Certified and efficient instruction scheduling. Application to interlocked VLIW processors.

Cyril Six, Sylvain Boulmé, David Monniaux

Certified and efficient instruction scheduling

Application to interlocked VLIW processors

CYRIL SIX, SYLVAIN BOULMÉ, and DAVID MONNIAUX

CompCert is a moderately optimizing C compiler with a formal, machine-checked, proof of correctness: after successful compilation, the assembly code has a behavior faithful to the source code. Previously, it only supported target instruction sets with sequential semantics, and did not attempt reordering instructions for optimization.

We present here a CompCert backend for a VLIW core (*i.e.* with explicit parallelism at the instruction level), the first CompCert backend providing scalable and efficient instruction scheduling. Furthermore, its highly modular implementation can be easily adapted to other VLIW or non-VLIW pipelined processors.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Retargetable compilers**; • **Theory of computation** → *Scheduling algorithms*; • **General and reference** → *Performance*; • **Computer systems organization** → Superscalar architectures; Very long instruction word.

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

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; yet it hinders performance on in-order processors, especially superscalar ones (multiple execution units able to execute 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 *bundle* of instructions is an 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 predictable 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 [1981]; Rau et al. [1982], followed by Feautrier [1991]; Lam [1988] and the Multiflow compiler

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

Authors' address: Cyril Six, Cyril.Six@kalray.eu, Cyril.Six@univ-grenoble-alpes.fr; Sylvain Boulmé, Sylvain.Boulme@univ-grenoble-alpes.fr; David Monniaux, David.Monniaux@univ-grenoble-alpes.fr.

[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, that it preserves the program semantics.

This paper presents an extension of CompCert with certified assembly generation for an interlocked VLIW processor (Kalray KVX 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, in particular to multiple-issue in-order cores (e.g. ARM Cortex A-53). Furthermore, we think general insights can be gained from our experiment, beyond the issue of instruction scheduling, such as efficiently certifying the output of compiler optimization phases by certified symbolic execution with hash-consing.

1.1 Overview of the Kalray KVX VLIW processor

The Kalray KVX core implements a 6-issue Fisher-style VLIW architecture [Fisher et al. 2005] (partial predication, dismissible loads, no rotating registers). It sequentially executes blocks of instructions called *bundles*, with parallel execution within them.

Bundles. A *bundle* is a block of instructions that are to be issued into 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 ; ; (Fig. 1). Compilers must ensure that each bundle does not require more resources than available—e.g., the KVX 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 KVX, bundles are executed through a 8-stage interlocked pipeline: the first stage prefetches the next bundle (PF stage), the second decodes it (ID stage), the third reads the registers (RR stage), then the last five stages (E1 through E5) perform the actual computation and write to the destination registers; depending on the instructions the writes occur sooner or later (e.g., an addition takes fewer stages than a multiplication). 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 until the register gets its result (Fig. 1).[3]

Processor implementations can be divided into: *out-of-order* processors (e.g. modern x86), which may locally re-schedule instructions to limit stalls; [4] *in-order*, which execute the instructions exactly in the order of the assembly code. On an in-order processor, an optimizing compiler should provide an efficient schedule; this is all the more important if the processor is multiple-issue or VLIW, since a single stalling cycle could have been used for executing multiple instructions.

1.2 Modular design of the CompCert 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*

[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 KVX, 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	E4+E5
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. Stage PF (not shown here) happens just before the ID stage.

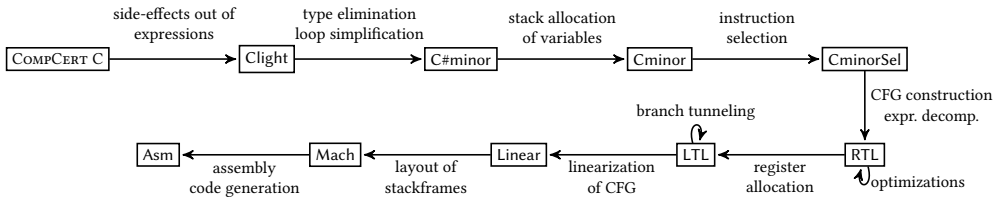

Fig. 2. The intermediate languages of CompCert.

(IR)—called Cminor—that is independent of the target machine [Blazy et al. 2006]. Then, a *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 Fig. 2, CompCert 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 introduce 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 and bundles instructions at the very end of the backend, after register allocation.

1.3 Porting CompCert to a VLIW architecture

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

- *How to represent bundles in CompCert?* The existing Asm languages are sequential. We need to define a parallel semantics within bundles for our VLIW processor.
- *How to include a scheduling pass within CompCert?* On in-order processors, particularly those capable of executing multiple instructions at the same time, it is of paramount importance for execution speed that 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, to minimize stalling. For instance, in Fig. 1, this pass could schedule B6 before B3. The task of grouping instructions into bundles (bundling) on a VLIW

processor is usually performed by a postpass scheduler: instructions are in the same bundle if they are scheduled in the same time slot.

Certified scheduling was already explored by Tristan and Leroy [2008], who extended CompCert with a certified postpass list-scheduler, split into (i) an untrusted oracle written in OCaml that computes a scheduling for each *basic block*[5] in order to minimize pipeline stalls (ii) a checker—certified in Coq—that verifies the oracle results. We identified three issues with their approach.

Firstly, their scheduling operates at the Mach level, simpler than Asm. Since some aspects (stack and control flow handling) are only detailed in the Mach to Asm pass, a model of latencies and pipeline use at the Mach level cannot be accurate. Furthermore, our scheduling needs the actual Asm instructions to construct well-formed bundles.

Secondly, their checker has exponential complexity w.r.t. the size of basic blocks, making it slow or even impractical as the number of instructions within a basic block grows. We thus needed to devise new algorithms that scale much better but that can still be proved correct in Coq.

Finally, Tristan and Leroy's proof only holds for a preliminary version of CompCert where non-terminating executions were not modeled (see Tristan [2009, Section 3.5.3]). CompCert now models diverging executions as well. This makes the semantic preservation proof more complex.

Tristan and Leroy's approach neither was integrated into CompCert, nor, to our best knowledge, seriously evaluated experimentally, probably due to prohibitive compile-times.

1.4 Contributions

Our main contribution is a *scalable*, *certified* and *highly modular* scheduler with bundling, combining an untrusted scheduling oracle with a verified scheduling checker. Both the oracle and checker are highly *generic*; we instantiated them with the instruction set and (micro-)architecture of the Kalray KVX core. We evaluated experimentally both scalability and the quality of the produced code.

Our solution solves the issues in Tristan and Leroy [2008]. Our certified scheduler is made of:

(1) An oracle, written in OCaml, producing a sequence of bundles for each basic block. We implemented a greedy list-scheduler with a priority heuristic based on latencies. [6]

(2) A generic certified scheduling checker, written in Coq, with a proof of semantic preservation, implementing 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.
- Verifying that, for each bundle, the sequential and parallel executions have the same semantics. This reduces to checking that each bundle never uses a register after writing to it.

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. The core of our certified scheduler is independent from the instruction set: it can be reused for other processors, or other IRs (e.g. in another prepass intrablock scheduling).

We compiled various software packages with our version of CompCert for the KVX, including our scheduler[7] and compared their execution time to that of the same software compiled with the reference compiler for the KVX (versions of the GNU C Compiler supplied by the chip designers).

[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).

[6] We briefly evaluate this choice compared to an optimal scheduler based on integer linear programming in Appendix D.

[7] Our full source code is available on https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx.

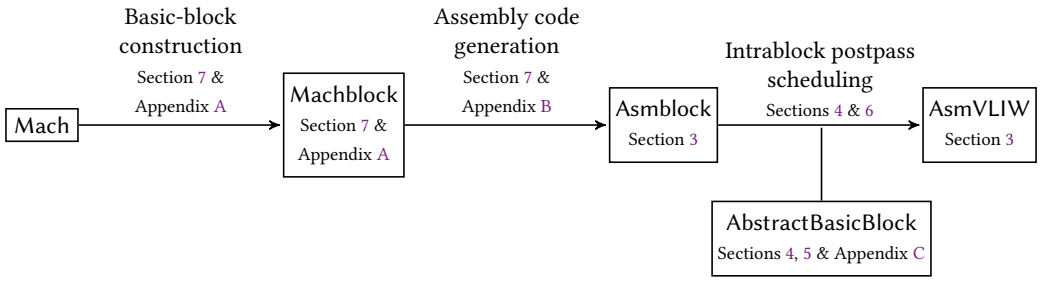

Fig. 3. Architecture of our solution in CompCert (and of this paper)

1.5 Related work

Our CompCert backend for the KV3 processor initially benefited from that of Barany [2018] for the KV2, though Barany's backend generates only one instruction per bundle, and does not model the VLIW semantics. He also faced the challenge of representing pairs of 32-bit registers in CompCert for handling 64-bit floating-point values on the KV2. The KV3 natively has 64-bit registers.

Scheduling with timing and resource constraints is a classical problem; [Micheli 1994, §5.4]. Ample work exists on scheduling for VLIW processors [Dupont de Dinechin 2004]—but with no machine-checked proof of correctness of the compiler implementation.

Tristan and Leroy [2008]; Tristan [2009]; Tristan and Leroy [2010] studied more advanced scheduling techniques, including software pipelining, which are particularly suited to prepass optimization. We plan to consider these in future work.

