# Certified Compiler Backends for VLIW Processors Highly Modular Postpass-Scheduling in the CompCert Certified Compiler

Cyril Six, Sylvain Boulmé, David Monniaux

# Certified Compiler Backends for VLIW Processors

Highly Modular Postpass-Scheduling in the CompCert Certified Compiler

CYRIL SIX, Kalray, France
S. BOULMÉ and D. MONNIAUX, Univ. Grenoble Alpes, CNRS, Grenoble INP*, VERIMAG, France

CompCert is a C compiler with a formal, machine-checked, proof of correctness: after successful compilation, the object code has a behavior faithful to the source code. It is moderately optimizing; in particular, it does not reorder instructions. To get good performance for in-order and, more specifically, VLIW processors (processors with explicit parallelism at the instruction level), it is necessary to reorder instructions. Previous attempts at reordering instructions in CompCert did not scale.

We present here a scalable, efficient approach for scheduling and reordering instructions in CompCert backends. We apply it to the VLIW assembly of the Kalray K1C processor, but our approach has wider scope.

Additional Key Words and Phrases: Formal verification of compiler optimizations, Instruction-level parallelism, Instruction pipelining, the Coq proof assistant.

## 1 INTRODUCTION

The CompCert certified compiler [Leroy 2009a,b] is the first optimizing C compiler with a formal proof of correctness that is used in industry [Bedin França et al. 2012; Kästner et al. 2018]. In particular, it does not have the middle-end bugs usually found in compilers [Yang et al. 2011], thus making it a major success story of software verification.

CompCert features a number of middle-end optimizations (constant propagation, inlining, common subexpression elimination, etc.) as well as some backend optimizations (register allocation using live ranges, clever instruction selection on some platforms). However, it does not attempt to reorder operations, which are issued in almost the same order as they are written in the source code. This may not be so important on processors with out-of-order or speculative execution (e.g. x86), since such hardware may dynamically find an efficient ordering on its own. Actually, it hinders performance on in-order processors, especially if they are superscalar (multiple execution units capable of executing several instructions at once, in parallel).

VLIW (Very Long Word Instruction) processors [Fisher 1983] require the assembly code to specify explicitly which instructions are to be executed in parallel. A VLIW instruction thus represents some aggregate of atomic computations running in parallel on the execution units of the processor. Compared to out-of-order architectures, an in-order VLIW processor has a simpler control logic, thus using less CPU die space and energy for the same computing power; it is more predictible with respect to execution time, which is important in safety-critical applications where a worst-case execution time (WCET) must be estimated or even justified by a sound analysis [França et al. 2011]. In addition, a simpler control structure may be more reliable.[1]

Due to their simpler design, such processors require more complex compilers to benefit from their potential. Compilers must indeed find an efficient way to decompose the behavior of the high-level program (typically in C) into a sequence of parallel atomic computations. Optimizing compilers for VLIW processors has a long and successful history since the seminal work of Fisher

[1]For instance, Intel's Skylake processor had a bug that crashed programs, under complex conditions [Leroy 2017].

Authors' addresses: Cyril Six, Kalray, 180 Avenue de l'Europe, Montbonnot-Saint-Martin, F-38330, France, csix@kalray.eu; S. Boulmé, Sylvain.Boulme@imag.fr; D. Monniaux, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, 38000, France, David.Monniaux@univ-grenoble-alpes.fr.

2019. Manuscript submitted to ACM

[1981]; Rau et al. [1982], followed by Feautrier [1991]; Lam [1988] and the Multiflow compiler Lowney et al. [1993]. In the case of CompCert, the problem is made harder by the need to formally verify that this transformation is sound, that is, preserving the program semantics.

This paper presents an extension of CompCert with certified assembly generation for a VLIW processor (Kalray K1c core), along with an intrablock postpass scheduling optimization (postpass meaning that it occurs after instruction selection, register allocation, and spilling). However, only a few parts are specific to this processor: many of the insights and a large part of the implementation are likely to be applicable to other architectures, either for postpass scheduling on other VLIW architectures, or for intrablock pre-pass scheduling on any kind of architecture.

### 1.1 Overview of the Kalray K1c VLIW processor

The Kalray k1c VLIW core implements a 6-issue Fisher-style VLIW architecture [Fisher et al. 2005] (partial predication, dismissable loads, no rotating registers). It executes blocks of instructions called *bundles*, in the same order as specified in the machine code.

*Bundles.* A *bundle* is a block of instructions that are to be issued in the pipeline at the same cycle. They execute in parallel with the following semantics: if an instruction writes into a register that is read by another instruction of the same bundle, then the value that is read is the value of the register prior to executing the bundle. If two instructions of the same bundle write to the same register, then the behavior at runtime is non-deterministic. For example, the bundle written in pseudo-code "$R_1 := 1; R_1 := 2$" assigns $R_1$ non-deterministically. On the contrary, "$R_1 := R_2; R_2 := R_1$" is deterministic and swaps the contents of $R_1$ and $R_2$ registers in one atomic execution step. In assembly code, bundles are delimited by the `;;` token (Fig. 1). Compilers must ensure that each bundle does not require more resources than physically available—for instance, the K1c has only one load/store unit, thus a bundle should contain at most one load/store instruction. The assembler refuses ill-formed bundles.

*Execution pipeline.* In the case of the K1c, bundles are executed through a 6-stage pipeline: the first two stages respectively decode the instruction (ID stage) and read the registers (RR stage), then the last four stages (E1 through E4) perform the actual computation and write to the destination registers. If, during the RR stage [2], one of the read registers of an instruction in the bundle is not available, the pipeline *stalls*: the bundle stops advancing through the pipeline, and only continues once the register gets its result. Figure 1 depicts this behavior.[3]

*In-order execution.* Processor implementations can be divided into: *out-of-order* processors (e.g. modern x86), which may locally re-schedule instructions to limit stalls, or to better exploit execution units;[4] *in-order*, which execute the instructions exactly in the same order as they are seen in the assembly code. As the K1c is an in-order processor, it is particularly important for the compiler to provide an efficient schedule.

### 1.2 Modular design of the CompCert certified compiler

Usual compilers (GCC, Clang/LLVM, ICC) split the compilation process into several components. In the case of CompCert, a *frontend* first parses the source code into an Intermediate Representation (IR)—called Cminor—that is independent from the target machine [Blazy et al. 2006]. Then, a

[2] Or the ID stage, for some instructions such as conditional branching.

[3] When a register is read before some prior instruction has written to it, *non-interlocked* VLIW processors use the old value. The compiler must then take instruction latencies and pipeline details into account to generate correct code, including across basic-blocks. This is not the case for the K1c, where these aspects are just matters of code efficiency, not correctness.

[4] For instance, in Fig. 1, seeing that the bundle B3 is stalled because its arithmetic instructions depend on a load in B2, an out-of-order processor could instead schedule the execution of B6.

| Cycle | ID | RR | E1 | E2 | E3 |
|---|---|---|---|---|---|
| 1 | B1 | | | | |
| 2 | B2 | B1 | | | |
| 3 | B3 | B2 | B1 | | |
| 4 | B4 | B3 | B2 | B1 | |
| 5 | B4 | B3 | STALL | B2 | B1 |
| 6 | B4 | B3 | STALL | STALL | B2 |
| 7 | B5 | B4 | B3 | STALL | STALL |

$$B1 : R_1 := load(R_0 + 0);;$$
$$B2 : R_2 := load(R_0 + 4);;$$
$$B3 : R_3 := R_1 + R_2;\ R_4 := R_1 * R_2;;$$
$$B4 : R_3 := R_3 + R_4;;$$
$$B5 : store(R_0, R_3);;$$
$$B6 : R_6 := R_7 + R_8;;$$

Fig. 1. The pipeline stalls at cycles 5 and 6 because B3 is waiting for the results of $R_1$ and $R_2$ from bundles B1 and B2, which are completed at stage E3

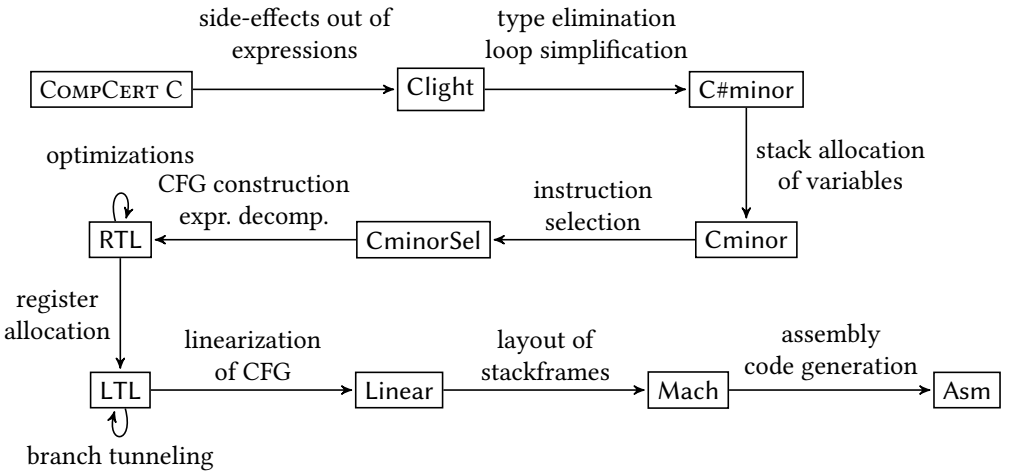



Fig. 2. The different languages and compilation passes of CompCert.

*backend* transforms the Cminor program into an assembly program for the target machine [Leroy 2009b]. Each of these components introduces several IRs, which are linked by *compilation passes*. A compilation pass can either transform a program from an IR to another (transformation pass), or optimize within an IR (optimization pass). As illustrated in Figure 2, CompCert actually introduces more IRs than usual compilers. This makes its whole proof more modular and manageable, because each compilation pass comes with its own proof of *semantic preservation*.

Within the backend, compilers usually first manipulate an *unbounded* number of *pseudo-registers*, which are then mapped to actual machine registers, with possible *spills* (saving on the stack, then reloading) when needed. This mapping is performed by the *register allocation* pass. Compiler backend passes are usually divided into two groups: those happening before register allocation, and those happening after. This paper presents a *postpass scheduling* optimization: it reorders instructions at the very end of the backend, after register allocation. This scheduler emits bundles: two instructions are in the same bundle if they are scheduled in the same cycle.

### 1.3 The challenges of porting CompCert to a VLIW architecture

Porting a VLIW architecture such as the K1c processor presents two main challenges:

- *How to represent bundles in CompCert?* The existing Asm languages of the underlying architectures define an instruction-per-instruction execution semantic, but ideally we would need a bundle-per-bundle semantic for our VLIW processor.
- *How to include a scheduling pass within CompCert?* On in-order architectures, it is of paramount importance for speed of execution that the instructions are ordered in a way that minimizes stalls, which is not, in general, the order in which they are written in the C program. A scheduling pass reorders the instructions, with knowledge of their execution latencies, so as to minimize stalling. For instance, in Fig. 1, this pass could schedule B6 before B3. Furthermore, the task of bundling (regrouping instructions in bundles) on a VLIW processor is usually performed by a postpass scheduler.

Certified scheduling was already explored by Tristan and Leroy [2008], where they extend CompCert with a certified postpass list-scheduler. They introduced a two-tier architecture for this purpose: an untrusted oracle written in OCaml computes a scheduling for each *basic-block*[5] in order to minimize pipeline stalls, and a checker—certified in Coq—verifies the results of this untrusted oracle. However, two problems were identified with this approach.

Firstly, the scheduling checker of Tristan and Leroy [2008] has an exponential complexity w.r.t. the size of basic-blocks, which makes it impractical. This is especially the case for certain kinds of programs, such as unrolled computational loops or some forms of cryptography, with many instructions within a basic-block (up to the order of a thousand instructions). We thus needed to devise new algorithms that scale much better but that can still be proved correct in Coq.

Secondly, the scheduling of Tristan and Leroy [2008] operates at the Mach level, which is simpler than the Asm level (fewer instructions). However, some Mach instructions, such as conditional jumps, actually correspond to several assembly instructions. It is thus hard to provide a precise latency model in that representation. Furthermore, we need the actual machine instructions to construct well-formed bundles only. We need to base our scheduling directly in the Asm language.

### 1.4 Contributions

Our main contribution is a *certified* and *highly modular* scheduler with bundling. Our scheduler combines an untrusted scheduling oracle with a verified scheduling checker. Both the oracle and checker are highly *generic*; we instantiated them with the instruction set, architecture and micro-architectural details of the Kalray VLIW core.

Our solution is inspired by that of Tristan and Leroy [2008], but solves the two issues mentioned above. Hence, our certified scheduler is also based on a two-tier architecture:

- An oracle, written in OCaml, producing a sequence of bundles for each basic-block. We provide four different implementations:
  (1) a dummy one that puts one instruction per bundle;
  (2) a naive greedy one that packs instructions into bundles without reordering them;
  (3) a default one: a greedy list-scheduler with a priority heuristic based on latencies;
  (4) an optimal one based on integer linear programming through an external ILP solver.
- A generic certified scheduling checker, written in Coq, with a proof of semantic preservation, consisting of two independent checks:
  – Verifying that, assuming sequential execution within each bundle, the reordered basic-block preserves the sequential semantics of the original one. This is achieved by comparing the symbolic execution of two basic-blocks, as did Tristan and Leroy. The exponential complexity of their approach is avoided by introducing (verified) hash-consing.

[5] A *basic-block* is defined as a sequence of instructions with a single entry point (possibly named by a label in front of the sequence) and a single exit point (e.g. a control-flow instruction at the end of the sequence).

– Verifying that, for each bundle, the sequential execution and the parallel execution have the same semantics. This simply reduces to check that each bundle never uses a register after writing to it (no Use-After-Write).[6]

These checks are performed on a new IR, called AbstractBasicBlock, which makes them easier to implement and prove, and which is moreover generic w.r.t the instruction set.
Our implementation is modular in three respects:

(1) the core of our certified scheduling scheduler is independent from the instruction set: it can be reused for other processors, or other IRs (e.g. in pre-pass scheduling);
(2) our scheduler fits within the modular backend of CompCert, and is largely independent from the rest of it; thus all improvements to the rest of the backend (e.g. better instruction selection) are carried over without the need to modify the scheduler or checker (except, of course, adding descriptions for new instructions being used);
(3) the certification process is independent from the untrusted scheduling oracle, and thus new oracles may be used without any change.

Using our version of CompCert for the Kalray VLIW, including our scheduler, we compiled a variety of software and compared their execution time, using a cycle-accurate simulator,[7] to the execution time of the same software compiled with the reference compiler for that platform (versions of the GNU C Compiler supplied by the chip designers).

### 1.5 Related work

Our CompCert backend for the Kalray K1c processor has initially benefited from that of Barany [2018] for the Kalray K1b processor, even if Barany's backend generates only one instruction per bundle—without instruction scheduling— and does not even model the VLIW semantics of the processor. Actually, Barany was faced to the challenge of representing in CompCert "superregisters"— that merge a pair of 32-bit "subregisters"— and which are mandatory for handling 64-bit floating-point values on the K1b. Fortunately for us, this constraint has disappeared on the K1c.

Patmos is another VLIW processor for which CompCert is currently being ported. For now, this backend only generate one instruction per bundle [Jacobsen 2019].

Scheduling in the presence of timing and resource constraints is a classical problem; [Micheli 1994, §5.4]. Ample work is available on scheduling for VLIW processors [Dupont de Dinechin 2004]. However, classically, there is no machine-checked proof of correctness of compiler implementation.

Tristan and Leroy [2008]; Tristan [2009]; Tristan and Leroy [2010] studied more advanced scheduling techniques, including software pipelining, which are particularly well-suited to pre-pass optimization. We plan to consider these in our future works.

Christian Schulte and collaborators have applied constraint programming to instruction selection, register allocation, code motion and other optimizations [Blindell et al. 2017; Castañeda Lozano et al. 2019]. Their process can even be optimal (with respect to their cost model) on medium-sized functions. They consider a wider class of optimizations than we do, but they provide no machine-verified proof of correctness; their approach has limited, albeit considerable scalability.[8]

[6] Hence, our backend will never emit a swapping bundle such as "$R_1 := R_2; R_2 := R_1$".
[7] At the time of submission, the Kalray MPPA3 processor had not been sampled yet, so all experiments were run on the simulator of FPGA emulator supplied by the chip designers.
[8] The primary goal of their approach is to identify weak points in production compilers. The machine code programs generated by their experimental compiler based on constraint solving and the production compiler are compared; if the production compiler is clearly sub-optimal, compiler designers investigate and devise an optimization pattern that may eventually be integrated into the production compiler.

