Formally Verified Speculation and Deoptimization in a JIT Compiler

AURÈLE BARRIÈRE, Univ Rennes, Inria, CNRS, IRISA, France
SANDRINE BLAZY, Univ Rennes, Inria, CNRS, IRISA, France
OLIVIER FLÜCKIGER, Northeastern University, USA
DAVID PICHARDIE, Univ Rennes, Inria, CNRS, IRISA, France
JAN VITEK, Northeastern University / Czech Technical University, USA

Just-in-time compilers for dynamic languages routinely generate code under assumptions that may be invalidated at run-time, this allows for specialization of program code to the common case in order to avoid unnecessary overheads due to uncommon cases. This form of software speculation requires support for deoptimization when some of the assumptions fail to hold. This paper presents a model just-in-time compiler with an intermediate representation that explicits the synchronization points used for deoptimization and the assumptions made by the compiler's speculation. We also present several common compiler optimizations that can leverage speculation to generate improved code. The optimizations are proved correct with the help of a proof assistant. While our work stops short of proving native code generation, we demonstrate how one could use the verified optimization to obtain significant speed ups in an end-to-end setting.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; • **Theory of computation → Program verification**.

Additional Key Words and Phrases: verified compilation, just-in-time compilation, CompCert compiler

ACM Reference Format:
Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. 2021. Formally Verified Speculation and Deoptimization in a JIT Compiler. *Proc. ACM Program. Lang.* 5, POPL, Article 46 (January 2021), 26 pages. https://doi.org/10.1145/3434327

1 INTRODUCTION

The essence of just-in-time (JIT) compilation is the interleaving of execution and code generation. It is this interleaving that allows JIT compilers to emit code specialized to the program's computational history. JIT compilers have been considered challenging targets for formal verification due to a combination of features that include self-modifying code to patch previously generated call instructions, mixed-mode execution to allow switching between interpretation and native execution, and on-stack replacement to transfer control from one version of a function to another.

To make matters worse, much of what sets apart JITs from traditional batch compilers is often relegated to implementation details that are neither clearly abstracted nor documented, and scattered around different levels in the compilers' architecture. For instance an optimization such

Authors' addresses: Aurèle Barrière, Univ Rennes, Inria, CNRS, IRISA, Rennes, France, aurele.barriere@irisa.fr; Sandrine Blazy, Univ Rennes, Inria, CNRS, IRISA, Rennes, France, sandrine.blazy@irisa.fr; Olivier Flückiger, Northeastern University, Boston, MA, USA, o@o1o.ch; David Pichardie, Univ Rennes, Inria, CNRS, IRISA, Rennes, France, david.pichardie@ens-rennes.fr; Jan Vitek, Northeastern University / Czech Technical University, Boston, MA, USA, j.vitek@neu.edu.

2475-1421/2021/1-ART46
https://doi.org/10.1145/3434327

as devirtualization might rely on the interplay of inline caches in the runtime, speculative optimizations in the optimizer and deoptimization triggered by the class loader. JIT compilers differ in their architecture and in the nature and extent of their reliance on the above mentioned features. The seminal papers on SELF [Hölzle et al. 1992; Hölzle and Ungar 1994] describe a system that limits self-modification to inline cache updates with a two-tiered compilation scheme. Compiled code that can be deoptimized with on-stack replacement when debugging is requested. The Java server compiler has both an interpreter and an optimizing compiler [Paleczny et al. 2001]. Jalapeño features a three-tier architecture and sample based profiling instead of instrumentation [Burke et al. 1999]. The early V8 JavaScript engine lowered both baseline and optimized functions to native code, and deoptimization toggled between the two; as of now the baseline is an interpreter [Chromium 2020]. The R compiler of Flückiger et al. [2019] performs speculation on a dedicated intermediate representation (IR) and offloads code generation to LLVM, with write-only code segments.

This paper contends that a JIT compiler can be designed so that verification becomes, if not easy, at least possible. To this end, we developed CoreJIT, a verified compiler that generates code at run-time and is able to speculate on arbitrary predicates as well leverage on-stack-replacement to deoptimize from optimized code back to an unoptimized base version. As in Graal [Duboscq et al. 2013], deoptimization is specified at the IR level by dedicated instructions which have a formally specified semantics. Without loss of generality, we restrict our attention to a uniform execution model in which both the unoptimized and the optimized code are expressed in the same intermediate language, similarly to the approach adopted by Béra et al. [2016] and Flückiger et al. [2019]. Furthermore, we carefully separate mechanism, *i.e.*, how code is generated, optimized and deoptimized, from policy, *i.e.*, when and what to optimize. This separation allows us to prove correctness of the implementation without having to trust the heuristics that define the policy.

Unless noted otherwise, all results presented in this paper have been mechanically verified using the Coq proof assistant. The complete Coq development is available as a supplementary material. Specifically, we claim the following novel contributions:

- **JIT design:** We introduce a JIT design, CoreJIT, that clearly separates heuristic-based optimization policy decisions and profiling needed to inform these decisions from program execution and the code generation mechanisms that are needed for speculation and optimization.
- **IR design:** We present a new high-level intermediate representation, CoreIR, that makes deoptimization and speculation explicit and separates the insertion of deoptimization points from the subsequent speculation checks. CoreIR is inspired by CompCert's RTL [Leroy 2006] and Sourir [Flückiger et al. 2018], an earlier IR with support for speculation.
- **New proof techniques:** We develop new proof techniques for proving the semantic preservation of CoreJIT. Our techniques reuse pre-existing proofs for compiler correctness and extend them for JIT compilers. Concretely, we propose nested simulations and instantiate them on CoreJIT passes, using advanced proof techniques such as loud instrumented semantics and delay invariants.
- **Verified transformations:** We prove the semantic preservation of all the passes of CoreJIT. This includes a constant propagation pass, an inlining pass and a speculative optimization pass: We show how to insert and maintain synchronization points to relate optimized and unoptimized code and use them for optimizations.
- **Efficient compilation:** To validate that our setting is realistic, we have extracted an executable artifact from our verified compiler and report on the benefit of JIT optimizations for dynamic languages on sample programs. We show how to combine the extracted artifact with an unverified frontend for a subset of Lua and an unverified LLVM backend. We identify several difficult open problems and report that performance is competitive in some cases.

Our work has limitations that we state here. We do not prove the native code generation that is discussed in the experimental section as we use LLVM for a backend. What we prove is the speculation and deoptimization happening in CoreIR. Ideally, the native code generation would be a black box that is correct as long as it preserves the semantics of CoreIR. Formally proving this is left for future work. We also do not prove the frontend that translates a minimal subset of Lua to CoreIR. We do not expect this to be challenging as the translation is straightforward. Finally, the optimizations that were proved correct are rather limited, a real compiler needs additional optimizations such as unboxing to be performant; a production JIT would also support transfers of control *into* optimized code, and non-local assumption checks. While these are limitations we expect to address in follow up works, we believe the challenges that were met in the present paper to be a significant step forward in the state of the art.

2 RELATED WORK

Mixing interpretation with run-time compilation goes back to the 60s. In his history of just-in-time, Aycock [2003] describes the evolution of JIT techniques. At first, the goal was to keep the size of program small while being able to generate efficient code for key functions. Nowadays, JIT compilers aim to have both fast start-up times and high peak performance after warm up.

Previous work has made in-roads in demystifying JIT compilation. Myreen [2010] presents a verified JIT compiler from a stack-based bytecode to x86. A challenge is being able to treat code as data. Indeed, in batch compilers, code is in a disjoint memory space from the program's data. But, with a JIT, code is produced by the compiler at run-time and has to be put in memory and instructions previously produced can be updated. Myreen defines an operational semantics for self-modifying x86 code. Memory is modeled by a function m along with instruction cache i. Instructions to execute are fetched in i, data is stored and read in m, and i can be updated with values of m. A Hoare Logic is defined to reason about these semantics. This logic includes a separating operator to describe the memory contents. The author makes a few design choices that allow the frame rule to be applied on both code and data. The correctness of the compiler is proved with the help of an invariant, *jit_inv*, which relates a state of the bytecode semantics with an equivalent state of the x86 semantics. *jit_inv s a* is a separation logic predicate which means that an x86 semantic state, corresponding to the bytecode semantic state s, is correctly represented in the memory at address a. This allows to prove that any execution of the bytecode is computed by the x86 code hidden in *jit_inv*. As the code invariant is $\emptyset$, the theorem holds for any content of the cache. In our work we avoid self-modifying code and introduce an optimizing compiler with speculative optimizations and dynamic deoptimization.

Dynamic languages are often highly polymorphic, at run-time this entails dynamic type tests and boxed primitive types. To reduce both, compilers will speculate on the most likely case and generate code specialized to that case. But there might come an execution where the speculation fails and the compiler must fall back to a more general version of the code. This process is called deoptimization and uses a technique called *on-stack-replacement* (OSR), where some stack frames are rewritten to point to the deoptimized code. OSR raises many technical issues, as the system must have full control over the layout of the internal state of the generated code. Lameed and Hendren [2013] and D'Elia and Demetrescu [2016] gave a simple implementation of OSR for LLVM that does not require self-modifying code. We follow their approach in this work.

One path towards specialization is trace-based compilation. Gal et al. [2009] introduce a JavaScript compiler based on an interpreter. Frequently executed code paths, or *traces*, are turned into efficient native code. Each trace is only correct under some assumptions about its environment. The compiler adds guards to check these assumptions hold or exit back to the interpreter. Guo and Palsberg [2011] discussed the soundness of trace-based compilers. When optimizing a trace, the rest of

the program is not known to the optimizer, so optimizations such as dead-store elimination are unsound: a store might seem useless in the trace itself, but actually impacts the semantics of the rest of the program. On the other hand, free variables of the trace can be considered constant for the entire trace. To characterize sound and unsound optimizations, the authors define semantics for trace recording and the bail-out mechanism. Finally, they define a bisimulation-based criterion for soundness. This work is not mechanically verified. The focus of our work is on a more general notion of speculation and we present a verified compiler implementation.

