PhD Student session: formally verified postpass scheduling with peephole optimization for AArch64

Léo Gourdin

Université Grenoble Alpes — Verimag & TIMA Laboratories

leo.gourdin@grenoble-inp.org

Abstract

COMPTh is a C compiler with a complete machine-checked proof of semantic preservation from C to assembly [Ler09]. We present here an extension of COMPTh for AArch64 processors, that optimizes the use of the pipeline in the processor (postpass scheduling) and performs instruction compaction (e.g. replaces pairs of simple load instructions, by single double load instructions), through a technique called peephole optimization. Our method is founded on a two-tier design, introducing an untrusted oracle performing the translation, and a formally-verified checker testing whether the code produced by the oracle simulates the original code. We reuse here the generic checker, based on symbolic execution with a hash-consing mechanism, of [SBM20]. The paper presents the correctness proof of my optimizations, and experimental measurements of performance improvements. More generally, my thesis explores how to apply and generalize such mechanisms of translation validation in order to extend COMPTh with target-dependent optimizations.

1 Introduction and related works

An instruction sequence may take significantly less time if executed according to a favorable schedule. Indeed, the simultaneous use of all processor units may be maximized with a smart interleaving of “parallelizable” computations. High-performance processors schedule instructions dynamically, but this complicates their design. COMPTh is often used by industrials working with Safety-critical systems (SCS) [BFBFF+12], that must remain reliable, not too complex, and predictable. In-order cores, which do not dynamically reorder instructions, are thus a common choice to meet these needs. On such processors, performance may be improved significantly if the compiler schedules instructions intelligently. Within compilers, instruction scheduling is usually split in two passes: a “coarse-grain” one, in an Intermediate Representation (IR), before register allocation and a “fine-grain” one, after register allocation, on the emitted assembly.[1]

Such a (verified) ”coarse-grain” prepass scheduling optimization has been recently proposed for COMPTh [SGBM21]. This paper presents a (verified) “fine-grain” postpass scheduling optimization.

[1]This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d’avenir.

[2]Combining scheduling and register allocation is useful to avoid a high register pressure, but existing works such as [LCBST9, MPSR95] are challenging and does not seem to scale.
and a peephole optimization (i.e. instruction compaction on load and store). Both optimizations are performed by untrusted OCAML oracles. We adapt the work of [SBM20], where they present a new IR Asmblock to define basic blocks\(^2\) at the assembly level, and a generic checker, formally proved correct in Coq, defined above a dedicated IR called AbstractBasicBlock. [Nec00] and [TGM11] have experimented that combining symbolic execution with rewriting is effective to validate the code produced by state-of-the-art optimizing compilers. In the meantime, [TL08] have used a formally-verified symbolic execution in order to formally certify the schedules produced by an untrusted oracle within the COMP\textsc{Cert} compiler. Unfortunately, their checker had exponential complexity [Trn09, §6.7.1], [TL08, §7] and their formally-verified scheduler was never released. [SBM20] tackle this issue with a dynamically verified hash-consing mechanism: an untrusted OCAML oracle memoizes symbolic terms, and its results are dynamically checked with an axiomatized pointer equality.\(^3\) We port their work on the AArch64 architecture, and also applies this translation validation solution to verify the correctness of peephole replacements. In contrast to the peephole optimizations of [MZTG16] (for x86-32), we do not consider register liveness nor arithmetic transformations of pointers, but they do not tackle instruction scheduling and their model of basic blocks relies on some unverified assumptions.

2 Architecture of our solution and its formal proof

Our solution, pictured in Fig. 1, reuses the generic basic blocks construction method of [SBM20] through Machblock. W.r.t. [SBM20], the whole proof effort consists in adapting to our target the Asmblock IR and the various translations from and to this IR. As pictured in Fig. 1, untrusted optimizations are together applied to each basic block \(B\) producing a basic block \(t_B\), which are then both translated to the generic AbstractBasicBlock IR for verification. Hence, the results of our combined oracles are verified in one pass, with a single checker. We apply the peephole

\(^2\)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.

\(^3\)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 model seems a quite reasonable, and enables an efficient symbolic simulation test.
optimization before the scheduling pass in order to leave more scheduling opportunities after load and store pairing.

Based on the existing Asm on the back-end, we build the Asmblock IR by defining a new instruction hierarchy\(^4\) We prove the correctness of our optimizations thanks to a simulation test on the AbstractBasicBlock IR, deducing the simulation from syntactical equalities on “symbolic states”, after symbolic execution.\(^5\) See [SBM20, §4.3]. The overall proof of the simulation of B by t\(B\) corresponds to compose the two commutative diagrams on the right-hand side of Figure 2.

