VERIFIED NATIVE CODE GENERATION IN A JIT COMPILER

JOURNÉE HYBRIDE LVP

AURÈLE BARRIÈRE  SANDRINE BLAZY  DAVID PICHARDIE

NOVEMBER 23RD, 2021
FORMALLY VERIFIED STATIC COMPILATION

Verified static compilers:

- CompCert [Leroy]
- CakeML [Kumar et al.]
- VeLLVM [Zhao et al.]

Compilation happens statically: the code is produced before its execution.

JIT compilation: Interleave execution and optimization of the program.
FORMALLY VERIFIED STATIC COMPILATION

Compilation happens **statically**: the code is produced before its execution.

**Verified static compilers**

- CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012].
Formally verified static compilation

CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012]. Compilation happens **statically**: the code is produced before its execution.
FORMALLY VERIFIED STATIC COMPILATION

Verified static compilers

CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012]. Compilation happens **statically**: the code is produced before its execution.
FORMALLY VERIFIED STATIC COMPILATION

Verified static compilers

CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012]. Compilation happens statically: the code is produced before its execution.

JIT compilation

Interleave execution and optimization of the program.
**Executing a program with a JIT with speculative optimizations**

**Execution Stack**

Interpreter: f

**Program**

Function f():
while(...):
g()

Function g():
g1
g2
**Executing a program with a JIT with speculative optimizations**

**Execution Stack**

- **Interpreter:** f
- **Interpreter:** g

**Program**

**Function** f():
while(...):
g()

**Function** g():
g1
g2
Executing a program with a JIT with speculative optimizations

Program

Function f():
while (...):
g()

Function g():
g1
g2

Function g_x86():
g1
Speculation (x=7)
g2'

Execution Stack

Interpreter: f

Optimizing Compiler
**Executing a program with a JIT with speculative optimizations**

**Execution Stack**

Interpreter: \( f \)

\( g_{x86} \)

**Program**

**Function** \( f() \):

\[
\text{while}(...) :
\]

\( g() \)

**Function** \( g() \):

\( g1 \)

\( g2 \)

**Function** \( g_{x86}() \):

\( g1 \)

Speculation \((x=7)\)

\( g2' \)
Execution Stack

Interpreter: f

g_x86

Speculation fails

Program

Function f():
    while(...):
        g()

Function g():
    g1
g2

Function g_x86():
    g1
    Speculation (x=7)
g2'
Execution Stack

Interpreter: f

Speculation fails

Interpreter: g

On-stack replacement

Program

Function f():
while(...):
g()

Function g():
g1
g2

Function g_x86():
g1
Speculation (x=7)
g2'
Deoptimization requires the JIT to
- Synthesize interpreter stackframes in the middle of a function.
- Possibly synthesize many stackframes at once.

With speculation, JITs need precise execution stack manipulation.
Our Goals

- A verified and executable JIT in Coq.
- With native code generation and execution.
- With speculation and on-stack replacement.
- Using CompCert as a backend compiler.
- Reusing CompCert’s proof and its proof methodology.
TOWARDS A FORMALLY VERIFIED JIT MIDDLE-END

JIT-specific verification problems

- Speculative optimizations.
- Dynamic Optimizations interleaved with execution.
- Impure and non-terminating components.
- Integrate the correctness proof of a static compiler backend.
JIT-specific verification problems

- Speculative optimizations.
- Dynamic Optimizations interleaved with execution.
- Impure and non-terminating components.
- Integrate the correctness proof of a static compiler backend.

Previous Work: Formally verified speculation and deoptimization in a JIT compiler, POPL21

Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, Jan Vitek.
https://github.com/Aurele-Barriere/CoreJIT

- CoreIR, inspired by RTL and speculative instructions ([Flückiger et al. 2018]).
- Correctness theorem of CoreJIT with interpretation, dynamic optimizations, and speculations.
JIT-specific verification problems

- Speculative optimizations.
- Dynamic Optimizations interleaved with execution.
- Impure and non-terminating components.
- Integrate the correctness proof of a static compiler backend.

Previous Work: Formally verified speculation and deoptimization in a JIT compiler, POPL21

Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, Jan Vitek.
https://github.com/Aurele-Barriere/CoreJIT

- CoreIR, inspired by RTL and speculative instructions ([Flückiger et al. 2018]).
- Correctness theorem of CoreJIT with interpretation, dynamic optimizations, and speculations.

A theorem about IR to IR transformation. No native code generation in the formal model.
A JIT ARCHITECTURE

JIT architecture
Extends the architecture from [Barrière et al. 2021] with native code generation and execution.

