Towards Formally Verified Just-in-Time Compilation

Aurèle Barrière, Sandrine Blazy, David Pichardie

IRISA, Celtique
CoqPL, January 25th, 2020
Formally verified static compilation

Verified static compilers
CompCert, CakeML, VeLLVM…
Compilation happens \textit{statically}.
No self-modification of code during execution.
Formally verified static compilation

Verified static compilers

CompCert, CakeML, VeLLVM...
Compilation happens *statically.*
No self-modification of code during execution.
Formally verified static compilation

Verified static compilers

CompCert, CakeML, VeLLVM...
Compilation happens *statically*.
No self-modification of code during execution.
What about Just-in-Time compilation?

Source Program

Interpreter

Veriﬁcation Challenge

How can we relate this execution (with interpretation, execution of compiled code, on-stack replacement) to the semantics of the original source program?
Veriﬁcation Challenge

How can we relate this execution (with interpretation, execution of compiled code, on-stack replacement) to the semantics of the original source program?

What about Just-in-Time compilation?
What about Just-in-Time compilation?

Veri/fication Challenge
How can we relate this execution (with interpretation, execution of compiled code, on-stack replacement) to the semantics of the original source program?
What about Just-in-Time compilation?

Verification Challenge
How can we relate this execution (with interpretation, execution of compiled code, on-stack replacement) to the semantics of the original source program?
JUST-IN-TIME COMPILATION

Definition
Compile parts of the program (source code or bytecode) during its execution. Interleaves interpreting the unoptimized code, compiling it, and executing the optimized code.

Exploiting Dynamic information
As the optimization is done during the execution, one can use dynamic information to speculate on the future behavior of the program.
Speculative Optimizations

Exploiting dynamic information recorded by a profiler allows you to create specialized versions of the program.

Example

Dynamically-typed language: each + and * polymorphic operator must check the types of its arguments each time.

Function f () {
    int i;
    for (i=0; i<N; i++) {
        g(a,b,array,i);
    }
}

Function g (a,b,array,i) {
    sum[i] = a + array[i];
    product[i] = a * (array[i] + b);
}
**Speculative Optimizations - Example**

Function \( f() \) {
    int i;
    for (i=0; i<N; i++) {
        g(a,b,array,i);
    }
}

Function \( g(a, array, i) \) {
    sum[i] = a + array[i];
    product[i] = a * (array[i] + b);
}

Speculate on the type of the arguments

We can generate dynamically the following code for \( g \):

- **Speculation** : \( a \) is \( \text{int} \) /\ array[i] is \( \text{int} \) /\ \( b = 0 \)
- \( ai = array[i]; \)
- \( sum[i] = \text{int_add}(a, ai); \)
- \( product[i] = \text{int_mult}(a, ai); \)
- \( i = i+1; \)

Deoptimization

We must provide a way to return to the original version if the speculation does not hold.
Interleaves execution of optimized and non-optimized functions.
- Keep several versions of each function.
- Instructions to deoptimize and restore environment.
Interleaves execution of optimized and non-optimized functions.
- Keep several versions of each function.
- Instructions to deoptimize and restore environment.
Interleaves execution of optimized and non-optimized functions.
- Keep several versions of each function.
- Instructions to deoptimize and restore environment.
Interleaves execution of optimized and non-optimized functions.
Keep several versions of each function.
Instructions to deoptimize and restore environment.
Interleaves execution of optimized and non-optimized functions.

Keep several versions of each function.

Instructions to deoptimize and restore environment.
Interleaves execution of optimized and non-optimized functions.

Keep several versions of each function.

Instructions to deoptimize and restore environment.
## Related Works on JIT formalization

### Verified Just-In-Time Compiler on x86


### Jitk: A Trustworthy In-Kernel Interpreter Infrastructure


### Correctness of Speculative Optimizations with Dynamic Deoptimization

Prototype of a formally verified JIT middle-end with speculative optimizations

2 compilation phases: a middle-end and a backend.
Our prototype

We focus on the manipulation of a JIT IR with speculation, including middle-end compiling, interpretation, profiling...
Towards formally verified JIT compilation

A formally verified JIT middle-end prototype

- Realistic architecture.
  Optimizations, interpretation and speculation.
- Modular correctness proofs.
- Can be extracted and executed.
- JIT correctness theorem.

<table>
<thead>
<tr>
<th>Component</th>
<th>Implementation</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser</td>
<td>OCaml</td>
<td></td>
</tr>
<tr>
<td>JIT step</td>
<td>Coq</td>
<td>✔</td>
</tr>
<tr>
<td>Interpreter</td>
<td>Coq</td>
<td>✔</td>
</tr>
<tr>
<td>Constant Propagation</td>
<td>Coq</td>
<td>✔</td>
</tr>
<tr>
<td>Adding speculation</td>
<td>Coq</td>
<td>✔</td>
</tr>
<tr>
<td>Inlining</td>
<td>Coq</td>
<td>✔</td>
</tr>
<tr>
<td>Profiler</td>
<td>Ocaml</td>
<td>Not needed</td>
</tr>
<tr>
<td>Component Proof</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Static Compiler correctness

If compilation succeeds, and the original program has a behavior (safe), then any behavior of the compiled program matches a behavior of the source program.