Schulte et al. apply 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 (w.r.t. their cost model) on medium-sized functions. They consider a wider class of optimizations than we do, but they do not provide any machine-verified proof of correctness.

Our CompCert optimization at assembly level postdates the one of Mullen et al. [2016]. They target x86-32 with more advanced peephole optimizations than we do: modulo register liveness and considering pointers as 32 bit-integers. But, they do not tackle instruction scheduling. Moreover, we show how to transfer basic blocks to the assembly, whereas they shortcut this issue by requiring unchecked assumptions (entitled "*calling conventions*") on the generated assembly.

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. It formalizes the assembly semantics of our VLIW target: the bundles are defined as basic blocks with a parallel execution inside.

Our postpass scheduler is formalized as a transformation on basic blocks. It takes as input our Asmblock IR, which shares its syntax with AsmVLIW, but with sequential execution inside basic blocks instead of a parallel one; these two languages are described in Section 3. Our postpass scheduling itself is described in Sections 4 to 6. Before that, Section 2 recalls the necessary details of CompCert. Finally, Section 8 presents experimental evaluations of our backend.

In summary, we extended the CompCert architecture with the passes shown in Fig. 3. The preliminary stage of our backend constructs the Asmblock program from the Mach program. As Section 7 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—whose syntax reflects the basic block structure of Mach programs, and then translates each basic block

from Machblock to obtain an Asmblock program. 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 Sect. 4, while those on the actual intra-block scheduling problem solved by our oracle are given in Sect. 6. The core of our scheduling checker—involving symbolic evaluation of basic blocks with hash-consing—operates on a new auxiliary IR, called AbstractBasicBlock, presented in Sect. 4 and 5, but further detailed in Appendix C.

2 COMPCERT BACKEND SEMANTICS

We present the correctness property of CompCert passes, then the Asm IR of existing backends.

2.1 Correctness of Compilation Passes

In CompCert [Leroy 2009b], the semantics of a program P consists of predicates for describing initial and final states, as well as a predicate $S \xrightarrow{t} S'$ (usually named `step`) indicating if *one* execution step *can* run from state S to state S' by generating a *trace* t—where t is either a single *observable event* (*e.g.* an external call or an access to a volatile variable) or ϵ (absence of observable event).

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 the observational behaviors of P_2 are the observable behaviors of P_1. Hence, this property involves a high-level notion of *observable behavior*, formalized by Leroy [2009a,b].

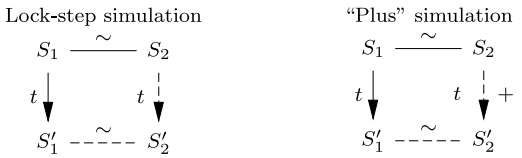

Fig. 4. Examples of simulation diagrams in CompCert

In order to simplify correctness proofs of its successive passes (Fig. 2), CompCert uses an alternative definition for the correctness. One of them is the *forward simulation* applicable on passes between *deterministic* languages. In its simplest form (*lockstep simulation*), given a relation ($\sim$) matching states between P_1 and P_2, a forward simulation involves proving that:

- The initial (resp. final) states of P_1 match those of P_2
- Given two matching states, if P_1 steps to a state S_1', then P_2 steps to a state S_2' matching S_1'.

Another form of forward simulation is the *plus simulation*, where the target program can execute one or more steps instead of just one. Fig. 4 schematizes both simulations.

2.2 The Asm IR

CompCert defines one Asm language per target processor. An Asm program consists of functions, each with a function body. Asm states are 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.

In each CompCert backend the instruction semantics are modeled by an `exec_instr` function which takes as argument an instruction i, a register state rs and a memory state m, and returns the next state (rs', m') given by the execution of i; or the special state `Stuck` if the execution failed. Here are some instructions that can be modeled in Asm:

- `Pcall(s)`: calls the function from symbol *s* (saves PC into RA and then sets PC to the address of *s*);
- `Paddw(rd, r1, r2)`: writes in *rd* the result of the addition of the lower 32-bits of *r1* and *r2*;
- `Pcb(bt, r, l)`: evaluates the test *bt* on register *r*—if it evaluates to true, PC jumps to the label *l*;
- `Pigoto(r)`: PC jumps to the value of the register *r*.

Most of the instructions modeled in Asm directly correspond to the actual instructions of the processor. There can also be a few pseudo-instructions like `Pallocframe` or builtins, which are specified in Asm, then replaced by a sequence of instructions in a non-certified part of CompCert. Due to distinctions in immediate vs register operand, word vs doubleword operand size etc. there are as many as 193 instructions to be modeled in the KVX processor.

3 SEMANTICS FOR A VLIW ASSEMBLY LANGUAGE

The Asm language of our target processor introduces a syntax and semantics for *bundles* of instructions. Bundles can be seen as a special case of *basic blocks*: 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*, such as conditional branching to 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 interesting for scheduling optimizations: reordering the sequence of *basic instructions* in a basic block without changing its (local) blockstep does not change the (global) semantics of the surrounding program.

We provide two such blockstep semantics which only differ on how they combine instructions within basic blocks: an IR with sequential semantics called Asmblock, and one with parallel semantics called AsmVLIW. Our backend first builds an Asmblock program from a Mach program, by detecting syntactically its basic block structure (Section 7). Then, each basic block is split into bundles of the AsmVLIW IR (Sections 4 & 6). 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

We first split the instructions into two syntactic categories: the basic ones and the control flow ones. 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 }
```

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 basic block into a sequence of smaller ones. In order to avoid infinite stuttering (due to incrementing PC by 0), we further require 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. A state in AsmVLIW and Asmblock is expressed the same way as in Asm: it is either a pair (rs, m) where *rs* (register state) maps registers to values and *m* (memory) maps addresses to values, or a `Stuck` in case of failure (e.g. division by zero). Hence, executing a single instruction in our semantics gives an `outcome` defined as either a (`Next rs m`) state, or a `Stuck` execution. Then, each blockstep 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 that block. Finally, that blockstep either returns the next state 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. The pipeline stages of our interlocked VLIW processor can be abstracted into:

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 the reading stage.
Writing stage the results are written to the registers.[8]

Our processor stalls at a reading stage whenever the result is not yet available. The exact number of cycles required to compute a value thus only has an impact on the performance: our formal semantics abstracts away the computing stages and only considers the reading and writing stages.

Reads are always deterministic: they happen at the start of the execution of our bundle. However, the write order is not necessarily deterministic, *e.g.* if the same register is written twice within the same bundle. We first introduce a deterministic semantics where the writes are performed in the order in which they appear in the bundle. For instance, the bundle "$R_0 := 1; R_0 := 2$" assigns 2 to R_0, in our in-order semantics. The actual 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. Fig. 5 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.
- The function (`parexec_wio` ... *rsr mr*) is the composition of the basic and control steps.

For example, the bundle "$R_0 := R_1$; $R_1 := R_0$; `jump@toto`" runs over an initial register state $rsw_0 = rsr$ in 3 steps:

(1) "$R_0 := R_1$" leads to $rsw_1 = rsw_0[R_0 \leftarrow rsr[R_1]]$
(2) "$R_1 := R_0$" leads to $rsw_2 = rsw_1[R_1 \leftarrow rsr[R_0]]$
(3) "`jump @toto`" leads to $rsw_3 = rsw_2[\mathtt{PC} \leftarrow \mathtt{@toto}]$

The final register state of the parallel in-order blockstep is

$$rsw_3 = rsr[R_0 \leftarrow rsr[R_1];\ R_1 \leftarrow rsr[R_0];\ \mathtt{PC} \leftarrow \mathtt{@toto}]$$

As expected, this bundle swaps the contents of R_0 and R_1.

The in-order parallel execution of a list of basic instructions is formally defined in Coq by the following function, where "`NEXT` *rs*,*m* ←*e*₁ `IN` *e*₂" is a notation for:

"`match` *e*₁ `with Next` *rs m*⇒ *e*₂ | _ ⇒ `Stuck end`"

```
Fixpoint parexec_wio_body bdy 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
```

[8] This includes the Program Counter register, which is updated at the end of each bundle execution.

The in-order parallel execution of a block (defined 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 `sz` is the offset by which PC is incremented in the absence of a control-flow instruction.

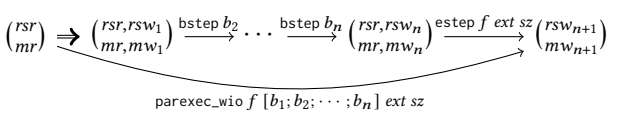

Fig. 5. Parallel in-order blockstep

```
Definition parexec_wio f bdy ext sz 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. The in-order parallel semantics defined above is not very representative of how a VLIW processor works, since concurrent writes may happen in any order. This issue is solved by relation (`parexec_bblock` f b rs m o), which holds if 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 b rs m o: Prop :=
∃ bdy1 bdy2, Sorting.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, the execution takes any permutation of the body and splits this permutation into two parts `bdy1` and `bdy2`. It first executes `bdy1`, then the control flow instruction, then `bdy2`. While PC is possibly written before the end of the execution of the bundle, the effect on the control-flow takes place when the next bundle is fetched, reflecting the behavior detailed in Footnote 8.

