Speculative Optimizations without Fear

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

Northeastern University, Boston, USA

November 3, 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

Source program

JITs:
Just-in-time compilation

Source program

Critical region

JITs: Profiling
Just-in-time compilation

Source program

Critical region

Machine code

JITs:

- Profiling
  - High/Low languages
  - Dynamic code generation/mutation
Just-in-time compilation

Source program

Critical region

Machine code

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.
Sourir

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

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

a single bytecode language
Sourir

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

a single bytecode language
one unrolled multi-version program
Sourir

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

a single bytecode language
one unrolled multi-version program
a **checkpoint** instruction
Sourir

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

a single bytecode language
one unrolled multi-version program

a **checkpoint** instruction

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

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

\[
e ::= \begin{align*}
se \\
x[se] \\
\text{length}(se) \\
\text{primop}(se^*) \\
\text{lit} \\
F \\
x \\
\text{nil} | \text{true} | \text{false}
\end{align*}
\]
Versions

\[ P ::= (F(x^*) \rightarrow D_F)^* \quad \text{program: a list of named functions} \]

\[ D_F ::= (V \rightarrow I)^* \quad \text{function definition: list of versioned instruction streams} \]

\[ I ::= (L : i)^* \quad \text{instruction stream with labeled instructions} \]

\[
\begin{align*}
F_{\text{fun}}(c) &\rightarrow \\
V_{\text{tough}} &\rightarrow \\
L_0 &\colon \quad \text{var } o = 1 \\
L_1 &\colon \quad \text{print } c + o \\
V_{\text{luck}} &\rightarrow \\
L_0 &\colon \quad \text{assume } c = 41 \ \text{else } F_{\text{fun}} \cdot V_{\text{tough}} \cdot L_1 \ [c = c, o = 1] \\
L_1 &\colon \quad \text{print } 42
\end{align*}
\]
Checkpoints

Checkpoint: **guards + bailout data.**

\[
\text{assume } c = 41 \text{ else } F_{\text{fun}}. V_{\text{tough}}. L_1 \ [c = c, o = 1]
\]

Guards: just a list of expressions returning booleans.

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

Checkpoint: **guards + bailout data.**

\[
\text{assume } c = 41 \text{ else } F_{\text{fun}}.V_{\text{tough}}.L_1 \ [c = c, o = 1]
\]

Guards: just a list of expressions returning booleans.

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

Not a branch. (inlining)
Checkpoints simplify optimizations...
Checkpoints

Checkpoint: **guards + bailout data.**

\[
\text{assume } c = 41 \text{ else } F_{\text{fun}}.V_{\text{tough}}.L_1 [c = c, o = 1]
\]

Guards: just a list of expressions returning booleans.

Bailout data:

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

Not a branch. (inlining)

Checkpoints simplify optimizations...and correctness proofs!
Speculative optimization pipeline

Critical version
Speculative optimization pipeline
Speculative optimization pipeline

Critical version

Add checkpoints and assumptions
Speculative optimization pipeline

Critical version

copy

Add checkpoints and assumptions

Optimize
Speculative optimization pipeline

Critical version

Final program:
Two versions

Active version
Speculative optimization pipeline

Critical version
Section 3

Formalization
Execution: Operational semantics

Configurations:

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

Actions:

\[ A ::= \text{read } lit \mid \text{print } lit \]

\[ A_{\tau} ::= A \mid \tau \]

\[ T ::= A^*. \]

Reduction:

\[ C_1 \xrightarrow{A_{\tau}}^* C_2 \]

\[ C_1 \xrightarrow{T}^* C_2 \]
Equivalence: (weak) bisimulation

Relation $R$ between the configurations over $P_1$ and $P_2$.

$R$ is a weak **simulation** if:

$$
\begin{array}{c}
C_1 \xrightarrow{A_\tau} C'_1 \\
\searrow R \\
C_2
\end{array} \implies
\begin{array}{c}
C_1 \xrightarrow{A_\tau} C'_1 \\
\searrow R \\
C_2 \xrightarrow{A_\tau^*} C'_2
\end{array}
$$

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

$$
\begin{array}{c}
C_1 \xrightarrow{A_\tau} C'_1 \\
\searrow R \\
C_2 \xrightarrow{A_\tau^*} C'_2
\end{array}
$$
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

\[ V_{base} \rightarrow \]

\[ L_1 : \text{branch} \; \text{tag} = \text{INT} \; L_{int} \; L_{nonint} \]

\[ L_{int} : \; \ldots \]

\[ L_{nonint} : \; \ldots \]
Branch pruning – from the kit

\[ V_{\text{base}} \rightarrow \]

\[ L_1 : \quad \text{branch } \textit{tag} = \text{INT} \ L_{\text{int}} \ L_{\text{nonint}} \]
\[ L_{\text{int}} : \quad \ldots \]
\[ L_{\text{nonint}} : \quad \ldots \]

\[ V_{\text{opt}} \rightarrow \]

\[ L_0 : \quad \text{assume } \textit{tag} = \text{INT} \ \text{else } F. V_{\text{base}}.L_1 \ \delta \]
\[ L_1 : \quad \text{branch } \textit{tag} = \text{INT} \ L_{\text{int}} \ L_{\text{nonint}} \]
\[ L_{\text{int}} : \quad \ldots \]
\[ L_{\text{nonint}} : \quad \ldots \]

