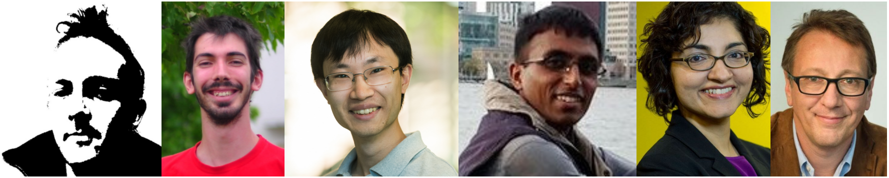
Correctness of Speculative Optimizations with Dynamic Deoptimization

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

Northeastern University, Boston, USA

October 12, 2017



Our work

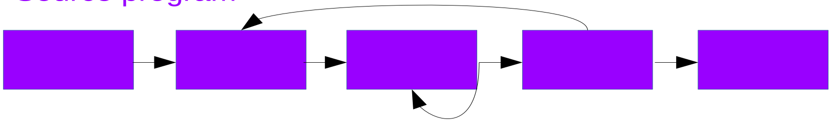
Just-in-time (JIT) compilation is essential to efficient dynamic language implementations.
(Javascript, Lua, R... Java)

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.

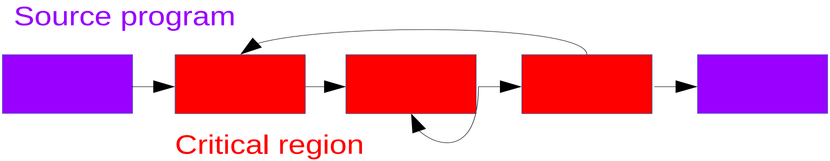
Just-in-time compilation

Source program



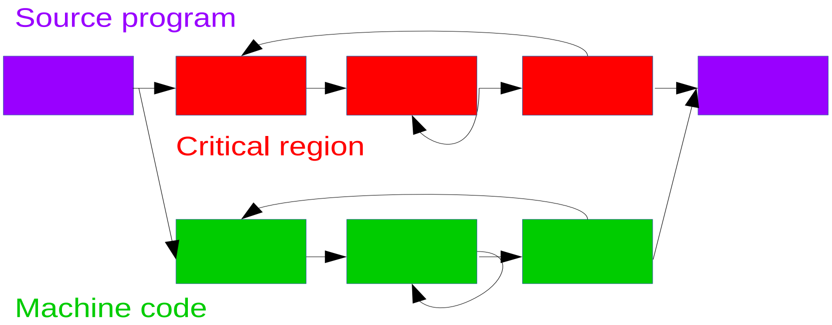
JITs:

Just-in-time compilation



JITs: Profiling

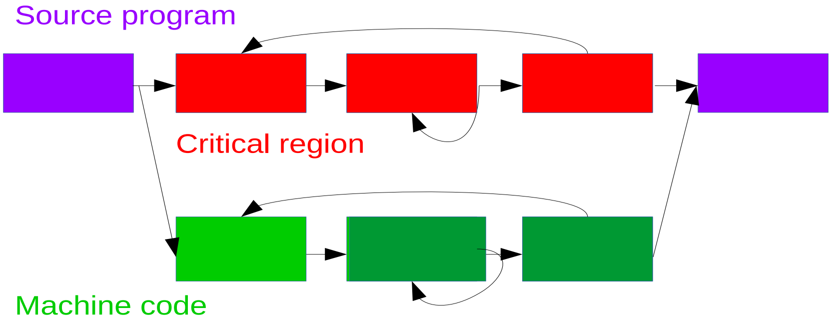
Just-in-time compilation



JITs: Profiling

+ High/Low languages
+ Dynamic code generation/mutation

Just-in-time compilation

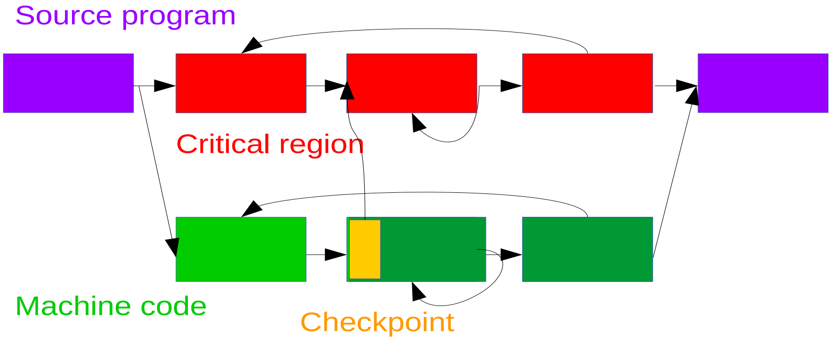


JITs: Profiling

+ 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.

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

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

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

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

fun(c)
  tough
    var o = 1
  L_1 **print** c + o
  luck
    assume c = 41 **else** fun.tough.L_1 [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

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

$$
\begin{aligned}
e \;::=\; & \\
\mid\; & se \\
\mid\; & x[se] \\
\mid\; & \textbf{length}(se) \\
\mid\; & primop\,(se^*)
\end{aligned}
$$

$$
\begin{aligned}
se \;::=\; & \\
\mid\; & lit \\
\mid\; & F \\
\mid\; & x
\end{aligned}
$$

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

Versions

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

fun(c)
  tough
    var $o = 1$
    L_1 **print** $c + o$
  luck
    assume $c = 41$ **else** fun.tough.L_1 $[c = c, o = 1]$
    print 42

Checkpoints

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

$$\textbf{assume}\ \mathrm{c} = 41\ \textbf{else}\ \mathrm{fun.tough.L_1}\ [\mathrm{c} = \mathrm{c}, \mathrm{o} = 1]$$

Guards: just a list of expressions returning booleans.