This semantics gives a fair abstraction of the actual VLIW processor. However, the proof of semantic preservation of CompCert requires the target language to be deterministic. Consequently, we force our backend to emit bundles that have the same semantics irrespectively of the order of writes. This is formalized by the relation below:

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

Given (rs', m'), the above holds only if all possible outcomes o satisfying (`parexec_bblock` f b rs m o) are exactly (`Next` rs' m'); that is, it only holds if (rs', m') is the only possible outcome. We then use the `det_parexec` relation to express the `step` of our AsmVLIW semantics: if `det_parexec` does not hold then it is not possible to construct a `step`.

3.3 Sequential semantics in Asmblock

Asmblock is the IR just before AsmVLIW: instructions are grouped in basic blocks. These are not re-ordered and split into bundles yet: execution within a block is sequential.

The given sequential semantics of a basic block, called `exec_bblock` below, is similar to the semantics of a single instruction in other Asm representations of CompCert. Just like AsmVLIW, its execution first runs the body and then runs the control flow. Our sequential semantics of single instructions 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 (Fig. 3). This pass has two goals: (1) reordering the instructions in each basic block to minimize the stalls; (2) grouping into 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 the untrusted oracle without modifying our Coq proofs.

The verifiers check semantic correctness only. If some generated bundle exceeds resource constraints, it will be rejected by the assembler.

Scheduling is performed block by block from the Asmblock program. As depicted in Fig. 6, 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 in Section 6). 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 stops the compilation if the verifier returned an error.

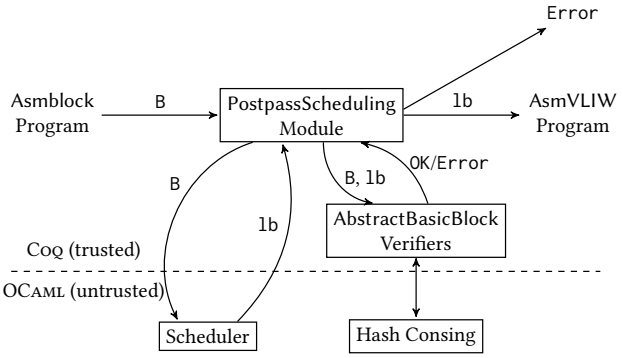

Fig. 6. Certified Scheduling from Untrusted Oracles

In Coq, the scheduler is declared as a function[9] splitting a basic block "`B:bblock`" into a value that is then transformed into a sequence of bundles "`lb: list bblock`".[10]

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

The proof of the pass uses a *"Plus" simulation* (Fig. 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, which proves 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. We first introduce AbstractBasicBlock, a helper IR that we use for both simulations. Then we describe the "parallelizability checker" ensuring a lockstep simulation (2). Finally, we describe the "simulation checker" ensuring a plus simulation (1).

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

[10] 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.

4.1 AbstractBasicBlock IR

The core of our verifiers lies in the AbstractBasicBlock specific representation. AbstractBasicBlock provides a simplified syntax, in which the registers that are read or assigned by each instruction appear syntactically. The memory is encoded by a pseudo-register denoted by m. [11] We illustrate in Example 4.1 how we have translated some instructions into AbstractBasicBlock assignments.

Example 4.1 (Syntax of AbstractBasicBlock). Examples of translations into AbstractBasicBlock:
(1) the addition of two registers r_2 and r_3 into r_1 is written "$r_1 := \texttt{add}[r_2, r_3]$";
(2) the load into register r_1 of memory address "$ofs[r_2]$" (representing $ofs+r_2$ where ofs is an integer constant) is written "$r_1 := (\texttt{load}\ ofs)[m, r_2]$";
(3) the store of register r_1 into memory address "$ofs[r_2]$" is written "$m := (\texttt{store}\ ofs)[m, r_1, r_2]$".

AbstractBasicBlock is dedicated to intra-block analyses. In AbstractBasicBlock, a block is encoded as a list of assignments, meant to be executed either sequentially or in parallel depending on the semantics. Each assignment involves an expression, composed of operations and registers.

The syntax and semantics of AbstractBasicBlock are generic, and have to be instantiated with the right parameters for each backend. Though we only did that task with the KV3 backend, we believe it could easily be extended to other backends, and possibly other IRs as well.

AbstractBasicBlock provides a convenient abstraction over assembly instructions like most IR of CompCert except Asm: it leverages the hundreds of AsmVLIW instructions into a single unified representation. Compared to the Mach IR—used in the initial approach of [Tristan and Leroy 2008]— AbstractBasicBlock is more low-level: it allows to represent instructions that are not present at the Mach level, like those presented in Section 4.6. It is also more abstract: there is no particular distinction between basic and control-flow instructions, the latter being represented as an assigment of a regular pseudo-register PC. The simplicity of its formal semantics (given in Section 5) is probably the key design point that allows us to program and prove efficiently the verifiers.

The following sections summarizes how we used AbstractBasicBlock to certify the scheduling of Asmblock/AsmVLIW. See Section 5 for a more detailed presentation.

4.2 Parallelizability Checker

To check whether the sequential and parallel semantics of a certain bundle give the same result, we translate the bundle into AbstractBasicBlock with a `trans_block` function, then we use the `is_parallelizable` function from AbstractBasicBlock.

Function `is_parallelizable` analyzes the sequence of AbstractBasicBlock assignments and checks that no pseudo-register is read or rewritten after being written once. For example, blocks "$r_1 := r_2$; $r_3 := r_2$" 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 C.

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` is proven to *bisimulate* the input block—both for the sequential and the parallel semantics. This bisimulation is also useful for the simulation checker described next section.

Proving the previously mentioned forward simulation (2) then relies on proving the following lemma `bblock_para_check_correct`, which is proven by using the above bisimulation property.

```
Definition bblock_para_check bundle: bool := is_parallelizable (trans_block bundle)
Lemma bblock_para_check_correct ge f bundle rs m rs' m': bblock_para_check bundle = true →
```

[11] This encoding should be refined in order to introduce alias analysis

```
exec_bblock ge f bundle rs m = Next rs' m' → det_parexec ge f bundle rs m rs' m'
```

4.3 Verifying Intrablock Reordering

In order to reason on reordering, we define a concatenation predicate: "`is_concat tb lb`" means that the `bblock tb` is the *concatenation* of the list of bundles `lb`. Formally, `lb` must be non-empty, only its head may have a non-empty header, only its tail may have a control flow instruction, and `tb.(body)` must be the concatenation of all the bodies of `lb`, and the header (resp. exit) of the head (resp. tail) of `lb` must correspond to the header (resp. exit) of `tb`.

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`, 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
```

The forward simulation (1) reduces to proving the correctness of our `verified_schedule` function on each basic block, as formalized by this property:

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

In detail, `(verified_schedule B)` calls our untrusted scheduler `(schedule B)` and then builds the sequence of bundles `lb` as well as their concatenation `tb`. Then, it checks each bundle for parallelizability with the `bblock_para_check` function described in the previous section. Finally, it calls a function `bblock_simub : bblock → bblock → bool` checking whether `tb` simulates `B`.[12]

Function `bblock_simub` is composed of two steps. First, each basic block is compiled (through the `trans_block` function mentioned in Section 4.2) into a sequence of AbstractBasicBlock assignments. Second, like in [Tristan and Leroy 2008], the simulation test symbolically executes each AbstractBasicBlock code and compares the resulting *symbolic memories* (Fig. 7). A symbolic memory roughly corresponds to a parallel assignment 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 its initial value.

Example 4.2 (Equivalence of symbolic memories). Let us consider the two blocks B_1 and B_2 below:
(B_1) $r_1 := r_1 + r_2;\ r_3 := \mathtt{load}[m, r_2];\ r_1 := r_1 + r_3$
(B_2) $r_3 := \mathtt{load}[m, r_2];\ r_1 := r_1 + r_2;\ r_1 := r_1 + r_3$
They are both equivalent to this parallel assignment:

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

Indeed, B_1 and B_2 bisimulate (they simulate each other).

Collecting only the final term associated with each pseudo-register is actually incorrect: 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[13] as pictured on Fig 7.

[12] More precisely, if the result of "`bblock_simub B tb`" is `true`, then "`bblock_simu ge f B tb`" holds.

[13] A "*symbolic memory*" corresponds here to a kind of parallel assignment, and does not represent a memory (despite the terminology). This confusion comes from the fact that "*symbolic execution*" as introduced by King [1976] refers to *how to* compute "symbolic memories" and does not refer to *what* they represent. Indeed, in our case, "*symbolic execution*" computes this alternative representation of each block by mimicking the sequential execution. See Sect. 5 for a formal overview.

Example 4.3 (Simulation on symbolic memories). Consider:
(B_1) $r_1 \coloneqq r_1 + r_2;\ r_3 \coloneqq \texttt{load}[m, r_2];\ r_3 \coloneqq r_1;\ r_1 \coloneqq r_1 + r_3$
(B_1) $r_3 \coloneqq r_1 + r_2;\ r_1 \coloneqq r_3 + r_3$
Both B_1 and B_2 lead to the same parallel assignment:

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

However, B_1 is simulated by B_2 whereas the converse is not true. This is because the memory access in B_1 may cause its execution to fail, whereas this failure cannot occur in B_2. Thus, the symbolic memory of B_1 should contain the term "$\texttt{load}[m, r_2]$" as a potential failure. We say that *a symbolic memory d_1 is simulated by a symbolic memory 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 Section 5 for the formal definitions.

As illustrated in Examples 4.2 and 4.3, the 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.

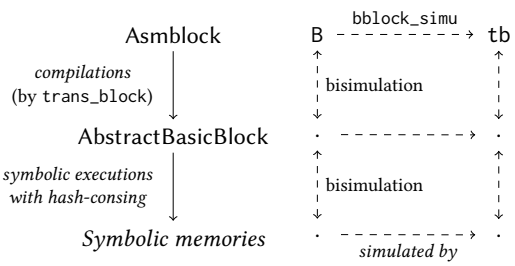

Fig. 7. Diagram of bblock_simub correctness

4.4 Generic and Verified Hash-Consing

We give here a brief overview of the hash-consing mechanism. More details on AbstractBasicBlock and its hash-consing mechanism are described in Section 5.

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.

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 are isomorphic, or aborts the computation if it cannot ensure they are. This check is kept constant-time by using OCaml *pointer equality* to compare *already memoized* subterms: $f(t_1, \ldots, t_m)$ and $g(u_1, \ldots, u_n)$ are tested for isomorphism by checking that the head symbols f and g are identical, the numbers of arguments m and n are the same, and for all i, t_i and u_i are the same pointer. 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.[14]

We solve this issue by using the *pointer equality* from the Impure library of [Boulmé and Vandendorpe 2019], which embeds OCaml functions in Coq through a *non-deterministic monad*. In particular, it represents OCaml *pointer equality* as a non-deterministic function.[15] We then use the axiom from Impure stating that, if pointer equality returns `true`, then the two values are (structurally) equal.

Here, "`??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`". In other words, even if "`phys_eq x y ⇝ true`" and "`phys_eq x y ⇝ b`", then we cannot conclude that "`b=true`" (we could also have "`b=false`"). The Impure library does not even assume that "`∀ x, phys_eq x x ⇝ true`".

