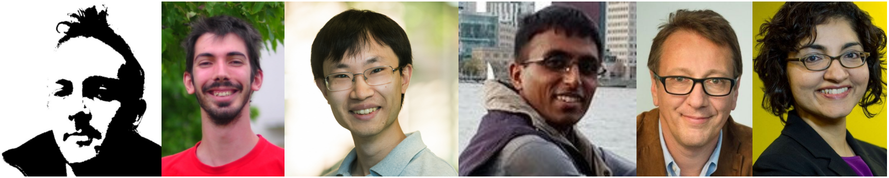
Speculative Optimizations without Fear

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

Northeastern University, Boston, USA

September 12, 2017



1 Context

2 Sourir

3 Formalization

4 Optimizations

Section 1

Context

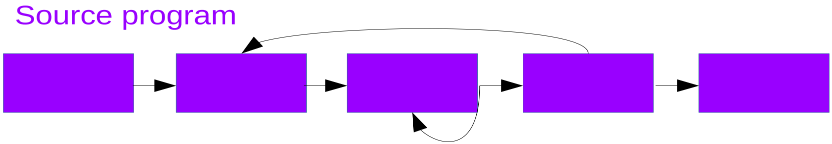
Our work

Just-in-time (JIT) compilation is essential to efficient dynamic language implementations.
(Javascript, Lua, R... Java)

There is a blind spot in our formal understanding of JITs: speculation.

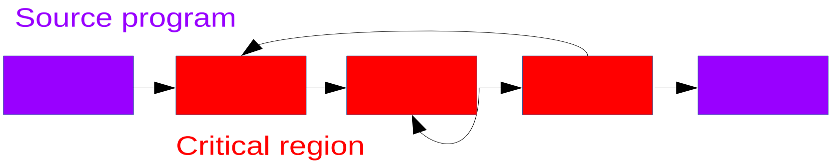
We present a language design to study speculative optimizations and prove them correct.

Just-in-time compilation



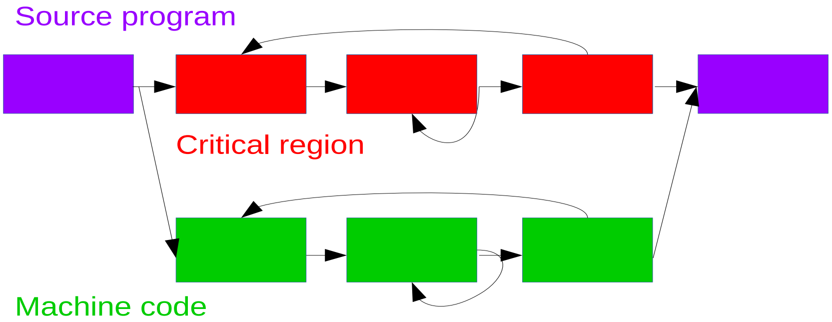
JITs:

Just-in-time compilation



JITs: Profiling

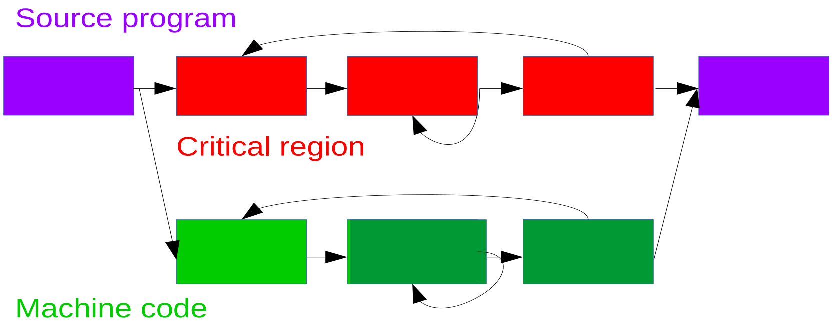
Just-in-time compilation



JITs: Profiling

+ High/Low languages
+ Dynamic code generation/mutation

Just-in-time compilation

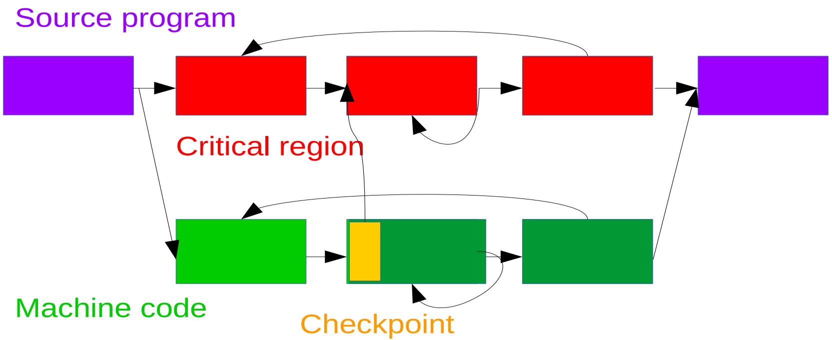


JITs: Profiling

+ High/Low languages
+ Dynamic code generation/mutation

+ Speculation

Just-in-time compilation



JITs: Profiling

+ High/Low languages
+ Dynamic code generation/mutation

+ Speculation and bailout

JITs:

- profiling
- high- and low-level languages (or multi-tiers, etc.)
- dynamic code generation + mutation

- speculation and bailout

JITs:

- profiling
- high- and low-level languages (or multi-tiers, etc.)
- dynamic code generation + mutation
  eliminates interpretation overhead (constant factor)
- speculation and bailout

JITs:

- profiling
- high- and low-level languages (or multi-tiers, etc.)
- dynamic code generation + mutation
  eliminates interpretation overhead (constant factor)
- speculation and bailout
  eliminates **dynamic features** overhead:
  dispatch (OO languages), type checks (Java),
  code loading (Java), redefinable primitives (R...)

JITs:

- profiling
- high- and low-level languages (or multi-tiers, etc.)
- dynamic code generation + mutation
  eliminates interpretation overhead (constant factor)
- speculation and bailout
  eliminates **dynamic features** overhead:
  dispatch (OO languages), type checks (Java),
  code loading (Java), redefinable primitives (R...)

JIT formalization: Myreen [2010]

- Stack language and x86 assembly
- dynamic code generation
- code mutation

mechanized in HOL!

JITs:

- profiling
- high- and low-level languages (or multi-tiers, etc.)
- dynamic code generation + mutation
  eliminates interpretation overhead (constant factor)
- speculation and bailout
  eliminates **dynamic features** overhead:
  dispatch (OO languages), type checks (Java),
  code loading (Java), redefinable primitives (R...)

JIT formalization: Myreen [2010]

- Stack language and x86 assembly
- dynamic code generation
- code mutation

mechanized in HOL!

JITs:

- profiling
- high- and low-level languages (or multi-tiers, etc.)
- dynamic code generation + mutation
  eliminates interpretation overhead (constant factor)
- speculation and bailout
  eliminates **dynamic features** overhead:
  dispatch (OO languages), type checks (Java),
  code loading (Java), redefinable primitives (R...)

JIT formalization: Myreen [2010]

- Stack language and x86 assembly
- dynamic code generation
- code mutation

mechanized in HOL!

What about **speculation**?

JITs:

- profiling
- high- and low-level languages (or multi-tiers, etc.)
- dynamic code generation + mutation
  eliminates interpretation overhead (constant factor)
- speculation and bailout
  eliminates **dynamic features** overhead:
  dispatch (OO languages), type checks (Java),
  code loading (Java), redefinable primitives (R...)

JIT formalization: Myreen [2010]

- Stack language and x86 assembly
- dynamic code generation
- code mutation

mechanized in HOL!

What about **speculation**? This work.

What do we want to know?

Speculation requires keeping **bailout data**.

How should optimizations maintain/transform bailout data?
(inlining is tricky)

Does the presence of checkpoint restrict optimizations?
(hoisting writes or IO is tricky)

When an assumption fails, how much of the other optimizations can keep?
(non-stack-order is tricky)

How should practitioners reason about correctness?

DEEP SPECULATION.

THE OIL-SPECULATOR'S DREAM.

Harper's Weekly cartoon of February 11, 1865. [8]

Sourir

- high- and low-level languages
- dynamic code generation
- speculative optimization and bailout

Sourir

- ~~high- and low-level languages~~ a single bytecode language
- dynamic code generation
- speculative optimization and bailout

Sourir

- ~~high- and low-level languages~~ a single bytecode language
- ~~dynamic code generation~~ one unrolled multi-version program
- speculative optimization and bailout

Sourir

- ~~high- and low-level languages~~ a single bytecode language
- ~~dynamic code generation~~ one unrolled multi-version program
- speculative optimization and bailout a **checkpoint** instruction

Sourir

- ~~high- and low-level languages~~ a single bytecode language
- ~~dynamic code generation~~ one unrolled multi-version program
- speculative optimization and bailout a **checkpoint** instruction