JIT loop
The monitor chooses the next step: execution or optimization.
Profiling: records information about the execution and suggest speculations.
A JIT ARCHITECTURE

Interpreter

Interpret the IR code that has not been compiled to native.
A JIT Architecture

POPL 2021

JIT monitor
Monitoring and Profiling

IR Execution
Interpreter

Optimization
Middle-end

Native Execution
Load Code
Run Native

Backend Compiler

Code Installation

Middle-end Optimizer
From the IR to the IR. Inserts speculation.

POPL21
The correctness theorem of our previous work is about these components. A Coq proof that any behavior of this JIT prototype is a behavior of the input program.
A JIT ARCHITECTURE

**Backend Compilation**
Generates native code, as in a static compiler backend.
Use the CompCert backend from RTL to x86.

**Code Installation**
Install the dynamically generated code in memory.
Make it executable.
Setting up native execution
Get a function pointer for the installed code.

Native Code Execution
Run the generated code.
Can we really write a JIT in Coq?

Some JIT components are **impure**. Global shared data-structures: execution stack and executable memory. The call to native code may even be **non-terminating**.
Can we really write a JIT in Coq?

Some JIT components are impure. Global shared data-structures: execution stack and executable memory. The call to native code may even be non-terminating.

The Free Monad

Interaction Trees [Xia et al. 2020] and FreeSpec [Letan and Régis-Gianas 2020] use a variation of the free monad to reason about impure programs in Coq.
Representing programs where some impure primitives have yet to be implemented.

\[
\text{Inductive } \text{free}(T : Type) : Type := \\
\quad \mid \text{pure}(x : T) : \text{free} T \\
\quad \mid \text{impure}\{R\} \\
\quad \quad (\text{prim}: \text{primitive } R)(\text{next}: R \rightarrow \text{free} T): \text{free} T.
\]

With different primitive implementations, the program can be executed differently.
Our strategy for a verified executable impure JIT

The Free JIT


Inspired by the Free Monad, but adapted to fit the simulation framework of CompCert.
The Free JIT


Inspired by the Free Monad, but adapted to fit the simulation framework of CompCert.
Every JIT component can be written as a Free Monad:

**Definition** optimizer (f:function): free unit :=

\[
\begin{align*}
& \text{do } f_{\text{rtl}} \leftarrow \text{ret}(\text{IRtoRTL } f); \\
& \text{do } f_{\text{x86}} \leftarrow \text{ret}(\text{backend } f_{\text{rtl}}); (* \text{ using CompCert backend } *) \\
& \text{Prim}_{\text{Install_CODE}} f_{\text{x86}}.
\end{align*}
\]
Every JIT component can be written as a Free Monad:

**Definition** optimizer (f: function): free unit :=
  do f_rtl ← ret (IRtoRTL f);
  do f_x86 ← ret (backend f_rtl); (* using CompCert backend *)
  Prim_Install_Code f_x86.

---

**New Calling Conventions**

We need to reason on and manipulate the execution stack (deoptimization). Our JIT works on a custom execution stack, that only the JIT modifies. We need to implement new calling conventions on this custom stack. The generated native code needs to call our primitives.
Generating Several RTL Programs

Generating RTL code that uses custom calling conventions with our primitives.

- Primitives are *external calls*.
- Each RTL function returns to the monitor.
- One Continuation per Call instruction.
Generating Native Code using Primitives

Generating Several RTL Programs

Generating RTL code that uses custom calling conventions with our primitives.

- Primitives are *external calls*.
- Each RTL function returns to the monitor.
- One Continuation per Call instruction.
## JIT Primitives

### Stack Primitives
- Pop and Push
- Push and pop entire interpreter stackframes

### Code Segments Primitives
- Install a native function in the executable memory.
- Load a function (or one of its continuations).
- Check if a function has been compiled.

---

### Running Native Code

We define a special primitive to run native code. Its specification is a monad describing the small-step semantics of x86 code.
### A Free JIT

- We can derive both small-step semantics and an executable OCaml JIT *(ongoing).*
- Native code generation and execution are part of the formal model.
- A correctness proof of the JIT small-step semantics.
- We reuse the simulation methodology of CompCert.
- We would like to reuse the simulation proof of CompCert’s backend *(ongoing).*

### Trusted Code Base

- Coq extraction to OCaml.
- The primitive impure implementations correspond to their monadic specifications.
- The call to the generated native code has been specified with a free monad.