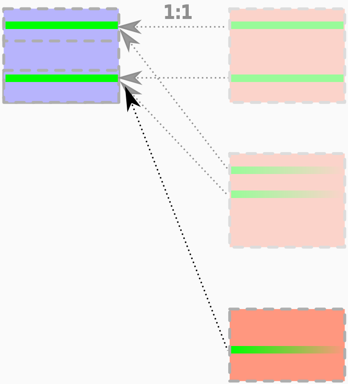
Preserve Invariants

Optimizations



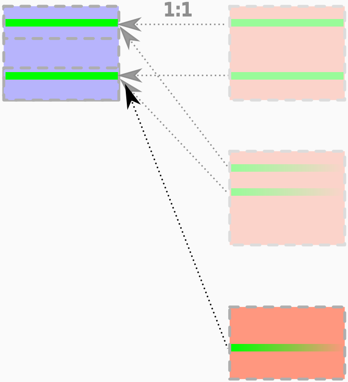
Finally

Most Optimized & Baseline Version



Finally

Equivalence result: Most Optimized & Baseline Version



Results

Explicit instruction for deoptimization

Invariants between versions

Optimizations are easy to adapt

Formalization

Execution: Operational semantics

Configurations:

$$C ::= \langle P\ I\ L\ K^*\ M\ E \rangle$$

Actions:

$$A ::= \text{read } \mathit{lit} \mid \text{print } \mathit{lit} \qquad A_\tau := A \mid \tau \qquad T ::= A^*.$$

Reduction:

$$C_1 \xrightarrow{A_\tau} C_2 \qquad\qquad C_1 \xrightarrow{T}{}^{*} C_2$$

Execution: A Peek

$$\frac{I(L) = \ \textbf{branch}\ e\ L_1\ L_2 \quad M\ E\ e \rightarrow \textbf{true}}{\langle P\ I\ L\ K^*\ M\ E\rangle \xrightarrow{\tau} \langle P\ I\ L_1\ K^*\ M\ E\rangle}\ [\textsc{BranchT}]$$

Execution: A Peek

[BranchT]

$$\frac{I(L) = \ \mathbf{branch}\ e\ L_1\ L_2 \quad M\ E\ e \to \mathbf{true}}{\langle P\ I\ L\ K^*\ M\ E\rangle \xrightarrow{\tau} \langle P\ I\ L_1\ K^*\ M\ E\rangle}$$

[Print]

$$\frac{I(L) = \ \mathbf{print}\ e \quad M\ E\ e \to \mathit{lit}}{\langle P\ I\ L\ K^*\ M\ E\rangle \xrightarrow{\mathsf{print}\ \mathit{lit}} \langle P\ I\ (L{+}1)\ K^*\ M\ E\rangle}$$

Equivalence: (weak) bisimulation

Relation R between the configurations over P_1 and P_2.

R is a weak **simulation** if:

$$
\begin{tikzcd}
C_1 \arrow[r, "A_\tau"] \arrow[d, dash, squiggly, "R"] & C_1' \\
C_2 &
\end{tikzcd}
\quad \Longrightarrow \quad
\begin{tikzcd}
C_1 \arrow[r, "A_\tau"] \arrow[d, dash, squiggly, "R"] & C_1' \arrow[d, dash, squiggly, "R"] \\
C_2 \arrow[r, "A_\tau"', "*" near end] & C_2'
\end{tikzcd}
$$

R is a weak **bisimulation** if R and R^{-1} are simulations.

Deoptimization invariants

Version equivalence All versions of a function are equivalent.

(Necessary to replace the active version)

Assumption transparence Bailing out **more** than necessary is correct.

(Necessary to add new assumptions)

yield

Optimization Pipeline: Create a new Version

$$\ldots \xrightarrow{\qquad\qquad\qquad\qquad A_\tau \qquad\qquad\qquad\qquad} \textbf{print}\,\mathsf{x}$$

Optimization Pipeline: Create a new Version

$$\ldots \xrightarrow{\qquad\qquad\qquad A_\tau \qquad\qquad\qquad} \textbf{print}\, \mathsf{x}$$

$$\ldots \xrightarrow{\qquad\qquad\qquad A_\tau \qquad\qquad\qquad} \textbf{print}\, \mathsf{x}$$

Optimization Pipeline: Create a new Version

$$\begin{tikzcd}
\ldots \arrow[r, "A_\tau"] & \textbf{assume true else } F.V.\mathrm{L1}\ [\ldots] \arrow[r, "\tau"] \arrow[dr, "\tau"] & \textbf{print}\ \mathrm{x} \\
\ldots \arrow[rr, "A_\tau"] & & \textbf{print}\ \mathrm{x}
\end{tikzcd}$$