Flückiger et al. [2018] investigate dynamic deoptimization and provide formal tools for speculative optimization. One of their key ideas that we adopt is the direct representation of assumptions in the compiler's IR. They add an `Assume` instruction for speculation which contains a condition (the assumption) and a deoptimization target. If the condition evaluates to true, the program proceeds to the next instruction. Otherwise, it deoptimizes to a more general version of the same function. The instruction contains information on how to synthesize missing stack frames and update registers. Also, the deoptimization target is invisible to the optimizer, any analysis can be local to the version being optimized. The authors prove the correctness of various optimizations, some standard and some specific to speculation, like moving or composing assume instructions. The correctness proof relies on invariants about different versions of the same function: all versions should be observationally equivalent, and adding assumptions to a version should not alter its semantics. The equivalence is maintained by means of weak bisimulations. This IR called Sourir seems well adapted to a JIT setting, speculative optimization is possible and can be reasoned on. An interpreter for Sourir has been implemented, but this is still far from being an actual JIT. In our work we designed a new IR that abstracts deoptimization and assumption checking into two instructions which are designed to embed some of the aforementioned invariants in their semantic. We use Coq to provide a verified implementation of key optimizations.

3 A VERIFIED JIT COMPILER

The key elements of most JIT compiled systems include one or more compilers that generate code at various degrees of optimization, a monitoring subsystem that gathers information about likely program invariants and provides them to the compiler as hints and a deoptimization mechanism that allows code to toggle between optimized and unoptimized versions. In this paper, we focus on the interplay between speculative optimizations and deoptimizations and the machinery needed for these in the compiler's intermediate language as well as its run-time support. To this end we have designed a compiler infrastructure, CoreJIT, that can directly execute programs in an intermediate representation, CoreIR. This intermediate representation can be optimized by the JIT and either executed as is, or translated to native. To obtain a complete system, we add a frontend that can take source code, written in a subset of Lua, and translate it to CoreIR. The overall system is illustrated by Figure 1.

The novelty of our work lies in the design of CoreIR and the optimizations performed by CoreJIT, which we express as CoreIR transformations[1]. These are the part that have been formalized and verified. The profiler and the heuristics chosen to decide what to recompile and what to specialize need not to be trusted, our development ensures that even when these are wrong, the generated code remains correct. Correspondingly, the profiler in our development is written in OCaml and untrusted. The frontend and backend portions of the system are more "standard", they are presently unverified and must be trusted (they are further described in Section 5).

Figure 1 illustrates the compilation steps of a single function. The source function is compiled to a `Base` CoreIR version by the frontend (0). As it executes, the profiler will observe some run-time

[1]This compiler architecture follows [Flückiger et al. 2019] where R programs are JIT compiled by IR rewriting.

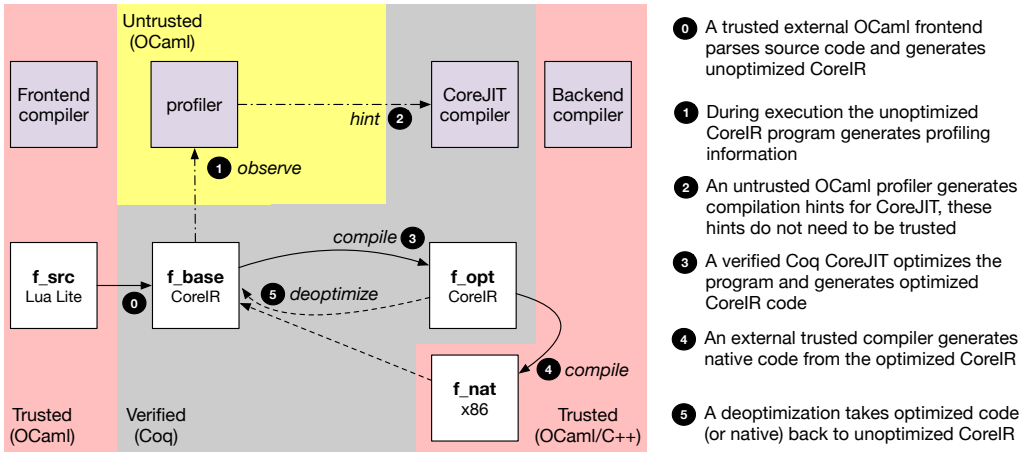


Fig. 1. CoreJIT architecture

invariant such as the type of values stored in particular variables (1). Based on heuristics, the profiler will forward some of those observations to the compiler in the form of hints (2). The compiler will optimize `Base` into an optimized `Opt` variant (3). At that point, `Opt` co-exists with `Base`, and can be executed directly or it can be compiled to native code by the trusted backend (4). At any point during execution, one of the profiler hints can be invalidated, this leads to deoptimization and transfer of control from optimized (or native) code back to the base version of the function (5). Further optimizations can replace `Opt` with more efficient versions, our development does not garbage collect compiled code, nor is compiled code ever modified.

Speculation. Two CoreIR instructions are related to speculation, `Anchor` and `Assume`. The `Anchor` instruction represents a potential deoptimization point, *i.e.*, a location in an `Opt` function where the correspondence with its `Base` version is known to the compiler and thus deoptimization can occur. For instance, in `Anchor F.l [r ← r+1]` the target `F.l` specifies the function (`F`) and label (`l`) to jump to, the mapping `[r ← r+1]`, called a *varmap* describes how to create the state of the baseline version from the optimized state. Namely, how to reconstruct the value of each register that is live at the target instruction label. As we will see later, this mapping becomes more elaborate with inlining as multiple stack frames must be synthesized, which complexifies our proof invariants.

Anchors are inserted first in the optimization pipeline, before any changes are made to the program. Choosing where to insert them is important as they determine where speculation *can* happen. Speculation itself is performed by inserting `Assume` instructions. To illustrate this second instruction, consider `Assume x=1 F.l [r ← r'+1]` which expresses the expectation that register `x` has value 1. When the instruction is executed, if `x` has any other value, we say that the assumption *failed* and deoptimization is triggered. If this occurs, the currently executing function is discarded and control is transferred to the `Base` version of `F` at label `l`, furthermore register `r` in that function is given the value `r'+1` where `r'` is looked in `Opt`. Unlike anchors, assumes are inserted at any time during compilation. To add an `Assume`, the compiler finds the dominating `Anchor` and copies its deoptimization metadata. If there is no anchor, then the assumption cannot be made.

The role of anchors is subtle. Maintaining the mapping between a `Base` and `Opt` versions is far from trivial as the optimized code gradually drifts away from its base, one transformation at a time. For the compiler, an anchor marks a point where it knows how to reconstruct the state needed by `Base` given the currently live registers in `Opt`. To be able to reconstruct that state, the anchor

keeps portions of the state alive longer than needed in Opt, and, as the compiler optimizes code, the varmap is updated to track relevant changes in the code. Thus, the cost of an Anchor is that it acts as a barrier to some optimizations. Once the compiler has finished inserting assumes, all anchors can be deleted. They will not be used in the final version of the function.

For our proofs, anchors have yet another role. They justify the insertion of Assume instructions. For this, we give the Anchor instruction a non-deterministic semantics, an anchor can randomly choose to deoptimize from Opt to Base. Crucially, deoptimization is always semantically correct, nothing is lost by returning to the baseline code eagerly other than performance. An inserted Assume is thus correct if it follows an Anchor and the observable behavior of the program is unchanged regardless which instruction deoptimizes. The benefit of having anchors is that the assumes they dominate can be placed further down the instruction stream. The compiler must make sure that the intervening instructions do not affect deoptimization. This separation is important in practice as it allows a single Anchor to justify speculation at a range of different program points. Initially the varmap of an Assume instruction will be identical to its dominating Anchor, but, as we will show shortly, this can change through subsequent program transformations. To sum up, in CoreJIT, the Anchor instruction is a helper for speculation and reasoning about correctness of speculation that is removed in the last step of the optimization pipeline.

Illustrative Example. All examples in the paper use a simplified concrete syntax for CoreIR. We omit labels when they refer to the next line and assume versions to start with the instruction on the first line. Moreover, we use the shorthand 'Anchor F.l [a,b]' to represent the varmap 'Anchor F.l [a ← a, b ← b]', and 'Assume ... ⚓l' to denote an Assume instruction with identical metadata as the Anchor instruction at label l.

```
Function F(x,y,z):
Version Base:
     d ← 1
 l1: a ← x*y
     Cond (z == 7) l2 l3
 l2: b ← x*x
     c ← y*y
     Return b+c+d
 l3: Return a
```

(a) Baseline

```
Function F(x,y,z):
Version Base:
     ...
Version Opt:
 l4: Anchor F.l1 [x,y,z,d←1]
     c ← y*y
     Assume [z=7, x=75] ⚓l4
     Return 5626+c
```

(b) Optimized

Fig. 2. Example of speculation

Assume that, for the program in Figure 2(a), a profiler detected that at label l2 of function F registers z and x always have values 7 and 75. Function F can thus be specialized. Figure 2(b) adds an Opt version to F where an anchor has been added at l4. In order to deoptimize to the baseline, the anchor must capture all of the arguments of the function (x, y, z) as well as the local register d. The compiler is able to constant propagate d, so the anchor remembers its value. The speculation is done by Assume [z=7, x=75] ⚓l4 which specifies what is expected from the state of the program and the dominating anchor. The optimized version has eliminated dead code and propagated constants. If the speculation holds, then this version is equivalent to Base. Despite the overhead of checking validity of the speculation, the new version should be faster: the irrelevant computation of a has been removed and x*x is speculatively constant folded. If the speculation fails, then the execution should return to Base, at label l1 where the closest Anchor is available, and

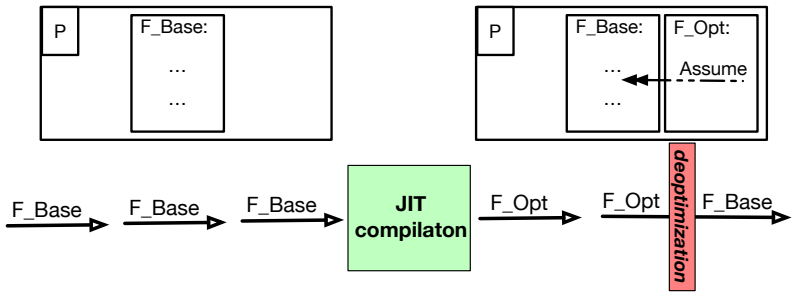


Fig. 3. Timeline

reconstruct the original environment. This involves for instance materializing the constant folded variable d. As we see here, `Assume` does not have to be placed right after an `Anchor` instruction. This will cause deoptimization to appear to jump back in time and some instructions will be executed twice. It is up to the compiler to ensure these re-executed instructions are idempotent.