**Theorem transf_c_program_correct:**

\[
\forall p \, tp, \quad \text{transf_c_program} \, p = \text{OK} \, tp \rightarrow \\
\text{backward_simulation} \,(\text{Csem.semantics} \, p) \,(\text{Asm.semantics} \, tp).
\]

### JIT correctness

We need an interpreter correctness theorem. If the original program is safe, then the JIT makes some progress and any of its possible executions matches a behavior of the source program semantics.
Original Program

Compiled Program

\( p \sim s_1 \sim s_2 \rightarrow t_p \)

Behavior refinement

Every compiled behavior is matched by a source behavior.

Same Program

In a static compiler, only the semantic state changes, not the program.
Behavior refinement
Every compiled behavior is matched by a source behavior.

In a static compiler, only the semantic state changes, not the program.
Behavior refinement

Every compiled behavior is matched by a source behavior.

In a static compiler, only the semantic state changes, not the program.
Behavior refinement

Every compiled behavior is matched by a source behavior.
Behavior refinement

Every compiled behavior is matched by a source behavior.

Same Program

In a static compiler, only the semantic state changes, not the program.
Building JIT Backward Simulations

Original Program

\[ p \]

\[ s \]

JIT state \( js \)

\[ \text{JIT prog} \]

\[ s' \]
BUILDING JIT BACKWARD SIMULATIONS

Original Program

\[ p \]

\[ s \sim \text{match\_states} \]

JIT state \( js \)

\[ \text{JIT prog} \]

\[ s' \]
Theorem jit_correctness:
\[\forall (p: \text{program})(s: \text{state})(js: \text{jit\_state})(ji: \text{jit\_index}),\]
\[\text{input\_prog} p \rightarrow\]
\[\text{match\_states} p s js ji \rightarrow\]
\[\text{safe} p s \rightarrow\]
\[\exists js', \exists e,\]
\[\text{jit\_jit\_step} js = \text{OK}(js',e) \land\]
\[((\exists s', \exists ji', \exists s (\text{traceof} e)s' \land \text{match\_states} p s' js' ji')) \lor\]
\[(\exists ji', \text{match\_states} p s js' ji' \land \text{jit\_order} ji' ji' \land \text{silent} e)).\]
**Theorem** \( \text{jit\_correctness} \):

\[
\forall (p:\text{program})(s:\text{state})(js:\text{jit\_state})(ji:\text{jit\_index}),
\quad \text{input\_prog} \ p \rightarrow
\quad \text{match\_states} \ p \ s \ js \ ji \rightarrow
\quad \text{safe} \ p \ s \rightarrow
\quad \exists \ js', \exists \ e,
\quad \text{jit\_jit\_step} \ js = \text{OK}(js',e) \land
\quad ((\exists \ s', \exists \ ji', \text{plus} \ p \ s (\text{traceof} \ e) s' \land \text{match\_states} \ p \ s' \ js' \ ji') \lor
\quad (\exists \ ji', \text{match\_states} \ p \ s \ js' \ ji' \land \text{jit\_order} \ ji' \ ji \land \text{silent} \ e)).
\]
**Theorem jit_correctness:**
\[
\forall (p: \text{program})(s: \text{state})(js: \text{jit\_state})(ji: \text{jit\_index}),
\text{input\_prog } p \rightarrow \\
\text{match\_states } p \ s \ js \ ji \rightarrow \\
\text{safe } p \ s \rightarrow \\
\exists js', \exists e, \\
\bullet \text{jit\_jit\_step } js = \text{OK} (js', e) \land \\
((\exists s', \exists ji', \text{plus } p \ s (\text{traceof } e) s' \land \text{match\_states } p \ s' js' ji') \lor \\
(\exists ji', \text{match\_states } p \ s js' ji' \land \text{jit\_order } ji' ji \land \text{silent } e)).
\]
**Theorem jit_correctness:**
\[
\forall (p:\text{program})(s:\text{state})(js:\text{jit\_state})(ji:\text{jit\_index}),
\text{input\_prog} p \rightarrow 
\text{match\_states} p s js ji \rightarrow 
\text{safe} p s \rightarrow 
\exists js', \exists e, 
\text{jit\_jit\_step} js = \text{OK}(js',e) \land \\
((\exists s', \exists ji', \text{plus} p s (\text{traceof} e) s' \land \text{match\_states} p s' js' ji') \lor \\
(\exists ji', \text{match\_states} p s js' ji' \land \text{jit\_order} ji' ji' \land \text{silent} e)).
\]
Theorem jit_correctness:
\[ \forall (p:\text{program})(s:\text{state})(js:\text{jit\_state})(ji:\text{jit\_index}), \]
\[ \text{input\_prog\ p \rightarrow} \]
\[ \text{match\_states\ p\ s\ js\ ji \rightarrow} \]
\[ \text{safe\ p\ s \rightarrow} \]
\[ \exists\ js',\ \exists\ e, \]
\[ \text{jit\_jit\_step\ js = OK(js',e) } \land \]
\[ ((\exists\ s',\ \exists ji',\ \text{plus\ p\ s}\ (\text{traceof\ e})s' \land \text{match\_states\ p\ s'}ji') \lor \]
\[ (\exists ji',\text{match\_states\ p\ s'}ji'ji' \land \text{jit\_order\ ji'ji' } \land \text{silent\ e})). \]
Theorem jit_correctness:
∀ (p:program)(s:state)(js:jit_state)(ji:jit_index),
input_prog p →
match_states p s js ji →
safe p s →
∃ js', ∃ e,
  jit.jit_step js = OK(js',e) ∧
  (( ∃ s', ∃ ji', plus p s (traceof e) s' ∧ match_states p s' js' ji') ∨
  • ( ∃ ji', match_states p s js' ji' ∧ jit_order ji' ji ∧ silent e)).
Summary

