Certifying assembly optimizations in Coq by symbolic execution with hash-consing

Léo Gourdin and Sylvain Boulmé *

Université Grenoble Alpes, CNRS, Grenoble INP, Verimag
{Leo.Gourdin,Sylvain.Boulme}@univ-grenoble-alpes.fr

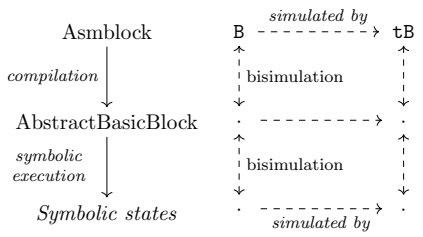
Abstract

We extend the CompCert C compiler for AArch64 processors with *instruction scheduling* and *instruction compaction*. We reuse the *translation validation* technique of [3]: an *untrusted* OCaml *oracle* performs the translation, and a *formally-verified* test checks that the code produced by the oracle simulates the original code. This verified test is composed of a processor-dependent frontend (ported for AArch64) with a generic backend based on *symbolic execution* with *hash-consing* (directly reused from [3]).

Previous works, such as [2], established that *symbolic execution* is effective in *validating* state-of-the-art compilers. Tristan and Leroy [4] used *formally-verified* symbolic execution to *certify* an untrusted scheduler within the CompCert compiler. Alas, their formally-verified checker had exponential complexity and was thus never integrated into mainline CompCert. Recently, our team [3] solved this performance issue with formally-verified hash-consing within the symbolic execution, and applied the resulting verifier to the certification of a scheduler for a VLIW processor. We report this approach and how we ported it to the more widespread AArch64 architecture. In average (this may vary depending on the benchmarks), the CompCert generated code for Cortex-A53 is sped up by around 10%, and its size is reduced by around 3%. This encourages the generalization of such an approach to more ambitious optimizations.

Overview of our design. Our optimization operates on *basic blocks*[1]. First, our CompCert backend recovers the basic block structure of the assembly program through a new processor-specific Intermediate Representation (IR), called Asmblock. Then, each basic block `B` of this Asmblock program is optimized (by an untrusted oracle) into a basic block `tB` that is checked to simulate `B` by a formally-verified test. Finally, the resulting Asmblock program is translated to the standard Asm IR of CompCert (by forgetting basic block boundaries).

The core of the simulation test is implemented on AbstractBasicBlock [3], a processor-generic IR, representing basic blocks as sequences of atomic assignments. As pictured on the right-hand side, each basic block is thus compiled from Asmblock to AbstractBasicBlock before being compared. The simulation test on AbstractBasicBlock uses *symbolic execution*, which simply consists in compiling each basic block into a big symbolic term—actually called a *symbolic state*—in order to deduce the simulation from syntactical equalities on symbolic states. Here, such a symbolic state simply corresponds to a kind of *preconditioned parallel assignment*, as illustrated on Example 1. The overall proof of the simulation of `B` by `tB` corresponds to compose the two commutative diagrams in the above figure.



*Work partially supported by LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by French program Investissement d'avenir.

[1] By definition, a *basic block* is a sequence of *assembly instructions* with at most one branching instruction, which is in this case in final position, and such that the ambient program only enters this sequence at the first instruction. Hence, optimizations (e.g. instructions rewriting or reorderings) that (locally) preserve the semantics of the basic block, also (globally) preserve the semantics of the ambient program.

Example 1 (Simulation on symbolic states)**.** Consider two "abstract" basic blocks B_1 and B_2:
(B_1) $r_1 \coloneqq r_1 + r_2;\ r_3 \coloneqq \texttt{load}[m, r_1];\ r_3 \coloneqq r_1;\ r_1 \coloneqq r_1 + r_3$
(B_2) $r_3 \coloneqq r_1 + r_2;\ r_1 \coloneqq r_3 + r_3$
B_1 and B_2 lead to the same parallel assignment: $r_1 \coloneqq (r_1 + r_2) + (r_1 + r_2) \parallel r_3 \coloneqq r_1 + r_2$. But, normal execution of B_1 is preconditioned by "$\texttt{load}[m, r_1 + r_2]$ has not trapped", whereas the precondition of B_2 is trivially true. Hence B_2 simulates B_1, but the converse is false.

A simple proof of hash-consing within symbolic execution. As suggested by duplication of term "$r_1 + r_2$" in Example 1, symbolic execution involves many replicas of terms. Thus, comparing symbolic states with structural equalities of terms, as performed in [4], takes exponential time. This issue is solved in [3] thanks to verified hash-consing. Hash-consing consists in memoizing the constructors of inductive data-types in order to ensure that two structurally equal terms are actually allocated to the same object in memory [1]. This enables to replace (expensive) structural equalities by (constant-time) pointer equalities.

Hash-consing is itself realized within an *untrusted* OCaml oracle whose result is *dynamically* verified by a *certified* constant-time test. This defensive test simply checks, using pointer equalities, that the memoization oracle turns each term into a structurally equivalent term.

Because representing pointer equality as a "pure" function would be unsound, OCaml pointer equality is instead axiomatized as returning a "non-deterministic" Boolean (within a dedicated monad) such that result "`true`" implies Coq equality. This approach provides a quite simple formal proof for an efficient symbolic simulation test (see [3]).

Our untrusted oracle for AArch64. On pipelined processors, an instruction sequence may take significantly less time if executed according to a favorable instruction order. High-performance processors schedule instructions dynamically, at the price of power and space consumption. On simpler processors, such as Cortex-A53 (AArch64), performance may be improved by static instruction scheduling, at compile-time. Our oracle instantiates the list-scheduling procedure described in [3] on latency and resource constraints specific to Cortex-A53 core (found in LLVM sources, because this processor lacks of public documentation).

Before scheduling, our oracle also reduces the code size by replacing pairs of simple load/store instructions to consecutive addresses by single double load/store instructions. In contrast to [3], our oracle is able to merge non-consecutive load/store within the original basic block and thus performs a kind of instruction reordering prior to scheduling. Our formally-verified simulation test validates these replacements by performing the *reverse* rewritings (i.e. from double load/store to pairs of simple load/store) in the Asmblock-to-AbstractBasicBlock pass. Indeed, proving semantic-preservation is *much simpler* in this way than on the replacements of the oracle. Our port for AArch64 represents about three man·months of development.

References

[1] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the ACM Workshop on ML, 2006*, pages 12–19. ACM, 2006.

[2] George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation (PLDI)*, pages 83–94. ACM Press, 2000.

[3] Cyril Six, Sylvain Boulmé, and David Monniaux. Certified and efficient instruction scheduling. Application to interlocked VLIW processors. *PACMPL (OOPSLA 2020)*, November 2020.

[4] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *POPL*, pages 17–27. ACM Press, 2008.