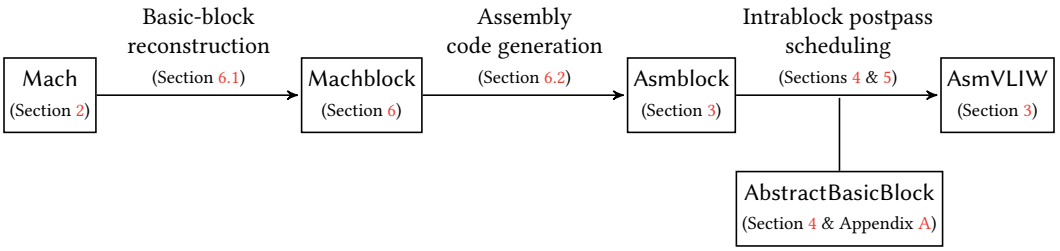



Fig. 3. Architecture of our solution in CompCert

### 1.6 Architecture of our solution (and of this paper)

Our ultimate goal is to generate efficient assembly code for our VLIW architecture: the AsmVLIW language is our final representation. This has the assembly semantics of our VLIW architecture, with parallel execution within each bundle. This parallel semantics relies on executing atomically basic-blocks of instructions—the bundles are treated as basic-blocks with a parallel execution inside.

Our postpass scheduling pass is formalized as a transformation on basic-blocs. It takes as input our IR called Asmblock, which shares its syntax with AsmVLIW, but with a sequential execution inside basic-blocks instead of a parallel one. These two central languages of our approach are described in Section 3. Our postpass scheduling itself is described in Section 4. Before that, Section 2 recalls the necessary details of CompCert. Finally, Section 7 presents our experimental evaluations of our backend. Sections 5 and 6 give more details on our backend: we summarized them below.

We extended the CompCert architecture with the passes shown in Figure 3. The preliminary stage of our backend constructs the Asmblock program from the Mach program. This stage is actually itself composed of two passes described in Section 6. As Section 6 explains, the basic-block structure cannot be recovered from the usual Asm languages of CompCert. Thus, we recover it from Mach, through a new IR—called Machblock—which provides a syntax reflecting the basic-block structure of Mach programs. Then, our postpass scheduling from Asmblock to AsmVLIW takes each block from Asmblock, performs intra-block scheduling via an external untrusted oracle, and uses a certified checker to verify the generated AsmVLIW bundles. The architecture of this pass and its verification are given in Section 4, while those on the actual intra-block scheduling problem solved by our external oracle are given in Section 5. The core of our scheduling checker—involving symbolic evaluation of basic-blocks with hash-consing—is achieved at the level of a new auxiliary IR, called AbstractBasicBlock, presented in Section 4, but further detailed in Appendix A.[9]

## 2 USUAL PASSES FROM MACH TO ASM IN COMPCERT

Our work replaces the usual passes between the Mach and Asm IRs of CompCert (there is one such pass for each target processor). These two IRs are presented in Section 2.2 and Section 2.3, respectively. Before that, Section 2.1 presents a general notion of correctness used in CompCert passes, and in particular for usual passes from Mach to Asm.

### 2.1 Correctness of Compilation Passes through Forward Simulations

In CompCert [Leroy 2009b], the semantics of a program $P$—in a given language where execution states inhabit a given type—consists of:

[9]Submitted as an anonymous appendix of this paper.

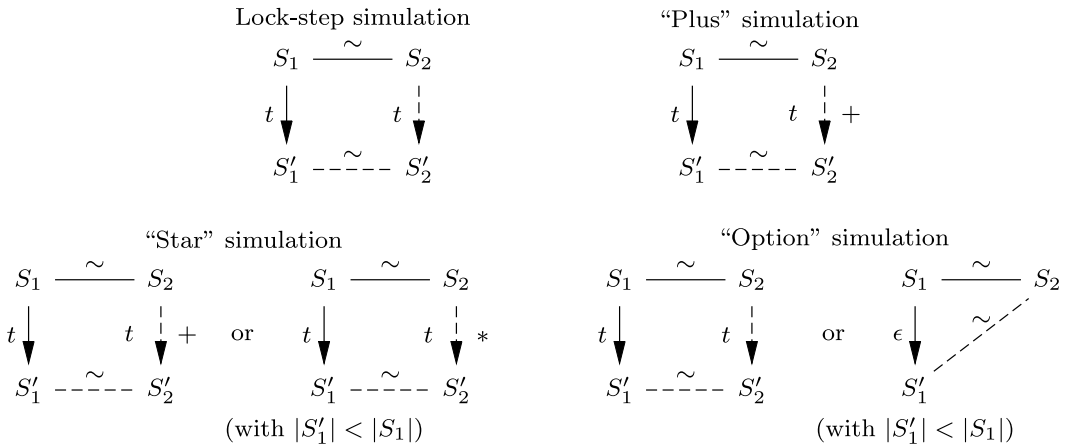



Fig. 4. Various simulation diagrams used by CompCert—copied from [Leroy 2009a,b].

- a predicate $S \xrightarrow{t} S$ indicating if *one* execution step *can* run from state $S$ to state $S'$ by generating a *trace* $t$—where such a $t$ is either a single *observable event* (e.g. an external call or an access to a volatile variable) or $\epsilon$, i.e. an internal computation step without observable event;
- a predicate ($P$.`initial_state` $S$) indicating that state $S$ is an initial state;
- a predicate ($P$.`final_state` $S$ $r$) indicating that state $S$ is a returning state (of the main function) with value $r$.

The formal correctness property of CompCert expresses that, given a source program $P_1$ *without undefined behavior* (i.e. that can always run a step from a non-final state), if the compilation of $P_1$ produces some assembly program $P_2$, then *observational behaviors* of $P_2$ are *observable behaviors* of $P_1$. Hence, this property involves a high-level notion of *observable behaviors*, formalized by Leroy [2009a,b]. In order to simplify correctness proofs of its successive passes (recalled in Figure 2), CompCert uses an alternative definition for the correctness. One of them is the *forward simulation* applicable on passes between *deterministic* languages—like all IRs of the backend.

The *forward simulation* of a program $P_1$ by a program $P_2$ is defined—for a relation $S_1 \sim S_2$ matching states $S_1$ from $P_1$ with states $S_2$ from $P_2$—as the conjunction of the following conditions:

- The initial states match
$$\forall S_1,\ P_1.\texttt{initial\_state}\ S_1 \implies \exists S_2,\ P_2.\texttt{initial\_state}\ S_2 \wedge S_1 \sim S_2$$
- The final states match
$$\forall S_1\ S_2\ r,\ S_1 \sim S_2 \wedge P_1.\texttt{final\_state}\ S_1\ r \implies P_2.\texttt{final\_state}\ S_2\ r$$
- The execution steps match through the "Star" simulation diagram (also depicted in Figure 4)
$$S_1 \sim S_2 \wedge S_1 \xrightarrow{t} S'_1 \implies \exists S'_2,\ S'_1 \sim S'_2 \wedge (S_2 \xrightarrow{t}{}^{+} S'_2 \ \vee\ S_2 \xrightarrow{t}{}^{*} S'_2 \wedge |S'_1| < |S_1|)$$

The "Star" simulation diagram expresses that each single step of $P_1$ producing a trace $t$ can be simulated by several steps of $P_2$ producing the same trace $t$. In particular, when $P_1$ performs an internal step (where $t = \epsilon$), $P_2$ can *stutter*, i.e. perform no computation step. But, if $P_1$ loops for ever without producing any observable event, then $P_2$ cannot stutter infinitely.[10] Indeed, stuttering is only allowed if the step of $P_1$ makes the state decrease for a well-founded order (hence, sequences of *successive* stutterings cannot be infinite). Actually, CompCert provides a collection of specialized diagrams of this simulation (see Figure 4): these make the proof simpler in specific cases.

[10] Otherwise an infinite silent loop $P_1$ could be compiled into a program $P_2$ returning in one step, and this would be incorrect.

### 2.2 Mach

The Mach IR consists of a set of functions, each with a function body. Roughly speaking, Mach is a simplified assembly language (with 3-address code instructions handling the actual registers of the target processor, except for some special registers like the program counter PC) where:

- The ABI (Application Binary Interface) is abstracted into Mach instructions allowing access to the stack and function parameters `Mgetstack`, `Msetstack` and `Mgetparam`.
- Loads and stores stay generic, and are not yet expanded into the "real" load/store instructions. The same applies to branching instructions.
- Calling instructions `Mcall` and `Mtailcall` can only branch either on a function symbol, or a register on an address that must be the first address of a function.
- Branching instructions such as `Mgoto` branch to labels in the current function (like in LLVM).
- There is neither a PC (Program Counter) nor a RA (Return Address) register. The remaining code to execute is an explicit part of the current state.

Mach states describe the register state `rs`, the global memory state `m`, and the stack state `st`. They are of three kinds, with the following meanings:

- `(State st f c rs m)`: the first instruction of code `c` is about to be run in current function `f`;
- `(Callstate st f rs m)`: function `f` is about to be run, the caller context has just been pushed on stack `st`;
- `(Returnstate st rs m)`: a caller context is about to be restored from stack `st`.

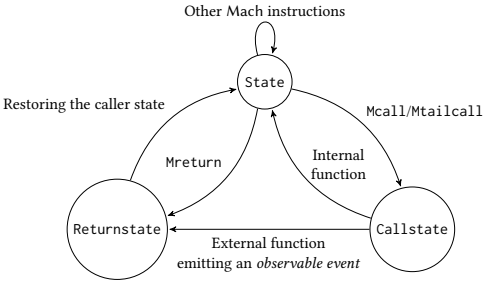



Fig. 5. Execution steps between Mach states

### 2.3 Asm

CompCert defines one Asm language per target processor. As in Mach, an Asm program consists of functions, each with a function body. Unlike in Mach, Asm states are only of a single kind "`State(rs, m)`" where `rs` is the register state (a mapping from register names to values), and `m` is the memory state (a mapping from addresses to values). An initial state is one where the PC register points to the first instruction of the `main` function, and the memory is initialized with the program code. The instructions of Asm are those of the target processor, each one with its associated semantics that specifies how the instruction modifies the registers and the memory.

The major difference between Mach and Asm really lies in the execution semantics. In Mach, the remaining code to execute is directly in the state and Mach semantics provide a clean notion of *internal function call*: execution can only enter into an internal function by its syntactic entry-point. In Asm, the code to execute resides in memory, and is pointed to by the PC register. Through jumps into registers and a bit of pointer arithmetic, an Asm program can jump into the middle of a function, like in Return-Oriented-Programming (ROP) [Buchanan et al. 2008]. Thus, proving the code generation pass from Mach to Asm implies ensuring that the generated code does not have such a behavior (assuming that the Mach program does not have any undefined behavior): it simulates Mach execution where such a behavior does not exist.[11]

Formally, it involves introducing a suitable "∼" relation matching Mach states with Asm states. The gist of it consists in expressing a correspondence between the register states as well as the memory, in addition to the following properties depending on the Mach state: if it is a `State`, then the PC register points to the Asm code generated from Mach; if it is a `Callstate`, then the

[11] Thus, ROP attacks on code generated by CompCert are only possible from undefined behaviors of the source code.

PC should point to the callee function, and the RA register to the return address (i.e. the address following the call instruction in the caller); otherwise, it is a `Returnstate` and the PC should point to the return address.

A "Star" simulation is then used, proving that one Mach step (i.e. one transition of Figure 5) gives the same result as several Asm instructions. For instance, the Mach step from `Callstate` into `State` is simulated by the steps of the Asm function prologue that allocate the stack-frame and save it into registers FP (Frame Pointer) and RA. The Mach conditional branching step is simulated by the Asm steps that compute the result of the condition, and then branch accordingly. Actually, the only stuttering step of Asm w.r.t Mach corresponds to the *Restoring* step from `Returnstate`.

## 3 A FORMAL BLOCKSTEP SEMANTICS FOR A VLIW ASSEMBLY LANGUAGE

The Asm language of our target processor introduces a syntax and semantics for *bundles* of instructions. This is very different from the existing Asm languages of CompCert (which are purely sequential). A bundle is a list of instructions that is ultimately considered for a parallel (VLIW) semantics, but that also allows for a sequential semantics in an intermediate step of the compiler. Hence, for the sequential semantics, a bundle is just a special case of *basic-block*: zero or more labels giving (equivalent) names to the *entry-point* of the block; followed by zero or more *basic instructions* – i.e. instructions that do not branch, such as arithmetic instructions or load/store; and ended with at most one *control flow instruction*, like conditional branching on a label.

Semantically, basic-blocks have a single entry-point and a single exit-point: branching from/to the middle of a basic-block is impossible. It is thus possible to define a semantics that steps through each block atomically, sequentially executing the program block by block. We call such a semantics a *blockstep semantics*. The notion of basic-block is thus relevant to scheduling optimizations: reordering the sequence of *basic instructions* in a basic-block without changing its (local) block-step, does not change the (global) semantics of the surrounding program either.

Hence, from the same assembly syntax, based on basic-block structures, we define two blockstep semantics: AsmVLIW with a parallel semantics inside blocks (our target language) and Asmblock with a sequential semantics inside blocks. During our compilation process, the Asmblock program is obtained by detecting the basic-block structure of the Mach program (as described in Section 6). Then, for each basic-block, our scheduling oracle reorders the list of its basic instructions, before splitting the reordered basic-block into a sequence of smaller basic-blocks (as described in Sections 4 & 5). Each of these small basic-blocks corresponds to a bundle in the final program.

In order to avoid information redundancies, AsmVLIW and Asmblock blockstep semantics share a common part: the semantics of single instructions. Indeed, the only difference between Asmblock and AsmVLIW semantics lies in how they combine instructions within basic-blocks. Below, Section 3.1 defines the syntax shared between AsmVLIW and Asmblock. Then, Section 3.2 defines AsmVLIW, and Section 3.3 defines Asmblock.

### 3.1 Syntax of bundles/basic-blocks

To define the syntax of basic-blocks, we first split the instructions into two syntactic categories: the basic instructions and the control flow instructions. Each of these categories is in turn divided into subcategories. Then, a basic-block (or a bundle) is syntactically defined as a record of type `bblock` with three fields: a list of labels, a list of basic instructions, and an optional control flow instruction.

```
Inductive basic: Type :=   (* basic instructions *)
Inductive control: Type := (* control-flow instructions *)
Record bblock := {
  header: list label;  body: list basic;  exit: option control;
  correct: wf_bblock body exit  (* must contain at least 1 instr. *)
}
```

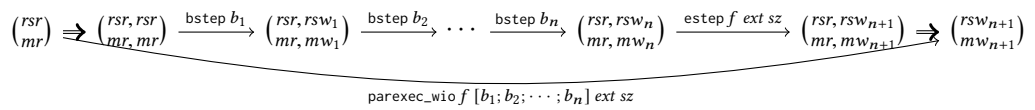



Fig. 6. Parallel in-order step from state $(rsr, mr)$ to state $(rsw_{n+1}, mw_{n+1})$

.

In our AsmVLIW and Asmblock semantics, on a `None` exit, the PC is incremented by the amount of instructions in the block. This convention makes reasoning easier when splitting a big basic-block into a sequence of smaller ones. In order to avoid infinite stuttering (due to incrementing the PC by 0), we have a further property that a block should contain at least one instruction.

Sections 3.2 and 3.3 define, respectively, the parallel and the sequential blockstep semantics of this syntax of basic-blocks. The state in AsmVLIW and Asmblock is expressed the same way as in Asm: it is a pair $(rs, m)$ where $rs$ (register state) maps registers to values, and $m$ (memory) maps addresses to values. As in Asm, we represent the failure (e.g., division by zero) of a single (basic or control-flow) instruction by a special value called `Stuck`. Hence, executing a single instruction in our semantics gives an `outcome` defined as a next state, or a stuck execution by "`Inductive outcome := Next (rs:regset) (m:mem) | Stuck`". Then, for both blockstep semantics, each block-step takes as input an initial state $(rs, m)$, fetches the block pointed by $rs$[PC] (the value of PC in $rs$), and executes the content of this block. Finally, that block-step either returns the next state $(rs', m')$—resulting from the parallel/sequential composition of single instructions—or propagates any encountered failure.

### 3.2 Parallel semantics of AsmVLIW

A bundle is a group of instructions that are to be *issued* in the same cycle through the pipeline. Each instruction goes through the following main stages (we ignore some stages irrelevant to the functional semantics):

- Reading stage: the contents of the registers are fetched
- Computing stages: the output values are computed, which can take several cycles. Once an output is computed, it is *available* to other bundles waiting at a reading stage.
- Writing stage: the computed results are written to the registers.

