Towards Formally Verified Just-in-Time compilation

Extended Abstract

Aurèle Barrière
Univ Rennes, Inria, CNRS, IRISA
aurele.barriere@irisa.fr

Sandrine Blazy
Univ Rennes, Inria, CNRS, IRISA
sandrine.blazy@irisa.fr

David Pichardie
Univ Rennes, Inria, CNRS, IRISA
david.pichardie@ens-rennes.fr

Abstract

Just-in-Time compilation consists in interleaving program interpretation and compilation at run-time, to achieve better performance than standard interpretation. While some of the execution time is spent compiling, a JIT compiler can leverage run-time information to make speculative optimizations. These optimizations create optimized versions of functions given some assumptions. While static compilers have been the topic of many formal verification works, few have tackled JIT compilation verification. We present our ongoing work about formal verification of a Just-in-Time compiler.

Keywords verified compilation, just-in-time, speculative optimizations

1 Introduction

Dynamic programming languages tend to postpone program optimization to run-time. These optimizations are typically done by a Just-in-Time (JIT) compiler in charge of creating, during a program execution, low-level optimized versions of some functions.

Using JIT compilation to execute a program thus amounts to using several components. One must have an interpreter to start the execution of the program. During interpretation, one must run a profiler to gather dynamic information about this execution. This profiler detects *hot* code, portions of code that would benefit from being optimized, for instance because they are run frequently. Then, a compiler can create optimized versions of hot code. When interpretation resumes, the next call to these instructions may call the optimized version. Using a JIT compiler, program execution interleaves interpretation, profiling, dynamic compilation and execution of optimized code.

As the optimizations are made dynamically, JIT compilation can use run-time information to create specialized versions. This technique is called *speculative optimization*, and is used in most JIT compilers, such as the JavaScript engine V8 [1], or the Java JIT Graal [2]. For instance, in a dynamically-typed language, one could create versions of polymorphic functions that assume the type or the value of their arguments and optimize them accordingly. To ensure correctness, the optimizer must add instructions that check the validity of assumptions each time the optimized version is called. If the assumption holds again, then one can keep executing the optimized code. Otherwise, the execution must go back to the original code, a process often called *deoptimization*.

Speculative optimization brings new challenges to a dynamic compiler. One must choose which information to speculate on, how to check the validity of assumptions, and how to deoptimize.

For static compilers, formal verification works aim at proving that a compiler does not introduce any bug in the code it produces. Works such as CompCert [5], CakeML [4] or VeLLVM [8] have shown that fully-verified, realistic static compilation is feasible. A verified compiler such as CompCert guarantees that properties proved at the source level hold for the compiled code as well.

While formally verified static compilation has been extensively studied, few works have tackled mechanized formal verification of JIT compilation. Myreen presented a fully verified JIT compiler [6], from a small stack-based bytecode to x86. While it tackles successfully the issues of verifying self-modifying code, no optimizations are made and the structure does not resemble these of realistic dynamic compilers. Wang et al. presented Jitk [7], a verified infrastructure for in-kernel interpreters built on top of CompCert. While it generates code at run-time, it does not implement speculative optimizations.

The objective of our ongoing work is to study a Coq implementation of a JIT compiler that preserves the behavior of its input program. We take interest in distinctive JIT features such as speculative optimizations and deoptimizations.

2 Formalizing Speculative Optimizations

Consider the following example, where `+` and `*` are polymorphic operators:

```
function f_original (a) {
  for (int i=0; i<length; i++) {
    sum[i]     = a + array[i];
    product[i] = a * array[i]; } }
```

If the language is dynamically typed, the interpreted code corresponding to a single iteration needs to check the type of `a` and `array`, then use the correct addition and product methods. Each iteration thus ends up with a more convoluted control flow:

```
if (a is int & array[i] is int) {
  sum[i] = int_add(a, array[i]); }
if (a is float & array[i] is float) {
  sum[i] = float_add(a, array[i]); }
if (a is string & array[i] is string) {
  sum[i] = string_add(a, array[i]); }
if (a is int & array[i] is int) {
  product[i] = int_mult(a, array[i]); }
if (a is float & array[i] is float) {
  product[i] = float_mult(a, array[i]); }
if (a is string & array[i] is string) {
  product[i] = string_mult(a, array[i]); }
i = i+1;
```

In this example, if previous calls to this code were always done on integers, one might want to create an optimized version specialized for integers, to speed up the next calls.

Speculative optimization must include several components. First, one needs instructions to check the validity of the assumptions. If the assumptions do not hold, one needs to know where to deoptimize to, to resume the execution in the unoptimized version. Finally, in some cases such as the function inlining optimization, one must be able to synthesize new stackframes to the stack.

Flückiger et al. formalize speculative optimizations in JIT compilers [3], but with only paper proofs. Their language, Sourir, is a

CoqPL'2020, January 25, 2020, New Orleans, LA, USA,
2020.

low-level language, close to RTL (Register Transfer Language), a standard intermediate representation used in many compilers. It presents two distinctive features useful for speculation. First, every function in the program can have both its original version and specialized ones. This allows creating specialized versions and still keep the original versions in case we need to deoptimize. Then, the language includes an `assume` instruction, which contains a list of expressions that describe the speculation, a deoptimization target, some bindings to reconstruct the original environment as well as additional continuations to add to the stack.