$F_{fun}(c) \rightarrow$
$V_{tough} \rightarrow$
$L_0 \quad : \quad \textbf{var } o = 1$
$L_1 \quad : \quad \textbf{print } (c + o)$
$V_{luck} \rightarrow$
$L_0 \quad : \quad \textbf{assume } [(c = 41)] \textbf{ else } \langle F_{fun}.V_{tough}.L_1 \ [c = c, o = 1] \rangle$
$L_1 \quad : \quad \textbf{print } 42$

Contribution

A language design to model speculative optimization: `Sourir`

A kit of correct program transformations and optimizations

A methodology to reason about correct speculative optimizations

Section 2

Sourir

A simple bytecode language

$$
\begin{array}{lll}
i & ::= & \\
 & | & \textbf{var}\ x = e \\
 & | & \textbf{drop}\ x \\
 & | & x \leftarrow e \\
 & | & \textbf{array}\ x[e] \\
 & | & \textbf{array}\ x = [e^*] \\
 & | & x[e_1] \leftarrow e_2 \\
 & | & \textbf{branch}\ e\ \ L_1\ \ L_2 \\
 & | & \textbf{goto}\ L \\
 & | & \textbf{print}\ e \\
 & | & \textbf{read}\ x \\
 & | & \textbf{call}\ x = e(e^*) \\
 & | & \textbf{return}\ e \\
 & | & \textbf{assume}\ [e^*]\ \textbf{else}\ \xi\ \tilde{\xi}^* \\
 & | & \textbf{stop}
\end{array}
$$

$$
\begin{array}{lll}
e & ::= & \\
 & | & se \\
 & | & x[se] \\
 & | & \textbf{length}(se) \\
 & | & primop\,(se^*)
\end{array}
$$

$$
\begin{array}{lll}
se & ::= & \\
 & | & lit \\
 & | & `F \\
 & | & x
\end{array}
$$

$$
\begin{array}{lll}
lit & ::= & \\
 & | & \ldots, -1, 0, 1, \ldots \\
 & | & \textbf{nil} \mid \textbf{true} \mid \textbf{false}
\end{array}
$$

Versions

P	$::=$	$(F(x^*) \to D_F)^*$	**program**: a list of named functions
D_F	$::=$	$(V \to I)^*$	**function definition**: list of versioned instruction streams
I	$::=$	$(L : i)^*$	**instruction stream** with labeled instructions

Checkpoints

Checkpoint: **guards** + **bailout data**.

$$\textbf{assume } [(c = 41)] \textbf{ else } \langle F_{fun}.V_{tough}.L_1 \; [c = c, o = 1] \rangle$$

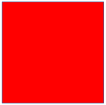
Guards: just a list of expressions returning booleans.

Bailout data:

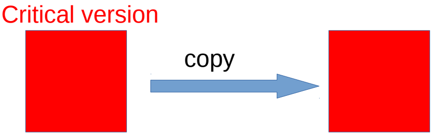
- where to go: $F_f.V_w.L_l$
- in what state: $[x_1 = e_1, .., x_n = e_n]$
- (plus more: see inlining)