Some instructions can bypass their results through the pipeline directly to the reading stage, without having to wait for the writing stage. Our processor stalls at a reading stage whenever the result is not yet available. The exact amount of cycles required to compute a value thus only has an impact on the performance: in our formal semantics, we ignore the computing stages and only keep the reading and writing stages.

Reads are always deterministic on our processor: they happen at the start of the execution of our bundle. However, the write order is not necessarily deterministic, if the same register is written twice in the same bundle. Defining directly a non-deterministic semantics is rather complex. Thus, we first introduce a deterministic semantics where the writes are performed in the order in which they appear in the bundle. For instance, in our in-order semantics, the bundle "$R_0 := 1; R_0 := 2$" assigns 2 to $R_0$. The non-deterministic semantics is then defined by allowing the execution to apply an arbitrary permutation on the bundle, before applying the in-order semantics.

*3.2.1 In-order parallel semantics.* We model the reading stage by introducing an internal state containing a copy of the initial state (prior to executing the bundle). Such an internal state is thus

of the form (*rsr*, *rsw*, *mr*, *mw*) where (*rsr*, *mr*) is the copy of the initial state, and (*rsw*, *mw*) is the running state where the values are written. Figure 6 schematizes the semantics:

- The function (`bstep` *b rsr rsw mr mw*) executes the basic instruction *b*, fetching the values from *rsr* and *mr*, and performing the writes on *rsw* and *mw* to give an `outcome`.
- The function (`estep` *f ext sz rsr rsw mw*) does the same with the optional control flow instruction *ext*. If there is no instruction, then it just increments PC by *sz*, the size of the block. Here, *f* is the current function—in which branching instructions look for labels, like in other Asm semantics of CompCert.
- The function (`parexec_wio` ... *rsr mr*) is the composition of the basic and control steps.

For example, executing the bundle "$R_0 := R_1; R_1 := R_0;$ `jump @toto`" with the initial register states "$rsw_0 = rsr = rs[R_0 \leftarrow r_0; R_1 \leftarrow r_1]$" leads to the following internal states:

(1) "$R_0 := R_1$" leads to $rsw_1 = rsw_0[R_0 \leftarrow rsr[R_1]] = rs[R_0 \leftarrow r_1; R_1 \leftarrow r_1]$
(2) "$R_1 := R_0$" leads to $rsw_2 = rsw_1[R_1 \leftarrow rsr[R_0]] = rs[R_0 \leftarrow r_1; R_1 \leftarrow r_0]$
(3) Finally, "`jump @toto`" leads to
$$rs' = rsw_3 = rsw_2[\mathsf{PC} \leftarrow \texttt{@toto}] = rs[R_0 \leftarrow r_1; R_1 \leftarrow r_0; \mathsf{PC} \leftarrow \texttt{@toto}]$$

Hence, parallel execution of this bundle swaps the content of registers $R_0$ and $R_1$ as expected.

Let "`NEXT` *rs*`,`*m* $\leftarrow e_1$ `IN` $e_2$" be a notation for "`match` $e_1$ `with Next` *rs m* $\Rightarrow e_2$ | _ $\Rightarrow$ `Stuck end`". The in-order parallel execution of a list of basic instructions is formally defined in Coq by this function:

```
Fixpoint parexec_wio_body (bdy:list basic) rsr rsw mr mw : outcome :=
  match bdy with
  | nil ⇒ Next rsw mw
  | bi::bdy' ⇒ NEXT rsw', mw' ← bstep bi rsr rsw mr mw IN
                parexec_wio_body bdy' rsr rsw' mr mw'
  end
```

The in-order parallel execution of a block—defined by `parexec_wio` below—first performs a parallel in-order execution on the body (the list of basic instructions), and then performs a parallel execution with the optional control flow instruction. Here, `f` is the current function and `size_b` is the offset by which PC is incremented in absence of a control-flow instruction.

```
Definition parexec_wio (f:function) (bdy:list basic) (ext:option control) (sz:ptrofs) rs m :=
  NEXT rsw', mw' ← parexec_wio_body bdy rs rs m m IN
  estep f ext sz rs rsw' mw'
```

*3.2.2 Deterministic out-of-order parallel semantics.* This in-order parallel semantics is not very representative of how a VLIW processor works, since concurrent writes may actually happen in any order. We model this non-deterministic writes order by the relation (`parexec_bblock` *f b rs m o*). This means that there exists a permutation of instructions such that the in-order parallel execution of block *b* with initial state (*rs*, *m*) gives the outcome *o*.

```
Definition parexec_bblock (f:function) (b:bblock) rs m o: Prop :=
∃ bdy1 bdy2, Permutation (bdy1 ++ bdy2) b.(body)
 ∧ o=(NEXT rsw', mw' ← parexec_wio f bdy1 b.(exit) (Ptrofs.repr (size b)) rs m IN
      parexec_wio_body bdy2 rs rsw' m mw')
```

Formally, execution takes any permutation of the body and splits this permutation into two parts `bdy1` and `bdy2`. Then, it runs in this order: `bdy1`, then the control flow instruction, then `bdy2`. Importantly, while PC is possibly written before the end of the execution of the bundle, the effect on the control-flow takes place only at the end of blockstep execution.

This gives a fair abstraction of the actual VLIW processor. However, for the target applications of CompCert (critical embedded systems), it is very desirable that the emitted program be deterministic: this is thus a prerequisite of all CompCert backends. Consequently, we force our backend to emit bundles that have the same semantics irrespective of the order of writes. This is formalized by:

```
Definition det_parexec f b rs m rs' m': Prop :=
  ∀ o, parexec_bblock f b rs m o → o = Next rs' m'
```

Given a supposed next state $(rs', m')$, the property (`det_parexec` $f$ $b$ $rs$ $m$ $rs'$ $m'$) holds only if all the possible outcomes $o$ satisfying (`parexec_bblock` $f$ $b$ $rs$ $m$ $o$) turn out to be exactly (`Next` $rs'$ $m'$). In other words, it only holds if $(rs', m')$ is the only possible outcome. We then use the `det_parexec` relation to express the step relation of our AsmVLIW semantics: if `det_parexec` does not hold then no step is possible (execution is stuck).

### 3.3 Sequential semantics in Asmblock

Asmblock is the IR just before AsmVLIW: instructions are grouped in basic-blocks, where each basic-block is atomically executed. However these are not re-ordered and split into bundles yet, and execution inside a block is sequential.

The sequential semantics of a basic-block, called `exec_bblock` below, is similar to the semantics of a single instruction in other Asm of CompCert. Just like AsmVLIW, its execution first runs the body and then runs the control flow. Our sequential semantics of single instructions simply reuses `bstep` and `estep` by using the same state for reads and for writes. Our semantics of single instructions is thus shared between the sequential Asmblock and the parallel AsmVLIW.

```
Fixpoint exec_body bdy rs m: outcome :=
  match body with
  | nil ⇒ Next rs m
  | bi::bdy' ⇒ NEXT rs' m' ← bstep bi rs rs m m IN exec_body bdy' rs' m'
  end
Definition exec_bblock f b rs m: outcome :=
  NEXT rs' m' ← exec_body b.(body) rs m IN estep f b.(exit) (Ptrofs.repr (size b)) rs' rs' m'
```

## 4 CERTIFIED INTRABLOCK POSTPASS SCHEDULING

Our postpass scheduling takes place during the pass from Asmblock to AsmVLIW (see Figure 3). This pass has two goals. First, re-order the instructions in each basic-block to minimize the stalls. Second, group in bundles the instructions that can be executed in the same cycle. Similarly to Tristan and Leroy [2008], our scheduling is computed by an untrusted oracle that produces a result which is checked by Coq-proved verifiers. A major benefit of this design is the ability to change or tune the untrusted oracle without modifying our Coq proofs.

Scheduling is performed block by block from the Asmblock program. As depicted in Figure 7, it generates a list `lb` of AsmVLIW bundles from each basic-block `B`. More precisely, a basic-block `B` from Asmblock enters the PostpassScheduling module. This module sends `B` to an external untrusted scheduler, which returns a list of bundles `lb`, candidates to be added to the AsmVLIW program (scheduling is detailed Section 5). The PostpassScheduling module then checks that `B` and `lb` are indeed semantically equivalent through dedicated verifiers. Then, PostpassScheduling either adds `lb` to the AsmVLIW program, or just stops the compilation if the verifier returned an error.

In Coq, the scheduler is declared as a function[12] splitting a basic-block "`B:bblock`" into a value that is then transformed into a sequence of bundles "`lb: list bblock`".[13]

[12]The scheduler is declared as a pure function like other CompCert oracles.

[13]It would be unsound to declare `schedule` returning directly a value of type "`list bblock`", since the "`correct`" proof field of `bblock` does not exist for the OCaml oracle.

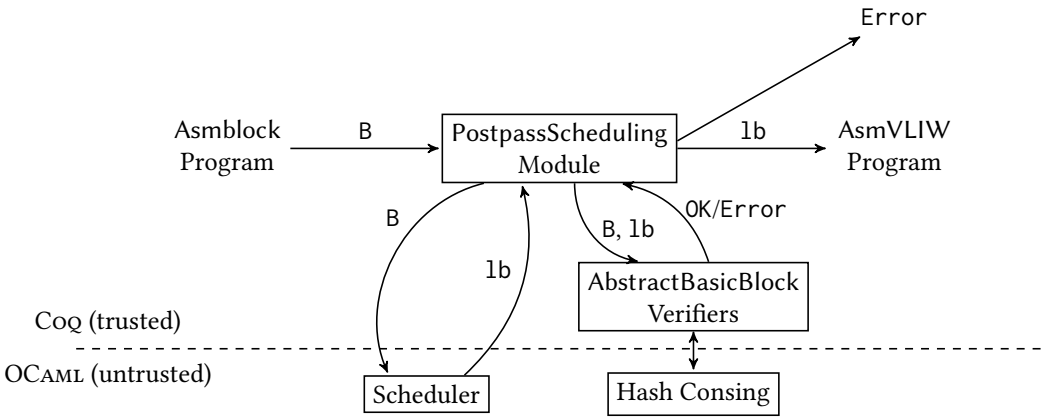



Fig. 7. Interactions between the untrusted oracle and the verification module

```
Axiom schedule: bblock → (list (list basic)) * option control
```

The proof of the pass uses a *"Plus" simulation* (see Figure 4): one step of the initial basic-block `B` in the sequential Asmblock semantics is simulated by stepping sequentially all bundles of `lb` for the parallel AsmVLIW semantics. This forward simulation results from composition of these two ones:

(1) A *plus simulation* ensuring that executing `B` is the same as executing `lb` in the sequential Asmblock semantics, basically proving the re-ordering part of the postpass scheduling.
(2) A *lockstep simulation* ensuring that executing each bundle of `lb` with the Asmblock semantics gives the same result as executing this bundle with the parallel AsmVLIW semantics.

Each of these two forward simulations is actually derived from the correctness property of a dedicated verifier. In other words, we prove that if each of these two verifiers returns "`OK`", then the corresponding forward simulation holds. The following sections describe these two verifiers and their correctness proof. Actually, we start with the simplest one: Section 4.1 describes the "parallelizability checker" ensuring forward simulation (2). Further sections describe the "simulation checker" ensuring forward simulation (1).

### 4.1 Parallelizability Checker through AbstractBasicBlock

We define below the `bblock_para_check` function checking that each generated bundle can be run in the parallel semantics without any change w.r.t the sequential semantics. Its correctness property, also given below, is sufficient to prove the above forward simulation (2).

```
Definition bblock_para_check (bundle: Asmvliw.bblock): bool :=
  is_parallelizable (trans_block bundle)

Lemma bblock_para_check_correct ge f bundle rs m rs' m':
  bblock_para_check bundle = true →
  exec_bblock ge f bundle rs m = Next rs' m' → det_parexec ge f bundle rs m rs' m'
```

Function `bblock_para_check` first translates—through function `trans_block`—the bundle into another IR called AbstractBasicBlock. The core of the checker—called `is_parallelizable`—operates on this IR.

Indeed, AbstractBasicBlock provides a simplified syntax, where registers read or assigned by each instruction appear syntactically. For this IR, the whole memory is itself accessed through a pseudo-register written here as *m*. See Example 4.1 below.

*Example 4.1 (Syntax of AbstractBasicBlock).* We illustrate below how we have translated some instructions of the K1c processor into AbstractBasicBlock assignments:

(1) the addition of two registers $r_2$ and $r_3$ into $r_1$ is written "$r_1 := \mathsf{add}[r_2, r_3]$";
(2) the load of memory address $16(r_2)$ into register $r_1$ is written "$r_1 := (\mathtt{load}\ 16)[m, r_2]$";
(3) the store of register $r_1$ into memory address $imm(r_2)$ is written "$m := (\mathtt{store}\ imm)[m, r_1, r_2]$".

In AbstractBasicBlock, a block (or a bundle) is simply a list of assignments (control-flow instructions are not distinguished from others: they simply assign a pseudo-register called PC). Like Asmblock/AsmVLIW, AbstractBasicBlock provides a sequential and parallel semantics for blocks. Unlike them, it does not provide any semantics for sequences of blocks. Hence, AbstractBasicBlock is dedicated to intrablock analyses. As explained in Section 4.2, we also use it for the simulation test. Actually, the syntax of AbstractBasicBlock and its semantics are generic, in the sense that the IR is parametrized by the names of pseudo-registers and the syntax and semantics of operators. Thus, as claimed in the introduction, it could be easily reused for other processors or other IRs of CompCert.

Finally, `is_parallelizable` simply analyzes the sequence of AbstractBasicBlock assignments and checks that no pseudo-register is read or re-written after being written once. For example, "$r_1 := r_2$; $r_3 := r_2$" and "$r_1 := r_2$; $r_2 := r_3$" are accepted as parallelizable. However, "$r_1 := r_2$; $r_2 := r_1$" and "$r_1 := r_2$; $r_1 := r_3$" are rejected, because $r_1$ is used after being written. See details in Appendix A.

When `is_parallelizable` returns true, the list of assignments has the same behavior in both the sequential and the parallel semantics. This property at the AbstractBasicBlock level can be lifted back to the AsmVLIW/Asmblock level, because the list of assignments returned by `trans_block` *bisimulates* the input block—both for the sequential and the parallel semantics. This bisimulation for the sequential semantics is also useful for the simulation checker described below.

### 4.2 Verifying Intrablock Reordering through a Generic Checker

To reason on executions of a bundle sequence, we define a predicate "`(is_concat tb lb)`" meaning that "`tb:bblock`" is the *concatenation* of "`lb:list bblock`". Formally, this means that `lb` is a non-empty list such that only its head $h$ may have a non-empty header, such that only its tail block $t$ may have some control-flow, and such that `tb.(header)`$=h$`.(header)` and `tb.(exit)`$=t$`.(exit)` and `tb.(body)` is the concatenation of all `lb` bodies.

We also define a *block simulation*: a block `b` is *simulated* by a block `b'` *if and only if* when the execution of `b` is not `Stuck`, then executing `b` and `b'` from the same initial state gives the same result (using the sequential semantics). That is, `b'` preserves any non-`Stuck` outcome of `b`.

```
Definition bblock_simu ge f b b' :=
  ∀ rs m, exec_bblock ge f b rs m <> Stuck → exec_bblock ge f b rs m = exec_bblock ge f b' rs m
```

With these definitions, the forward simulation (1) reduces to proving the following correctness property of our `verified_schedule` pass on each basic-block:

```
Theorem verified_schedule_correct: ∀ ge f B lb,
  (verified_schedule B) = (OK lb) → ∃ tb, is_concat tb lb ∧ bblock_simu ge f B tb
```

First, `(verified_schedule B)` calls `(schedule B)` and then builds the sequence of bundles `lb` and their concatenation `tb`. Finally, it calls a function `bblock_simub` (detailed below) checking that `tb` simulates `B`. Its correctness theorem states that if it returns `(OK lb)`, then `tb` simulates `B`.[14]

The core of the verification is thus achieved by `bblock_simub`, which is largely inspired by the list-scheduling verifier of Tristan and Leroy [2008], but with three major differences. First,

[14]With our current scheduler, `tb` is actually semantically equivalent to `B`. Future versions may be able to eliminate some avoidable memory accesses, and hence not necessarily preserve all `Stuck` outcomes.

they define their verifier for the Mach IR, while ours is defined at a lower-level language, where scheduling is more accurate. Second, we have introduced a dedicated IR, called AbstractBasicBlock, which provides a generic list-scheduling verifier, parametrized by the targeted language. Third, our simulation test is expected to be linear-time thanks to a (dynamically) verified hash-consing procedure, while theirs is exponential-time in the worst case.