3 The Postpass Scheduler

This oracle is declared as a COQ function taking a basic block in input, and returning a tuple containing a list of basic instructions (i.e. the basic block body) and an optional (using the option monad) control-flow instruction (i.e. the basic block exit). The main obstacle is then to retrieve the most precise information possible about latencies of instructions, to be able to correctly “tune” the oracle. The difficulty resides in the fact that measuring correctly the number of execution cycles for each instruction is hard, and the manufacturer in the case of the Cortex-A53 (i.e. ARM) does not provide such information. However, the AArch64 LLVM back-end is using a similar postpass scheduling optimization, and the source code\(^6\) contains some latency information. Another source we used is an article by [Wig19] where some latencies are manually measured.

Concerning the set of read and written registers for each instruction, it could be deduced from the Asm semantics. When implementing our solution, we have discovered bugs in this semantics: indeed, some instructions such as Pfmovimms, Pfmovimmd and Pbtl were incorrectly described. The first two are destroying a scratch register before writing the result in the destination register, and the latter is in fact preserving a scratch register contrary to what its semantics describes. These three instructions are macros expanded later on in an unverified part of COMP\(\text{CERT}\), the TargetPrinter, into several real Asm instructions. The formalization of their behavior in COQ was not correct, and it was possible to generate incorrect code by interleaving them with other macros that are using the same scratch register, and which are expanded in COQ, at the Asm level (so before our scheduling). The bug was invisible in the sense that as instructions were never reordered at the Asm level, an incorrect code could not appear (the only way was either to reschedule instructions as we do, or to write it manually). Thus, our verifier combined with the postpass scheduling can help us find errors in the trusted base of COMP\(\text{CERT}\).\(^7\)

\(^4\) We do not detail the chosen hierarchy here, but the interested reader can note that a smart grouping of instructions by operands (according to the number and type of input registers) helps to produce more compact definitions in the checker and a shorter overall proof.

\(^5\) This method simply consists in compiling each program into a big symbolic term, called a symbolic state

\(^6\) Accessible here as a TABLEGEN code.

\(^7\) Those bugs are now fixed in the COMP\(\text{CERT}\) mainline repository.
Figure 3: Four Examples of load/store Compaction on AArch64

4 The peephole optimizer

In contrast to the peephole optimizer of [SBM20], ours is able to merge non-consecutive load or store within the original basic block, as long as they respect the semantic dependencies and offset constraints on double load/store specific to AArch64 Instruction Set Architecture (ISA). Our algorithm traverses the basic block in both directions, while remembering every encountered compatible load and store as potential candidates (and forgetting them if another instruction breaks a needed dependency in-between). The first pass (forward) tries to replace the last encountered load or store by the double instruction, and the first one by a Nop (no operation) instruction. The second pass (backward) tries the opposite.

Figure 3 illustrates four situations found by our peephole optimizer. On the left column: 1. backward load pairing, with increasing offset (the offset of the second load is greater than that of the first one); 2. consecutive load pairing, with increasing offset. On the right column: 1. consecutive load pairing, with decreasing offset (the offset of the second load is lower than that of the first one); 2. forward store pairing, with increasing offset.

Currently, the main benefit of our peephole optimizer for AArch64 is a reduction of code size: it reduces the number of generated memory transfer instructions by about 10%, which represents approximately 3% of the total code length (on average across all our benchmarks). Like [SBM20], our formally-verified simulation test validates these rewritings by performing the reverse rewriting (i.e. from double load/store to pairs of simple load/store) in the AsmBlock-to-AbstractBasicBlock pass (see Fig. 2).

5 Conclusion and future work

Using such a low level scheduling pass allows a finer tuning of instructions latencies compared to the existing prepass. On average, running on a Raspberry Pi 3 with a Cortex-A53 core, our oracle alone raises performance by 9.11% across all our benchmarks, and using prepass scheduling [SGBM21] on top of postpass makes us reach 22% of performance improvement. The postpass itself brings a gain of about 5.46% comparing to COMP-cert with all optimizations turned on, as they allow generating larger, more profitable, basic blocks. The overall implementation of our formally-verified optimization pipeline on AArch64 represents a bit more than three man-months of development. This constitutes for us the first step for future optimizations exploiting the same principle of a posteriori verification, which we will study in the rest of the thesis.

---

8Based on Polybench [Pou12], TACLeBench [FAH+16], and some additional benches described in [SBM20].
9Including among others Loop Invariant Code Motion (LICM), loop-unrolling, loop-rotate, and tail duplication.
References


