Correctness of Speculative Optimizations with Dynamic Deoptimization

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

Northeastern University, Boston, USA

October 12, 2017
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

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!

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
- 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.
Sourir

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

fun(c)
tough
var o = 1
L1
print c + o
luck
assume c = 41
else
fun.tough.L1[c = c, o = 1]
print 42
Sourir

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

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

a single bytecode language
one unrolled multi-version program

fun(c)

v = 1
L1
println(c + v)

assume c = 41
else
  fun().tough().L1[c = c, v = 1]
println(42)
Sourir

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

(See Myreen [2010] for the first two.)
Sourir

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

(See Myreen [2010] for the first two.)

```plaintext
fun(c)
tough

    var o = 1
    L1
    print c + o

luck

    assume c = 41 else fun.tough.L1 [c = c, o = 1]
    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
A simple bytecode language

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

\[
e ::= \\
\text{se} \\
x[se] \\
\text{length}(se) \\
\text{primop}(se^*)
\]

\[
se ::= \\
\text{lit} \\
F \\
x
\]

\[
lit ::= \\
\ldots, -1, 0, 1, \ldots \\
nil | \text{true} | \text{false}
\]
Versions

\[ P ::= F(x^*) \rightarrow D_F, \ldots \textbf{program}: \text{a list of named functions} \]
\[ D_F ::= V \rightarrow I, \ldots \textbf{function definition}: \text{list of versioned instruction streams} \]
\[ I ::= L \rightarrow i, \ldots \textbf{instruction stream} \text{ with labeled instructions} \]

\[
\begin{align*}
\text{fun}(c) \\
\text{tough} \quad & \text{var } o = 1 \\
\text{L}_1 \quad & \text{print } c + o \\
\text{luck} \quad & \text{assume } c = 41 \textbf{ else fun}.\text{tough}.L_1 \ [c = c, o = 1] \\
& \text{print } 42
\end{align*}
\]
Checkpoint: guards + bailout data.

\[
\text{assume } c = 41 \text{ else fun.tough.L}_1 \ [c = c, o = 1]
\]

Guards: just a list of expressions returning booleans.

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

Checkpoint: **guards + bailout data.**

\[
\text{assume } c = 41 \text{ else fun.tough.L}_{1}[c = c, o = 1]
\]

Guards: just a list of expressions returning booleans.

Bailout data:
- where to go: \(F.V.L\)
- 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.**

\[
\text{assume } c = 41 \text{ else fun.tough.L}_1 [c = c, o = 1]
\]

Guards: just a list of expressions returning booleans.

Bailout data:
- where to go: \( F.V.L \)
- 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
Execution: Operational semantics

Configurations:

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

Actions:

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

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:

$C_1 \xrightarrow{A_\tau} C_1' \quad \Downarrow \quad \Downarrow R \quad \Downarrow R \quad \Downarrow R \quad \Downarrow R$

$C_2 \xrightarrow{A_\tau} C_2' \quad \Downarrow R \quad \Downarrow R \quad \Downarrow R \quad \Downarrow R$

$R$ 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)
Branch pruning – from the kit

\[
\begin{array}{c|c}
\text{base} & \text{branch} \tag{INT \ int \ nonint} \\
L_1 & \text{int} \quad \ldots \\
\text{int} & \ldots \\
\text{nonint} & \ldots \\
\end{array}
\]
Branch pruning – from the kit

**base**

```
L₁ \quad \textbf{branch} \; \text{tag} = \text{INT} \; \text{int} \; \text{nonint}
```

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

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

**opt**

```
L₁ \quad \textbf{assume} \; \text{tag} = \text{INT} \; \textbf{else} \; \text{F.base.L₁} \; \ldots
```

```
\quad \textbf{branch} \; \text{tag} = \text{INT} \; \text{int} \; \text{nonint}
```

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

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

Checkpoint + guard inserted

Bailout invariant!
Branch pruning – from the kit

```plaintext
base
L1 | branch tag = INT int nonint
  | int    ... 
  | nonint ... 

opt
L1 | assume tag = INT else F.base.L1 [...]
  | branch true int nonint
  | int    ... 
  | nonint ... 

constant folding
```
Branch pruning – from the kit

base
  | \texttt{L}_1 \quad \texttt{branch} \tag = \texttt{INT} \ int \ \texttt{nonint}
  | \texttt{int} \quad \ldots
  | \texttt{nonint} \quad \ldots

opt
  | \texttt{L}_1 \quad \texttt{assume} \tag = \texttt{INT} \ \texttt{else} \ F.\texttt{base}.\texttt{L}_1 \ [\ldots]
  | \texttt{int} \quad \ldots

unreachable code elimination
Conclusion

All you need for speculation: versions + checkpoints.

Future work: bidirectional transformations.

Thanks!
Questions?
**Bonus: inlining**

```plaintext
main()

array pl = [1, 2, 3, 4]
array vec = [length(pl), pl]
var size = nil
var obj = vec
assume obj ≠ nil else ... 
var len = obj[0]
size ← len * 32
drop len
drop obj
goto ret
ret

print size
stop

size(obj)
opt

assume obj ≠ nil else ... 
var len = obj[0]
return len * 32

base ...
**Bonus: inlining**

```plaintext
main()

array pl = [1, 2, 3, 4]
array vec = [length(pl), pl]
var size = nil
var obj = vec
assume obj ≠ nil else ...
var len = obj[0]
size ← len * 32
drop len
drop obj
goto ret
ret

base ...

main(base)

array pl = [1, 2, 3, 4]
array vec = [length(pl), pl]
call size = size(vec)
ret
print size
stop

size(obj)

opt

assume obj ≠ nil else ...
var len = obj[0]
return len * 32
base ...

assume obj ≠ nil else ξ main.base.ret size [vec = vec]
```