Function `bblock_simub` is in turn composed of two steps. First, each basic-block is compiled—through the `trans_block` function presented in Section 4.1—into a sequence of AbstractBasicBlock assignments.[15] Second, like in [Tristan and Leroy 2008], the simulation test symbolically executes each AbstractBasicBlock code and compares the resulting *symbolic memories* (see Figure 8). Roughly speaking, such a symbolic memory[16] corresponds to a kind of parallel assignment that is equivalent to the input block. More precisely, this symbolic execution computes a term for each pseudo-register assigned by the block: this term represents the final value of the pseudo-register in function of the initial values of the pseudo-registers.

*Example 4.2 (Equivalence of symbolic memories).* Let us consider the two blocks below:
$r_1 \coloneqq r_1 + r_2;\ r_3 \coloneqq \texttt{load}[r_2, m];\ r_1 \coloneqq r_1 + r_3 \qquad r_3 \coloneqq \texttt{load}[r_2, m];\ r_1 \coloneqq r_1 + r_2;\ r_1 \coloneqq r_1 + r_3$
These two blocks are both equivalent to the following parallel assignment:

$$r_1 \coloneqq (r_1 + r_2) + \texttt{load}[r_2, m] \parallel r_3 \coloneqq \texttt{load}[r_2, m]$$

Indeed, these two blocks simulate each other (they bisimulate).

Collecting only the final term associated with each pseudo-register is actually not correct: an incorrect scheduling oracle could insert additional failures. The symbolic memory must thus also collect a list of all intermediate terms on which the sequential execution may fail and that have disappeared from the final parallel assignment. See Example 4.3 below. Formally, the symbolic memory and the input block must be bisimulable (see Figure 8).

*Example 4.3 (Simulation on symbolic memories).* Let us consider the two bblocks $p_1$ and $p_2$ below:
$r_1 \coloneqq r_1 + r_2;\ r_3 \coloneqq \texttt{load}[r_2, m];\ r_3 \coloneqq r_1;\ r_1 \coloneqq r_1 + r_3 \quad (p_1) \qquad r_3 \coloneqq r_1 + r_2;\ r_1 \coloneqq r_3 + r_3 \quad (p_2)$
Again, $p_1$ and $p_2$ lead to the same parallel assignment:

$$r_1 \coloneqq (r_1 + r_2) + (r_1 + r_2) \parallel r_3 \coloneqq r_1 + r_2$$

However, $p_1$ is simulated by $p_2$ whereas the converse is not true. This is because the "useless" memory access in $p_1$ may cause its execution to fail, whereas this failure cannot occur in $p_2$. Thus, the symbolic memory of $p_1$ should contain the term "$\texttt{load}[r_2, m]$" as a potential failure. Finally, we say that *a symbolic memory $d_1$ is simulated by a symbolic $d_2$* if and only if their parallel assignment are equivalent, and the list of potential failures of $d_2$ is included in the list of potential failures of $d_1$. See Appendix A for the formal definitions.

As illustrated in Examples 4.2 and 4.3, computation of symbolic memories involves many duplications of terms. Thus, comparing symbolic memories with structural equalities of terms, as performed in [Tristan and Leroy 2008], is exponential-time in the worst case. In order to solve this issue, we have developed a generic verified hash-consing factory for Coq. Hash-consing consists in memoizing the constructors of some inductive data-type in order to ensure that two structurally equal terms are actually allocated to the same object in memory. This enables us to replace (expensive) structural equalities by (constant-time) pointer equalities.

The details on AbstractBasicBlock and its hash-consing mechanism are described in Appendix A. Below, we simply give a brief overview of the hash-consing mechanism.

[15] Let us remark that the invocation of `trans_block` is, however, not the same here as in Section 4.1. Here it is invoked on the "big" basic-blocks `B` and `tb`, while in Section 4.1, it is invoked on each bundle of `lb`.

[16] The terminology "*symbolic memory*" means that this alternative representation of each block is obtained by mimicking their sequential execution with "symbolic memories" instead of "concrete memories". See Appendix A for a formal overview.

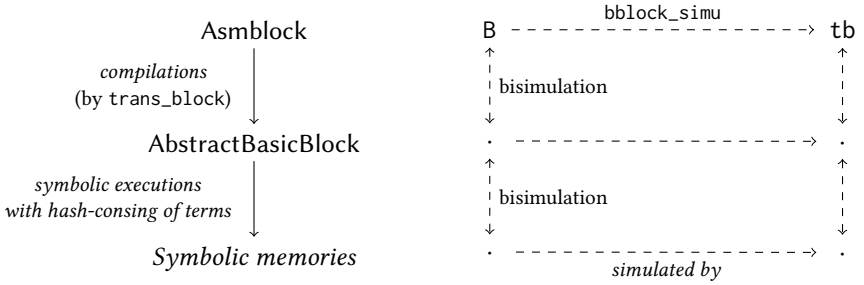



Fig. 8. Diagram of bblock_simub correctness

### 4.3 Generic and Verified Hash-Consing

Hash-consing a data-type simply consists in replacing the usual constructors of this data-type by *smart constructors* that perform the memoization of each constructor. This memoization is usually delegated to a dedicated function that can in turn be generated from a generic factory [Filliâtre and Conchon 2006]. Our hash-consing technique follows this principle. However, whereas the memoization factory of Filliâtre and Conchon [2006] (in OCaml) has no formal guarantee, ours satisfies a simple correctness property that is formally verified in Coq: each memoizing function *observationally behaves* like an identity. [17]

Typically, our Coq memoization function on terms invokes an external *untrusted* OCaml oracle that takes as input a given term, and returns a memoized term (possibly memoizing the input term in the process). Then, our Coq memoization function *dynamically* checks that the memoized term and the input term have the same *evaluation* function, or aborts the computation if it cannot ensure this. This check is kept constant-time by using OCaml *pointer equality* to compare *already memoized* subterms. We have thus imported OCaml pointer equality into Coq.

Importing an OCaml function into Coq is carried out by declaring the type of this OCaml function through an axiom: the Coq axiom is replaced by the actual OCaml function at extraction. Using a pure function type in this Coq axiom implicitly assumes that the OCaml function is *logically deterministic* (like any Coq function): calling the function twice on *equal* inputs should give *equal* outputs – where *equality* is Coq equality: structural equality. In contrast, the OCaml *pointer equality* does not satisfy this property: two *structurally equal* values do not necessarily have the same pointer. We solve this issue by using the *pointer equality* from the Impure library of [Boulmé and Vandendorpe 2019], which represents OCaml *pointer equality* as a non-deterministic function. We then use the axiom from Impure stating that, if pointer equality returns `true`, then the two values are (structurally) equal:

```
Axiom phys_eq: ∀ {A}, A → A → ??bool
Extract Constant phys_eq ⇒ "(==)"
Axiom phys_eq_true: ∀ A (x y: A), phys_eq x y ⇝ true →  x=y
```

Above, "`??bool`" is logically interpreted as the type of all "subsets" of Booleans, `phys_eq` is the physical equality, later extracted as the OCaml `(==)`, and "⇝" is the may-return relation of the Impure library: "`phys_eq x y ⇝ true`" means that "`true`" is a possible result of "`phys_eq x y`".

Hence, we have efficient and formally verified hash-consing. For more details, see Appendix A.

[17] This simple correctness property suffices for the formal correctness of our simulation test. However, it is too weak to ensure a strong invariant like "*two distinct BDDs w.r.t pointer equality are semantically different*".

## 5 INTRABLOCK SCHEDULING ORACLE

As detailed in Section 4, our postpass scheduling of basic-blocks is computed by an untrusted oracle. We provide 4 implementations of this oracle, selectable by command-line options. The first two implementations, presented in Section 5.1, do not reorder the instructions: we use them to measure the impact of reordering in experiments. Like the verifiers in Section 4, the reordering oracles are made up of two components: a *frontend* specific to the processor, and a generic *backend* independent from the processor. From the input basic-block, the frontend builds an *optimization problem*. Then, it invokes the backend that *solves* this problem. Finally, from this *solution*, the frontend returns a sequence of bundles. Section 5.2 presents the optimization problem, and how bundles are built from its solutions. Currently, our reordering oracles share the same (quite naive) frontend (a smarter one is ongoing). They only differ by their *solver* presented in Sections 5.3 and 5.4, respectively.

### 5.1 Trivial and naive greedy bundlers without reordering

The trivial bundler puts one instruction per bundle without any reordering. The naive greedy bundler does not reorder instructions either: it builds well-formed bundles out of successive instructions, while preserving sequential semantics. Both algorithms are easy.

### 5.2 Scheduling as an optimization problem

We refer the reader to [Micheli 1994, Ch. 5] for a general background on scheduling problems in hardware, which is not far from our software problem [Dupont de Dinechin 2004]. Here, we explain the exact problem we need to solve on the Kalray VLIW architecture.

We have $n$ instructions to schedule, that is, compute a function $t : 0 \dots n-1 \to \mathbb{N}$ assigning a date to each instruction. These dates will be used to group instructions into bundles: first bundle is all instructions $j$ such that $t(j) = 0$, next bundle all those such that $t(j) = 1$ etc.

Each instruction $j$ is characterized by a kind $K(j)$ (whether it is an addition, a multiplication, a load, etc.). This schedule must satisfy three classes of constraints:

**Semantic dependencies** Read and write dependencies are examined for each register of the processor, as well as the pseudo-register MEM (standing for the whole addressable memory, that is, the memory accessed by LOAD and STORE instructions). These dependencies are functionally relevant: code reordered without paying attention to them is generally incorrect.

- *Read after write*: If instruction $j$ writes to register $r$ and this is the last write to $r$ before an instruction $j'$ reading from $r$, then the schedule should respect $t(j') - t(j) \geq 1$.
- *Write after write*: If instruction $j$ writes to register $r$ and this is the last write to $r$ before an instruction $j'$ writing to $r$, then the schedule should respect $t(j') - t(j) \geq 1$.
- *Write after read*: If instruction $j$ reads from $r$ and the next write to $r$ is instruction $j'$, then the schedule should respect $t(j') - t(j) \geq 0$.

**Latency constraints** The description of the processor microarchitecture states, for each instruction, the number of clock cycles after which the values it produces are ready. More precisely, it states that if an instruction of kind $k'$ is scheduled at least $\delta$ cycles after an instruction of kind $k$ then it incurs no waiting for reading the output of the other instruction. In most cases, $\delta$ depends only on $k$, but there are "bypasses" for some $k'$ with lower $\delta$ than for others. All these are mentioned in the processor documentation.[18] For memory loads, the number of cycles after which data are ready depends on whether data are in the cache or not. In this case, we use the minimum number: this assumes that data are in the L1 cache.

[18]For simplicity's sake, bypasses are not at present taken into account in our implementation. We consider building a system for converting tables supplied by the chip designers into more detailed latency information.

Find $t : 0 \ldots n \to \mathbb{N}$ satisfying:

| A resource usage constraint | Latency constraints of the kind (where $j' > j$): |
|---|---|
| $\forall i \sum_{j \mid t(j)=i} \mathbf{u}(K(j)) \leq \mathbf{r}$ (1) | $t(j') - t(j) \geq \delta$ (2) |

Fig. 9. Our family of scheduling problems

We have already mentioned that on K1c, latencies do not affect architectural semantics. If an instruction is scheduled before its operands are ready, the result is unchanged, the only consequence is that the whole bundle to which the instruction belongs is stalled. Thus, mistakes in the latencies encoded into the contraints may lead only to suboptimal performance, not to incorrect results.

**Resource usage constraints** The processor has a limited number of processing units. Therefore, a bundle of instructions must not request more processing units of a given kind than available. Also, there is a limit on the number of instruction words inside a bundle. Bundles that do not abide by these rules will be rejected by the assembler.
The architecture documentation describes these limitations as a constant vector $\mathbf{r} \in \mathbb{N}^m$ and, for each kind $k$ of instruction, a vector $\mathbf{u}(k) \in \mathbb{N}^m$. The constraint is that the sum of all $\mathbf{u}(K(j))$ for all instructions $j$ scheduled within the same bundle $i$ should be coordinate-wise less than or equal to the vector of available resources $\mathbf{r}$. This is expressed by Inequality (1) in Figure 9. This part of the problem is thus a form of *vector bin packing*.

The semantic dependencies and the latency constraints are instances of Inequality (2). In actual fact, most of the "read after write" dependencies are subsumed by the latency constraints between the output values and the read operands.

Finally, we introduce an extra date $t(n)$ representing the time at which all instructions have already been completely executed, in the sense that all their outputs have been computed. We thus add extra latency constraints of the form $t(n) - t(j) \geq \delta$ to express that output operands should be available at time $t(n)$. Hence, $t(n)$ is the *makespan* of our basic-block, which we wish to minimize.

In other words, our scheduling problem is an instance of the system of inequalities in Figure 9: a correct sequence of bundles is directly built from any solution $t$, and $t(n)$ represents the (minimal) number of cycles run by an execution of this sequence.

### 5.3 List scheduler with respect to critical paths

We provide a solver based on a variant of Coffman-Graham list scheduling [Dupont de Dinechin 2004] [Micheli 1994, §5.4] with one heuristic: instructions with the longest latency path to the exit get priority. This is our default solver for the scheduling oracle: it is fast (linear-time) and computes an optimal schedule in almost all practical cases.

We consider each time $i$ starting from 0, and we choose at each step which instructions $j$ to schedule at time $i$ (those for which $t(j) = i$). Our Algorithm 1 is based on two simple ideas:

**Maximal scheduling sets** Assume we have already chosen a set $S$ of instructions to be scheduled at time $i$, such that $\sum_{j \in S} \mathbf{u}(K(j)) \leq \mathbf{r}$. Assume there is $j' \notin S$ such that all its operands are ready, and $\sum_{j \in S \cup \{j'\}} \mathbf{u}(K(j)) \leq \mathbf{r}$. Then it is always at least as good to schedule $j'$ in the same time slot as the instructions in $S$, compared to scheduling only $S$: this can never make

**Algorithm 1** A simplified version of the "critical-path" scheduler. The actual implementation pre-computes all $l(j,n)$ by a simple graph traversal. It also avoids scanning systematically for all $j' \in Q$ by updating an array, indexed by $i$, of sets of instructions ready to be scheduled $R(i)$ at time $i$: an instruction is added to the appropriate $R(i)$ when its last predecessor has been scheduled.

```
i := 0
while Q ≠ ∅ do
    R := ∅
    for j' ∈ Q do
        ready := true
        for j --δ--> j' ∈ G do
            if t(j) > j' − δ then
                ready := false
            end if
        end for
        if ready then
            R := R ∪ {j'}
        end if
    end for
    a := r
    for j ∈ R do (in descending l(j,n) order)
        if a ≥ u(K(j)) then
            a := a − u(K(j))
            Q := Q \ {j}
            t(j) := i
        end if
    end for
    i := i + 1
end while
```

the makespan increase. Thus, at every step we can restrict the search to $S$ maximal with respect to the inclusion ordering among the feasible $S$.

**Critical path heuristic** The question is then which $S$ to consider if there are many of them, which is generally the case for the first bundles of a block—since all instructions operating only on unassigned registers (in the basic-block) can be scheduled in the first bundle.

Consider the (multi)graph $G$ obtained by turning each inequality (2) into an edge $j \xrightarrow{\delta} j'$. By construction this graph is acyclic, since all these edges satisfy $j' > j$.

In a valid schedule, $t(j)$ is at most $t(n) - l(j,n)$ where $l(j,n)$ is the maximal length of paths from $j$ to $n$ in $G$. If we had no resource constraints, in an optimal schedule we would have $t(j) = t(n) - l(j,n)$. When constructing a maximal $S$, we thus consider $j$ in decreasing order of $l(j,n)$; in other words, we try to schedule first the instructions on the critical path.

This algorithm too is greedy: it never backtracks. Thus, if the heuristic choice is non-optimal, it may miss a better solution.

### 5.4 Optimal list scheduler by Integer Linear Programming

We provide an *optimal* solver based on Integer Linear Programming (ILP). This ILP solver is not intended for production use as it is too costly. Rather, it is used to validate how often the "critical-path" scheduler of Section 5.3 actually computes an optimal solution.

Roughly speaking, our optimal solver will turn the scheduling problem into an ILP problem, and then, invoke an external ILP solver to find a solution. More precisely, we first compute an upper bound $B$ on the makespan $t(n)$, typically by running the "critical-path" scheduler of Section 5.3: if the latter computes a solution with makespan $B+1$, then we can restrict our ILP search to solutions of makespan at most $B$.