```
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
```

These axioms are proved to be non-contradictory w.r.t. the Coq logic. They express a correctness property of OCaml physical equality, from which we derive efficient and formally verified hash-consing.

4.5 Peephole optimization

We have expressed the semantics of assembly instructions by decomposing them into atomic operations, which we used to define the symbolic execution. This means that distinct groups of instructions that decompose into the same atomic operations are considered equivalent. We exploit this to implement *peephole optimization*: local replacement of groups of instructions by faster ones, prior to scheduling. On Fig. 6, this untrusted optimization is performed by a preliminary pass of the "Scheduler" oracle, and thus dynamically checked by our `bblock_simub` trusted verifier.

Currently our only peephole optimizations are the replacement of two (respectively, four) store instructions from an aligned group of double-word (64-bit) registers (e.g. `$r18`, `$r19`) to a succession of offsets from the same base register (e.g. `8[$r12]`, `16[$r12]`) by a single quadruple-word (respectively, octuple-word) store instruction; and the same with loads. In practice, this quickens register spilling and restoring sequences, such as those around function calls: CompCert spills and restores registers in ascending order, whereas in general it would only be sheer chance that register allocation placed data that is consecutive in memory into a aligned group of consecutive registers. Similar optimizations could be conducted on the ARM (multiple loads and stores) and AAarch64 (loads and stores of arbitrary pairs of registers) architectures.

4.6 Atomic Sequences of Assignments in AbstractBasicBlock

In the previous examples of this paper, each AsmVLIW instruction is translated to a single assignment of AbstractBasicBlock. However, in many cases (extended-word accesses of Sect. 4.5, control-flow instructions, frame-handling pseudo-instructions, etc), AsmVLIW instructions cannot correspond to a single assignment: they are rather translated to AbstractBasicBlock as an *atomic*

[14] Hence, if we model pointer equality (OCaml's ==) as an infix function "`phys_eq:∀ {A}, A → A → bool`", then we are able to prove this wrong proposition (when `x` and `y` are structurally equals but are distinct pointers):

`y=x ∧ (phys_eq x y)=false ∧ (phys_eq x x)=true → false=true`

[15] In the Coq logic, two occurrences of variable "`x`" may correspond to two distinct objects in memory (e.g. after substituting `y=x` in "`P x y`"). This is why `phys_eq` must appear as "non-deterministic" to Coq's eyes.

sequence of assignments (ASA). A form of parallelism may occur in such a sequence through the special operator $\texttt{Old}(e)$ where e is an expression, meaning that the evaluation of e occurs in the initial state of the ASA. And an AsmVLIW bundle (resp. an Asmblock basic-block) actually corresponds to the parallel (resp. sequential) composition of a list of such ASA.

For example, the parallel load from a 128-bit memory word involves two contiguous (and adequately aligned) destination registers d_0 and d_1 that are distinct from each other by construction—but not necessarily from the base address register a. This parallel load is thus translated into the following ASA, that emulates a parallel assignment of d_0 and d_1, even if $a = d_0$:

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

See Section 5.1 for a formal definition of atomic sequences of assignments.

5 FORMALIZING A SYMBOLIC EXECUTION WITH HASH-CONSING

This section sketches how the symbolic execution with hash-consing of AbstractBasicBlock is formalized. This requires to first present the formal sequential semantics of this IR (Sect. 5.1). See Appendix C for more details on AbstractBasicBlock (in particular its parallel semantics and the formalization of the parallelizability test).

5.1 Syntax and Sequential Semantics of AbstractBasicBlock

We sketch below the formal definition of AbstractBasicBlock syntax and its sequential semantics. Its syntax is parametrized by a type `R.t` of pseudo-registers (positive integers in practice) and a type `op` of operators. Its semantics is parametrized by a type `value` of values, a type `genv` for global environments, and a function `op_eval` evaluating operators to an "`option value`".

Let us introduce the semantics in a top-down (i.e. backward) style. Function `run` defines the semantics of a `bblock` by sequentially iterating over the execution of instructions, called `inst_run`. 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`). It invokes the evaluation of an expression, called `exp_eval`. Similarly, the `exp_eval` function takes two memory states as input: the current memory is replaced by `old` when entering under the `Old` operator.

```
(* Abstract Syntax parametrized by type R.t of registers and op of operators *)
Inductive exp := PReg(x:R.t) | Op (o:op) (le:list_exp) | Old (e:exp)    with list_exp :=...
Definition inst := list (R.t * exp). (* inst = atomic sequence of assignments *)
Definition bblock := list inst
(* Semantical parameters and auxiliary definitions *)
Parameter value genv: Type
Parameter op_eval: genv → op → list value → option value
Definition mem := R.t → value. (* concrete memories *)
Definition assign (m:mem) (x:R.t) (v:value): mem := fun y ⇒ if R.eq_dec x y then v else m y
(* Sequential Semantics *)
Fixpoint exp_eval (ge: genv) (e: exp) (m old: mem): option value :=
 match e with PReg x ⇒ Some (m x) | Old e ⇒ exp_eval ge e old old
 | Op o le ⇒ SOME lv ← list_exp_eval ge le m old IN op_eval ge o lv
 end with list_exp_eval ge (le: list_exp) (m old: mem): option (list value) :=...
Fixpoint inst_run (ge: genv) (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
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
```

5.2 Sketch of our verified hash-consed terms

Using King [1976] terminology, a *symbolic value* is a kind of term. In such terms, a pseudo-register represents its value in the *initial memory* of block execution. Hence, the structure of our terms is similar to type `exp` without the `Old` operator. Below, we define an inductive type `hterm` (together with `list_hterm`) for hash-consed terms: it adds an hash-tag information (of type `hashcode`) to

each constructor. These hash-tags are computed by our external memoizing factory and intend to identify each hash-consed term as a unique integer. Actually, a misuse on these hash-tags will not affect the correctness of our verifier but only its performance (including its success). Indeed, it is ignored by the formal semantics of `hterm`, called `ht_eval`.

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

Our symbolic execution with hash-consed terms is parametrized by two memoizing functions `hC_term` and `hC_list_term`. Indeed, our simulation test ultimately performs two symbolic executions, one for each block: these two symbolic executions share the same memoizing functions, leading to an efficient comparison of the symbolic memories through pointer equality. The correctness property associated with each of these functions is directly derived from our generic certified memoization factory (which is detailed in Appendix C.4.2). Here is the specification of `hC_term` using notations of Impure (`hC_list_term` is similar).

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

These memoizing functions are invoked 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 memoizing oracle): `hC_term` replaces this special hash-tag by the one actually allocated for this term.

```
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, ht_eval ge t m = (SOME v ← lht_eval ge l m IN op_eval ge o v)
```

5.3 An abstract Model of our Simulation Test (without hash-consing)

The formal proof of our simulation test is decomposed into two parts using a *data-refinement* style. In the first part, we define an abstract model of the symbolic execution and the simulation test (without hash-consing): this allows to reduce the simulation of two basic blocks for their sequential semantics to the simulation of their symbolic memories computed through an abstract definition of the symbolic execution. In a second part, sketched in Section 5.4, this abstract symbolic execution is refined using concrete data-structures and in particular hash-consing.

The symbolic execution of a block is modelled as a function `bblock_smem: bblock → smem`, where a symbolic memory of type `smem` is abstractly modelled by 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. For the sake of this presentation, we model terms with type `hterm`, but without real hash-consing (all hash-tags are set to `unknown_hid`).

```
Record smem:= {pre: genv → mem → Prop; post: R.t → hterm}. (* abstract symbolic memories *)
```

Then, the bisimulation property between symbolic and sequential execution is expressed by:

```
Lemma bblock_smem_correct p d: bblock_smem p = d →
  ∀ m m', run ge p m=Some m' ↔ (d.(pre) ge m ∧ ∀ x, ht_eval ge (d.(post) x) m = Some (m' x))
```

This lemma allows to reduce the simulation of block executions to the simulation of symbolic memories, formalized by `smem_simu` below.