Semantic Preservation Theorem. Imagine a program that consists of five invocations of the function `F`, the first four calls have the same arguments and the last one differs. Figure 3 shows a possible execution with a JIT that generates an optimized version of `F` after three invocations and then deoptimizes in the last call. For the JIT to be correct, the observable behavior of that execution should line up with an execution where we ran the base version five times in a row and with no occurrence of speculation instructions.

The semantic state of CoreJIT consists of the program being executed and the state of that program. Each execution step then consists in either calling the compiler, and thus extending the current program with new versions, or executing the next instructions. Defining CoreJIT execution as a sequence of such steps allows us to define a small-step semantics for the execution of a program. The observable behaviors of a program are to terminate, to diverge or to err. Moreover, the trace of input/output actions is also observed during execution (and it belongs to behaviors). The correctness of CoreJIT is stated as follows: For any optimization heuristics h and any source program p, let $\mathcal{B}(p)$ be the set of behaviors of p. Let $\mathcal{B}_{JIT}(p, h)$ be the set of behaviors of CoreJIT with its initial current program set to p. If we have $\mathcal{B}(p) \neq \emptyset$, then $\mathcal{B}_{JIT}(p, h) \subset \mathcal{B}(p)$, where $\mathcal{B}(p) \neq \emptyset$ means that p is a safe program (*i.e.*, with no error behavior). Restricting the theorem to safe programs and stating a backward property (*i.e.*, every behavior of the compiled program is also a behavior of the source program) a standard technique to ensure overall behavior preservation [Leroy 2009]. This theorem guarantees that using any heuristics, if the source program is safe, then the execution of CoreJIT matches one of its behaviors.

3.1 CoreIR: an Intermediate Language for Speculation

CoreIR is inspired by CompCert's RTL [Leroy 2009]. Its formal syntax is given in Figure 4. Code is represented by control-flow graphs with explicit program labels l. Each instruction i explicitly lists its successor(s). The instructions operate over an unbounded number of pseudo-registers, r, holding scalar values v (we use a for values used as addresses). A program is a map from function identifier to functions. Each function f has a default `Base` version V, its original version, and one optional optimized version `Opt`. This version may deoptimize back to its baseline if speculation fails. Versions contain code and an entry label. `Base` versions do not use the `Anchor` and `Assume` instructions. The deoptimization metadata in those operations consists of the function identifier and label where to jump to $(F.l)$ along with a varmap $((r \leftarrow e)^*)$ that specifies the value e of every

Operands:

op	$::=$	r	Register
	$\mid$	v	Value

Expressions:

e	$::=$	$op + op \mid op - op \mid op * op$	Arithmetic
	$\mid$	$op < op \mid op = op$	Relational
	$\mid$	op	Register or value

Instructions:

i	$::=$	**Nop** l	Noop
	$\mid$	$(r \leftarrow e)^* \; l$	Operations
	$\mid$	**Cond** $e \; l_t \; l_f$	Branch
	$\mid$	$r \leftarrow$ **Call** $f \; e^* \; l$	Call
	$\mid$	**Return** e	Return
	$\mid$	$r \leftarrow$ **Load** $e \; l$	Memory load
	$\mid$	$e \leftarrow$ **Store** $e \; l$	Memory store
	$\mid$	**Print** $e \; l$	Output value
	$\mid$	**Anchor** $deop \; l$	Deoptimization anchor
	$\mid$	**Assume** $e^* \; deop \; l$	Speculation

Metadata:

vm	$::=$	$(r \leftarrow e)^*$	Varmap
syn	$::=$	$f.l \; r \; vm$	Stack frame
$deop$	$::=$	$f.l \; vm \; syn^*$	Deopt metadata

Programs:

V	$::=$	$l \mapsto i$	Code
F	$::=$	$\{r^*, l, V, option \; V\}$	Function
P	$::=$	$f \mapsto F$	Program

Fig. 4. Syntax of CoreIR

target register r. The metadata can contain information to synthesize multiple frames, in which case we also specify the register that will hold the result. Some simplifications have been made to avoid tying the development to any particular language; these include the memory model and the arithmetic operations.

The small-step semantics of CoreIR instructions is detailed in Figure 5. We define its judgement as $S \; V \; l \; R \; M \xrightarrow{t} S' \; V' \; l' \; R' \; M'$, where t is a trace of observable events, and a semantic state consists of a stack S, the current version V, the current label l, registers R and a memory M. A stack is a sequence of frames (r, V, l, R) containing the register r where the result will be stored, the caller V, the label in the caller l, and the registers to restore R. States can also be final ($Final(v, M)$), in case the main function has returned, and then only contain a value and a memory. The memory is abstract and incomplete: an abstract type `mem_state`, a constant representing the initial memory state and two operations over memory states provided as partial functions for accessing values ($M[a]$) and updating them ($M[a \leftarrow v]$). Option types are used to represent potential failures. When

$$\text{Nop}\ \frac{V[l_{pc}] = \texttt{Nop}\ l_{next}}{S\ V\ l_{pc}\ R\ M \rightarrow S\ V\ l_{next}\ R\ M}$$

$$\text{Op}\ \frac{V[l_{pc}] = r_1 \leftarrow e_1, \ldots, r_n \leftarrow e_n\ l_{next} \quad (e_1, R) \downarrow v_1, \ldots, (e_n, R) \downarrow v_n}{S\ V\ l_{pc}\ R\ M \rightarrow S\ V\ l_{next}\ R\ [r_1 \leftarrow v_1 \ldots r_n \leftarrow v_n]\ M}$$

$$\text{ConT}\ \frac{V[l_{pc}] = \texttt{Cond}\ e\ l_t\ l_f \quad (e, R) \downarrow \mathit{true}}{S\ V\ l_{pc}\ R\ M \rightarrow S\ V\ l_t\ R\ M}$$

$$\text{ConF}\ \frac{V[l_{pc}] = \texttt{Cond}\ e\ l_t\ l_f \quad (e, R) \downarrow \mathit{false}}{S\ V\ l_{pc}\ R\ M \rightarrow S\ V\ l_f\ R\ M}$$

$$\text{Call}\ \frac{\begin{array}{c}V[l_{pc}] = r \leftarrow \texttt{Call}\ f\ e^*\ l_{next} \\ \texttt{current\_version}_P\ f = V' \quad \texttt{init\_regs}\ e^*\ R\ f = R'\end{array}}{S\ V\ l_{pc}\ R\ M \rightarrow (r, f, l_{next}, R +\!\!+ S)\ V'\ \mathit{entry}\,(V')\ R'\ M}$$

$$\text{Ret}\ \frac{V[l_{pc}] = \texttt{Return}\ e \quad (e, R) \downarrow v}{(r, V', l_{next}, R' +\!\!+ S)\ V\ l_{pc}\ R\ M \rightarrow S\ V'\ l_{next}\ R'\ [r \leftarrow v]\ M}$$

$$\text{RetFinal}\ \frac{V[l_{pc}] = \texttt{Return}\ e \quad (e, R) \downarrow v}{[\,]\ V\ l_{pc}\ R\ M \rightarrow \mathit{Final}(v, M)}$$

$$\text{Print}\ \frac{V[l_{pc}] = \texttt{Print}\ e\ l_{next} \quad (e, R) \downarrow v}{S\ V\ l_{pc}\ R\ M \xrightarrow{v} S\ V\ l_{next}\ R\ M}$$

$$\text{Store}\ \frac{\begin{array}{c}V[l_{pc}] = e_1 \leftarrow \texttt{Store}\ e_2\ l_{next} \quad (e_2, R) \downarrow v \\ (e_1, R) \downarrow a \qquad M' = M[a \leftarrow v]\end{array}}{S\ V\ l_{pc}\ R\ M \rightarrow S\ V\ l_{next}\ R\ M'}$$

$$\text{Load}\ \frac{V[l_{pc}] = r \leftarrow \texttt{Load}\ e\ l_{next} \quad (e, R) \downarrow a \quad M[a] = v}{S\ V\ l_{pc}\ R\ M \rightarrow S\ V\ l_{next}\ R\ [r \leftarrow v]\ M}$$

$$\text{Ignore}\ \frac{\begin{array}{c}V[l_{pc}] = \texttt{Anchor}\ f.l\ vm\ st^*\ l_{next} \\ \texttt{deopt\_regmap}\ vm\ R = R' \quad \texttt{synthesize\_frame}\ R\ st^* = S'\end{array}}{S\ V\ l_{pc}\ R\ M \rightarrow S\ V\ l_{next}\ R\ M}$$

$$\text{Deopt}\ \frac{\begin{array}{c}V[l_{pc}] = \texttt{Anchor}\ f.l\ vm\ st^*\ l_{next} \\ \texttt{deopt\_regmap}\ vm\ R = R' \quad \texttt{synthesize\_frame}\ R\ st^* = S'\end{array}}{S\ V\ l_{pc}\ R\ M \rightarrow (S' +\!\!+ S)\ (\texttt{base\_version}_P\ f)\ l\ R'\ M}$$

$$\text{AssumePass}\ \frac{V[l_{pc}] = \texttt{Assume}\ e^*\ f.l\ vm\ st^*\ l_{next} \quad (e^*, R) \Downarrow \mathit{true}}{S\ V\ l_{pc}\ R\ M \rightarrow S\ V\ l_{next}\ R\ M}$$

$$\text{AssumeFail}\ \frac{\begin{array}{c}V[l_{pc}] = \texttt{Assume}\ e^*\ f.l\ vm\ st^*\ l_{next} \qquad (e^*, R) \Downarrow \mathit{false} \\ \texttt{deopt\_regmap}\ vm\ R = R' \quad \texttt{synthesize\_frame}\ R\ st^* = S'\end{array}}{S\ V\ l_{pc}\ R\ M \rightarrow (S' +\!\!+ S)\ (\texttt{base\_version}_P\ f)\ l\ R'\ M}$$

Fig. 5. CoreIR semantics

a rule does not emit any event, the trace is omitted. The judgement for evaluating an expression e with registers R is $(e, R) \downarrow v$. Similarly, lists of expressions can be evaluated as *true* or *false*, noted $(e^*, R) \Downarrow b$.