We first compute, for every instruction $j$, the maximum length $\alpha(j)$ of a path ending at $j$ in the graph, and the maximum length $l(j)$ of a path starting at $j$ (necessarily to $n$). Let us define $\beta(j) = B - l(j)$. Then we know that any solution $t$ satisfies $\alpha(j) \leq t(j) \leq \beta(j)$ for all $j$.

$$\sum_{i=\alpha(j)}^{\beta(j)} x(j,i) = 1 \quad (3)$$

$$t(j) = \sum_{i=\alpha(j)}^{\beta(j)} x(j,i)i \quad (4)$$

$$x(j,i) \leq \sum_{i'=\max(i+\delta,\alpha(j'))}^{\beta(j')} x(j',i') \quad (5)$$

$$\sum_{j|\alpha(j)\leq i\leq\beta(j)} x(j,i)u_h(K(j)) \leq r_h \quad (6)$$

Fig. 10. Turning the scheduling problem into an ILP problem

For every $j$ and every $\alpha(j) \leq i \leq \beta(j)$, we introduce a Boolean variable $x(j,i)$, meaning that instruction $j$ is scheduled at time $i$. An instruction is scheduled at a single date, thus all these variables are exclusive, as expressed by Equation (3) of Figure 10. Then, $t(j)$ is easily recovered from the Boolean variables (Equation 4). The latency (and dependency) constraints $t(j') - t(j) \geq \delta$ are directly copied into the ILP problem. Alternatively, a constraint $t(j') - t(j) \geq \delta$ can for instance be encoded as Inequality (5). The resource constraints are implemented by introducing, for all $0 \leq i \leq B$ and all $1 \leq h \leq m$, an instance of Inequality (6) where $u_h(k)$ denotes the $h$-th coordinate of $\mathbf{u}(k)$ and $r_h$ the $h$-th coordinate of $\mathbf{r}$.
The resulting ILP problem can be used in two ways:

**Optimization** Minimize $t(n)$.

**Decision** Test for the existence of a solution with makespan $t(n) \leq B$. Then set $B$ to be $t(n) - 1$ and restart the process, until no better solution is found.

While it may appear that optimization, requiring one ILP call, would be more efficient than a sequence of decision problems, but it seems that, experimentally, this is not the case.

Such an ILP problem can also be formulated as *pseudo-Boolean*, that is, a problem where all variables are Boolean and the constraints are linear inequalities. It suffices to replace $t(j)$ by its definition from Equ. (4) everywhere.

We support Gurobi and CPlex as ILP backends, and we have run experiments with Sat4J and others as pseudo-Boolean backends.

## 6 BASIC-BLOCK RECONSTRUCTION

This section explains how we transform a Mach program into an Asmblock program. As explained in Section 2.3 page 8, on an arbitrary Asm program, we cannot prove that execution cannot jump to the middle of a function, and thus to the middle of a basic-block. Thus, the basic-block structure can only be recovered syntactically from the function structure of Mach programs. Moreover, proving that a block-step semantics simulates an instruction-step semantics is not very easy. Hence, it seems interesting to capitalize this effort for a generic component w.r.t the processor.

To this end, we have introduced a new IR between Mach and Asmblock, called Machblock (see Figure 3). This introduces a sequential blockstep semantics on Mach programs. The translation from Mach to Machblock—presented in Section 6.1—exhibits the basic-block structure of Mach programs. Then, each block of Machblock is translated into one (most often) or several (in a few cases) blocks of Asmblock, as explained in Section 6.2.

### 6.1 Mach to Machblock translation

The basic-blocks syntax in Machblock is very similar to that of Asmblock, except that instructions are Mach instructions and that empty basic-blocks are allowed (but not generated by our translation

from Mach). We have only defined a sequential semantics for Machblock, which is Mach one, except that a whole basic-block is run in one single step. This block-step is internally made up of several computation steps, one by basic or control-flow instruction.

```
Record bblock := { header: list label; body: list basic_inst; exit: option control_flow_inst }
```

The code of our translation from Mach to Machblock is straightforward: it groups successive Mach instructions into a sequence of "as-big-as-possible" basic-blocks, while preserving the initial order of instructions. Indeed, each Mach instruction corresponds syntactically to either a label, a basic instruction, or a control-flow instruction of Machblock.

Proving that this straightforward translation is a forward simulation is much less simple than naively expected. Our proof is based on a special case of "Option" simulation (see Figure 4). Intuitively, the Machblock execution stutters until the Mach execution reaches the last execution of the current block, then while the Mach execution runs the last step of the current block, the Machblock execution runs the whole block in one step. Hence, the measure over Mach states—that indicates the number of Machblock successive stuttering steps—is simply the size of the block (including the number of labels) minus 1. Formally, we have introduced a dedicated simulation scheme, called "Block" simulation in order to simplify this simulation proof. This specialized scheme avoids defining the simulation relation—written "$\sim$" in Figure 4—relating Mach and Machblock states in the stuttering case. In other words, our scheme only requires relating of Mach and Machblock states at the beginning of a block. Indeed, the "Block" simulation scheme of a Mach program $P_1$ by a Machblock program $P_2$ is defined by the two conditions below (where $S_1$ and $S_1'$ are states of $P_1$; $S_2$ and $S_2'$ are states of $P_2$; and $t$ is a trace):

(1) stuttering case of $P_2$ for one step of $P_1$:

$$|S_1| > 0 \ \wedge \ S_1 \xrightarrow{t} S_1' \quad \Longrightarrow \quad t = \epsilon \ \wedge \ |S_1| = |S_1'| + 1$$

(2) one step of $P_2$ for $|S_1|+1$ steps of $P_1$:

$$S_1 \sim S_2 \ \wedge \ S_1 \xrightarrow{t}{}^{|S_1|+1} S_1' \quad \Longrightarrow \quad \exists S_2', S_2 \xrightarrow{t} S_2' \ \wedge \ S_1' \sim S_2'$$

Naively, relation $S_1 \sim S_2$ would be defined as "$S_2$ = `trans_state`($S_1$)" where `trans_state` translates the Mach state in $S_1$ into a Machblock state, only by translating the Mach codes of $S_1$ into Machblock codes. However, this simple relation is not preserved by `goto` instructions on labels. Indeed, in Mach semantics, a `goto` branches to the instruction *following* the label. On the contrary, in Machblock semantics, a `goto` branches to the block *containing* the label. Hence, actually, we define $S_1 \sim S_2$ as the following relation: "*either* $S_2$ = `trans_state`($S_1$)*, or the next* Machblock *step from* $S_2$ *reaches the same* Machblock *state as the next step from* `trans_state`($S_1$)". The condition (2) of the "Block" simulation is then proved according to the decomposition of Figure 11.

### 6.2 Machblock to Asmblock translation

The usual Mach-to-Asm pass of CompCert backends is an instruction-by-instruction translation. It is proved by composing together the large number of correctness lemmas about the independant parts of the translation. For example, one of them expresses that the Mach conditional branching is simulated by several Asm instructions (see Section 2.3).

The Machblock-to-Asmblock pass is directly adapted from these usual passes, but with one major change: its proof is a block-step simulation instead of an instruction-step simulation. This has led us to introduce an intermediate type of *ghost* states, called "Codestate". A Codestate extends a usual Asmblock state by duplicating the Asmblock instructions of the current block still to be executed into explicit fields. Then, the $\sim$ relation between Machblock and Asmblock states is composed of two relations: a relation from a Machblock state to a Codestate expressing the correctness of the translation, and a relation from a Codestate to an Asmblock state ensuring consistency between

On the right-hand side, c and b::bl are, respectively, the initial Mach and Machblock codes where b is the basic-block at the head of the Machblock code. Relation match_state is the Coq name of the simulation relation (also noted "~" in the paper). The Machblock step from b::bl simulates the following |b| Mach steps from c. First, skip all labels: this leads to Mach code c0. Second, run all basic instructions: this leads to Mach code c1. Finally, run the optional control-flow: this leads to code c2. Each of these three subdiagrams is actually an independent lemma of our Coq proof.

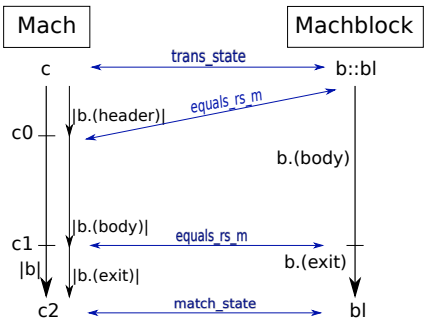



Fig. 11. Overview of our proof for condition (2) of the "Block" simulation of Mach by Machblock

the actual code in memory, pointed by register PC, and the duplicated one in Codestate. This allows us to decompose the block-step simulation into instruction-step simulations and to adapt relatively effortlessly the usual basic lemmas of Mach-to-Asm passes. However, the "glue" around these lemmas has been completely rewritten.

## 7 EXPERIMENTAL EVALUATION OF PERFORMANCES

We compared compilation times and timings of compiled code to that of the default compiler for the target platform. Timings are obtained from a cycle-accurate CPU simulator. In addition to our postpass optimization, we also benefit from the if-conversion pass recently added to CompCert, which transforms certain branches and ternary (if-then-else) operators into conditional moves, thus increasing the size of basic-blocks.

### 7.1 Benchmarks

*Heptagon-**, *Lustrev4-**, *Lustrev6-** are benchmarks compiled from synchronous data-flow programming languages. Such languages are for instance used to specify and implement fly-by-wire aircraft controls [França et al. 2011]. In this context, the C program obtained by compilation of the synchronous program is often compiled *without optimization* so that it can be easily matched to the resulting assembly code, in order to satisfy qualification requirements [França et al. 2011]. CompCert's advantage in this area is that it allows use of optimizations, while its semantics preservation proof replaces the structural manual matching between assembly and C code [França et al. 2011]. Any performance gain above, say, GCC -O0, is good in this context.

*lift* is a lift controller program from TACLeBench, a collection of benchmarks used for worst-case execution time research [Falk et al. 2016].

*binary_search*, *quicksort* and *heapsort* are textbook implementations of familiar algorithms from Rosetta Code.[19]

*idea* and *sha-256* are cryptographic primitives. *bitsliced-aes* and *bitsliced-tea* are bitsliced implementations of the AES and TEA block ciphers.

*complex-mat* and *float-mat* are floating-point implementations of matrix multiplication over complex and real numbers (no special instructions are used for complex floats). *xor-mat* is matrix multiplication in the ring $(\{0,1\}^{64}, \text{XOR}, \text{AND})$. *complex-mat*, *float-mat v2* and *xor-mat* feature loop transformations on the C code to have a better performance.

*ntt* naively performs a number theoretic transform in $\mathbb{Z}/(2^{16}+1)\mathbb{Z}$ for a vector of size $2^{16}$.

[19] https://www.rosettacode.org

### 7.2 Scheduling optimality

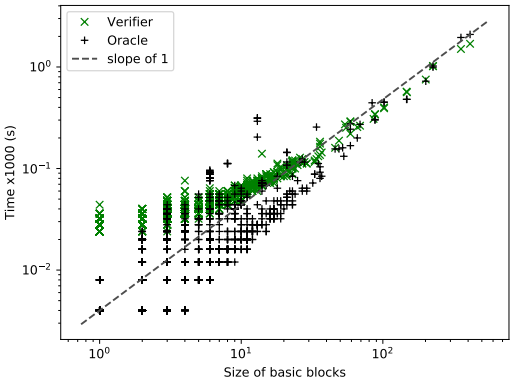
We compiled our benchmarks with ILP scheduling of Section 5.4. The ILP scheduler outperformed our list scheduler of Section 5.3 in only 8 of 26,161 basic-blocks: for 1 block, removing 2 cycles (from 106 to 104); in others, removing 1.

As detailed in Section 5.4, we first compute an initial schedule by list scheduling, then try to find improvements using ILP. The ILP solver is not called if that initial schedule is obviously optimal since its makespan is the minimum makespan imposed by a critical path. Among the 26,161, we called the ILP solver only 2,368 times.

In most cases, solving ILP problems using Gurobi takes a few milliseconds of CPU time. However, it can be considerably costlier in some rare cases: for instance, Gurobi took more than 7 hours to check that the initial schedule for a 147 instruction block was optimal.

### 7.3 Experimental time complexity of the list scheduling oracle and its verifier

We experimentally checked that our oracle and its verifier have linear running times, by instrumenting the generated OCaml code of the compiler to get the user timings and basic-block sizes. The figure on the right-hand side shows our measurements in logarithmic scales. Each point in this figure corresponds to an actual basic-block from our benchmarks, verified or scheduled (for the list-scheduling) 1000 times. The verifier is generally a little slower than the oracle, but both are experimentally of linear complexity. The biggest basic-block we came accross, of around 500 instructions, was scheduled and verified in approximately 4 ms.




### 7.4 Impact of our postpass scheduling pass in CompCert

Figure 12 illustrates the impact of our optimization pass on the performance of the generated code. It measures 3 versions of the scheduling oracle in our “ccomp” compiler. The default version uses the list scheduler of Section 5.3. The others use the *bundlers without reordering* of Section 5.1: the “nobundle” version emits one instruction per bundle, and the “pack” version uses the naive greedy bundler. In addition to these, the “noif” version corresponds to the default list scheduler, but without any if-conversion. For each of these 4 versions, Figure 12 displays the ratio of the best time (out of the 4 versions) versus the time of the given version. Higher percentages mean better timings.

First of all, the if-conversion leaves most of our benchmarks unchanged - however, some of them do see quite an increase in performance, in particular one of them gains 55%, and the Lustrev4 "heater control" code gains 14%. The Lustre code contains a large amount of if/else branches with a single move inside - as for the filling part of our binary search, it has a ternary instruction in the random generation part.

Postpass scheduling has a bigger impact on performance: compared to the “nobundle” version, most benchmarks get a code at least 20% more performant, and 8 out of 18 benches observe a performance boost of more than 50%, up to 110%. However, some benches like Lustrev6-convertible are barely affected by the optimization - indeed, the main loop features around 800 different

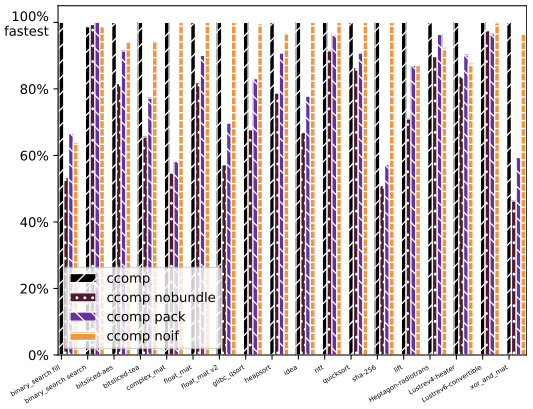



Fig. 12. Comparing various CompCert versions

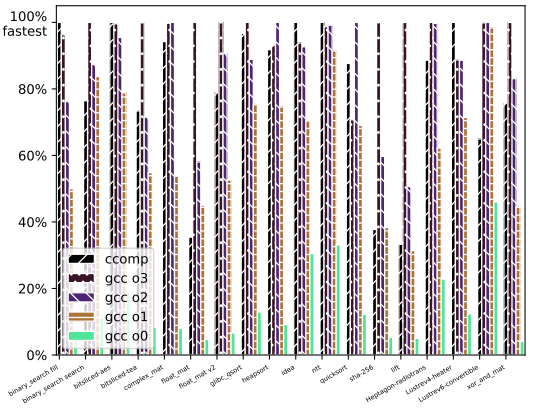



Fig. 13. CompCert versus Kalray's GCC.

variables in the same scope, which does not fit into the 64 registers of the k1c. Register spillings are thus generated, which in turn prevent scheduling since we do not yet have any alias analysis[20].

The "pack" version gains a slight increase in performance w.r.t "nobundle", but not by much compared to our actual postpass scheduling. Half of our benchmarks gain an increase in performance of more than 20% by using list scheduling with reordering instead of naive greedy bundling.

### 7.5 Comparison of CompCert with Kalray's GCC compiler

We also compared our CompCert compiler to the GCC[21] compiler supplied by Kalray, and adapted from version 7.4.1. We compared it to the -O3, -O2, -O1 and -O0 optimizations of GCC. It is worth noting that, since -O1 deactivates scheduling, it only generates bundles of one instruction. Also, -O0 gives particularly bad code, and notably only uses a handful of registers.

The results vary considerably depending on the benchmark - furthermore, at the time of this writing, the GCC backend is still being developed by Kalray: in particular some optimizations are not yet functional, and code selection could be improved in a few places. It is thus hard to draw meaningful conclusions on the comparison with GCC, though it allowed us to outline some optimizations that could be made by CompCert to improve performance.