```
Definition smem_valid ge (d: smem) (m:mem): Prop :=
  d.(pre) ge m ∧ ∀ x, ht_eval ge (d.(post) x) m <> None
Definition smem_simu (d1 d2: smem): Prop := ∀ ge m, smem_valid ge d1 m →
```

```
(* initial symbolic memory *)
Definition smem_empty := {| pre:=(fun _ _ ⇒ True); post:=(fun x ⇒ Input x unknown_hid) |}
(* symbolic evaluation of the right-hand side of an assignment *)
Fixpoint exp_term (e: exp) (d old: smem) : hterm :=
 match e with PReg x ⇒ d.(post) x | Old e ⇒ exp_term e old old
 | Op o le ⇒ App o (list_exp_term le d old) unknown_hid
 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 ⇒ (t_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. 8. Basic operations of the symbolic execution in the abstract model

```
(* 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 Old e ⇒ exp_hterm e hod hod
 | PReg x ⇒ match Dict.get hd.(post) x with Some ht ⇒ RET ht
             | None ⇒ hInput x  (* smart constructor for Input *) end
 | Op o le ⇒ DO lt ↜ list_exp_hterm le hd hod;;
              hApp o lt (* smart constructor for App *)
 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 → ht_eval ge t m = ht_eval ge (exp_term e d od) m
(* effect of an assignment on the symbolic memory (naive wrt the actual implementation) *)
Definition hsmem_set (hd:hsmem) x (ht:hterm): ?? hsmem :=
  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 → ht_eval ge ht m = ht_eval ge t m) →
   hsmem_set hd x ht ⇝ hd' → smem_model ge (smem_set d x t) hd'
```

Fig. 9. Data-refinement of symbolic execution with hash-consing

```
  ( smem_valid ge d2 m ∧ ht_eval ge (d1.(post) x) m = ht_eval ge (d2.(post) x) m) )
Theorem bblock_smem_simu p1 p2: smem_simu (bblock_smem p1) (bblock_smem p2) →
  ∀ ge m, (run ge p1 m) <> None → (run ge p1 m) = (run ge p2 m)
```

Internally, as coined in the name of "symbolic execution" by King [1976], `bblock_smem` mimics `run` (the sequential execution of the block), by replacing operations on memories of type `mem` by operations on type `smem` given in Fig. 8. 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 (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 bisimulation of symbolic memories wrt the sequential execution, on each assignment.

5.4 Refining symbolic execution with hash-consed terms

We now refine the type `smem` into type `hsmem`. The latter involves a dictionary of type `(Dict.t hterm)` (positive maps in practice) associating pseudo-registers of type `R.t` to hash-consed terms. Type `hsmem` is related to `smem` (in a given environment `ge`) by relation `smem_model`.

```
(* 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) → ht_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 ⇒ ht_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 = ht_eval ge (d.(post) x) m
```

Fig. 9 provides an implementation of the operations of Fig. 8 that preserves the data-refinement relation `smem_model`. It uses the monadic operators provided by Impure: its unit noted "`RET _`" and its bind operator noted "`DO _ ⇜ _ ;; _`". The smart constructors building hash-consed terms are invoked by the `exp_hterm` (i.e. the evaluation of expressions on symbolic memories).

Then, the symbolic execution `bblock_hsmem : bblock → ??hsmem` invokes these operations on each assignment of the block. We prove that it refines `bblock_smem` from lemma of Fig. 9.

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

Finally, the main function of the simulation test (detailed in Appendix C.4.3) creates two memoizing functions `hC_term` and `hC_list_term` as presented Sect. 5.2. Then, it invokes the symbolic execution `bblock_hsmem` on each block. 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 thus two `hsmem` and we compare them efficiently using physical equality on hash-consed terms.

6 INTRABLOCK SCHEDULING ORACLE

The postpass schedule is computed by an untrusted oracle, which first invokes a processor-dependent frontend that turns the scheduling problem of a given basic-block into an optimization problem (Sec. 6.2), which is then solved by one of our processor-independent oracles: (1) one instruction per bundle (2) greedy bundling without reordering (3) *list scheduling* (default) (4) reduction to ILP (Integer Linear Programming; Appendix D), solved by an external tool (e.g., Gurobi).

6.1 Bundlers without reordering

In order to measure the performance impact of scheduling, we provide two simple backends without any reordering: the first one trivially issues one instruction per bundle, while the second one attempts to greedily "pack" successive instructions without altering the sequential semantics.

6.2 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 time slot to each instruction. These time slots 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 processor register, as well as the pseudo-register m, standing for the whole addressable memory. 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*: Instruction j reads from r, the next write to r is instruction j', then $t(j') - t(j) \geq 0$.

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

A resource usage constraint

$$\forall i \quad \sum_{j \mid t(j)=i} \mathbf{u}(K(j)) \leq \mathbf{r} \qquad (1)$$

Latency constraints of the kind (where $j' > j$):

$$t(j') - t(j) \geq \delta \qquad (2)$$

Fig. 10. Our family of scheduling problems

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 δ cycles after an instruction of kind k then it incurs no waiting for reading the output of the other instruction. In most cases, δ depends only on k, but there are "bypasses" for some k' with lower δ than for others. All these are mentioned in the processor documentation. For memory loads, we take the timing for a L1 cache hit.
The KVX processor is *interlocked*: 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 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 ("syllables") 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 of available resources $\mathbf{r} \in \mathbb{N}^m$ and, for each instruction kind k, 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 $\mathbf{r}$, as expressed by Inequality (1) in Fig. 10.

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

Finally, we introduce an extra time slot $t(n)$ 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.

Algorithm 1: Simplified view of our list scheduler

```
i := 0 ‖ Q := {0 ... n − 1}
while Q ≠ ∅ do
    R := ∅ ‖ a := r
    for j′ ∈ Q do
        ready := true
        for j —δ→ j′ ∈ G do
            if t(j) > j′ − δ then ready := false;
        if ready then R := R ∪ {j′};
    for j ∈ R (in descending l(j, n) order) do
        if a ≥ u(K(j)) then
            a := a − u(K(j)) ‖ Q :=
              Q \ {j} ‖ t(j) := i
    i := i + 1
```

Our scheduling problem is thus an instance of the system of inequalities in Fig. 10: a correct sequence of bundles using $t(n)$ cycles in total is directly built from any solution t.

```
0      make   $r3 = stringlit2
1      make   $r2 = stringlit1
2      compw.gtu $r11 = $r18, 65535
3      cmoved.weqz $r11? $r3 = $r2
4      ld $r0 = 952[$r19]
5      addd   $r1 = $r12, 40
6      ld $r2 = 0[$r19]
7      ld $r4 = 32[$r4]
8      call   TIFFErrorExt
```

Fig. 11. Example of a basic-block

```
   make   $r3 = stringlit_2
   make   $r2 = stringlit_1
   compw.gtu $r11 = $r18, 65535
;;
   cmoved.weqz $r11? $r3 = $r2
   ld $r0 = 952[$r19]
   addd   $r1 = $r12, 40
;;
   ld $r2 = 0[$r19]
;;
   ld $r4 = 32[$r4]
   call   TIFFErrorExt
```

Fig. 12. Its list scheduling

```
   compw.gtu $r11 = $r18, 65535
   ld $r4 = 32[$r4]
;;
   make   $r3 = stringlit_2
   make   $r2 = stringlit_1
   ld $r0 = 952[$r19]
;;
   cmoved.weqz $r11? $r3 = $r2
   addd   $r1 = $r12, 40
   ld $r2 = 0[$r19]
   call   TIFFErrorExt
```

Fig. 13. Its ILP scheduling

6.3 (Critical paths) List scheduler

Our default solver is 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 fast (quasi linear-time) and computes an optimal schedule in almost all practical cases.

We consider that time i starts 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 chains two ideas[16]:

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 cannot increase the makespan. Thus, at every step we can restrict the search to S maximal w.r.t. the inclusion ordering among the feasible S.

Critical path heuristic The question is then which S to consider if there are many of them, often the case for the first bundles of a block—since all instructions using only registers with their values at the start of the block can be scheduled in the first bundle.

Consider the (multi)graph G with an edge $j \xrightarrow{\delta} j'$ for each inequality (2). It 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 never backtracks. If the choice is non-optimal, it may miss a better solution. This happens on example of Figure 11 (sequential code shown, one line per bundle). The timing constraints of this examples are: $t_8 \geq$ all other times, $t_6 \geq t_3$ (write-after-read), $t_6 - t_1 \geq 1$ (write-after-write, WAW), $t_3 - t_0 \geq 1$ (WAW), $t_3 - t_1 \geq 1$ (read-after-write, RAW), $t_3 - t_2 \geq 1$ (RAW). The critical path lengths to t_8 are 1 for t_0, t_1, t_2, t_6 and 0 for t_3, t_4, t_5, t_7.

The list scheduler breaks ties between instructions 0, 1, 2, 6 according to their order in the original program and schedules instructions 0, 1, 2 at time slot 0, saturates the "maximum number of syllables" resource at this time slot and yields a schedule with a makespan of 4 (Fig. 12). In contrast, the optimal scheduler, using integer linear programming, yields a makespan of 3 (Fig. 13).