Checkpoint + guard inserted

\textbf{Bailout invariant!}
Branch pruning – from the kit

\[ V_{\text{base}} \rightarrow \]

\[ L_1 : \textbf{branch} \ tag = \text{INT} \ L_{\text{int}} \ L_{\text{nonint}} \]
\[ L_{\text{int}} : \ldots \]
\[ L_{\text{nonint}} : \ldots \]

\[ V_{\text{opt}} \rightarrow \]

\[ L_0 : \textbf{assume} \ tag = \text{INT} \ \textbf{else} \ \text{F}.V_{\text{base}}.L_1 \ \delta \]
\[ L_1 : \textbf{branch true} \ L_{\text{int}} \ L_{\text{nonint}} \]
\[ L_{\text{int}} : \ldots \]
\[ L_{\text{nonint}} : \ldots \]

constant folding
Branch pruning – from the kit

\[ V_{\text{base}} \rightarrow \]

\[ L_1 : \quad \text{branch} \; \text{tag} = \text{INT} \; L_{\text{int}} \; L_{\text{nonint}} \]

\[ L_{\text{int}} : \quad \ldots \]

\[ L_{\text{nonint}} : \quad \ldots \]

\[ V_{\text{opt}} \rightarrow \]

\[ L_0 : \quad \text{assume} \; \text{tag} = \text{INT} \; \text{else} \; F \; V_{\text{base}}.L_1 \; \delta \]

\[ L_{\text{int}} : \quad \ldots \]

unreachable code elimination
Inlining

\[ F_{\text{main}}() \rightarrow \]
\[ V_{\text{inlined}} \rightarrow \]
\[ L_0 : \quad \text{array vec} = [1, 2, 3, 4] \]
\[ L_2 : \quad \text{var size} = \text{nil} \]
\[ L_3 : \quad \text{var obj} = \text{vec} \]
\[ L_{cp1} : \quad \text{assume obj} \neq \text{nil} \text{ else} \ldots \]
\[ L_5 : \quad \text{var len} = \text{length}(obj) \]
\[ L_6 : \quad \text{size} \leftarrow \text{len} \times 4 \]
\[ L_7 : \quad \text{drop len} \]
\[ L_8 : \quad \text{drop obj} \]
\[ L_9 : \quad \text{goto} L_{\text{ret}} \]
\[ L_{\text{ret}} : \quad \text{print size} \]
\[ V_{\text{base}} \rightarrow \ldots \]

\[ F_{\text{main}}() \rightarrow \]
\[ V_{\text{base}} \rightarrow \]
\[ L_0 : \quad \text{array vec} = [1, 2, 3, 4] \]
\[ L_2 : \quad \text{call size} = F_{\text{size}}(\text{vec}) \]
\[ L_{\text{ret}} : \quad \text{print size} \]

\[ F_{\text{size}}(\text{obj}) \rightarrow \]
\[ V_{\text{opt}} \rightarrow \]
\[ L_{cp1} : \quad \text{assume obj} \neq \text{nil} \text{ else } \xi \]
\[ L_{\text{vec}} : \quad \text{var len} = \text{length}(\text{obj}) \]
\[ L_3 : \quad \text{return} \text{len} \times 4 \]
\[ V_{\text{base}} \rightarrow \ldots \]
Inlining

\[ F_{\text{main}}(\ ) \rightarrow \]
\[ V_{\text{inlined}} \rightarrow \]
\[ L_0 : \quad \text{array} \ v = [1, 2, 3, 4] \]
\[ L_2 : \quad \text{var} \ s = \text{nil} \]
\[ L_3 : \quad \text{var} \ o = v \]
\[ L_{cp1} : \quad \text{assume} \ o \neq \text{nil} \text{ else } \ldots \]
\[ L_5 : \quad \text{var} \ l = \text{length}(o) \]
\[ L_6 : \quad s \leftarrow l \times 4 \]
\[ L_7 : \quad \text{drop} \ l \]
\[ L_8 : \quad \text{drop} \ o \]
\[ L_9 : \quad \text{goto} \ L_{\text{ret}} \]
\[ L_{\text{ret}} : \quad \text{print} \ s \]

\[ V_{\text{base}} \rightarrow \ldots \]

\[ F_{\text{main}}(\ ) \rightarrow \]
\[ V_{\text{base}} \rightarrow \]
\[ L_0 : \quad \text{array} \ v = [1, 2, 3, 4] \]
\[ L_2 : \quad \text{call} \ s = F_{\text{size}}(v) \]
\[ L_{\text{ret}} : \quad \text{print} \ s \]

\[ F_{\text{size}}(o) \rightarrow \]
\[ V_{\text{opt}} \rightarrow \]
\[ L_{cp1} : \quad \text{assume} \ o \neq \text{nil} \text{ else } \xi \]
\[ L_{vec} : \quad \text{var} \ l = \text{length}(o) \]
\[ L_3 : \quad \text{return} \ l \times 4 \]
\[ V_{\text{base}} \rightarrow \ldots \]

\[ \text{assume} \ o \neq \text{nil} \text{ else } \xi \langle F_{\text{main}} \cdot V_{\text{base}} \cdot L_{\text{ret}} \ s \rangle [v = v] \]
Conclusion

All you need for speculation: versions + checkpoints.

Future work: bidirectional transformations.

Thanks!
Questions?