Regardless, for most of benchmarks, CompCert generated code is between 20% and 30% slower than that of GCC -O3, but faster for a couple of benchmarks, up to 20%. Except for a few benchmarks, CompCert is generally around 10% slower than GCC -O2, while it is faster for 5 benchmarks. For most of the benchmarks, CompCert is more than 20% faster than GCC -O1, as much as twice as fast. In the worst case, CompCert is currently 65% behind -O3, 40% behind -O2, 30% behind -O1, and 40% *better* than -O0. This last figure is a really pathological case - for all the other benchmarks, CompCert is between 2 and 17 times better than GCC -O0.

### 7.6 Remarks and limitations

One major difficulty is that, even though we aim at comparing the quality of instruction scheduling, we are actually comparing two very different whole compilers. In particular, the machine-independent parts of CompCert do not perform certain optimizations that GCC does:

[20] In our verifier, the memory is viewed as a unique resource, see section 4.1 for more details.
[21] The GNU Compiler Collection, https://gcc.gnu.org/

- certain *strength reductions*: for instance, GCC converts a multiplicative expression $ci$, where $c$ is a loop-invariant constant and $i$ is a loop index with a step of one into a new variable $x$ stepping in increments of $c$;
- *loop invariant code motion* and, more generally, any form of code motion across basic-blocks;
- loop *unrolling* and other loop optimizations; CompCert compiles loops straightforwardly.

In contrast, CompCert will replace a 32-bit signed division by a constant with a small efficient sequence of code [Granlund and Montgomery 1994], whereas this version of GCC calls a generic library function. Certain compiler differences are subtler but may have dramatic effects in some cases. For instance, our version of CompCert does not simplify an inlined function if a parameter is a constant value allowing simpler instructions to be used, e.g., replacing a floating-point division by 2 by a multiplication by 0.5.

Some of these discrepancies have great importance on some benchmarks. For instance, textbook matrix multiplication can be greatly optimized by strength reduction (removal of multiplications for computing the address of array cells according to indices and stride), loop unrolling and loop invariant code motion. In some benchmarks, we consider both the original code and some slight manual optimization of this code, reproducing the optimizations that GCC would perform.

In some cases, we were able to identify register reuse as the cause of disappointing performance. Our postpass scheduler has to obey read-over-write and write-over-write dependencies. CompCert's register allocator sometimes reuses registers in ways that prevent some better scheduling from being adopted. We plan to add a pre-pass scheduler to address that problem.

Our backend cannot at present reorder a memory read and a memory write, or two memory writes, when their addresses cannot overlap. Also, CompCert sometimes does not recognize that it is reloading a value that it recently stored. We plan to add some form of alias analysis to our system to resolve both these issues.

Finally, despite our efforts, our instruction selection is still immature: some instructions that could be of use are still not selected. This is due to a lack of development time.

## 8 CONCLUSION AND FUTURE WORK

Our implementation adds to the source code of CompCert around 20Kloc of Coq. This is much more than the usual 5Kloc added for each CompCert target. Our target assembly is described by around 1.2K lines of specification in the AsmVLIW module. This is a little more than other Asm, specified by between 0.7K and 1K lines. Our scheduling oracle is implemented by 2.2Kloc of OCaml (half for its frontend, and half for its backend).

*Lessons learned.* Formal proof forces developers to rigorously document the compiler, with precise semantics and invariants. Proving programs in Coq is heavyweight, but there is almost no bug-finding after testing on real programs: the compiler just works. This allowed us, compilation novices and with no experience in CompCert, to extend it with another backend within a matter of one year. Most bugs we had to consider were in the parts of CompCert not concerned by formal proofs: printing of assembly instructions, sequences of instructions allocating and deallocating stack frames. On one occasion we used an instruction incorrectly due to a misunderstanding on its semantics. Apart from the difficulty of finding suitable invariants and proof techniques, the other main hurdle was properly exploiting the platform and interpreting benchmark results.

*Future work.* It seems that a major cause of inefficient schedules, which may account for disappointing performances in some benchmarks, is that we cannot reorder memory stores, or memory stores and loads—we plan to extend our scheduler with an alias analysis to allow such reordering. Another cause is register reuse, which prevents some instructions from being reordered—we plan

to add another scheduler to schedule instructions approximately prior to register allocation, which could perhaps improve performance for all architectures.

Further optimizations will probably involve inter-block code motion and changes to control flow. In this context, it will be interesting to see what can be achieved by oracle and checkers, and what is better carried out by classical total correctness proofs.

## ACKNOWLEDGMENTS

We thank Benoît De Dinechin and Xavier Leroy for their advices and suggestions all along this work. We thank Gergö Barany for his help while we were starting to port CompCert for the K1c. We also thank Thomas Vandendorpe for his improvements on the pass from Mach to Machblock.

## REFERENCES

Gergö Barany. 2018. A more precise, more correct stack and register model for CompCert. In *LOLA 2018 - Syntax and Semantics of Low-Level Languages 2018*. Oxford, United Kingdom. https://hal.inria.fr/hal-01799629

Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. 2012. Formally verified optimizing compilation in ACG-based flight control software. In *Embedded Real Time Software and Systems (ERTS2)*. AAAF, SEE. arXiv:hal-00653367

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings (Lecture Notes in Computer Science)*, Vol. 4085. Springer, 460–475. https://doi.org/10.1007/11813040_31

Gabriel Hjort Blindell, Mats Carlsson, Roberto Castañeda Lozano, and Christian Schulte. 2017. Complete and Practical Universal Instruction Selection. *ACM Trans. Embedded Comput. Syst.* 16, 5 (2017), 119:1–119:18. https://doi.org/10.1145/3126528

Sylvain Boulmé and Thomas Vandendorpe. 2019. Embedding Untrusted Imperative ML Oracles into Coq Verified Code. (March 2019). https://hal.archives-ouvertes.fr/hal-02062288 preprint.

Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. 2014. Implementing and Reasoning About Hash-consed Data Structures in Coq. *J. Autom. Reasoning* 53, 3 (2014), 271–304.

Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*. ACM, 27–38. https://doi.org/10.1145/1455770.1455776

Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial Register Allocation and Instruction Scheduling. *Transactions on Programming Languages and Systems* (April 2019). arXiv:1804.02452 https://chschulte.github.io/papers/castanedacarlssonea-toplas-2019.html Accepted for publication.

Benoît Dupont de Dinechin. 2004. From Machine Scheduling to VLIW Instrution Scheduling. *ST Journal of Research* 1, 2 (September 2004), 1–35. https://www.cri.ensmp.fr/classement/doc/A-352.ps Also as Mines ParisTech research article A/352/CRI.

Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016) (OpenAccess Series in Informatics (OASIcs))*, Martin Schoeberl (Ed.), Vol. 55. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:10.

Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. ACM, 12–19. https://doi.org/10.1145/1159876.1159880

J. Fisher. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.* 30, 07 (jul 1981), 478–490. https://doi.org/10.1109/TC.1981.1675827

Joseph A. Fisher. 1983. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the 10th Annual Symposium on Computer Architecture, 1983*. ACM Press, 140–150.

Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. 2005. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. 2011. Towards Formally Verified Optimizing Compilation in Flight Control Software. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, March 18, 2011, Grenoble, France. (OASICS)*, Philipp Lucas, Lothar Thiele,

Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm (Eds.), Vol. 18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 59–68. https://doi.org/10.4230/OASIcs.PPES.2011.59

Torbjörn Granlund and Peter L. Montgomery. 1994. Division by Invariant Integers using Multiplication. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa (Eds.). ACM, 61–72. https://doi.org/10.1145/178243.178249

Frederik Krogsdal Jacobsen. 2019. A formally verified compiler back-end for a time-predictable processor. B.S. Thesis. Unpublished.

Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*. 3AF, SEE, SIE, Toulouse, France, 1–9. https://hal.inria.fr/hal-01643290

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27, 3 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

Monica S. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Programming Language Design and Implementation (PLDI)*). ACM Press.

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009). arXiv:inria-00415861

Xavier Leroy. 2009b. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://xavierleroy.org/publi/compcert-backend.pdf

Xavier Leroy. 2017. How I found a crash bug with hyperthreading in Intel's Skylake processors. https://thenextweb.com/contributors/2017/07/05/found-crash-bug-hyperthreading-intels-skylake-processors/

P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John Ruttenberg. 1993. The Multiflow Trace Scheduling Compiler. *J. Supercomput.* 7, 1-2 (May 1993), 51–142. https://doi.org/10.1007/BF01205182

Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill.

B. Ramakrishna Rau, Christopher D. Glaeser, and Raymond L. Picard. 1982. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *9th International Symposium on Computer Architecture (ISCA 1982), Austin, TX, USA, April 26-29, 1982*. IEEE Computer Society, 131–139. https://dl.acm.org/citation.cfm?id=801721

Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: a Case Study on Instruction Scheduling Optimizations. In *Principles of Programming Languages (POPL)*. ACM Press, 17–27.

Jean-Baptise Tristan. 2009. *Formal verification of translation validators*. Ph.D. Dissertation. Université Paris 7 Diderot.

Jean-Baptiste Tristan and Xavier Leroy. 2010. A simple, verified validator for software pipelining. In *Principles of Programming Languages (POPL)*. ACM Press, 83–92. http://gallium.inria.fr/~xleroy/publi/validation-softpipe.pdf

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI)*). ACM Press, 283–294.

# A OVERVIEW OF THE ABSTRACTBASICBLOCK INTERMEDIATE REPRESENTATION

AbstractBasicBlock is an IR (Intermediate Representation) – independent from the target processor and from the remainder of CompCert, dedicated to verification of the results of scheduling/bundling oracles. This IR is only used for verification: there is no translation from AbstractBasicBlock to another IR of CompCert.

Section A.1 explains informally how our assembly instructions are compiled into AbstractBasicBlock: this introduces the syntax of AbstractBasicBlock instructions. Section A.2 formally defines this syntax and its associated semantics. Section A.3 presents the *parallelizability test*, which checks that a bundle/basic-block has the same behavior in sequential and in parallel executions. Section A.4 presents the *simulation test*, which checks that the sequential semantics of basic-blocks is preserved by scheduling.

## A.1 Introduction through the translation from Asmblock and AsmVLIW

AbstractBasicBlock defines a (deeply-embedded) language for representing the semantics of single assembly instructions as the assignment of one or more pseudo-registers. For example, an instruction "add $r1, r2, r3$" is represented as an assignment "$r1 \coloneqq \mathsf{add}[r2, r3]$". Hence, AbstractBasicBlock distinguishes syntactically which pseudo-registers are in input or output of each instruction. Moreover, it gives to all operations (including load/store and control-flow ones) a single signature "`list exp` $\rightarrow$ `exp`". Hence, a binary operation like add will just dynamically fail, if applied to an unexpected list of arguments. This makes the syntax of AbstractBasicBlock very simple.

Let us consider less straightforward examples. Our translation from Asmblock to AbstractBasicBlock represents the whole memory as a single pseudo-register called here $m$. Hence, instruction "load $r_1, r_2, i$" (where $i$ is an integer constant representing offset) is encoded an assignment "$r_1 \coloneqq (\mathtt{load}\ i)[m, r_2]$" where the underlying operation is "(load $i$)". In other words, the syntax of AbstractBasicBlock provides an infinite number of operations "(load $i$)" (one for each $i$). Similarly, a "store $r_1, r_2, i$" is encoded an assignment "$m \coloneqq (\mathtt{store}\ i)[m, r_1, r_2]$" reflecting that the whole memory is potentially modified.

We also encode control-flow instructions in AbstractBasicBlock: a control-flow instruction modifies the special register PC (the program counter). Actually, we consider that each bundle of a VLIW processor has one control-flow instruction: when the latter is implicit in the assembly code, it corresponds to the increment of PC by the size of the bundle. Hence, in our translation of bundles to AbstractBasicBlock, each control-flow instruction performs at least the assignment "$\mathsf{PC} \coloneqq (\mathtt{incr}\ i)[\mathsf{PC}]$" where $i$ is an integer representing the size of the bundle. Typically, a conditional branch such as "lt $r, l$" (where $l$ is the label and $r$ a register) is translated as the *sequence* of two assignments in AbstractBasicBlock:

$$\mathsf{PC} \coloneqq (\mathtt{incr}\ i)[\mathsf{PC}]\ ;\quad \mathsf{PC} \coloneqq (\mathtt{lt}\ l)[\mathsf{PC}, r]$$

It could equivalently be coded as the assignment "$\mathsf{PC} \coloneqq (\mathtt{lt}\ l)[(\mathtt{incr}\ i)[\mathsf{PC}], r]$". However, we find it more convenient to insert the incrementation of PC before the assignments specific to each control-flow instruction. A more complex control-flow instruction such as "call $f$" (where $f$ is a function symbol) actually modifies two registers: PC and RA (the returned address). Hence "call$f$" is translated as the *sequence* of 3 assignments in AbstractBasicBlock:

$$\mathsf{PC} \coloneqq (\mathtt{incr}\ i)[\mathsf{PC}]\ ;\quad \mathsf{RA} \coloneqq \mathsf{PC}\ ;\quad \mathsf{PC} \coloneqq (\mathtt{cte}\ \mathrm{address}_f)[]$$

To resume, an *instruction* of AbstractBasicBlock is a sequence of assignments. An *abstract basic-block* is simply a list of such instructions: this list is run in sequence (for the sequential semantics), or in parallel (for the parallel semantics). Hence, there is a single translation from our assembly to

AbstractBasicBlock: this translation produces a bisimulable basic-block, both for the sequential semantics and the parallel semantics.

Finally, Asmblock contains instructions modifying several pseudo-registers in parallel. One of them is an atomic parallel load from a 128-bit memory word in two contiguous (and adequately aligned) destination registers $d_0$ and $d_1$. These two destination registers are distinct from each other by construction—but not necessarily from the base address register $a$. These parallel assignments are expressed in the sequential semantics of AbstractBasicBlock *instructions* with the special `Old` operator of AbstractBasicBlock *expressions*: an expression "(`Old` $e$)" evaluates "$e$" in the initial state of the surrounding AbstractBasicBlock instruction.[22] Hence, the parallel load of 128-bit words is given in terms of two loads of 64-bit words:[23]

$$d_0 \coloneqq (\texttt{load}\ i)[a, m]\ ;\quad d_1 \coloneqq (\texttt{load}\ (i+8))[(\texttt{Old}\ a), m]$$

Similarly, our assembly provides a pseudo-instruction `freeframe` modifying both the memory and some registers. It is involved in the epilogue of functions. In the semantics, `freeframe` modifies the memory $m$ by deallocating the current stack frame in the memory model of CompCert. It also updates register `SP` (the stack pointer) accordingly and destroys the contents of a scratch register called here *tmp*. The modifications to `SP` and $m$ are performed in "parallel", since `SP` indicates the current stack frame in $m$, and the new value of `SP` is read from this stack frame. For the pseudo-instruction "`freeframe` $i_1$ $i_2$" (where $i_1$ and $i_2$ are two integers), our translation from Asmblock to AbstractBasicBlock introduces two intermediate operations: first, "(`freeframe_m` $i_1$ $i_2$)" for the effect on memory, and second, "(`freeframe_SP` $i_1$ $i_2$)" for the effect on the stack pointer. Then, the pseudo-instruction "`freeframe` $i_1$ $i_2$" is translated as the *sequence* of 3 assignments in AbstractBasicBlock:

$$m \coloneqq (\texttt{freeframe\_m}\ i_1\ i_2)[\texttt{SP}, m]\ ;\quad \texttt{SP} \coloneqq (\texttt{freeframe\_SP}\ i_1\ i_2)[\texttt{SP}, (\texttt{Old}\ m)]\ ;\quad tmp \coloneqq \texttt{Vundef}[]$$

### A.2 Syntax, sequential and parallel semantics

The syntax of AbstractBasicBlock is parametrized by: a type `R.t` of pseudo-registers (the type of positive integers in practice) and a type `op` of operators. The semantics of AbstractBasicBlock is parametrized by: a type `value` of values and a type `genv` for global environments; and an evaluation function:

```
Parameter op_eval: genv → op → list value → option value
```

By convention, a `None` value in the semantics represents an error. For the underlying assembly instruction, it is either a dynamic error (like an invalid pointer dereference) or a syntactic error (the operation is not called on the right numbers of parameters).
The syntax of the language is given by:

```
Inductive exp := PReg(x:R.t) | Op (o:op) (le:list_exp) | Old (e:exp) with list_exp :=...
Definition inst := list (R.t * exp). (* = a sequence of assignments *)
Definition bblock := list inst
```

The semantics introduces a notion of memory from pseudo-registers into values.

```
Definition mem := R.t → value
Definition assign (m: mem) (x:R.t) (v: value): mem
  := fun y ⇒ if R.eq_dec x y then v else m y
```

[22] Such an operator `Old` is quite standard in Hoare logic assertions. For example, see the ACSL annotation language of Frama-C [Kirchner et al. 2015].
[23] A benefit of this translation is that our scheduling oracle may replace two loads of 64-bit words into one load of a 128-bit words, and our verifier is able to check "for free" whether the replacement is semantically correct.