Speculative optimization pipeline

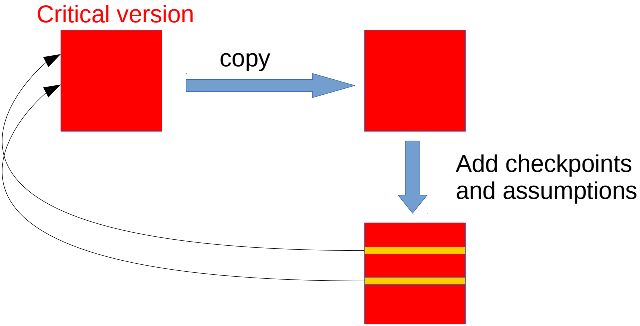
Critical version



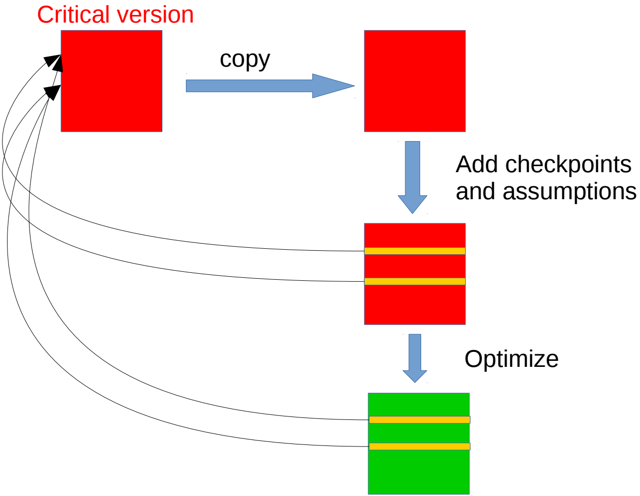
Speculative optimization pipeline



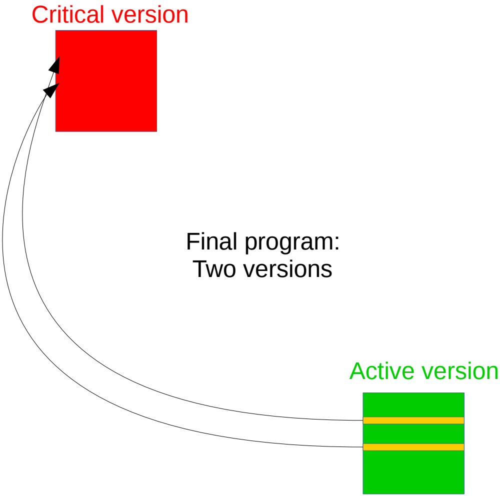
Speculative optimization pipeline



Speculative optimization pipeline



Speculative optimization pipeline



Speculative optimization pipeline

Section 3

Formalization

Execution: Operational semantics

Configurations:

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

Actions:

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

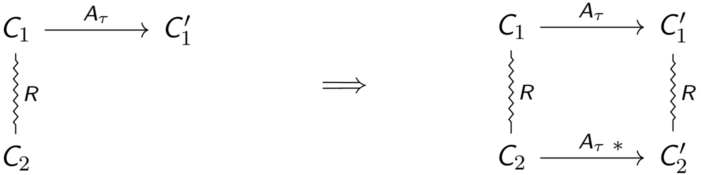
Reduction:

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

Equivalence: (weak) bisimulation

Relation R between the configurations over P_1 and P_2.

R is a weak **simulation** if:



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

Bailout invariants

Version invariant: All versions of a function are equivalent.
(Necessary to replace the active version)

Bailout invariant: Bailing out **more** than necessary is correct.
(Necessary to add new assumptions)

Section 4

Optimizations

Branch pruning – from the kit

$$V_{base} \rightarrow$$

$$\begin{array}{lcl} L_1 & : & \textbf{branch}\,(tag\ =\ INT)\ L_{int}\ L_{nonint} \\ L_{int} & : & \ldots \\ L_{nonint} & : & \ldots \end{array}$$

Branch pruning – from the kit

$V_{base} \rightarrow$

$$\begin{array}{lcl} L_1 & : & \textbf{branch}\,(tag\ =\ INT)\ L_{int}\ L_{nonint} \\ L_{int} & : & \ldots \\ L_{nonint} & : & \ldots \end{array}$$

$V_{opt} \rightarrow$

$$\begin{array}{lcl} L_0 & : & \textbf{assume}\ [(tag\ =\ INT)]\ \textbf{else}\ \langle F.V_{base}.L_1\ \delta \rangle \\ L_1 & : & \textbf{branch}\,(tag\ =\ INT)\ L_{int}\ L_{nonint} \\ L_{int} & : & \ldots \\ L_{nonint} & : & \ldots \end{array}$$

Checkpoint + guard inserted

Bailout invariant!

Branch pruning – from the kit

$$V_{base} \rightarrow$$

$$\begin{array}{lcl} L_1 & : & \textbf{branch}\,(tag\ =\ INT)\ L_{int}\ L_{nonint} \\ L_{int} & : & \ldots \\ L_{nonint} & : & \ldots \end{array}$$

$$V_{opt} \rightarrow \quad \begin{array}{lcl} L_0 & : & \textbf{assume}\ [(tag\ =\ INT)]\ \textbf{else}\ \langle F.V_{base}.L_1\ \delta\rangle \\ L_1 & : & \textbf{branch}\,\textbf{true}\ L_{int}\ L_{nonint} \\ L_{int} & : & \ldots \\ L_{nonint} & : & \ldots \end{array}$$

constant folding

Branch pruning – from the kit

$$V_{base} \rightarrow$$

$$\begin{array}{lcl} L_1 & : & \textbf{branch}\,(tag\ =\ INT)\ L_{int}\ L_{nonint} \\ L_{int} & : & \ldots \\ L_{nonint} & : & \ldots \end{array}$$