[16] This algorithm gives a simplified view of our implementation. The latter pre-computes all $l(j, n)$ by graph traversal. And, it avoids scanning 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.

7 BASIC BLOCK RECONSTRUCTION

We motivate and briefly present our solution to reconstruct basic blocks in CompCert, necessary for the later scheduling pass. More details can be found in appendices A and B.

7.1 Necessity of constructing basic blocks at the Mach level

Mach is the IR closest to assembly with a significant level of abstraction. It features 3-address code instructions, with generic instructions (such as `Mload`, `Mstore` and `Mop`) that are to be translated into their Asm architecture specific equivalents. The ABI (Application Binary Interface) is also abstracted away into specific Mach instructions handling the stack and function parameters.

One major difference between Mach and Asm lies in their semantics: the "next" Mach instructions to be executed are directly in a Mach state (as a list of instructions), whereas the Asm instructions are stored in memory, accessed by the `PC` register. In such Asm semantics, the `PC` register can jump anywhere within the function body, not necessarily to a label.[17] On the contrary, the Mach semantics ensures that jumps can only branch to particular points (labels, function entry points and return addresses). This property is not carried within the Asm IR. This makes reconstructing basic blocks from Asm (in a hypothetical Asm to Asmblock pass) impossible: our AsmVLIW semantics requires that `PC` never jump inside a basic block.

Our solution is to construct the basic blocks earlier, at the Mach level, by introducing a new Machblock IR as well as an architecture independent Mach to Machblock translation, and then adapting the former Mach to Asm pass to a new Machblock to Asmblock pass. Not only does introducing Machblock allows separating the proof of the basic block reconstruction from the proof of the Mach to Asm translation, but it also makes part of the process reusable for other backends.

7.2 Translating and proving Mach to Machblock

The Mach to Machblock translation is a purely syntactic one: Mach instructions are separated into labels, basic and control-flow instructions (much like section 3). We create an initial basic block and start filling basic instructions inside. The translation ends the current basic block whenever a label or control-flow instruction is met - after which it creates a new basic block.

The semantic preservation proof then checks that stepping through a certain number of instructions at the Mach level is equivalent to stepping through a corresponding basic block in Machblock. This is done with an Option simulation: executing a Mach instruction either leads to a "stuttering" with a decreasing measure, [18] or the execution of the whole Machblock block once the end is reached. See Appendix A.2 for details.

7.3 Translating and proving Machblock to Asmblock

The Machblock to Asmblock translation relies on translating each Machblock basic block into an Asmblock basic block. Each Machblock basic instruction is translated into one or more Asmblock basic instructions—and each Machblock control flow instruction is translated into any number of Asmblock basic instructions, followed by a control flow instruction. [19]

The proof of Machblock to Asmblock is a star simulation: each Machblock basic block is simulated by one (or more, in the case of a builtin) Asmblock basic blocks; like in usual Mach to Asm translation there is only a single stuttering case, on the Mach step restoring the caller state (see Appendix A.1.2

[17] See the `igoto` instruction in section 2.2

[18] In our case, the measure is the (statically known) number of instructions left to run before the end of the basic block.

[19] For instance, translating a Machblock conditional instruction may require basic instructions to evaluate the condition, followed by the actual `Pcb` conditional branch instruction.

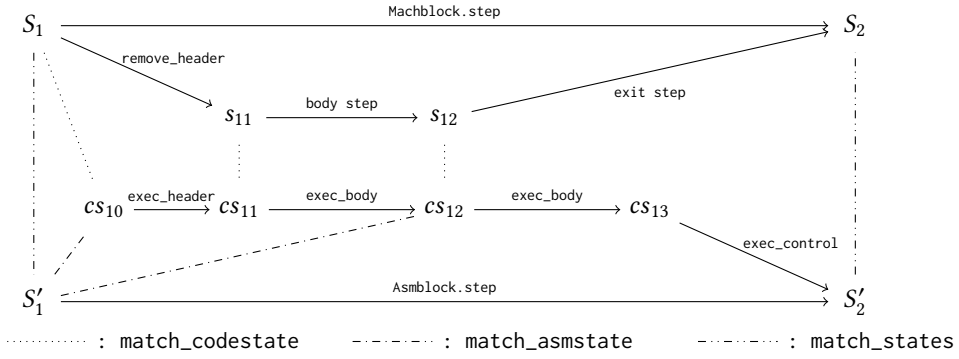

Fig. 14. Diagram of the simulation proof

for detailed explanations). The simulation used for that proof is a blockstep simulation (stepping one basic block at a time) that we have to decompose in terms of instruction-step simulations.

In usual Mach to Asm passes, the `match_states` simulation specifies that the value of registers and memory stay correct throughout the execution. It also specifies that the code to which the PC register points must correspond to the code that is executed. Here, our instruction-step simulations cannot directly use such a `match_states` relation, since the PC register would have to point in the middle of a basic block (to fetch the current instruction), however our semantics disallow that.

We thus split the `match_states` relation into two different simulations: `match_codestate` which handles the simulation between Machblock and Asmblock regardless of the presence of the instructions in memory (these are instead saved in an intermediate "ghost state" called `codestate`), and a `match_asmstate` relation ensuring that the instructions stored in the `codestate` are present at the memory address pointed to by PC.

The actual blockstep simulation theorem is then cut into several smaller theorems, following the simulation diagram of Fig. 14. This approach re-uses most of the existing Mach to Asm proofs with a minimum amout of readaptations. More details can be found in Appendix B.

8 EXPERIMENTAL EVALUATION

Our implementation[7] adds to CompCert around 28Kloc of Coq and 5Kloc of OCaml, much more than e.g. 10 Kloc of Coq and 2Kloc of OCaml each for the Risc-V and x86 targets. Our target assembly is described by around 1.8K lines of specification in the AsmVLIW module. This is a little more than other Asm (1–1.3Klines). Our scheduling oracle is implemented by 2.4Kloc of OCaml (half for its frontend, half for its backend).

The remaining of this section describes our evaluation of this implementation. Firstly, we measure the compilation time of our optimization *w.r.t* the other optimization passes of CompCert. Then, we compare timings of compiled code with our CompCert to that of the default compiler for the target platform, measuring clock cycles using performance counters. We also compare our timings in different configurations to study the impact of postpass scheduling in CompCert.

8.1 Experimental compiling time

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.

Fig. 15 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, which is the same time required for other CompCert optimizations such as constant propagation or common subexpression elimination.
These compile times are in line with other optimization passes of CompCert.

8.2 Benchmarks used

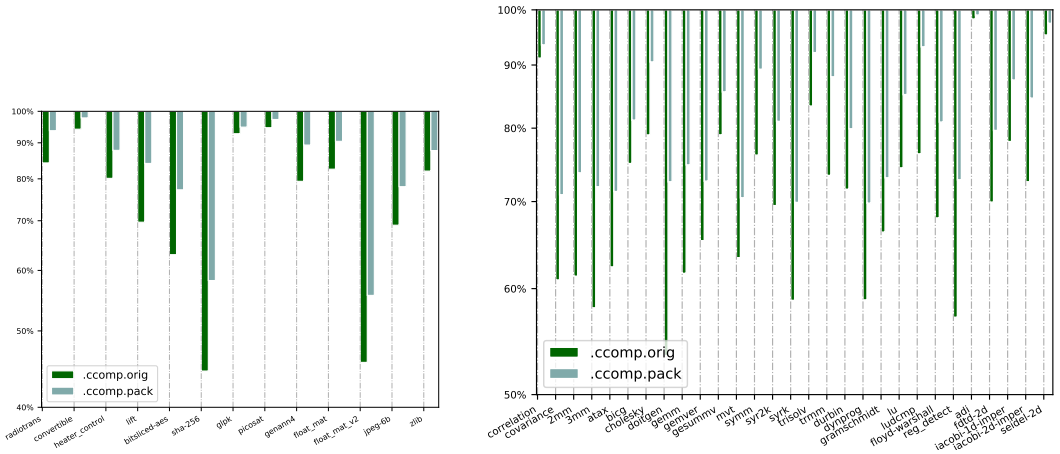

Fig. 16. Relative speed of CompCert's generated code depending on scheduling options. The speed with list scheduling—the default option— is "100%" (lower is slower). The "orig" option outputs the original assembly code, unscheduled, one instruction per bundle. "pack" packs instructions into bundles without reordering them. The left picture represents our target benchmark, whereas the right picture represents Polybench.

We evaluated our optimization on a range of applications that could be used on the Kalray KVX core: critical embedded systems and computational benchmarks. This is our target benchmark. We also evaluated CompCert on the Polybench benchmarks [Pouchet 2012].

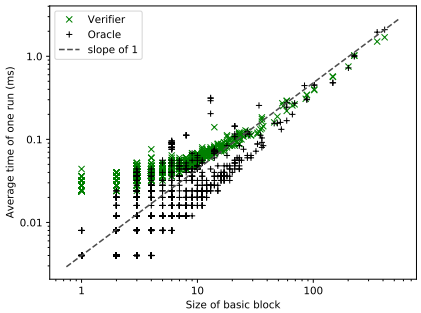

Fig. 15. Scheduling times

radiotrans, *convertible* and *heater-control* are benchmarks compiled from synchronous dataflow programming languages (respectively Heptagon, Lustre v6 and Lustre v4). Such languages are for instance used to specify and implement fly-by-wire aircraft controls [França et al. 2011]. The C source code compiled from the high-level specification is then often compiled *without optimization* so that it can be easily matched to the resulting assembly code. CompCert's advantage in this area is that it allows using optimizations, its semantics preservation proof replacing the manual structural matching between assembly and C code. *lift* is a lift controller program from TACLeBench, a collection of benchmarks used for worst-case execution time research [Falk et al. 2016].