Then, the sequential semantics of a bblock takes a memory m as input and returns an optional memory. It simply iterates sequentially over the execution of instructions, called inst_run, and detailed below. Here,"SOME $v \leftarrow e_1$ IN $e_2$" means "match $e_1$ with Some $v \Rightarrow e_2$ | _ ⇒ None end".

```
Fixpoint run (ge: genv) (p: bblock) (m: mem): option mem :=
  match p with
  | nil ⇒ Some m
  | i::p' ⇒ SOME m' ← inst_run ge i m m IN run ge p' m'
  end
```

The inst_run function takes two memory states as input: m as the current memory, and old as the initial state of the instruction run (the duplication is carried out in run above). It invokes the evaluation of expression, called exp_eval and defined below.

```
Fixpoint inst_run ge (i: inst) (m old: mem): option mem :=
  match i with
  | nil ⇒ Some m
  | (x,e)::i' ⇒ SOME v' ← exp_eval ge e m old
                 IN inst_run ge i' (assign m x v') old
  end
```

Similarly, the exp_eval function takes two memory states as input: the current memory is replaced by old when entering under the Old operator.

```
Fixpoint exp_eval ge (e: exp) (m old: mem): option value :=
  match e with
  | PReg x ⇒ Some (m x)
  | Op o le ⇒ SOME lv ← list_exp_eval ge le m old IN
               op_eval ge o lv
  | Old e ⇒ exp_eval ge e old old
  end
with list_exp_eval ge (le: list_exp) (m old: mem): option (list value) :=
 ...
```

Now, we define the non-deterministic out-of-order parallel semantics of AbstractBasicBlock as the prun relation below. Like the semantics of AsmVLIW defined at Section 3.2, it is defined from the in-order parallel semantics, called prun_iw below. This out-of-order execution simply invokes the prun_iw on an arbitrary permutation p' of the bblock and after duplicating the initial memory.

```
Definition prun ge p m (om: option mem) := ∃ p', om = (prun_iw ge p' m m) ∧ Permutation p p'
```

Indeed, prun_iw is parametrized by m for the write-only memory and by old for the read-only memory (which is thus the initial memory of the block)

```
Fixpoint prun_iw ge p m old: option mem :=
  match p with
  | nil ⇒ Some m
  | i::p' ⇒ SOME m1 ← inst_prun ge i m old old IN
             prun_iw ge p' m1 old
  end
```

The parallel semantics of an instruction now takes three memories as input: m for the write-only memory, old for the read-only memory (which is thus the initial memory of the block), and tmp a duplication of the old memory, with modifications that are purely local to the instruction.

```
Fixpoint inst_prun ge (i: inst) (m tmp old:mem) : option mem :=
  match i with
  | nil ⇒ Some m
  | (x,e)::i' ⇒ SOME v' ← exp_eval ge e tmp old IN
                 inst_prun i' ge (assign m x v') (assign tmp x v') old
  end
```

Note that, like in AsmVLIW, the sequential semantics of an instruction is a particular case of the parallel one. We have `(inst_run ge i m old) = (inst_prun ge i m m old)`. Moreover, in the sequential and parallel semantics of a block, instructions are considered *atomically*: splitting/merging instructions in the block does generally not preserve the semantics.

### A.3 Parallelizability Testing

Our parallelizability test is a function `is_parallelizable` taking a basic-block `p` and returning a Boolean. If this Boolean is true, then any out-of-order parallel execution returns the same[24] result (possibly `None`) as the sequential execution. In this case, out-of-order parallel execution is thus deterministic.

```
Theorem is_parallelizable_correct (p:bblock):
   is_parallelizable p = true → ∀ ge m om', prun ge p m om' ↔ om'=run ge p m
```

The `is_parallelizable` test analyzes the sets of pseudo-registers used by each instruction. The type of such sets of pseudo-registers is noted here `S.t` and is implemented by prefix-trees from module `PositiveSet` of the Coq standard library. Function `is_parallelizable` invokes actually two functions, `inst_wframe` and `inst_frame`, of type `inst → S.t`:

- `(inst_wframe i)` is the set of all pseudo-registers *written* by instruction `i`.
- `(inst_frame i)` is the set of all pseudo-registers *used*—i.e. read or written—by `i`;

Then, `(is_parallelizable p)` simply checks the absence of Use-After-Write: no instruction of `p` uses a pseudo-register after a previous instruction of `p` has written in it.

```
Fixpoint is_pararec (p: bblock) (previously_written: S.t): bool :=
  match p with
  | nil ⇒ true
  | i::p' ⇒ S.is_disjoint (inst_frame i) previously_written
             &&& is_pararec p' (S.union (inst_wframe i) previously_written)
  end

Definition is_parallelizable (p: bblock) := is_pararec p S.empty
```

The proof of `is_parallelizable_correct` results from the conjunction of two properties: the absence of Write-After-Write ensures that out-of-order parallel execution is deterministic; the absence of Read-After-Write ensures that sequential execution gives the same result as in-order parallel execution. To simplify this proof, we use a *data-refinement* style: first, we prove it when *frames* are implemented by lists instead of prefix-trees; then, we prove that the handling of frames implemented by prefix-trees emulates the handling of frames using lists. There is thus little proof about prefix-trees. A more detailed and significant example of data-refinement style is given in the next section.

### A.4 Simulation Testing

The sequential simulation of a block `p1` by a block `p2` is defined by the following pre-order:

```
Definition bblock_simu (p1 p2: bblock): Prop :=
  ∀ ge m, (run ge p1 m) <> None → (run ge p1 m) = (run ge p2 m)
```

We have implemented a simulation test: it takes two blocks `p1` and `p2`, and returns a Boolean, such that if this latter is true then `(bblock_simu p1 p2)`. This test is largely inspired by the list-scheduling verifier of Tristan and Leroy [2008], but with two major differences. First, they define their verifier for the Mach IR, while ours defined for AbstractBasicBlock is more generic. Second, we use hash-consing in order to avoid a combinatorial explosion of the test.

[24]Here, we admit functional extensionality to compare memories, like other parts of CompCert.

Like in Tristan and Leroy [2008], our simulation test symbolically executes each block, and then simply compares the resulting *symbolic memories*. As introduced in Section 4.2, such a symbolic memory bisimulates the input block by combining a parallel assignment together with a list of potential failures. We recall the examples of Section 4.2 below.

*Example (Reminder of Example 4.2).* Let us consider the two blocks below (in informal syntax):
$r_1 := r_1 + r_2;\ r_3 := \texttt{load}[r_2, m];\ r_1 := r_1 + r_3 \qquad r_3 := \texttt{load}[r_2, m];\ r_1 := r_1 + r_2;\ r_1 := r_1 + r_3$
These two blocks are both equivalent to the parallel assignment (in an informal syntax):

$$r_1 := (r_1 + r_2) + \texttt{load}[r_2, m] \parallel r_3 := \texttt{load}[r_2, m]$$

Indeed, these two blocks simulate each other (they bisimulate).

*Example (Reminder of Example 4.3).* Let us consider the two bblocks $p_1$ and $p_2$ below:
$r_1 := r_1 + r_2;\ r_3 := \texttt{load}[r_2, m];\ r_3 := r_1;\ r_1 := r_1 + r_3 \quad (p_1) \qquad r_3 := r_1 + r_2;\ r_1 := r_3 + r_3 \quad (p_2)$
Again, $p_1$ and $p_2$ lead to the same parallel assignment:

$$r_1 := (r_1 + r_2) + (r_1 + r_2) \parallel r_3 := r_1 + r_2$$

However, $p_1$ is simulated by $p_2$ whereas the converse is not true. This is because the "useless" memory access in $p_1$ may cause its execution to fail, whereas this failure cannot happen in $p_2$. Thus, the symbolic memory of $p_1$ should contain the term "$\texttt{load}[r_2, m]$" as a potential failure.

Our formal development is decomposed into two parts using a *data-refinement* style. In a first part, presented in Section A.4.1, we define a model of the symbolic execution and the simulation test. In a second part, sketched by Section A.4.3, we refine this model with efficient data-structures and algorithms, involving hash-consing of terms. Indeed, as illustrated by the previous examples, without a mechanism dealing efficiently with duplication of terms, symbolic execution produces terms that may be exponentially big w.r.t to the size of the source block. Our technique for hash-consing terms is explained in Section A.4.2.

*A.4.1 A Model of our Simulation Test.* The type `term` of terms—defined below—is similar to type `exp` without the `Old` operator. In a term, a pseudo-register represents its value in the initial memory of block execution.

```
Inductive term := Input (x:R.t) | App (o: op) (l: list_term)  with list_term :=...

Fixpoint term_eval (ge: genv) (t: term) (m: mem): option value :=...
```

In this model, the symbolic execution of a block is a function `bblock_smem: bblock → smem`, where a symbolic memory of type `smem` is the pair of a predicate `pre` expressing at which condition the intermediate computations of the block do not fail, and of a parallel assignment `post` on the pseudo-registers.

```
Record smem:= {pre: genv → mem → Prop; post: R.t → term}
```

Then, the bisimulation property between symbolic execution and sequential execution is expressed by the `bblock_smem_correct` lemma below. It uses the `smem_correct` predicate, relating the symbolic memory `d` with an initial memory `m` and a final optional memory `om`.

```
Definition smem_correct ge (d: smem) (m: mem) (om: option mem): Prop :=
  ∀ m', om=Some m' ↔ (d.(pre) ge m ∧ ∀ x, term_eval ge (d.(post) x) m = Some (m' x))

Lemma bblock_smem_correct p m: smem_correct ge (bblock_smem p) m (run ge p m)
```

By using this lemma, we transfer the notion of simulation of block executions into the simulation of symbolic memories, through the predicate `smem_simu` below. In particular, proposition `(smem_valid ge d m)` holds iff the underlying execution does not return a `None` result from the initial memory `m`.

```
Definition smem_valid ge (d: smem) (m:mem): Prop :=
  d.(pre) ge m ∧ ∀ x, term_eval ge (d.(post) x) m <> None

Definition smem_simu (d1 d2: smem): Prop :=
   (∀ ge m, smem_valid ge d1 m → smem_valid ge d2 m)
 ∧ (∀ ge m0 x, smem_valid ge d1 m0 →
               term_eval ge (d1.(post) x) m0 = term_eval ge (d2.(post) x) m0)

Theorem bblock_smem_simu p1 p2:
   smem_simu (bblock_smem p1) (bblock_smem p2) → bblock_simu ge p1 p2
```

Internally, as indicated in the name of "symbolic execution", `bblock_smem` mimics the sequential execution of the block, by replacing operations on memories of type `mem` by operations of type `smem`: these operations on the symbolic memory are given in Figure 14. The initial symbolic memory is defined by `smem_empty`. The evaluation of expressions on symbolic memories is defined by `exp_term`: it outputs a term (e.g. a *symbolic value*). Also, the assignment on symbolic memories is defined by `smem_set`. To conclude, starting from `smem_empty`, the symbolic execution preserves the `smem_correct` relation w.r.t the initial memory and the current (optional) memory, on each assignment.

```
(* initial symbolic memory *)
Definition smem_empty := {| pre:=(fun _ _ ⇒ True); post:=(fun x ⇒ Input x) |}

(* symbolic evaluation of the right-hand side of an assignment *)
Fixpoint exp_term (e: exp) (d old: smem) : term :=
  match e with
  | PReg x ⇒ d.(post) x
  | Op o le ⇒ App o (list_exp_term le d old)
  | Old e ⇒ exp_term e old old
  end
with list_exp_term (le: list_exp) (d old: smem) : list_term :=...

(* effect of an assignment on the symbolic memory *)
Definition smem_set (d:smem) x (t:term) :=
  {| pre:=(fun ge m ⇒ (term_eval ge (d.(post) x) m) <> None ∧ (d.(pre) ge m));
     post:=(fun y ⇒ if R.eq_dec x y then t else d.(post) y) |}
```

Fig. 14. Basic operations of the symbolic execution in the abstract model

*A.4.2 Formally Verified Hash-Consed Terms in Coq.* Hash-consing is a standard technique of imperative programming, which in our case has two benefits: it avoids duplication of structurally equal terms in memory, and importantly, it reduces (expansive) *structural equality* tests over terms, to (very cheap) *pointer equality* tests. In our verified backend, we thus need to import pointer equality from OCaml. However, importing pointer equality as a pure function, such as "`∀ {A}, A → A → bool`", would be unsafe. Indeed, such a Coq function is—by construction—compatible with the logical equality of Coq (i.e. the structural equality), which is not the case of pointer equality. Thus, we instead import pointer equality from the Impure library of [Boulmé and Vandendorpe 2019].

*The Impure library and its pointer equality test.* The Impure library provides an approach to safely embed type-safe OCaml impure computations into Coq-verified code: impure computations are abstracted as non-deterministic computations. For a given type `A`, type `??A` represents the type of non-deterministic computations returning values of type `A`: it can be interpreted as $\mathcal{P}(A)$, the type `A → Prop` of predicates over `A`. Formally, the type transformer "`??`" is axiomatized as a monad that provides a *may-return* relation "$\rightsquigarrow$" of type `?? A → A → Prop`. Intuitively, when "`A`"

is seen as "$\mathcal{P}$(A)", then "$\rightsquigarrow$" simply corresponds to identity. At extraction, ??A is extracted like A, and its binding operator is efficiently extracted as an OCaml let-in. See details in [Boulmé and Vandendorpe 2019]. Moreover, this library declares a *trusted* pointer equality with the following axioms.

```
Axiom phys_eq: ∀ {A}, A → A → ?? bool
Extract Constant phys_eq ⇒ "(==)"
Axiom phys_eq_true: ∀ A (x y: A), phys_eq x y ⇝ true →  x=y
```

In other words, in our Coq model, the pointer equality test is seen as a non-deterministic function, since it can distinguish more objects than the logical equality of Coq. Moreover, when it answers true, we know that the two objects under test are logically equals.

*A generic and verified factory of memoizing functions for hash-consing.* Hash-consing is a fundamentally impure construction, and it is not easy to retrofit it into a pure language; Braibant et al. [2014] propose several approaches for hash-consing in Coq and in code extracted from Coq to OCaml. However, we need weaker properties than what they aim for. They wish to use physical equality (or equality on an "identifier" type) as equivalent to semantic equality; they use this to provide a fast equality test for Binary Decision Diagrams (BDD)—two Boolean functions represented by reduced ordered binary decision diagrams are equal if and only if the roots of the diagrams are physically the same. In contrast, we just need physical equality to imply semantic equality. This allows a lighter approach.

Hash-consing consists in memoizing the constructors of some inductive data-type —such as the terms described above—in order to ensure that two structurally equal terms are actually allocated to the same object in memory. In practice, this technique simply replaces the usual constructors of the data-type by *smart constructors* that perform memoization. Memoization is usually delegated to a dedicated function in turn generated from a generic factory.