For readability, we use identifiers to name functions. In the semantics, a function has to be found in the program. $V[l_{pc}]$ denotes the instruction at label l_{pc} of version V, and $R[r \leftarrow v]$ is the update to register map R where r has value v. The function called `current_version` returns the current version of a function (the optimized version if it exists, the base otherwise). `entry` returns the entry label. To initialize registers of function f, we write `init_regs` e^* R f, where e^* is a list of expressions to be evaluated under R. When deoptimizing, we need to reconstruct a register state with the mapping of the instructions. This is done with function `deopt_regmap`. Finally, `synthesize_frame` creates the stack frames to synthesize during deoptimization.

If R contains value 17 for register r_1 and value 3 for r_3, then `deopt_regmap [r1, r2 ← r3 + 2]` R will create a new register mapping R' where r_1 has value 17, r_2 has value 5 and all other registers are undefined. To create additional frames, `synthesize_frame` R `[F.l retreg vm]` will create the stack frame $(retreg, F, l, R')$ if `deopt_regmap vm` returns R'. At the next `Return` instruction, this will tell the execution to return to function F, at label l, with registers R' where *retreg* has been set to the return value of the function. Additionally, `synthesize_frame` can create multiple stack frames if given multiple *synth* metadata.

Most instructions have an unsurprising semantics. `Anchor` is the only non-deterministic instruction of CoreIR. The semantics of an `Anchor` is such that execution proceeds either with the next label (Ignore), or with a transition (*i.e.*, a deoptimization) to unoptimized code (Deopt). This is a helper instruction used during optimizations such as the speculation insertion pass and all `Anchor` instructions are removed at the end of optimizations. The role of this instruction is to capture what Flückiger et al. [2018] referred to as *transparency invariant*: Given an `Anchor` instruction it is always correct to add more assumptions, since the `Anchor` ensures matching states at both ends of deoptimization for all executions. `Assume` behaves deterministically, only deoptimizing if the guard fails (AssumeFail).

3.2 CoreJIT, a JIT Compiler

The main iteration loop of JIT compilation is implemented as a `jit_step` function that chooses between executing an instruction to make some progress in the execution, or calling the compiler to update its current program. This `jit_step` modifies JIT states (of type `jit_state`), where a JIT state holds the data needed to go forward in a JIT execution, namely the current program being executed and its state. `jit_step` is typed `jit_state` → (`jit_state` * `trace`).

A `jit_state` is a record (jp, ps, M, S, *synchro*, N_{opt}). jp is the current program of the JIT. As the optimizer gets called, jp will get modified. ps represents the current state of the profiler that is used by the optimization heuristics. M is the current memory state and S is the current stack of the execution as in the semantic states of CoreIR. *synchro* is called a *synchronization state*, and contains some information about the current position in the program. With the addition of S and M, it is equivalent to a semantic state of CoreIR. Last, N_{opt} represents the maximum number of optimizations that the JIT can still perform.

The profiler is called at each step. It updates the profiler state, and suggests to either optimize or execute an instruction. The program gets modified only during optimization steps. The memory state may get modified during execution. N_{opt} decreases monotonically during optimization steps. When the profiler suggests to optimize, we check that N_{opt} is still strictly positive. This ensures that execution does not get stuck if the profiler keeps suggesting optimizations.

To verify the deoptimization mechanism, we need control over the execution stack to perform on-stack replacement. Using the Coq extraction mechanism comes with limitations. Coq code is

Algorithm 1: The jit_step function

```
input  : jit_state js = (jp, ps, M, S, synchro, N_opt)
output : Next jit_state and the output trace
let new_ps = profiler ps synchro;
match (next_status new_ps N_opt) with
  case Exe ⇒
    let (int_state, newS) = forge_int_state jp S synchro;
    let (newsynchro, newM, output) = interpreter jp int_state M;
    let newjs = (jp, new_ps, newM, newS, newsynchro, N_opt);
    Return (newjs, output)
  case Opt ⇒
    let newp = safe_optimize new_ps jp;
    let newjs = (newp, new_ps, M, S, synchro, N_opt − 1);
    Return (newjs, [])
end
```

extracted to a functional subset of OCaml that does not allow to read and write from the machine stack as we please. We decide to model the execution stack as a Coq object, and implement a synchronization interface. We define synchronization states (*synchro*) to be states where we want the execution to return to the JIT. This happens at function calls, function return, and deoptimizations. Instruction execution with an interpreter can then be defined as a function that evaluates the instructions of the current program, and returns with the next encountered synchronization state. Its specification guarantees that if a synchronization state is returned, then there existed a sequence of equivalent steps in the semantics that led to a matching semantic state. Finally, the JIT state is said to be final if its synchronization state is a return state, and its stack is empty.

The `jit_step` function is described in Algorithm 1. The `next_status` function returns *Exe* if N_{opt} is down to 0, and otherwise listens to the profiler heuristics suggestion. The `profiler` function is the external parameter that updates the profiler state. The `forge_int_state` function is used when executing an instruction. It constructs the program state needed to resume execution. It also modifies the stack model and thus performs on-stack replacement if the current synchronization state is a deoptimization. Instruction execution may modify the program state and may return some observable behavior that is then output. When optimizing, we use the information from the profiler state that suggests which optimization to perform (*e.g.*, what to speculate on).

Since the JIT must be correct regardless of the implementation of the profiler, we must account for bugs in the heuristics. For instance, the profiler may suggest to inline a call that does not exist. Our correctness theorem states that if compilation succeeded, the optimized program matches the base version. In a JIT, if some optimization passes were to fail, this should not crash the entire JIT execution, otherwise the behavior of the program would not be preserved. We use the `safe_optimize` function that simply does not modify the program in case an optimization fails. This ensures that in a adversarial setting where the profiler is deliberately trying to suggest incorrect optimization to the dynamic optimizer, our JIT would still produce a behavior that matches the source program.

3.3 CoreJIT Optimizations

As discussed in the previous section, CoreJIT is defined in terms of a `jit_step` relation interfacing between optimization, execution and profiling. The optimizations happen at run-time, for instance before calling a function, the function might be optimized first. The profiling part of CoreJIT is

separate and does not need to be trusted. As such the optimizer takes optimization requests from the profiler which it may honor or not, the importance is that under any kind of optimization request the optimizer only performs correct transformations to the program. Naturally a reasonable profiler is expected to suggest optimizing hot functions and provide speculations based on the most likely future behavior. To model this, our dynamic optimizer expects the profiler to provide a list of *optimization wishes* for each optimization step. The optimizer tries to execute all of them in sequence and update the program accordingly. In the following sections we present all the verified optimization passes implemented at the time of writing, that can be requested. The proofs for those passes are described in the later Section 4.

3.3.1 Anchor insertion. Anchor insertion must be run as the first pass in the optimization pipeline as it is the only pass that relies on a fresh copy of the base version, before any changes are made by other optimizations. Given a function and a list of labels, this pass creates a copy of the base version of the function, and inserts `Anchor` instructions at every label in the list. The `Anchor` instructions are a prerequisite for the later speculation pass, called assume insertion. The profiler can suggest to insert `Anchor` instructions anywhere in a function. Selecting good locations for the insertion is the role of the profiler, and is out of scope of this work.

The following sections feature a running example with a `Function F` and its two versions. The second version was obtained by inserting an `Anchor` instruction after the `Call` instruction. The two versions do not differ except for that instruction.

```
Function F(a):
Version Base:
      t ← Call G(a,0)
  l1: Return a*t
Version Opt:
      t ← Call G(a,0)
  l2: Anchor F.l1 [a,t]
      Return a*t
```

The deoptimization target of the `Anchor` points to the base version of `F`, at `l1`, where the instruction was inserted. The deoptimization metadata includes everything needed to reconstruct the original environment. This is done according to the varmap `[a,t]`, which is sugar for `[a ← a, t ← t]` and denotes that expression `a` is evaluated in `Opt`'s environment and bound to `a` in `Base`'s environment; the same applies to `t`. Since the rest of the code is unchanged, it suffices to reconstruct the values of each defined register with its current values. Further optimization passes that would want to modify these values should also modify them in the deoptimization metadata to keep the synchronization between the two versions. The metadata is constructed using an analysis of defined registers in the current environment. In this example, the analysis sees that after the `Call`, register `t` has been assigned a value, and register `a` is also part of the environment as a parameter to the function.

3.3.2 Immediate Assume insertion. Using these `Anchor` instructions, the dynamic optimizer is able to insert `Assume` instructions. The profiler may provide the label of an `Anchor`, and a guard expression, and the optimizer will try to insert an `Assume` with that guard right after that `Anchor` instruction. In our example a request that can be satisfied would be to add the guard `t=0` at label `l2`.

```
Version Opt:
      t ← Call G(a,0)
  l2: Anchor F.l1 [a,t]
      Assume t=0  ⚓l2
      Return a*t
```

The required metadata is copied from the Anchor instruction. As syntactic sugar to avoid repetition we just refer to the Anchor by ⚓l2, to denote that the metadata is identical.

Since the profiler can suggest any guard to insert in an Assume instruction, the dynamic optimizer must ensure that this guard may not introduce bugs in the program. The semantics of the Anchor justifies that the execution will be equivalent to the original one whether the Assume deoptimizes or succeeds. The guard must however evaluate successfully, without errors. The Assume insertion pass thus includes an analysis to make sure that the guard will evaluate to some value. In this case, checking that register t has a value is enough. We reuse the same analysis as used in the previous Anchor insertion pass.

3.3.3 Constant propagation. This next optimization pass is a standard dataflow analysis seen in many compilers. It closely resembles the constant propagation pass from CompCert, and reuses its Kildall fixpoint solver library. If a register can be statically known to contain a value, it will get replaced. Expressions and instructions can be simplified.