- Untyped, simple integer values, simple memory.
- Similar to CompCert RTL.
- An Assume instruction, the same as in Sourir ([Flückiger et al. 2018]).
- Function versions.

The only language of our JIT

- No backend compilation yet. Optimized code is also interpreted.
- The initial program should not have any speculation, and only one version per function.
### Syntax

\[
\text{Assume} \ (\text{expr list}) \ \text{target} \ (\text{varmap}) \ [\text{synth frame list}]
\]

- **expr list**: the speculation
- **target**: deoptimization target
- **varmap**: restore the register environment
- **synth frame list**: restore extra stack frames

### Example

\[
\text{Assume} \ (x = 0, \ y = 3) \ F.V1.lbl5 \{(a,10)\} \ []
\]

- First, test if \((x = 0)\) and \((y = 3)\) hold.
- If not, deoptimize to function \(F\), version \(V1\), line \(<\text{lbl5}>\).
- Put value \(10\) in register \(a\).
Speculating on the values of function arguments. The profiler records the values at each function call.

**Example**

```plaintext
Function F (r1, r2) :
    Version V1:
    <lbl1> Return (r1 + r2)
```
Speculating on the values of function arguments. The profiler records the values at each function call.

Example

Function F (r1, r2) :
Version V1:
<lbl1> Return (r1 + r2)

The new Version

Version V2:
<lbl0> Assume (r2 = 10) F.V1.lbl1 {(r1,r1) (r2,r2)} []
<lbl1> Return (r1 + r2)

F.V1.lbl1: deoptimize to Function F, Version V1, line <lbl1>.
Optimizes the function based on the previously inserted speculation.

Example

Function F (r1, r2, r3) :
Version 1:
r1 = 4
Assume (r2 = 0) G.V2.lbl3 {(r1,r1) (r2,r2)} []
Return r1 + r2 + r3

The optimized version

Version 2:
r1 = 4
Assume (r2 = 0) G.V2.lbl3 {(r1,4) (r2,r2)} []
Return 4 + r3

Verification

Uses a fixpoint solver library from CompCert.
Replaces a function call by its code.
Name-mangling and synthesizing new stackframes in Assume.

Changing Assumptions in the inlined code

Assume \((r1 = 4) H.V2.lbl7 (r1,r1)\) in the inlined code becomes
Assume \((R1 = 4) H.V2.lbl7 (r1,R1) [f.v.l ret]\)

Where

- \(R1\) is the mangled name of \(r1\).
- \(f.v.l\) is the location of the instruction after the call in the original caller function.
- \(ret\) is the variable of the caller function that receives the callee’s return.
## Proving optimizations correct in our JIT

### Reusing CompCert Forward Simulation Methodology

Show that each step of the program before the optimization matches some steps in the program after optimization. 
Forward to backward theorem: a forward simulation implies a backward simulation.

### Proving the JIT correct

We showed that, if each optimization pass is proved, the entire JIT is correct. Every behavior of the JIT matches a behavior of the original program.

**Theorem** `optimization_correctness`:
\[\forall p \; ps \; newp,\]
\[\text{optimize} \; ps \; p = \text{OK} (newp) \rightarrow\]
\[\text{spec_wf} \; p \rightarrow\]
\[\exists \; \text{order}, \exists (r: \text{relation}),\]
\[\text{bwd_sim} \; p \; newp \; \text{order} \; r \land \text{reflexive_wf} \; p \; r.\]
CONCLUSION

A Coq JIT

- A Coq model of a realistic JIT architecture.
- An executable prototype.
- A backward simulation for JIT correctness.

Verification work

Adding an optimization pass in the JIT middle-end can be proved with the same forward simulation methodology as CompCert.
A Coq JIT

- A Coq model of a realistic JIT architecture.
- An executable prototype.
- A backward simulation for JIT correctness.

Verification work

Adding an optimization pass in the JIT middle-end can be proved with the same forward simulation methodology as CompCert.
Sourir Transparency Invariant

From [Flückiger et al. 2018].
Prove that deoptimizing, even when the conditions hold, does not change the behavior of the program.
Useful in some speculation-specific optimizations.

Backend compilation

Using the translation of CompCert? Its specification doesn’t suit our needs.
INLINING AND SYNTHESIZING STACK FRAMES