Bailout data:

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

Checkpoints

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

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

Guards: just a list of expressions returning booleans.

Bailout data:

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

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

Checkpoints

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

$$\textbf{assume } \mathrm{c} = 41 \textbf{ else } \mathrm{fun.tough.L_1}\ [\mathrm{c} = \mathrm{c}, \mathrm{o} = 1]$$

Guards: just a list of expressions returning booleans.

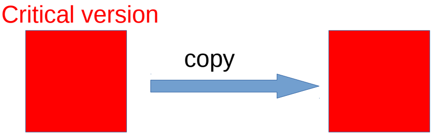
Bailout data:

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

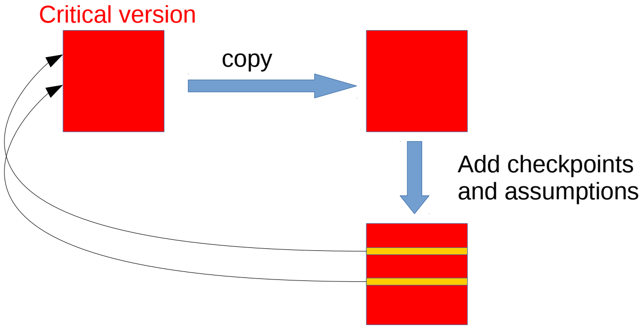
Not a branch. (inlining)
Checkpoints simplify optimizations...and correctness proofs!

Speculative optimization pipeline

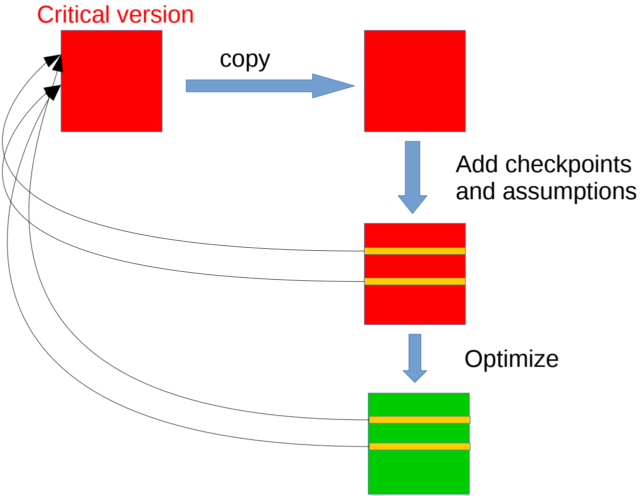
Speculative optimization pipeline



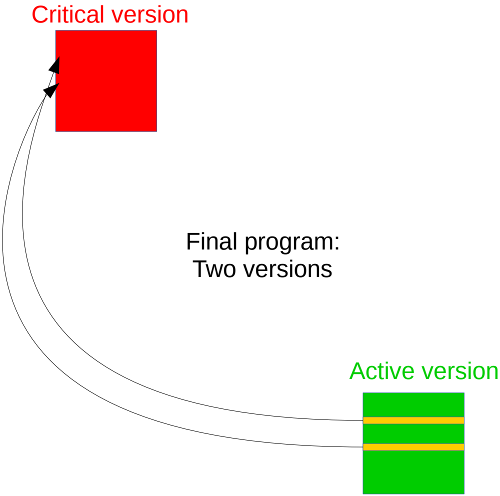
Speculative optimization pipeline



Speculative optimization pipeline



Speculative optimization pipeline



Speculative optimization pipeline

Execution: Operational semantics

Configurations:

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

Actions:

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

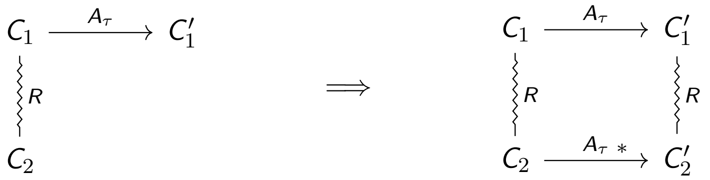
Reduction:

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

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)

Branch pruning – from the kit

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

Branch pruning – from the kit

base

L_1	**branch** tag = INT int nonint
int	...
nonint	...

opt

L_1	**assume** tag = INT **else** F.base.L_1 [...]
	branch tag = INT int nonint
int	...
nonint	...

Checkpoint + guard inserted

Bailout invariant!

Branch pruning – from the kit

base

L_1	**branch** tag = INT int nonint
int	...
nonint	...

opt

L_1	**assume** tag = INT **else** F.base.L_1 [...]
	branch true int nonint
int	...
nonint	...

constant folding

Branch pruning – from the kit

base

L_1	**branch** tag = INT int nonint
int	...
nonint	...

opt

L_1	**assume** tag = INT **else** F.base.L_1 [...]
int	...

unreachable code elimination

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 **Principles of Programming Languages (POPL)**, 2010. doi: 10.1145/1706299.1706313.

Bonus: inlining

```
main( )
  inlined
          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
  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 ...
```

Bonus: inlining

```
main( )
  inlined
          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
  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 $\neq$ **nil else** ξ main.base.ret size [vec = vec]