On the computational part, *bitsliced-aes* and *sha-256* are cryptography primitives taken from [Mosnier 2019] and [Patrick 2015]. *glpk* runs GLPK (GNU Linear Programming Kit [Makhorin 2012]) on an example. *picosat* is an optimized SAT solver [Biere 2008], ran over a Sudoku example. *genann* is a minimal artificial neural network [Van Winkle 2018]. *float-mat* is a textbook implementation of floating-point matrix multiplication; *float-mat-v2* is a version with high level transformations such as loop unrolling done at the source level. *jpeg-6b* is from the `Libjpeg` [Lane and the Independent JPEG Group (IJG) 1998]. *zlib* [Gailly and Adler 2017] is a data-compression library.

8.3 Impact of optimizations

Figure 16 illustrates the impact of our optimization pass on the performance of the generated code. The reference version uses the list scheduler of Section 6.3. Two others use the *bundlers without reordering* of Section 6.1: "orig" emits one instruction per bundle (close to what straightforwardly generating instuctions one by one as other CompCert backends would produce), and "pack" uses the greedy bundler. In Figure 16, the execution time of each version is compared to those of the reference. Higher percentages mean better timings.

Postpass scheduling has a noticeable impact on performance: compared to the reference "orig" version, we get an average performance increase of 41%. However, some benchmarks such as *convertible* are barely affected by the optimization—indeed, the main loop features around 800 different variables in the same scope, which do not fit into the 64 registers of the KVX. Register spills are thus generated, which in turn prevent scheduling since we do not yet have any alias analysis yet, which would allow reordering memory accesses.

The "pack" version increases performance slightly w.r.t. "orig", but not by much compared to true scheduling. We gain an average of 23% by scheduling instead of naive greedy bundling.

A word of warning: since our scheduler operates after register allocation,[20] it is highly sensitive to register reuse: $write(v_1, r_1) \dots read(r_1) \dots write(v_2, r_2) \dots read(r_2)$ (with no other accesses to r_1, r_2) can be rescheduled to $write(v_1, r_1) \dots write(v_2, r_2) \dots read(r_1) \dots read(r_2)$ but $write(v_1, r_1) \dots read(r_1) \dots write(v_2, r_1) \dots read(r_1)$ cannot. In some cases, optimizations prior to register allocation, with actual improvements in intermediate code, lead to worse final performance because the optimized code, after register allocation, happens to reuse a register in a "hot" loop in a way that prevents instructions from being scheduled optimally, whereas the less optimized code does not. This makes it difficult to measure the impact of optimizations, because whether or not registers are reused in this way is a matter of luck. Adding a prepass scheduler could solve this problem.[21]

8.4 Comparison of CompCert with Kalray's GCC

We also compared our CompCert compiler to the GCC[22] compiler supplied by Kalray, adapted from version 7.5.0, at -O0…-O2 optimization levels. -O1 deactivates scheduling and thus only generates bundles of one instruction. -O0 loads and stores variables from memory at every use.

The results (see Fig. 17) 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.

[20] We have not modified CompCert's register allocator, except for allowing float and integer values to be allocated to a single bank of registers.

[21] But, even with a prepass scheduler, a postpass scheduler on the assembly code would still be required: the assembly code is the level where instructions (e.g. loads of spilled registers) are precisely scheduled.

[22] The GNU Compiler Collection, https://gcc.gnu.org/

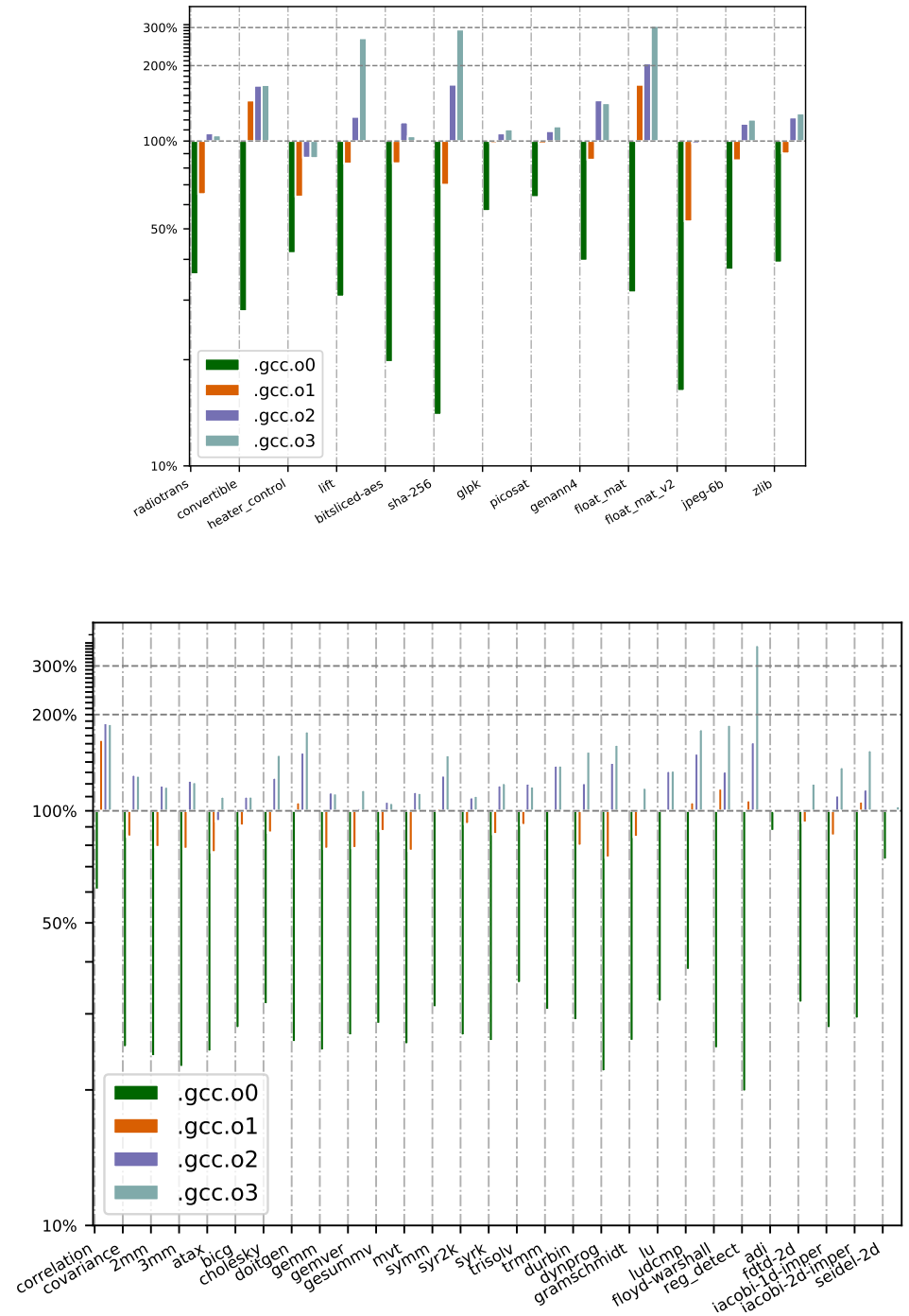

Fig. 17. Relative speed of generated code with Kalray's GCC, from -O0 to −O3 optimization levels; 100% being the speed with CompCert by list scheduling (lower is slower). Note that for some safety-critical applications with mandatory traceability from source to object code (e.g., DO178 level-A avionics), only gcc -O0 is likely to be usable, whereas CompCert's correctness proof is accepted for traceability [Bedin França et al. 2012]. Our target benchmark is pictured at the top, Polybench is at the bottom picture.

For example, while CompCert is outperformed by GCC on *float-mat*, it comes close in performance on *float-mat-v2*, meaning that we identified which high-level transformations need to be integrated into CompCert to improve performance on this kind of functions.

Regardless, in average, we produce code 276% faster than code produced by -O0, 19% faster than -O1, 18% slower than -O2, and 25% slower than -O3. In some cases, we produce faster code than GCC, in the best case 8% faster than -O3.

8.5 Remarks and limitations

Even though we aim at comparing the quality of instruction scheduling, we actually compare two very different whole compilers. In particular, the machine-independent parts of CompCert do not perform certain optimizations that GCC does, among which:

- certain *strength reductions*: 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;
- structure or array "disaggregation": expanding an aggregate (accessed through *base* + *index* loads and stores) into independent scalar variables, which can be allocated to registers;
- *interprocedural optimisations*: the only one performed by CompCert is *inlining*, and CompCert's inlining heuristic is less aggressive than GCC's.

In contrast, CompCert replaces a 32-bit signed division by a constant with a small efficient sequence of code [Granlund and Montgomery 1994], whereas GCC calls a generic library function.

Certain compiler differences are subtler but may have dramatic effects in some cases: e.g., our version of CompCert sometimes 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 thereof, reproducing optimizations that GCC would perform.