On the top of the Impure library, we have defined in Coq a generic and verified memoization factory. This factory is inspired by that of Filliâtre and Conchon [2006] in OCaml. However, whereas their factory was not formally verified, ours satisfies a simple correctness property that is formally verified in Coq (and shown sufficient for the formal correctness of our simulation test). Actually, we use an external *untrusted* OCaml oracle that creates memoizing functions and we only *dynamically* check that these untrusted functions behave *observationally* like an identity. Let us insist on this point: the formal correctness of our memoization factory does not assume nor prove that our oracle is correct; it only assumes that the embedding of OCaml untrusted oracles in Coq verified code through the Impure library is correct (see the details in [Boulmé and Vandendorpe 2019]). We now detail a slightly simplified version of this factory.[25]

Our generic memoization factory is parametrized by a record of type (hashP A), where A is the type of objects to memoize. Below, hashcode is an abstract data type on the Coq side, extracted as an OCaml int. Function hash_eq is typically a fast equality test, for comparing a new object to already memoized ones in smart constructors. This test typically compares the sons of the root node w.r.t pointer equality (the example for terms is given below by term_hash_eq function). Function hashing is expected to provide a unique hashcode for data that are equal modulo hash_eq. Finally, set_hid is invoked by memoizing functions to allocate a fresh and unique hash-tag to new objects (this hash-tag is used by efficient implementations of hashing).

```
Record hashP (A:Type) := {
  hash_eq: A → A → ?? bool;
  hashing: A → ?? hashcode;
  set_hid: A → hashcode → A;
```

[25] Our actual factory also provides some debugging features, which are useful for printing a trace when the whole simulation test fails. We omit these implementation details in this presentation.

```
}
```

The details on `hashing` and `set_hid` are only relevant for efficiency: these functions are simply ignored in our formal proofs. Hence, given such `(hashP A)` structure, our OCaml oracle `xhCons` returns a (fresh) memoizing function of type `(A → ??A)`.

```
Axiom xhCons: ∀ {A}, hashP A → ??(A → ??A). (* declares our OCaml oracle in Coq *)
```

Such a memoizing function of type `(A → ??A)` is expected to behave as an identity w.r.t `hash_eq`. Actually, as we do not trust `xhCons`, we dynamically check this property.[26] Hence, our *verified* generic memoization factory in Coq –called `hCons` below—simply wraps each function returned by `xhCons` with this defensive check: it raises an exception if the memoizing function does not return a result equal to its input (w.r.t `hash_eq`). Below, the notation "`DO` $x$ `⇜` $e_1$ `;;` $e_2$" stands for a *bind* operation of the may-return monad of the Impure library (it is extracted as "**let** $x = e_1$ **in** $e_2$"). Moreover, "`RET` $e$" is the *unit* of this monad (it is extracted as "$e$"). Function "`assert_b`" is also provided by Impure.

```
Definition hCons {A} (hp: hashP A): ??(A → ??A) :=
  DO hC ⇜ xhCons hp;;
  RET (fun x ⇒
       DO y ⇜ hC x;;
       DO b ⇜ hp.(hash_eq) x y;;
       assert_b b "xhCons: hash-eq differs";; (* exception raised if Boolean [b] is [false] *)
       RET y)
```

We are thus able to formally prove the following (trivial) correctness property on `hCons`, which is sufficient in our development to reason about hash-consing. Here, the relation `R` is typically an equivalence under which we want to observe hash-consed objects.

```
Lemma hCons_correct A (hp: hashP A) (R: A → A → Prop):
  (∀ x y, hp.(hash_eq) x y ⇝ true → R x y) → ∀ hC, hCons hp ⇝ hC → ∀ x y, hC x ⇝ y → R x y
```

*Smart constructors for hash-consed terms.* In our development, we actually need hash-consing on two types of objects: `term` and `list_term`, because they are mutually inductive. First, we redefine type `term` and `list_term` into `hterm` and `list_hterm` by inserting a hash-tag—called below `hid`—at each node.

```
Inductive hterm :=
  | Input (x:R.t) (hid:hashcode)
  | App (o: op) (l: list_hterm) (hid:hashcode)
with list_hterm :=
  | LTnil (hid:hashcode)
  | LTcons (t:hterm) (l:list_hterm) (hid:hashcode)
```

Thus, we also have to redefine `term_eval` and `list_term_eval` for their "`hterm`" versions. Note that these functions simply ignore hash-tags.

```
Fixpoint hterm_eval (ge: genv) (t: hterm) (m: mem): option value :=
  match t with
  | Input x _ ⇒ Some (m x)
  | App o l _ ⇒ SOME v ← list_hterm_eval ge l m IN op_eval ge o v
  end
with list_hterm_eval ge (l: list_hterm) (m: mem) {struct l}: option (list value) :=...
```

Then, we define two records of type `(hashP hterm)` and `(hashP list_hterm)`. Below, we only detail the case of `(hashP hterm)`, as the `(hashP list_hterm)` case is similar. First, the `hash_eq` field of `(hashP hterm)` is defined as function `term_hash_eq` below. On the `Input` case, we use the structural equality over pseudo-registers. On the `App` case, we use an equality

[26] As `hash_eq` is expected to be constant-time, this dynamic check only induces a small overhead.

`op_eq` on type `op` in parameters of the simulation test, and we use the pointer equality over the list of terms.

```
Definition term_hash_eq (ta tb: hterm): ?? bool :=
  match ta, tb with
  | Input xa _, Input xb _ ⇒ if R.eq_dec xa xb then RET true else RET false
  | App oa lta _, App ob ltb _ ⇒
     DO b ↜ op_eq oa ob ;;
     if b then phys_eq lta ltb else RET false
  | _,_ ⇒ RET false
  end
```

Second, the `hashing` field of `(hashP hterm)` is defined as function `term_hashing` below. This function uses an untrusted oracle "`hash: ∀{A}, A → ??hashcode`" extracted as the polymorphic `Hashtbl.hash` of the OCaml standard library. It also uses `list_term_get_hid` defined below—that returns the hash-tag at the root node. To ensure memoization efficiency, two terms that are distinct w.r.t `term_hash_eq` are expected to have distinct `term_hashing` with a high probablity.[27] This property relies here on the fact that when `term_hashing` is invoked on a node of the form "`(App o l _)`", the list of terms `l` is already memoized, and thus `l` is the unique `list_hterm` associated with the hash-tag `(list_term_get_hid l)`.

```
Definition list_term_get_hid (l: list_hterm): hashcode :=
  match l with
  | LTnil hid ⇒ hid
  | LTcons _ _ hid ⇒ hid
  end

Definition term_hashing (t:hterm): ?? hashcode :=
  match t with
  | Input x _ ⇒
     DO hc ↜ hash 1;;
     DO hv ↜ hash x;;
     hash [hc;hv]
  | App o l _ ⇒
     DO hc ↜ hash 2;;
     DO hv ↜ hash o;;
     hash [hc;hv;list_term_get_hid l]
  end
```

Finally, the `set_hid` field of `(hashP hterm)` updates the hash-tag at the root node. It is defined by:

```
Definition term_set_hid (t: hterm) (hid: hashcode): hterm :=
  match t with
  | Input x _ ⇒ Input x hid
  | App op l _ ⇒ App op l hid
  end
```

Having defined two records of type `(hashP hterm)` and `(hashP list_hterm)` as sketched above, we can now instantiate `hCons` on each of these records. We get two memoizing functions `hC_term` and `hC_list_term` specified in Figure 15. The correctness property associated with each of these functions is derived from `hCons_correct` with an appropriate relation `R`: the semantical equivalence of terms (or list of terms). These memoizing functions and their correctness properties are parameters of the code building `hterm` and `list_hterm` described below.

Indeed, these functions are involved in the smart constructors of `hterm` and `list_hterm`. Below, we give the smart constructor—called `hApp`—for the `App` case with its correctness property. It uses a special hash-tag called `unknown_hid` (never allocated by our `xhCons` oracle). The three other smart constructors are similar.

[27] Two terms equals w.r.t `term_hash_eq` *must* also have the same `term_hashing`.

```
Variable hC_term: hterm → ?? hterm
Hypothesis hC_term_correct: ∀ t t', hC_term t ⤳ t' →
  ∀ ge m, hterm_eval ge t m = hterm_eval ge t' m

Variable hC_list_term: list_hterm → ?? list_hterm
Hypothesis hC_list_term_correct: ∀ lt lt', hC_list_term lt ⤳ lt' →
  ∀ ge m, list_hterm_eval ge lt m = list_hterm_eval ge lt' m
```

Fig. 15. Memoizing functions for hash-consing of terms (and list of terms)

```
Definition hApp (o:op) (l: list_hterm) : ?? hterm := hC_term (App o l unknown_hid)
Lemma hApp_correct o l: ∀ t, hApp o l ⤳ t →
   ∀ ge m, hterm_eval ge t m = (SOME v ← list_hterm_eval ge l m IN op_eval ge o v)
```

In the next section, we only build `hterm` and `list_hterm` by using the smart constructors defined above. This ensures that we can replace the structural equality over type `hterm` by the physical equality. However, this property does not need to be formally proved (and we have no such formal proof, since this property relies on the correctness of our untrusted memoization factory).

*A.4.3 Implementing the Simulation Test.* Our implementation can be decomposed in two parts. First, we implement the symbolic execution function as a *data-refinement* of the `bblock_smem` function of Section A.4.1. Then, we exploit the `bblock_smem_simu` theorem to derive the simulation test.

*Refining symbolic execution with hash-consed terms.* Our symbolic execution builds hash-consed terms. It invokes the smart constructors of Section A.4.2, and is thus itself parametrized by the memoizing functions `hC_term` and `hC_list_term` defined in Figure 15. Note that our simulation test will ultimately perform two symbolic executions, one for each block. Furthermore, these two symbolic executions share the same memoizing functions, leading to an efficient comparison of the symbolic memories through pointer equality. In the following paragraph, functions `hC_term` and `hC_list_term` remain implicit parameters as authorized by the section mechanism of Coq.

Figure 16 refines the type `smem` of symbolic memories into a type `hsmem`. The latter involves a dictionary with pseudo-registers of type `R.t` as keys, and terms of `hterm` as associated data. These dictionaries of type `(Dict.t hterm)` are implemented as prefix-trees, through the `PositiveMap` module of the Coq standard library.

Figure 16 also relates type `hsmem` to type `smem` (in a given environment `ge`), by a relation called `smem_model`. The `hpre` field of the symbolic memory is expected to contain a list of all the potential failing terms in the underlying execution. Hence, predicate `hsmem_valid` gives a precondition on the initial memory `m` ensuring that the underlying execution will not fail. This predicate is thus expected to be equivalent to the `smem_valid` predicate of the abstract model. Function `hsmem_post_eval` gives the final (optional) value associated with pseudo-register `x` from the initial memory `m`: if `x` is not in the `hpost` dictionary, then its associated value is that of the initial memory (it is expected to be unassigned by the underlying execution). This function is thus expected to simulate the evaluation of the symbolic memory of the abstract model.

Hence, `smem_model` is the (data-refinement) relation for which our implementation of the symbolic execution simulates the abstract model of Section A.4.1. Figure 17 provides an implementation of the operations of Figure 14 that preserves the data-refinement relation. The smart constructors building hash-consed terms are actually invoked by the `exp_hterm` (i.e. the evaluation of expressions on symbolic memories). The `hsmem_set` implementation given in Figure 17 is an intermediate refinement toward the actual implementation, which improves it on two points. First, in some specific cases—like when `ht` is an input or a constant, we know that `ht` cannot fail. In

```
(* The type of our symbolic memories with hash-consing *)
Record hsmem:= {hpre: list hterm; hpost: Dict.t hterm}

(* implementation of the [smem_valid] predicate *)
Definition hsmem_valid ge (hd: hsmem) (m:mem): Prop :=
   ∀ ht, List.In ht hd.(hpre) → hterm_eval ge ht m <> None

(* implementation of the symbolic memory evaluation *)
Definition hsmem_post_eval ge (hd: hsmem) x (m:mem): option value :=
   match Dict.get hd.(hpost) x with
   | None ⇒ Some (m x)
   | Some ht ⇒ hterm_eval ge ht m
   end

(* The data-refinement relation *)
Definition smem_model ge (d: smem) (hd:hsmem): Prop :=
  (∀ m, hsmem_valid ge hd m↔ smem_valid ge d m)
∧ ∀ m x, smem_valid ge d m → hsmem_post_eval ge hd x m = term_eval ge (d.(post) x) m
```

Fig. 16. Data-refinement of symbolic memories, with handling of hash-consed terms

these cases, we avoid adding it to `hd.(hpre)`. Second, when `ht` is structurally equal to `(Input x)`, the actual implementation removes `x` from the dictionary: in other words, an assignment like "$x := y$"—where $y \mapsto$ (`Input` $x$) in the current symbolic memory—resets $x$ as unassigned. In actual fact, there is much room for improvement on the `hsmem_set` operation, like applying rewriting rules on terms. However, such an extension is left for future works.

```
(* initial symbolic memory *)
Definition hsmem_empty: hsmem := {| hpre:= nil ; hpost := Dict.empty |}
Lemma hsmem_empty_correct ge: smem_model ge smem_empty hsmem_empty

(* symbolic evaluation of the right-hand side of an assignment *)
Fixpoint exp_hterm (e: exp) (hd hod: hsmem): ?? hterm :=
  match e with
  | PReg x ⇒
     match Dict.get hd.(post) x with
     | None ⇒ hInput x  (* smart constructor for Input *)
     | Some ht ⇒ RET ht
     end
  | Op o le ⇒
     DO lt ↜ list_exp_hterm le hd hod;;
     hApp o lt (* smart constructor for App *)
  | Old e ⇒ exp_hterm e hod hod
  end
with list_exp_hterm (le: list_exp) (d od: hsmem): ?? list_term :=
 ...
Lemma exp_hterm_correct ge e hod od d ht:
   smem_model ge od hod → smem_model ge d hd → exp_hterm e hd hod ⇝ ht →
   ∀ m, smem_valid ge d m → smem_valid ge od m →
            hterm_eval ge t m = term_eval ge (exp_term e d od) m

(* effect of an assignment on the symbolic memory *)
Definition hsmem_set (hd:hsmem) x (ht:hterm): ?? hsmem :=
  (* a weak version w.r.t the actual implementation *)
  RET {| hpre:= ht::hd.(hpre); hpost:=Dict.set hd x ht |}

Lemma hsmem_set_correct hd x ht ge d t hd':
   smem_model ge d hd → (∀ m, smem_valid ge d m → hterm_eval ge ht m = term_eval ge t m) →
   hsmem_set hd x ht ⇝ hd' → smem_model ge (smem_set d x t) hd'
```

Fig. 17. Refinement of the operations of Figure 14 for symbolic memories with hash-consing

Finally, we define the symbolic execution that invokes these operations on each assignment of the block. It is straightforward to prove that `(bblock_hsmem p)` refines `(bblock_smem p)` from the correctness properties of Figure 17.

```
Definition bblock_hsmem: bblock → ?? hsmem :=...

Lemma bblock_hsmem_correct p hd: bblock_hsmem p ⤳ hd → ∀ ge, smem_model ge (bblock_smem p) hd
```

*The main function of the simulation test.* Let us now present the main function of the simulation test, called `bblock_simu_test` below[28]. First, it creates two memoizing functions `hC_term` and `hC_list_term` (specified in Figure 15) from the generic factory `hCons` (see Section A.4.2 for details). Then, it invokes the symbolic execution `bblock_hsmem` on each block. Notice that these two symbolic executions share the memoizing functions `hC_term` and `hC_list_term`, meaning that each term produced by one of the symbolic executions is represented by a unique pointer. The symbolic executions produce two symbolic memories `d1` and `d2`. We compare them using two auxiliary functions specified in Figure 18. Hence, `(Dict.eq_test d1.(hpost) d2.(hpost))` compares whether each pseudo-register is assigned to the *same* term w.r.t pointer equality in both symbolic memories. Finally, `(test_list_incl d2.(hpre) d1.(hpre))` compares whether each term of `d2.(hpre)` is also present in `d1.(hpre)`: i.e. whether all potential failures of `d2` are potential failures of `d1`. Again, in `test_list_incl`, terms are compared for pointer equality. Let us note that `test_list_incl` is itself efficiently implemented (with a linear execution time), by using an untrusted OCaml oracle with a hash-table. More precisely, the formal proof of `test_list_incl_correct` relies on a property derived by parametricity from the polymorphic type of this untrusted oracle. This applies a "*theorems for free*" technique described in [Boulmé and Vandendorpe 2019].

```
Definition bblock_simu_test (p1 p2: bblock): ?? bool :=
  DO hC_term ↜ hCons {|hash_eq:=term_hash_eq; hashing:=term_hashing; set_hid:=term_set_hid|};;
  DO hC_list_term ↜ hCons...(* omit a record of type [(hashP list_hterm)] *)
  DO d1 ↜ bblock_hsmem hC_term hC_list_term p1;;
  DO d2 ↜ bblock_hsmem hC_term hC_list_term p2;;
  DO b ↜ Dict.eq_test d1.(hpost) d2.(hpost);;
  if b then test_list_incl d2.(hpre) d1.(hpre);;
       else RET false

Lemma bblock_simu_test_correct (p1 p2 : bblock):
   bblock_simu_test reduce p1 p2 ⤳ true → ∀ ge, bblock_simu ge p1 p2
```

The proof of `bblock_simu_test_correct` directly results from the conjunction of the two correctness properties of Figure 18 with `bblock_smem_correct` and `bblock_hsmem_correct`.

```
Definition Dict.eq_test: ∀ {A}, Dict.t A → Dict.t A → ?? bool
Lemma Dict.eq_test_correct A (d1 d2 : Dict.t A): Dict.eq_test d1 d2 ⤳ true →
  ∀ x:R.t, Dict.get d1 x = Dict.get d2 x

Definition test_list_incl: ∀ {A}, list A → list A → ?? bool
Lemma test_list_incl_correct A (l1 l2:list A): test_list_incl l1 l2 ⤳ true →
  ∀ t:A, List.In t l1 → List.In t l2
```

Fig. 18. Formal specification of the two auxiliary functions used by the simulation test

[28]The code of `bblock_simu_test` has been largely simplified, by omitting the complex machinery which is necessary to produce an understandable trace for CompCert developers in the event of a negative answer.