Its interaction with the Assume instruction turns it into a speculative optimization. When constant propagation analyses an optimized version containing Assume instructions, it knows that the control flow will remain in that version only if the guard holds. Our previous example can thus be further optimized by transforming the return expression using the speculation.

```
Version Opt:
      t ← Call G(a,0)
  l2: Anchor F.l1 [a,t]
      Assume t=0  ⚓l2
      Return 0
```

Another feature of constant propagation in CoreIR is its ability to simplify deoptimization metadata. For instance, an instruction Anchor F.l [x] could be simplified to Anchor F.l [x ← 3] if the analysis shows that register x will hold value 3 at that point.

3.3.4 Lowering. Anchor instructions are used to justify Assume insertion, but should not be executed. After all optimization wishes have been treated by the dynamic optimizer the lowering pass removes all Anchor instructions, and replaces them with Nop instructions.

```
Function F(a):
Version Base:
      t ← Call G(a,0)
  l1: Return a*t
Version Opt:
      t ← Call G(a,0)
      Assume t=0  F.l1 [a,t]
      Return 0
```

In that new function, the original version has been kept intact, and a new optimized version has been inserted.

3.3.5 Delayed Assume insertion. So far we have not used the full flexibility of Assume insertion provided by Anchor instructions. For that we need to extend our running example slightly.

```
Version Base:
      t ← Call G(a,0)
  l1: Cond a=0 l2 l3
  l2: Return a*t
  l3: Return 1
```

In this variant there is an additional condition on register a, influencing the result value. We would like the profiler to have as much freedom as possible for inserting assumptions. This means for example being able to insert assumptions at l1,l2, and l3. As seen in Section 3.3.2 this could be achieved by preprocessing this version to include an Anchor at each of those labels. But, such an approach would have downsides, since additional deoptimization points constrain optimizations and unnecessarily bloat the code. Instead it is sufficient to place an Anchor at L1, and show that we can use it to justify Assume instructions at l2 and l3. As a concrete example, the following is a valid use of the Anchor by a delayed Assume.

```
Version Opt:
      t ← Call G(a,0)
      Anchor F.l1 [a,t]
      Cond a=0 L4 L5
  l4: Assume t=0  F.l1 [a ← 0,t]
      Return 0
  l5: Return 1
```

In case this assumption fails, the execution travels back in time to F.l1 executing the condition at l1 a second time. Since the condition is a silent operation and does not alter any registers referenced later, the behavior is preserved. Therefore a single Anchor instruction can serve all possible locations for Assume instructions in this example. The delayed Assume insertion pass allows the profiler to suggest arbitrary locations for an assumption. It features a verification step that rejects all requests where the in-between instructions cannot be proved to be non-interfering.

3.3.6 Inlining. The optimizer also features an inliner, a standard compiler optimization. However, inlining Assume and Anchor has to be done with caution. Consider the following program, where inlining replaces the Call instruction with the code of G. If the Assume instruction in G fails and deoptimizes, it returns to the original version of G. But if the Assume of G inlined into F fails, the execution would returns to the original version of G, which upon returning skips the rest of the execution of F.

```
Function F(a,b):
Version Base:
      a ← a*3
      a ← Call G(a,b)
      ...
  l1: Return a
Version Opt:
      a ← a*3
      a ← Call G(a,b)
      Anchor F.l1 [a,b]
      ...
      Return a
```

```
Function G(c,d):
Version Base:
  l2: Return 2*c
Version Opt:
      Assume c=12  G.l2 [c]
      Return 24
```

The solution is to use the Anchor instruction right after the call in Opt. This is where the deoptimization metadata of CoreIR really shines: we can copy the metadata from the Anchor to synthesize a stack frame for the original version of F. The parallel assignment instructions initializes the parameters of the inlined call. The Return instruction of G is now replaced by a simple assignment to the return register of the Call. If the Assume fails, we deoptimize to the original version of G.

```
Function F(a,b):
Version Opt:
     a ← a*3
     c ← a, d ← b
     Assume c=12  G.l2 [c], F.l1 a [a,b]
     a ← 24
     Anchor F.l2 [a,b]
     ...
     Return a
```

The interesting part is the rest of the metadata `F.l1 a [a,b]` allowing deoptimization to reconstruct an additional stack-frame that returns to `F` at label `l1`, return register `a`, and using a varmap `[a,b]`.

4 PROVING THE CORRECTNESS OF COREJIT

In this section we show how to prove CoreJIT correct. At first Section 4.1 introduces the relevant background from CompCert's library for proving simulations between program executions. Then, Section 4.2 and Section 4.3 explain our main two changes in these simulations so that they can be used to prove the correctness of CoreJIT: firstly we define a new simulation relation where the matching relation is another simulation relation, and secondly we adapt the notion of initial state of a program in our simulations. Last, in Section 4.4 we prove the correctness of our optimizations using our new simulations.

4.1 Simulation Relations Between Program Executions

In CompCert, a small-step semantics defines an execution relation between semantic states and associates to each program the set of its possible behaviors, including its trace of observable actions. There are numerous intermediate languages in CompCert and a generic notion of transition semantics is defined in a library devoted to semantic preservation. A semantics consists of a type of program states, a step relation over these states, and two predicates representing the initial state of a program and the final state of a terminating execution. In the remainder of this paper, for the sake of brevity, we omit other minor features and denote (`Semantics step init_state final_state`) such a semantics. CompCert defines several transition relations (*e.g.*, star and plus transitive closures) from the generic step relation, together with their properties.

The main correctness theorem of CompCert is a backward simulation that is proved to imply a property similar to the behavior preservation theorem introduced in Section 3. As a compiler, CompCert is naturally decomposed into several passes, and the main correctness theorem results from the correctness of each compiler pass. The standard technique to prove the correctness of a pass is to prove a backward simulation (*i.e.*, every behavior of a transformed program C is a behavior of the original program S).

In general, it is hard to prove a backward simulation relation, especially in the frequent case where a step in the original program is implemented by several steps in the compiled program. For passes that preserve nondeterminism, it is easier to reason on forward simulations (*i.e.*, stating that every behavior of S is also a behavior of C). A backward simulation from C to S can be constructed from a forward simulation from S to C when S is receptive and C is determinate [Sevcík et al. 2013]. S is receptive when each step produces at most one action in the observed trace, and when two matching traces t_1 and t_2 (*i.e.*, they represent the same actions in the same order) are such that any step $s \xrightarrow{t_1} s_1$ implies that there exists a state s_2 such that we have $s \xrightarrow{t_2} s_2$. C is determinate (a weaker version of determinism) when states are only allowed to take several different steps if these steps produce different observable actions.

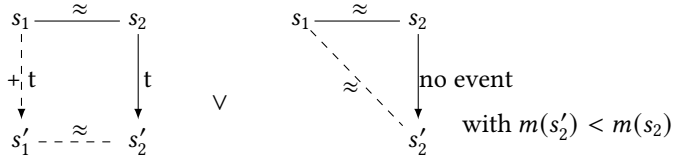


Fig. 7. A backward simulation diagram. Solid lines are hypotheses and dashed lines are conclusions. On the left of each diagram are the source program and its current state s_1; the target program and its current state s_2 are on the right. Horizontal lines represent the matching relation ≈, and vertical lines the semantic steps.

The most general backward-simulation diagram is defined in CompCert as follows. The correctness proof of a compiler pass from language L_1 to language L_2 relies on a backward simulation diagram shown in Figure 7 and expressed in the following theorem. Given a program P_1 and its transformed program P_2, each transition step in P_2 with trace t must correspond to transitions in P_1 with the same trace t and preserve as an invariant a relation ≈ between states of P_1 and P_2. In order to handle diverging execution steps and rule out the infinite stuttering problem (that may happen when infinitely many consecutive steps in P_2 are simulated by no step at all in P_1), the theorem uses a measure over the states of L_2 that strictly decreases in cases where stuttering could occur. It is generically noted $m(\cdot)$ and is specific to each compiler pass. In CompCert's parlance, this diagram is denoted by `backward_simulation (sem1 P1) (sem2 P2)`, where `semi` defines the semantics of L_i.

As explained previously, the main correctness theorem corresponds to a backward simulation and is stated in Coq as follows.

```
Lemma compiler_correct:
  ∀ p_src p_opt,
  compile p_src = p_opt →
  backward_simulation (source_sem p_src) (opt_sem p_opt).
```

As illustrated by many tricky published proofs conducted within CompCert, the gist of proving a simulation for an optimization pass relies on designing a suitable matching relation ≈ and measure, specific to this pass, then proving the preservation of the relation ≈ for each possible step.

4.2 A New Simulation Relation for JIT Correctness

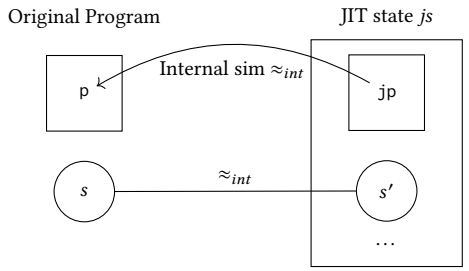
The backward simulations of CompCert need to be adapted in the context of dynamically evolving programs that change during their execution. Making the program part of the semantic states of the JIT (*i.e.*, the `jp` field of the `jit_state` record defined in Section 3.2) allows us to prove a backward simulation detailed in Figure 8, where the JIT semantics `jit_sem` goes from `jit_state` to `jit_state`. In Figure 8a, `jit_sem` is defined as a semantics (in CompCert's parlance) from the `jit_step` function defined in Section 3.2. What is called `(jit_sem p)` is then the small-step semantics associated to the JIT initialized with program `p`, in which the program included in the `jit_state` will change during execution as it gets optimized. We can prove the simulation theorem of Figure 8a, where `(core_sem p)` is the CoreIR semantics of `p`, defined in Figure 5, where the program is not part of the semantic states and remains the same during execution. This theorem states a simulation diagram, hence implying semantics preservation as in CompCert.