In our previous example, using such a formalism, one could create the following version:

```
function f_optimized (a) {
  assume (a is int && array[i] is int) f_original.0 []

  int ai = array[i];
  sum[i] = int_add(a, ai);
  product[i] = int_mult(a, ai);
  i = i+1; }
```

For all subsequent calls to this function, one can now call `f_optimized`. If the speculation holds (`a` and `array[i]` are indeed integers), the execution proceeds with this faster version. Otherwise, one must dynamically redirect the execution to the first instruction at line `0` of `f_original`. The empty brackets indicate that in this case, no modification to the environment or stack is needed to maintain correctness.

We draw inspiration from this formalism, and include in our intermediate representation a similar `assume` instruction.

3 Proving optimizations of a JIT compiler

In CompCert, every optimization pass is proved with a simulation [5]. To prove such simulations, one needs to prove that for each step of the source semantics, there exists a series of corresponding steps in the compiled program. This requires having defined the small-step semantics of all intermediate languages.

We intend to use this semantic simulation approach to prove correct the optimizations of a JIT compiler. However, we are faced with some differences. First, our final correctness theorem is expressed as an interpreter correctness theorem, since the JIT is active during the entire execution of the program. We must prove that any JIT execution, with dynamic optimizations, matches the execution of the original program. Besides, our semantic simulation has to account for the possibility to go back to unoptimized versions of functions, which isn't the case in static compilers.

Finally, speculative optimizations present new challenges to the verification of optimizations. As presented in Sourir [3], there exists some optimizations specific to the handling of our `assume` instruction, such as inserting them, merging them, moving them etc.

Moreover, traditional static optimizations become more complex when handling `assume` instructions. For instance when performing inlining, consider the case where we have the following functions:

```
function f_original (x,y) {
  return x + y; }

function f_optimized (x,y) {
  assume (x=0) f_original.0 []
  return y; }

function g_original (a) {
  b = f (a, 4);
  return (2 * b); }
```

Function `f` has two versions. The optimized one speculates on the value of the first argument. If the assumption does not hold, the execution returns to `f_original`.

Now if a JIT compiler wants to inline the call of `f` in `g`, it could simply create the following version, copying the code of `f_optimized`:

```
function g_optimized (a) {
  x = a;
  y = 4;
  assume (x=0) f_original.0 []
  b = y;
  return (2 * b); }
```

While this version is correct if the speculation holds, there will be an issue when deoptimizing. During the execution of `g_optimized`, if the `assume` fails, we deoptimize to the function `f_original` where we successfully compute the sum. But to match the execution of the original program, one should return from `f_original` to `g_original`. The solution, as done in Sourir [3], consists in synthesizing a new stackframe when deoptimizing. Intuitively, when deoptimizing, the execution should match the execution of the original program without inlining, and thus with an extra stackframe.

When copying the code of `f_optimized`, we replace the `assume` instruction with `assume (x=0) f_original.0 [g_original.1]`, so that after executing `f_original`, the execution comes back to `g_original`, just after the call to `f`, at line `1`.

Proving speculative optimization passes with a simulation requires proving that any execution of the original version is matched by an execution of the optimized one, whether the assumptions hold or not.

4 A formally verified JIT in Coq

Our ongoing work thus consists in implementing and proving a JIT compiler in Coq. A prototype is being developed, it can be extracted to Ocaml and executed. Its input and output language draws inspiration from CompCert RTL and Sourir for the `assume` instruction and the function versions. While an input program contains only one version per function and no such `assume`, the JIT will progressively add new optimized versions of functions with speculation, during the execution. One of our next goals is to generate lower-level code when optimizing, so the execution can either use an interpreter for original versions, or call machine instructions for optimized ones.

Our prototype has proofs of correctness for the interpreter and some optimizations. Constant propagation has been proved. We also proved an optimization that adds an `assume` instruction to a function by speculating on the values of its parameters. Some optimizations have yet to be proved, such as function inlining which synthesizes stackframes. Moreover, some parts are implemented in OCaml without invalidating our correctness theorem. For instance the profiler, gathering dynamic information for our speculations, can return incorrect results. It should affect the performance but not the correctness of our JIT execution, as deoptimization always presents a safe way to return to the original program.

We intend to remain as close as possible to CompCert to reuse libraries and proofs. Our main objective is to show how traditional static compilation verification techniques can be adapted to prove the correctness of a JIT compiler.

Acknowledgments This work is supported by a European Research Council (ERC) Consolidator Grant for the project "VESTA", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 772568).

References

[1] [n. d.]. V8 - Google's high-performance open source JavaScript and WebAssembly engine. https://v8.dev/.

[2] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*. ACM, New York, USA.

[3] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of speculative optimizations with dynamic deoptimization. *PACMPL* (2018).

[4] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proceedings of POPL*.

[5] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[6] Magnus O. Myreen. 2010. Verified just-in-time compiler on x86. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*.

[7] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *OSDI '14*.

[8] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*.