$$V_{opt} \rightarrow \quad \begin{array}{lcl} L_0 & : & \textbf{assume}\ [(tag\ =\ INT)]\ \textbf{else}\ \langle F.V_{base}.L_1\ \delta\rangle \\ L_{int} & : & \ldots \end{array}$$

unreachable code elimination

Inlining

$$
\begin{array}{l}
F_{main}(\,) \rightarrow \\
\quad V_{inlined} \rightarrow \\
\qquad \begin{array}{lcl}
L_0 & : & \textbf{array } vec = [1, 2, 3, 4] \\
L_2 & : & \textbf{var } size = \textbf{nil} \\
L_3 & : & \textbf{var } obj = vec \\
L_{cp_1} & : & \textbf{assume } [(obj \neq \textbf{nil})] \textbf{ else } \ldots \\
L_5 & : & \textbf{var } len = \textbf{length}(obj) \\
L_6 & : & size \leftarrow (len * 4) \\
L_7 & : & \textbf{drop } len \\
L_8 & : & \textbf{drop } obj \\
L_9 & : & \textbf{goto } L_{ret} \\
L_{ret} & : & \textbf{print } size
\end{array} \\
\quad V_{base} \rightarrow \; \ldots
\end{array}
$$

$$
\begin{array}{l}
F_{main}(\,) \rightarrow \\
\quad V_{base} \rightarrow \\
\qquad \begin{array}{lcl}
L_0 & : & \textbf{array } vec = [1, 2, 3, 4] \\
L_2 & : & \textbf{call } size = \text{‘}F_{size}(vec) \\
L_{ret} & : & \textbf{print } size
\end{array} \\
, F_{size}(obj) \rightarrow \\
\quad V_{opt} \rightarrow \\
\qquad \begin{array}{lcl}
L_{cp_1} & : & \textbf{assume } [(obj \neq \textbf{nil})] \textbf{ else } \ldots \\
L_{vec} & : & \textbf{var } len = \textbf{length}(obj) \\
L_3 & : & \textbf{return } (len * 4)
\end{array} \\
\quad V_{base} \rightarrow \; \ldots
\end{array}
$$

Inlining

$$
\begin{array}{l}
F_{main}(\,) \rightarrow \\
\quad V_{inlined} \rightarrow \\
\qquad \begin{array}{lcl}
L_0 & : & \textbf{array}\ vec = [1,2,3,4] \\
L_2 & : & \textbf{var}\ size = \textbf{nil} \\
L_3 & : & \textbf{var}\ obj = vec \\
L_{cp_1} & : & \textbf{assume}\ [(obj \neq \textbf{nil})]\ \textbf{else}\ \ldots \\
L_5 & : & \textbf{var}\ len = \textbf{length}(obj) \\
L_6 & : & size \leftarrow (len * 4) \\
L_7 & : & \textbf{drop}\ len \\
L_8 & : & \textbf{drop}\ obj \\
L_9 & : & \textbf{goto}\ L_{ret} \\
L_{ret} & : & \textbf{print}\ size
\end{array} \\
\quad V_{base} \rightarrow \ \ldots
\end{array}
$$

$$
\begin{array}{l}
F_{main}(\,) \rightarrow \\
\quad V_{base} \rightarrow \\
\qquad \begin{array}{lcl}
L_0 & : & \textbf{array}\ vec = [1,2,3,4] \\
L_2 & : & \textbf{call}\ size = `F_{size}(vec) \\
L_{ret} & : & \textbf{print}\ size
\end{array} \\
, F_{size}(obj) \rightarrow \\
\quad V_{opt} \rightarrow \\
\qquad \begin{array}{lcl}
L_{cp_1} & : & \textbf{assume}\ [(obj \neq \textbf{nil})]\ \textbf{else}\ \ldots \\
L_{vec} & : & \textbf{var}\ len = \textbf{length}(obj) \\
L_3 & : & \textbf{return}\ (len * 4)
\end{array} \\
\quad V_{base} \rightarrow \ \ldots
\end{array}
$$

Need for an extra frame in the inlined version:

$\langle F_{main}.V_{base}.L_{ret}\ size\ [vec = vec]\rangle$

Conclusion

All you need for speculation: versions + checkpoints.

Future work: bidirectional transformations.

Thanks!
Questions?

Magnus O. Myreen. Verified just-in-time compiler on x86. In **Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**, POPL '10, pages 107–118, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706313.