To prove such a simulation theorem, one needs to define a simulation invariant. Such an invariant must be adapted to our JIT setting, where the JIT states include the current state of the dynamically optimized program. Figure 8b shows the matching relation we define to prove this backward simulation. The relation $\approx_{ext}$ is a simulation invariant of the JIT execution, relating semantics states s of the original program `p` to JIT states js (consisting in the current JIT program `jp` and

```
Definition jit_sem (p:program) : semantics :=
  Semantics jit_step (init_jit_state p) (final_jit_state).

Theorem jit_simulation:
  ∀ (p: program),
  backward_simulation (core_sem p) (jit_sem p).
```

(a) Simulation theorem written in Coq



(b) In the external simulation relation, $s \approx_{ext} js$ iff 1) there exists an (internal) backward simulation between jp and p using relation $\approx_{int}$ and 2) s is matched with s' using $\approx_{int}$.

Fig. 8. Main simulation theorem and corresponding simulation relation

synchronization state s'). To match progress in jp to progress in p, this invariant should express that executing jp, possibly optimized, is equivalent to executing p. This is best described by another simulation called in the figure (internal sim $\approx_{int}$). As the program jp gets optimized during JIT execution, the internal simulation $\approx_{int}$ changes to reflect that. As a result, our proof uses an *internal* backward simulation (with a matching relation $\approx_{int}$), as an invariant to prove the main *external* backward simulation, that is used to get the correctness theorem. Our invariant also includes that the current synchronization state s' is matched using $\approx_{int}$, to some semantic state s of the original program. In other words, $s \approx_{ext} js$ iff there exists a simulation between p and jp using some invariant $\approx_{int}$ (both programs behave the same), and $s \approx_{int} s'$.

As a result, if a JIT step consists in calling the interpreter to update the current synchronization state s', we can use the internal simulation to deduce that this behavior matches some behavior s of p, and that the new synchronization state s'_1 is also matched with the same simulation relation $\approx_{int}$ to a new semantic state s_1 of p. This case is depicted on the left of Figure 9, where the dashed lines are deduced from the invariant, and interpreter correctness is required to exhibit state s_1.

If the JIT step is an optimizing one, calling the dynamic optimizer to update the current JIT program jp, then we prove that this corresponds to a stuttering step in our external backward simulation. Since the number of optimizations left decreases by one in the JIT step, this ensures that the JIT execution can not get stuck in such stuttering steps. Proving preservation of the $\approx_{ext}$ invariant amounts to proving that the new program jp' is also related to p with an internal backward simulation, as seen on the right of Figure 9. Since backward simulations compose, and jp is simulated with p from the invariant, it suffices to show that the dynamic optimizer will produce a program jp' that is itself simulated with jp with an internal backward simulation. We compose this new simulation with $\approx_{int}$ and get a new internal backward simulation $\approx'_{int}$ relating jp' to p.

Last, proving our dynamic optimizer correct then amounts to proving the following theorem, where ps is any profiler state. This specification is designed to bear a striking resemblance with the

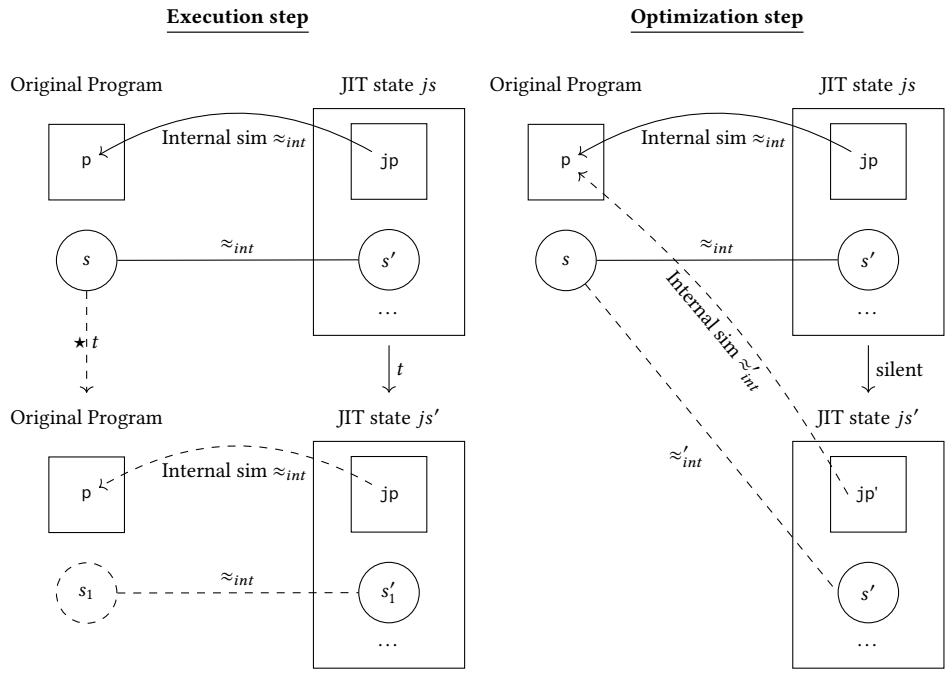


Fig. 9. The external simulation diagram

static `compiler_correct` theorem presented earlier, allowing us to reuse most of the methodology used to prove static optimization passes. In this theorem, the JIT-specific issue of dynamic optimization has been removed as the backward simulation is proved using CoreIR semantics, and not the more complex JIT semantics.

```
Theorem safe_optimize_correct:
    ∀ jp ps jp',
    safe_optimize ps jp = jp' →
    backward_internal_simulation jp jp'.
```

4.3 Internal Simulation Relations for Dynamic Optimizations

Proving the dynamic optimizer correct is then proving an internal backward simulation, between the new current JIT program, and the previous JIT program. At this point in the proof, one does not have to consider the original unoptimized program anymore, but simply the correctness of a single optimization step. Such simulations are almost identical to CompCert backward simulations, with one difference: in CompCert, the simulation relation is required to match initial states of the program. This is used to initialize the invariant when calling the optimized program. In our dynamic optimizer case, the program obtained after an optimizing step can be accessed from any synchronization state. For instance, if the interpreter comes back to the JIT with a `Call` state (meaning just before calling a function), and the called function gets optimized by the JIT, then the invariant should hold moving into that new function. As a result, we require our internal simulation relations to match all synchronization states. In practice, proving this for each pass did not require a substantial proof effort.

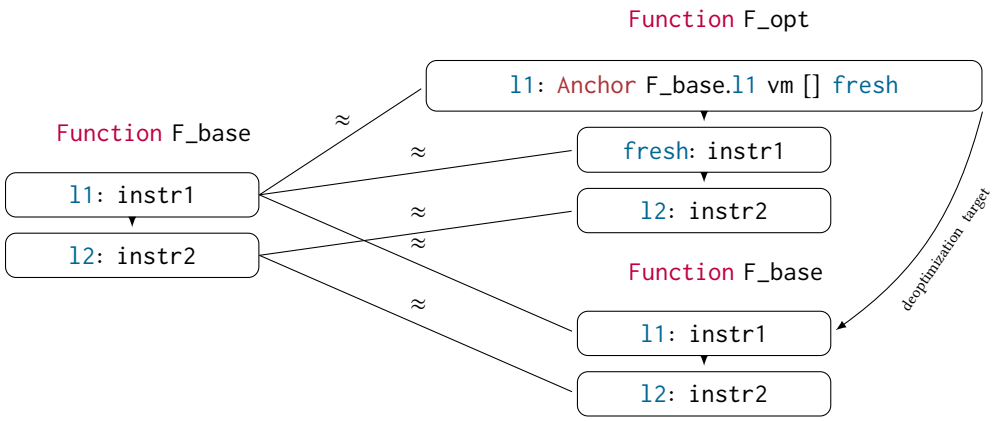


Fig. 10. Example of ≈ relation for Anchor insertion

While the proofs of internal simulations closely resemble those of a static compiler, some complexity is added by the dynamicity of optimizations. In the static case, optimizing the entire program ahead-of-time means that in the optimized execution, any call to any function `f` will be replaced by a call to a function `f_opt`. In the dynamic case, maybe function `f` is part of the execution stack before its optimization. Optimizations change the program, but not the stack. In that case, executing `f` in the source program can be both related to executing `f_opt` (when doing a new call), or `f` (when returning from the stack) in the new program. In practice, this means adding a single case to our matching inductive relations ≈ in our optimization proofs.

Internal backward simulations can also be composed. Our optimization passes can thus be proved independently, each with a backward simulation. This allows for modular correctness arguments: inserting an `Anchor` is correct for other reasons than inserting an `Assume`. The next section presents the simulation proofs for our six optimization passes.

4.4 Proving the Correctness of CoreJIT Optimizations

Finally, as all the surrounding infrastructure is in place, we are now able to present our correctness arguments for the individual optimization passes mentioned in Section 3.3, following the chronological order of this section.

4.4.1 Anchor insertion. This pass (see Section 3.3.1) is proved with an internal backward simulation. Since the pass adds new instructions, an optimized program execution will take additional steps. These additional steps are stuttering steps of the simulation, meaning that the invariant ≈ sometimes relates two consecutive states of the original execution. This invariant ≈ is depicted on an example in Figure 10, where the optimization inserted an `Anchor` at label `l1`. The original program on the left has a single version for function `f`, while the program after the `Anchor` insertion pass now has an optimized version `F_opt` of `f`. The new `Anchor` instruction can either go on or deoptimize, and in both cases, the invariant is preserved. Moreover, the deoptimization metadata includes the varmap `vm`, that assigns to each defined register its current value. To show preservation of the invariant, we prove that deoptimizing using this metadata preserves the contents of the registers, as expected. The non-determinism of the `Anchor` semantics allows to capture the synchronization between versions needed in a speculative optimizer and this invariant shows that it is adapted to a simulation methodology.

$$\frac{V[l_{pc}] = \texttt{Anchor}\ f.l\ vm\ st^*\ l_{next} \quad \texttt{deopt\_regmap}\ vm\ R = R' \quad \texttt{synthesize\_frame}\ R\ st^* = S'}{S\ V\ l_{pc}\ R\ M \xrightarrow{\textbf{GoOn}} S\ V\ l_{next}\ R\ M}$$

$$\frac{V[l_{pc}] = \texttt{Anchor}\ f.l\ vm\ st^*\ l_{next} \quad \texttt{deopt\_regmap}\ vm\ R = R' \quad \texttt{synthesize\_frame}\ R\ st^* = S'}{S\ V\ l_{pc}\ R\ M \xrightarrow{\textbf{Deopt}} (S' +\!\!+ S)\ (\texttt{base\_version}_P\ f)\ l\ R'\ M}$$

Fig. 11. Loud semantic rules for Anchor instructions

4.4.2 Immediate assume insertion. This pass (see Section 3.3.2) is also proved with an internal backward simulation. The profiler suggests a guard (a list of expressions speculated to be true) and the location of an Anchor instruction. As the dynamic optimizer should not have to trust the profiler, it checks that the suggested label actually corresponds to an Anchor, and runs an analysis to ensure that the guard will evaluate without errors. If the analysis succeeds, the Assume is inserted right after the Anchor. Inserting an Assume instruction next to an Anchor means adding a speculation check in the optimized execution, also represented by a stuttering step in the proof.

