Speculative Optimizations without Fear

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

Northeastern University, Boston, USA

July 19, 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

Source program

Critical region

Machine code

JITs:
+ Profiling
+ High/Low languages
+ Dynamic code generation/mutation
+ Speculation and bailout

Checkpoint
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...)

JIT formalization: Myreen [2010]
Stack language and x86 assembly
dynamic code generation
code mutation
mechanized in HOL!

What about speculation?
This work.
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
- **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.

© 2001 HARPWEEK

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
- 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

one unrolled multi-version program

a single bytecode language

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

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

A language design to model speculative optimization: Sourir

A kit of correct program transformations and optimizations
Section 2

Sourir
A simple bytecode language

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

\[
e ::= \begin{align*}
&\text{se} \\
&x[se] \\
&\text{length}(se) \\
&\text{primop}(\text{primopargs}) \\
&\text{se} ::= \begin{align*}
&\text{lit} \\
&x \\
&\text{primop}(\text{primopargs}) \\
&\text{lit} ::= \begin{align*}
&\ldots, -1, 0, 1, \ldots \\
&\text{nil} \\
&\text{true} \\
&\text{false}
\end{align*}
\end{align*}
\]
Versions

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

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

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

Checkpoint: \textbf{guards} + \textbf{bailout data}.

\begin{align*}
\text{assume } [(c = 41)] \text{ else } \langle F.V_{\text{tough}}.L_1 \rangle [c = c, o = 1]
\end{align*}

Guards: just a list of expressions returning booleans.

Bailout data:
- \textbf{where to go}: \langle F.V.L \rangle
- \textbf{in what state}: \[x_1 = e_1, \ldots, x_n = e_n]\n- (plus more: see inlining)
Speculative optimization pipeline
Speculative optimization pipeline
Speculative optimization pipeline

Critical version → copy → 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
Section 3

Formalization
Execution: Operational semantics

Configurations:

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

Actions:

\[ A ::= \text{read lit} \mid \text{print lit} \quad A_\tau ::= A \mid \tau \quad T ::= A^* \]

Reduction:

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

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

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

$$
\begin{array}{c}
C_1 \xrightarrow{A_\tau} C'_1 \\
\downarrow R \\
C_2
\end{array}
\quad \Rightarrow
\quad
\begin{array}{c}
C_1 \xrightarrow{A_\tau} C'_1 \\
\downarrow R \\
C_2 \xrightarrow{A_\tau \ast} C'_2
\end{array}
$$

$R$ 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

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

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

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

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

\[ \mathcal{V}_{base} \rightarrow \]

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

\[ \mathcal{V}_{opt} \rightarrow \]

\[ L_0 : \text{assume } [(tag = INT)] \ \text{else } \langle F.\mathcal{V}_{base}.L_1 \rangle \ \delta \]
\[ L_1 : \text{branch}(tag = INT) \ L_{\text{int}} \ L_{\text{nonint}} \]
\[ L_{\text{int}} : \ldots \]
\[ L_{\text{nonint}} : \ldots \]

Checkpoint + guard inserted

Bailout invariant!
Branch pruning – from the kit

\[ \nu_{base} \rightarrow \]

\[ \begin{align*}
L_1 & : \textbf{branch} (\text{tag} = \text{INT}) \; L_{\text{int}} \; L_{\text{nonint}} \\
L_{\text{int}} & : \ldots \\
L_{\text{nonint}} & : \ldots 
\end{align*} \]

\[ \nu_{opt} \rightarrow \]

\[ \begin{align*}
L_0 & : \textbf{assume} \ [(\text{tag} = \text{INT})] \; \textbf{else} \; \langle F \cdot \nu_{base}.L_1 \rangle \; \delta \\
L_1 & : \textbf{branch true} \; L_{\text{int}} \; L_{\text{nonint}} \\
L_{\text{int}} & : \ldots \\
L_{\text{nonint}} & : \ldots 
\end{align*} \]

constant folding
Branch pruning – from the kit

\[ V_{base} \rightarrow \]

\begin{align*}
L_1 & : \text{branch}(tag = \text{INT}) \ L_{\text{int}} \ L_{\text{nonint}} \\
L_{\text{int}} & : \ldots \\
L_{\text{nonint}} & : \ldots
\end{align*}

\[ V_{opt} \rightarrow \]

\begin{align*}
L_0 & : \text{assume} [(tag = \text{INT})] \ \text{else} \ \langle F.V_{base}.L_1 \rangle \ \delta \\
L_{\text{int}} & : \ldots
\end{align*}

unreachable code elimination
Inlining

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

\[ F_{\text{main}}(\ ) \rightarrow \]
\[ V_{\text{base}} \rightarrow \]
\[ L_0 : \text{array vec} = [1, 2, 3, 4] \]
\[ L_2 : \text{call size} = 'F_{\text{size}}(\text{vec})' \]
\[ L_{\text{ret}} : \text{print size} \]
\[ , F_{\text{size}}(\text{obj}) \rightarrow \]
\[ V_{\text{opt}} \rightarrow \]
\[ L_{cp1} : \text{assume} [(\text{obj} \neq \text{nil})] \text{ else } \ldots \]
\[ L_{\text{vec}} : \text{var len} = \text{length(obj)} \]
\[ L_3 : \text{return} (\text{len} \ast 4) \]
\[ V_{\text{base}} \rightarrow \ldots \]
## Inlining

\[ F_{\text{main}}() \rightarrow \]

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

\[ L_0 : \text{array} \ vec = [1, 2, 3, 4] \]
\[ L_2 : \text{var} \ size = \text{nil} \]
\[ L_3 : \text{var} \ obj = vec \]
\[ L_{cp_1} : \text{assume} \ [(obj \neq \text{nil})] \ \text{else} \ldots \]
\[ L_5 : \text{var} \ len = \text{length}(obj) \]
\[ L_6 : \text{size} \leftarrow (len \ast 4) \]
\[ L_7 : \text{drop} \ len \]
\[ L_8 : \text{drop} \ obj \]
\[ L_9 : \text{goto} \ L_{\text{ret}} \]
\[ V_{\text{base}} \rightarrow \ldots \]

\[ F_{\text{main}}() \rightarrow \]

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

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

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

\[ L_{cp_1} : \text{assume} \ [(obj \neq \text{nil})] \ \text{else} \ldots \]
\[ L_{vec} : \text{var} \ len = \text{length}(obj) \]
\[ L_3 : \text{return} \ ((len \ast 4)) \]
\[ V_{\text{base}} \rightarrow \ldots \]

Need for an extra frame in the inlined version:
\[ \langle F_{\text{main}}.V_{\text{base}}.L_{\text{ret}} \ size \ [vec = vec] \rangle \]
Conclusion

All you need for speculation: versions + checkpoints.

Future work: bidirectional transformations.

Thanks!
Questions?