Optimization Pipeline: Create a new Version

$$
\begin{tikzcd}
\dots \arrow[r, "A_\tau"] \arrow[d, squiggly, dash] & \textbf{assume true else } F.V.\mathrm{L1}\ [\dots] \arrow[r, "\tau"] \arrow[dr, "\tau"] \arrow[dr, squiggly] & \textbf{print}\ \mathrm{x} \arrow[d, squiggly, dash] \\
\dots \arrow[rr, "A_\tau"] & & \textbf{print}\ \mathrm{x}
\end{tikzcd}
$$

Conclusion

All you need for speculation: versions + checkpoints

Correctness of Speculative Optimizations with Dynamic Deoptimization (POPL' 18)
https://arxiv.org/abs/1711.03050

https://www.o1o.ch/talk-sourir-rmod.pdf

Advanced Topics

Adding more assumptions

$$
\begin{tikzcd}
\textbf{P2}\quad \dots \arrow[r, "\tau"] \arrow[d, squiggly, no head] & \textbf{assume}\ e^* \ \textbf{else}\ F.V.\mathrm{L1}\ [\ldots] \arrow[d, squiggly, no head] & \\
\textbf{P1}\quad \dots \arrow[r, "\tau"] & \textbf{assume true else}\ F.V.\mathrm{L1}\ [\ldots] \arrow[dr, "\tau"] & \\
\dots \arrow[rr, "\tau"] & & \textbf{print}\ \mathrm{x}
\end{tikzcd}
$$

Adding more assumptions

$$
\begin{tikzcd}
\textbf{P2} & \dots \arrow[r, "\tau"] \arrow[d, squiggly, no head] & \textbf{assume } e^* \textbf{ else } F.V.\mathrm{L1}\ [\dots] \arrow[d, squiggly, no head] \arrow[ddr, "\tau"] & \\
\textbf{P1} & \dots \arrow[r, "\tau"] & \textbf{assume true else } F.V.\mathrm{L1}\ [\dots] & \\
& \dots \arrow[rr, "\tau"] & & \textbf{print}\ \mathsf{x}
\end{tikzcd}
$$

Is deoptimizing in P2 correct, even if P1 does not deoptimize?

Adding more assumptions

$$
\begin{tikzcd}
\mathbf{P2} & \dots \arrow[r, "\tau"] \arrow[d, dash, squiggly] & \textbf{assume } e^* \textbf{ else } F.V.\mathrm{L1}\ [\dots] \arrow[d, dash, squiggly] \arrow[ddr, "\tau"] & \\
\mathbf{P1} & \dots \arrow[r, "\tau"] & \textbf{assume true else } F.V.\mathrm{L1}\ [\dots] \arrow[dr, squiggly] & \\
& \dots \arrow[rr, "\tau"] & & \textbf{print}\ \mathrm{x}
\end{tikzcd}
$$

Is deoptimizing in P2 correct, even if P1 does not deoptimize?

Yes, because of assumption transparency in P1.

How many deoptimization points are necessary?

Deoptimization points are expensive. How many are necessary?

Should assume be split into framestate and guard instructions?
(unrestricted deoptimization)

Unrestricted deoptimization is just a transformation

before:

$$
\begin{array}{ll}
 & \textbf{assume true else } \mathsf{size.V_b.L_0}\ [x = x] \\
 & \textbf{branch } x = \textbf{nil } \mathsf{L_2}\ \mathsf{L_1} \\
\mathsf{L_1} & x \leftarrow x[0] \\
 & \textbf{return } x * \mathsf{el} \\
\mathsf{L_2} & \ldots
\end{array}
$$

after:

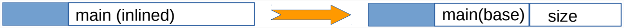
$$
\begin{array}{ll}
 & \textbf{var } x0 = x \\
 & \textbf{branch } x = \textbf{nil } \mathsf{L_2}\ \mathsf{L_1} \\
\mathsf{L_4} & x \leftarrow x[0] \\
 & \textbf{assume } x = 1 \textbf{ else } \mathsf{size.V_b.L_0}\ [x = x0] \\
 & \textbf{return } 1 * \mathsf{el} \\
\mathsf{L_3} & \ldots
\end{array}
$$

We can inline with deoptimization points

```
main( )
  Vinlined
  |      array vec = [1, 2, 3, 4]
  |      var size = nil
  |      var obj = vec
  |      assume obj ≠ nil else   size.Vbase.L1 [. . .]
  |                              main.Vbase.Lret size [. . .]
  |      var len = length(obj)
  |      size ← len ∗ 4
  |      drop len
  |      drop obj
  |      goto Lret
  | Lret print size
  Vbase . . .
```

```
main( )
  Vbase
  |      array vec = [1, 2, 3, 4]
  |      call size = size(vec)
  | Lret print size
size(obj)
  Vopt
  |   assume obj ≠ nil else . . .
  |   var len = length(obj)
  |   return len ∗ 4
  Vbase . . .
```

Need for an extra frame in the inlined version



Future Work

experimental validation

bidirectional transformations