This proof uses the non-determinism of the Anchor insertion as a way to guarantee behavior equivalence regardless of the validity of the guard. For instance, if the speculation check fails in the optimized execution and the inserted Assume deoptimizes, then this behavior is related to the source behavior where the Anchor deoptimizes. Since the inserted Assume and the Anchor used for the insertion share the same deoptimization metadata, they deoptimize to the same semantic state.

4.4.3 Constant Propagation. This pass (see Section 3.3.3) closely resembles the CompCert constant propagation, but with some added interaction with the speculative instructions. For this proof, we decided to reuse the *forward to backward* methodology that is used extensively in CompCert. However, the CoreIR semantics (see Figure 5) is not determinate (see Section 4.1): the Anchor instruction can take two different steps that both produce a silent trace (outputting no event). For many optimization passes, it is important for the Anchor steps to be silent, as the behavior of Anchor should not be observable. For instance, a pass such as Anchor insertion would not be correct otherwise: it inserts new Anchor steps into the optimized execution but should not change the observable behavior of the program.

To circumvent this issue, we define a new temporary semantics called *loud* semantics for the Anchor instructions. The Anchor rules are simply augmented with a visible distinct event, as shown in Figure 11. Intuitively, an optimization pass such as constant propagation does not change the behavior of Anchor instructions: any behavior of an Anchor in the source program will behave just as the same Anchor in the optimized program. Making these behaviors explicit in the semantics allows us to comply with determinacy and still preserve a program behavior. On these new semantics, we can use the forward to backward methodology. Finally, we show that a backward simulation on loud semantics implies a backward simulation on silent semantics. In the end, we prove the following theorem.

```
Theorem fwd_loud_bwd:
  ∀ psrc popt,
  forward_internal_loud_simulation psrc popt →
  backward_internal_simulation psrc popt.
```

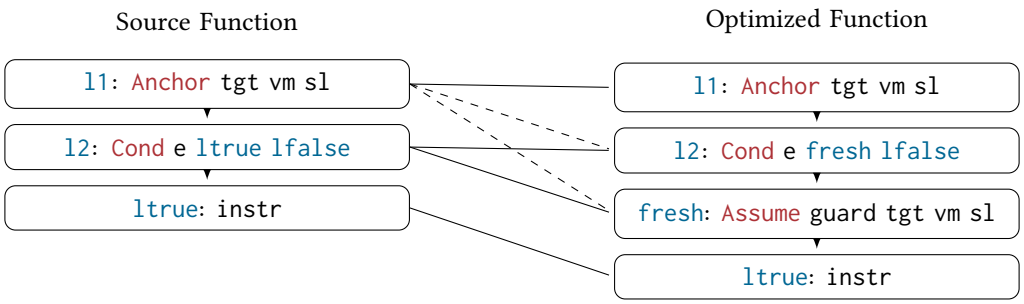


Fig. 12. The ≈ relation for delayed Assume insertion

Using this methodology, the correctness proof for constant propagation closely resembles its proof in CompCert. The other optimization passes we implemented have to be proved in a backward manner as they do not always preserve all the source Anchor behaviors or insert new ones. However, this proof suggests that most standard optimization passes could be implemented in our setting and proved just as in a static compiler.

4.4.4 Lowering. This pass (see Section 3.3.4) is also proved with an internal backward simulation. As it simply replaces Anchor instructions with Nop instructions, its correctness comes from the fact that the behavior of one such Nop instruction matches one of the possible behaviors of the Anchor: the one that goes on in the execution of the optimized version.

4.4.5 Delayed assume insertion. This pass (see Section 3.3.5) is similar to the previous Assume insertion pass in its proof. However, an execution of the optimized program where the inserted Assume deoptimizes should be related to an execution of the source where the Anchor deoptimized earlier. To this end, it is essential that no side-effect occurs between an Assume and its Anchor. Our development, for instance, proves that it is safe to insert a branch between the two speculative instructions. In our simulation invariant ≈ presented in Figure 12, the optimized function stays matched to the Anchor until the guard of the Assume is evaluated. But to be able to catch up if the guard of the Assume succeeds, we also include in our invariant that for any step taken between the Anchor and the Assume in the optimized version, there exists a corresponding step in the source version that ends up in a matching state. In the example in Figure 12, the Cond instruction of the optimized program is matched to both Anchor and Cond instructions of the source program. In the in-between states, we are then proving two simulation diagrams at once: each of these states is related both to the Anchor (in dashed lines), and to a corresponding state that has executed the same instructions. That way, if the Assume fails, then we deoptimize to the exact same semantic state in both the source and optimized versions, and the in-between instructions did not emit any observable event. And if the Assume succeeds, then the preservation of the invariant has been constructed in a way to catch up in the source version. This technique is reminiscent of simulation relations used in CompCertTSO [Sevcík et al. 2013].

In addition, the dynamic optimizer checks that the Cond instruction used for the delay is strictly dominated by the Anchor, so that every execution going through the new Assume has seen the Anchor. To show that deoptimizing from the new Assume instruction goes to the same state as if the execution deoptimized at the Anchor instruction, we show that the Cond instruction does not have any observable behavior, and does not change the evaluation of the deoptimization metadata. This part would need to be adapted if we were to extend this optimization pass to other instructions

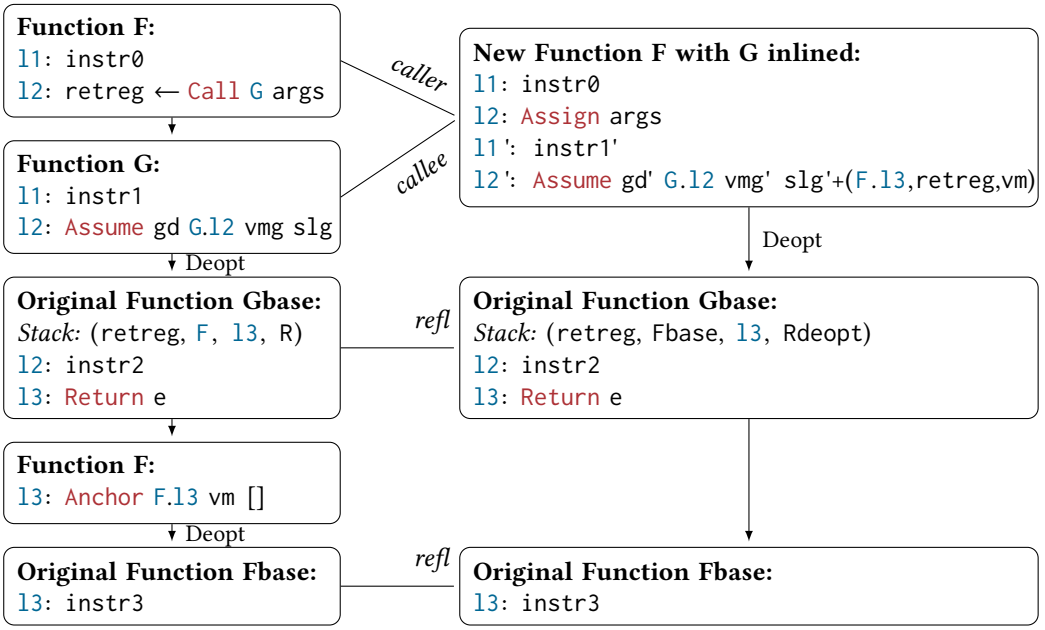


Fig. 13. Matching a deoptimization in the inlined code

(delaying the Assume after an Op instruction for instance). As in the previous proof, the optimizer also checks that the inserted guard in the Assume will evaluate without errors.

4.4.6 Inlining. This pass (see Section 3.3.6) must check that the inlined call is followed by an Anchor instruction, and use its metadata to synthesize an additional stack frame in the optimized program. Such a manipulation of the speculative instructions requires complex invariants. This pass is proved with an internal backward simulation. Our $\approx$ invariant comes in three shapes. The semantic states might represent the execution of the same version (*refl*). The target state might be in the new inlined version while the source one is in the caller function (*caller*). And the target state might be in the new version while the source one is in the function that we inlined (*callee*). In the last two cases, we design invariants to represent that the environment of the new version with inlining holds the environments of both the caller and the callee functions.

Consider Figure 13, where we show on a simplified example how this invariant can be preserved. We abbreviate Assign args the instruction that assigns each parameter of G to the corresponding expression of the list args. On the left, the source execution starts in the caller function F. As the execution progresses in F, we use the *caller* invariant. When the source calls into the callee function G, we show that the *callee* invariant holds: the new Assign args instruction successfully updates the optimized environment, which now contains the register mappings of both F and G. If a speculative instruction deoptimizes in the inlined code, it also deoptimizes in G. The optimized execution now deoptimizes back to the original version Gbase. The new metadata that has been inserted synthesizes a new stack frame, pointing to Fbase. When the execution of Gbase finishes, we prove that returning into that synthesized stack frame matches the behavior of the source where we return to function F, and use the Anchor after the call to deoptimize once more into Fbase. This example only focuses on deoptimization, but many other behaviors must be matched. For instance if the Assume succeeded, then one needs to prove that stepping out of the inlined called

```
local function fib(n)
  if n<2 then return n end
  return fib(n-1)+fib(n-2)
end
```

(a) Fibonacci in Lua Lite

```
Function fib(n_val, n_tag)
Version Opt:
    Assume (n_tag=3)
F.L [n_tag, n_val]
    Cond n_tag=3 L2 L1
L1: Call DynamicTypeError(...)
L2: ...
```

(b) Speculating that n is an integer in CoreIR

Fig. 14. Speculating on the type of Lua Lite variables

ensures a new *caller* invariant. Or if another `Call` happens during the inlined part, the optimized version now has one less stack-frame than the source in its stack. While being the most complex correctness proof in our current development, this optimization allows us to inline speculations, but also to inline `Anchor` instructions and use them later to insert new speculations.

