CoreJIT

Aurèle Barrière Sandrine Blazy Olivier Flückiger
David Pichardie Jan Vitek

This is the development of CoreJIT, a verified JIT compiler. This document contains an overview of the code and build instructions.

Running CoreJIT

This artifact comes with a prebuilt version of CoreJIT packaged as an OCI compliant container. To follow these instructions you need to install a container runtime, such as `docker` or `podman`. To follow the local build instructions in the end, you need `opam` installed.

To run this artifact it suffices to run:

```
CR=docker    # or podman
VS=90bd61a3e99a3fae1c78d62c60bd69b8d742e485
RG=quay.io/corejit/jit
$CR run $RG:$VS
```

The final submission of this artifact will include a copy of the container which can be imported using `$CR load < image.tgz`.

This container by default executes the `aec.sh` script, which compiles the proofs, runs all tests and the performance experiments.

There are a number of example programs in CoreJITs IR format in `progs_specIR`. To run one of these programs do the following:

```
$CR run $RG:$VS bash -c "~/coqjit/jit ~/coqjit/progs_specIR/constprop.specir"
```

To get the list of options use

```
$CR run $RG:$VS bash -c "~/coqjit/jit -h"
```

There are a number of lua programs in `progs_lua`. To run one of these programs do the following:

```
$CR run $RG:$VS bash -c "~/coqjit/jit -f ~/coqjit/progs_lua/scopes.lua"
```

To run these steps with the native backend enabled additionally pass the `-n` flag.

Reproducing performance numbers

```
$CR run $RG:$VS bash -c "~/coqjit/experiments.sh 10"
```

Building CoreJIT

The container of this artifact includes build dependencies and source.

To edit and build inside the container use:

```
$CR run -it $RG:$VS bash
# Then in the container:
cd ~/coqjit
make clean
vim ....    # edit some files
make
```

To build the sources from this repository from your host filesystem inside the container:

```
$CR run -v $PWD/src:/home/opam/src -it $RG:$VS bash
# Then in the container
cd ~/src/coqjit
make
```

Building the Container

This artifact also includes a `Dockerfile` to build the container using:

```
docker build . --file Dockerfile
```

The interesting steps are to be found in the `docker-install.sh` script.

Alternatively and optionally CoreJIT can be built without docker on any system with `opam` as follows:

```
cd src/coqjit
wget https://releases.llvm.org/9.0.0/clang+llvm-9.0.0-x86_64-pc-linux-gnu.tar.xz
tar xf clang+llvm-9.0.0-x86_64-pc-linux-gnu.tar.xz
PATH="$PATH:$PWD/clang+llvm-9.0.0-x86_64-pc-linux-gnu/bin"
make install-deps
eval $(opam env)
make
```

CoreJIT code overview

This section details the different components of CoreJIT. Some definitions have been renamed in the submission. Here are their equivalent:

- CoreIR <-> `SpecIR`
- core_sem <-> `specir_sem`

- Anchor <-> `Framestate`

Coq Development

The `src/coqjit` directory contains the Coq development for CoreJIT, operating on CoreIR, our intermediate representation for speculative optimizations and deoptimizations. CoreIR syntax and semantics are defined in `specIR.v`.

The JIT step that is looped during JIT execution is defined in `jit.v`. This either calls a CoreIR interpreter `interpreter.v` or a dynamic optimizer `optimizer.v` The different passes of the dynamic optimizer are in separate files (`const_prop.v, inlining.v`...).

Our developpment uses a few Coq libraries from CompCert. These are located in `src/coqjit/lib`.

Coq Proofs

Our final Semantic Preservation Theorem is proved in `jit_proof.v`. Each file ending in `_proof.v` contains the correctness proof of a CoreJIT component. The Internal Simulation Framework for Dynamic Optimizations is located in `internal_simulations.v`.

Extraction

The Coq development is extracted to OCaml as specified by the `extract.v` file. This creates an `extraction` directory where the extracted code is located. The extraction of `jit` is patched with `jit.ml.patch` to integrate the native backend, which has no representation in coq (see below).

OCaml Frontend

CoreJIT can be run using the extracted OCaml code. The additional OCaml code is out of scope of our verification work. The `parsing` directory contains a parser of CoreIR (see examples in `progs_specIR` directory). The `frontend` directory contains a frontend from miniLua (see examples in `progs_lua`) to CoreIR.

The extracted `jit_step` from `jit.v` is looped in `main.ml`. A simple profiler implementation is defined in `profiler.ml`.

Native Backend

The `backend` directory contains an optional native backend written in OCaml where CoreIR is translated to LLVM IR and then to native code.

To call from the interpreter into native code, the extracted jit is modified using the patch in `jit.ml.patch`. This patch adds an alternate execution step, which

instead of using the interpreter to evaluate instructions of a function, hands control to native code. This part is out of scope of our verification work.

The generated native code relies on some builtin functions in `native_lib/native_lib.c` used for I/O and interfacing with the OCaml runtime.

Coq Axioms and Parameters

The profiler, optimization heuristics and memory model are external parameters. This ensures that the correctness theorems do not depend on their implementation. These parameters have to be realized for the JIT to be extracted. The Load and Store implementations may fail (return None), for instance for an out-of-bound access. This corresponds to blocking behaviors in the semantics, and these behaviors are preserved.

Here is a list of Parameters realized during the Coq extraction:

- `profiler_state`: a type that the profiler can update
- `initial_profiler_state`: how to initialiaze the profiler state
- `profiler`: updating the profiler state
- `optim_policy`: suggesting to optimize or execute
- `optim_list`: the list of optimization wishes the profiler wants to perform
- `framestates_to_insert`: the list of locations the profiler wants to insert framestates at
- `mem_state`: an abstract type for the memory
- `initial_memory`: how to initialize the memory
- `Store_`: how to implement the Store instruction (may fail)
- `Load_`: how to implement the Load instruction (may fail)
- `spacing`: a number used as an heuristic for Assume insertion
- `max_optim`: maximum number of optimization steps the JIT can do
- `interpreter_fuel`: maximum number of steps the interpreter can perform before going back to the JIT
- `hint`: a type to annotate the Nop instructions
- `fuel_fresh`: a number used as an heuristic for Assume insertion

The CompCert libraries also use 2 Axioms: Functional Extensionality and Classical Logic.