5 EXPERIMENTAL RESULTS

This section discusses the size of our Coq development, and then answers how well CoreJIT models an actual JIT. It is set out to investigate a series of increasingly more advanced aspects of this main questions: Is CoreIR a realistic compile target for a dynamic language? Can we observe speculative optimizations in the executable artifact? Is the CoreJIT design compatible with native compilation?

5.1 Coq Development

Our Coq development reuses several Coq libraries from CompCert: data structures such as Patricia Trees, Kildall fixpoint solver for resolution of dataflow inequations by fixpoint iteration and the simulations for small-step semantics. These simulation definitions and proofs are also copied and slightly adapted to fit the *internal* backward simulations of Section 4. Finally, we reuse the behaviors definition and the proof that a backward simulation implies the behavior preservation theorem.

The rest of our development has been developed specifically for our setting. The different optimization passes, the CoreIR execution semantics and the JIT step have all been implemented and verified in Coq. This represents more than 12k lines of Coq code and proofs. Once the executable part extracted, the CoreJIT represents around 5k lines of OCaml. This extracted JIT is augmented with basic implementations of the abstract and incomplete specifications: profiling heuristics, a main OCaml function and a parser for the CoreIR language to run the JIT on user-defined programs.

5.2 A JIT for Lua Lite

CoreIR is minimal by design, but it can be a target for high-level languages, in a way that is similar to WebAssembly [Haas et al. 2017]. To evaluate the practicality of CoreIR, we implemented an unverified frontend for Lua Lite, an ad hoc subset of Lua. This subset includes a subset of values (nil, booleans, integers, tables with integer indices), no closures, no methods, and only programs where function call targets can be statically resolved. These choices are not fundamental. The frontend models Lua values as tuples of integers, where the first value holds a type tag and the second value the actual value. Speculation can be showcased on the type tags. In our experiments, we hand-craft profiler hints to speculate on the types of Lua Lite variables.

In the example of Figure 14, we show the beginning of the CoreIR function after inserting speculation. Later, the assumption `n_tag=3` (speculating that `n` is an integer) allows the subsequent

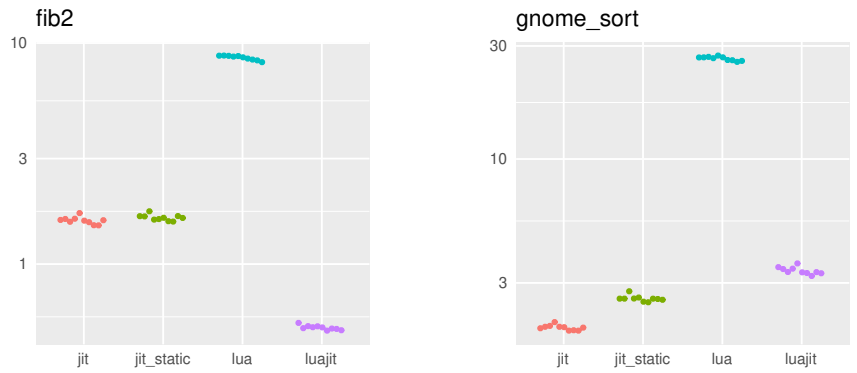


Fig. 15. Performance comparison, runtime in seconds

constant propagation pass to remove all typechecks from `n<2`, as well as `n-1` and `n-2`. We therefore observe a three-fold reduction of executed condition instructions in optimized code.

The final ingredient for our experiment is a native backend. We implemented a translation pass from CoreIR to LLVM IR. For the most part the translation is straightforward, however the points where the execution interfaces with the `jit_step` function of CoreJIT are the crux. This part of our experiments are purely to gauge feasibility. `jit_step` is designed such that there is a simple interface manifested by the synchronization state. The native backend interfaces with `jit_step` by synthesizing a synchronization state. The `jit_step` does not model the native backend, instead we add a four line patch to the extracted function that upon encountering a `Call` synchronization state hands control to the native backend instead of the interpreter. Another open issue is the memory which is modified in-place during native execution. Additional verification would need to show that this conforms how CoreIR semantics modify the memory.

By essence, the benefits of inserting speculative instructions are better observed in synergy with other optimizations. Equipped with the native backend and the powerful optimizer from LLVM we are now able to measure the impact of speculative optimizations on overall code quality for our dynamic Lua subset. We measure the runtime for two benchmarks, the first one being fibonacci from Figure 14a and the second one being an implementation of gnome sort. This implementation of the sorting algorithm is able to sort arrays where the contents are nil, booleans, or integers. In the benchmark we only pass arrays of integers and we make the profiler speculate on this type.

We ran those benchmarks each in the following configurations: (jit) CoreJIT with the unverified Lua frontend, the verified speculative and non-speculative optimizations, and the unverified native backend; (jit_static) the same as the former, but disabling speculative optimizations; (lua) the official Lua interpreter version 5.3.5; (luajit) the JIT compiler for the Lua language version 2.1.0. We ran the programs 10 times on a Laptop with a i7-7600U CPU at 2.80GHz, stepping 6, microcode version 0xc6 and 16 GB of RAM. The reported run times in seconds are measuring one execution of the whole process, including startup, compilation, and so on. In the case of `fib2` it consists of one call to `fib(39)`, in the case of gnome_sort sorting an array of length 46, 20000 times.

For fibonacci (Figure 14a) we observe that the speculative optimizations do not yield large gains, despite removing all but a third of all type checks in the optimized version. The reason is that those type checks can be removed by common subexpression elimination as well, which in fact LLVM does, therefore the similarity in performance is as expected.

For gnome sort, we observe a large gain by speculative optimizations. Here the speculation happens in a loop on values loaded from the heap and the data shows that a non-speculating compiler such as our LLVM backend cannot optimize this code well without the speculation. In this light, we can see how our design for speculation in synergy with the LLVM optimizer can lead to speedups. We observed a performance overhead of speculation due to the metadata captured by the `Assume` instruction in CoreIR, which sometimes can prolong the liveness of registers. If we (unsoundly) elide all metadata in the deoptimization points in this example we observe another two fold speed up. Finally, if CoreJIT outperforms luajit in this example, this is expected since we do not have support for actual lua tables, but instead only for the subset that uses integer keys and can therefore be represented as arrays.

6 CONCLUSION

We have proved the correctness of CoreJIT, a JIT compiler for CoreIR that performs speculative optimizations and deoptimizations. Speculative optimizations and deoptimizations are used in industrial practice, and CoreJIT can be used as as correct reference description for demystifying the interplay between them. Moreover, our proofs reuse the simulation framework of CompCert, thus showing that verification techniques for traditional static compilation can be adapted to JIT compilation.

In the future, it would be interesting to prove more optimizations and extend CoreIR and CoreJIT to a more realistic language such as RTL. For instance, inserting and proving a liveness analysis in our `Anchor` insertion pass would prevent the speculation instructions from extending the liveness of registers. Extending our `Assume` insertion with delay to insert over other instructions could also increase performance by reducing the number of `Anchor` to insert in a program. We believe that our current proof methodology fits to support these simple extensions. We also intend to extend our proof methodology to establish the correctness of a translation to a native code backend. This raises the challenge of modular verification and CoreIR-assembly linking. A promising step toward this direction would be to study the RUSC refinement technique of CompCertM [Song et al. 2019].

ACKNOWLEDGMENTS

This work 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 772568), the National Science Foundation awards 1544542 and 1618732, the Czech Ministry of Education from the Czech Operational Programme Research, Development, and Education, under agreement CZ.02.1.01/0.0/0.0/15_003/0000421, and the European Research Council under grant agreement 695412.

REFERENCES

John Aycock. 2003. A brief history of just-in-time. *ACM Comput. Surv.* (2003). https://doi.org/10.1145/857076.857077

Clément Béra, Eliot Miranda, Marcus Denker, and Stéphane Ducasse. 2016. Practical validation of bytecode to bytecode JIT compiler dynamic deoptimization. *Journal of Object Technology (JOT)* 15, 2 (2016). https://doi.org/10.5381/jot.2016.15.2.a1

Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. 1999. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Conference on Java Grande (JAVA)*. https://doi.org/10.1145/304065.304113

Project Chromium. 2020. V8 JavaScript Engine. https://chromium.googlesource.com/v8/v8.git.

Daniele Cono D'Elia and Camil Demetrescu. 2016. Flexible on-stack replacement in LLVM. In *Code Generation and Optimization (CGO)*. https://doi.org/10.1145/2854038.2854061

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 *Virtual Machines and Intermediate Languages (VMIL)*. https://doi.org/10.1145/2542142.2542143

Olivier Flückiger, Guido Chari, Jan Jecmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R melts brains: an IR for first-class environments and lazy effectful arguments. In *International Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/3359619.3359744

Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of speculative optimizations with dynamic deoptimization. 2, POPL (2018). https://doi.org/10.1145/3158137

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1542476.1542528

Shu-yu Guo and Jens Palsberg. 2011. The essence of compiling with traces. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*. https://doi.org/10.1145/1926385.1926450

Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062363

Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/143095.143114

Urs Hölzle and David Ungar. 1994. A Third-generation SELF Implementation: Reconciling Responsiveness with Performance. In *Object-oriented Programming Systems, Language, and Applications (OOPSLA)*. https://doi.org/10.1145/191080.191116

Nurudeen A. Lameed and Laurie J. Hendren. 2013. A Modular Approach to On-stack Replacement in LLVM. In *Virtual Execution Environments (VEE)*. https://doi.org/10.1145/2451512.2451541

Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1111037.1111042

Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

Magnus Myreen. 2010. Verified Just-in-time Compiler on x86. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1706299.1706313

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot Server Compiler. In *Java Virtual Machine Research and Technology (JVM)*. http://www.usenix.org/events/jvm01/full_papers/paleczny/paleczny.pdf

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22:1–22:50. https://doi.org/10.1145/2487241.2487248

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091