9 CONCLUSION AND FUTURE WORK

Trusted Computing Base. Customizing Coq's extraction mechanism to call external OCaml procedures increases the trusted computing base. Yet we limited this increase: the only property we trust is that OCaml's pointer equality implies structural equality (of Coq).[23]

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. We however had a few bugs in the parts of CompCert not checked by formal proofs—printing of assembly instructions, stack

[23] Our correctness proof does not rely on other properties that our OCaml code provides (in particular, the correctness proof of our verifier does not assume that two isomorphic hash-consed data structures in existence at the same time are always allocated to the same place in memory). In order not to have to convert CompCert's code generation flow to the full monadic style of Impure, we also unsafely exit from the Impure monad in order to obtain the scheduled code as though the verified scheduler were guaranteed to be (logically) deterministic. We do not think that this weakness hides a real issue: even if an unexpected bug in some of our OCaml oracles makes them non-deterministic, we do not call the scheduler twice on the same code, so there is no absurd case where we could go to if two different calls gave different results. This is in line with similar implicit assumptions elsewhere in CompCert that oracles are deterministic. These assumptions result from a shortcut in the formalization and are expected to be useless, without any bad consequence if they are wrong.

frame (de)allocation. Most bugs in the scheduling oracles were found by testing them on random examples; an untrusted checker with detailed error messages is useful for such early testing.

Apart from the difficulty of finding suitable invariants and proof techniques, another main hurdle was interpreting benchmark results. Missed optimization bugs (e.g. an inferior schedule was selected) were particularly hard to catch.

Future work. Our scheduler is intra-block; this implies it cannot for instance anticipate computations by moving them to free time slots before a branching instruction. We plan to implement a superblock scheduler allowing such movements on the critical path, before register allocation.

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; again, prepass scheduling should help in this respect: it would generate a reasonable schedule, register allocation would be performed on that schedule and the postpass scheduler would then perform local adjustments.

Our backend cannot at present reorder a memory read and a memory write, or two memory writes, even 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.

Despite our efforts, our instruction selection is still perfectible: a few instructions that could be of use are still not selected. We shall work on this, though we do not hope much improvement can be gained. Selecting vector instructions automatically (loop vectorization) could improve performance in some cases, but doing so would entail considerable changes to CompCert's notions for mapping variables to registers. Hardware loops could save some cycles on tight loops, but again these would require considerable changes to CompCert's backend.

The main cause of inefficiency of the code generated for some examples, compared to GCC's, is the lack of some high level optimizations in CompCert, for instance better inlining heuristics, structure or array "disaggregation" (expanding an aggregate into scalar variables), loop-invariant code motion and strength reduction. Again, this is left to future work.

ACKNOWLEDGMENTS

We wish to thank Benoît Dupont de Dinechin, Xavier Leroy, Gergö Barany, Thomas Vandendorpe as well as the anonymous reviewers for their helpful remarks.

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

Armin Biere. 2008. PicoSAT Essentials. *JSAT* 4 (01 2008), 75–97.

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

Joseph A. 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

Jean-Loup Gailly and Mark Adler. 2017. *zlib*. https://www.zlib.net/

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

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

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

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.

Tom Lane and the Independent JPEG Group (IJG). 1998. *Libjpeg*. http://libjpeg.sourceforge.net/

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

Andrey Makhorin. 2012. *GNU Linear Programming Kit.* Free Software Foundation. https://www.gnu.org/software/glpk/

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

Alain Mosnier. 2019. *SHA-256 implementation.* https://github.com/amosnier/sha-2

Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 448–461. https://doi.org/10.1145/2908080.2908109

Conor Patrick. 2015. *Bitsliced AES implementation.* https://github.com/conorpp/bitsliced-aes

Louis-Noël Pouchet. 2012. *the Polyhedral Benchmark suite.* http://web.cs.ucla.edu/~pouchet/software/polybench/

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

Lewis Van Winkle. 2018. *Genann — minimal artificial neural network.* https://github.com/codeplea/genann

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 BASIC-BLOCK RECONSTRUCTION

This section explains how we recover the basic-block structure of a Mach program. Section A.1 first recalls some preliminaries on Mach and Asm IR in CompCert. This helps to motivate our new Machblock IR, between Mach and Asmblock, introduced at Section A.2. The latter also presents the translation between Mach and Machblock. The pass of Machblock to Asmblock is presented in Section B.

A.1 Preliminaries : from Mach to Asm in CompCert

Section A.1.1 presents the Mach IR of CompCert. Section A.1.2 explains the difference between Mach and Asm and presents the proof from Mach to Asm in existing CompCert backends.

A.1.1 The Mach IR of CompCert. 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`.

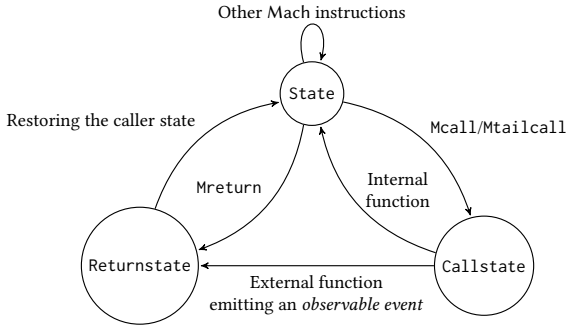

Fig. 18. Execution steps between Mach states

A.1.2 Proving a translation from Mach to Asm. 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,

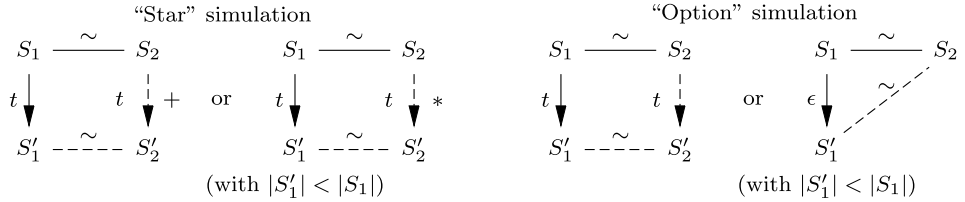

Fig. 19. Simulation diagrams with stuttering in CompCert

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.[24]

Formally, it involves introducing a suitable "$\sim$" 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 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.

Then, the proof involves a "Star" simulation. Such a 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 — by the following conditions:

- The initial states match

$$\forall S_1,\ P_1.\texttt{istate}\ S_1 \implies \exists S_2,\ P_2.\texttt{istate}\ S_2 \wedge S_1 \sim S_2$$

- The final states match

$$\forall S_1\ S_2\ r,\ S_1 \sim S_2 \wedge P_1.\texttt{fstate}\ S_1\ r \implies P_2.\texttt{fstate}\ S_2\ r$$

- The execution steps match through the "Star" simulation diagram (also depicted in Figure 4)

$$\forall S_1\ S_2\ S_1'\ t,\ S_1 \sim S_2 \wedge S_1 \xrightarrow{t} S_1' \implies \exists S_2',\ S_1' \sim S_2' \wedge (S_2 \xrightarrow{t}{}^{+} S_2' \vee (S_2 \xrightarrow{t}{}^{*} S_2' \wedge |S_1'| < |S_1|))$$

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 forever without producing any observable event, then P_2 cannot stutter infinitely.[25] 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).

The "Star" simulation from Mach to Asm thus corresponds to prove that one Mach step (i.e. one transition of Fig. 18) 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

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

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

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`.

A.2 Mach to Machblock translation

As explained in A.1.2, on an arbitrary Asm program, we cannot prove that an execution cannot jump to the middle of a function, and in particular 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 trivial. Hence, it seems interesting to capitalize this effort for a generic component w.r.t the processor. This motivates us to introduce a new IR between Mach and Asmblock, called Machblock (Fig. 3), which introduces a sequential blockstep semantics on Mach programs.

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 (Fig. 19). 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 Fig. 19—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:

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

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

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

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, 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 20.

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 an independent lemma of our Coq proof.

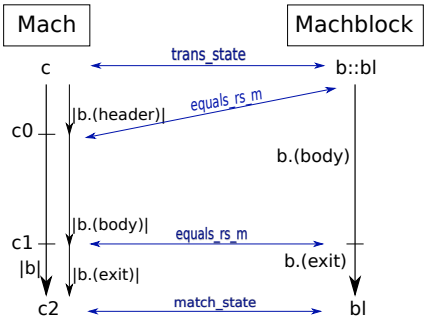

Fig. 20. Overview of our proof for condition (2) of the "Block" simulation of Mach by Machblock

B MACHBLOCK TO ASMBLOCK

The Machblock to Asmblock pass is adapted from the Mach to Asm pass of other backends (see Section A.1.2), but with a consequential change: instead of manipulating instructions (and reasoning on the execution of single instructions), we are now manipulating basic blocks of instructions, and must reason on the execution of an entire basic block (in one single step). We are giving here details to hint the reader on the differences between the two approaches, in particular for the star simulation proof.

This appendix details the biggest differences, to give the viewer an idea of the difficulty involved in porting an existing Mach to Asm translation into a Machblock to Asmblock variant.

B.1 Machblock to Asmblock translation

B.1.1 Overview. Given a basic block from Machblock, the translation consists in emitting an Asmblock basic block which performs the same operation.

The Mach to Asm pass of other backends translates code function by function, then within each function, instruction by instruction.

Our translation goes through the Machblock code function by function, then basic block by basic block. Within each basic block, the labels, basic instructions, then the control-flow instruction are translated consecutively.

The label translation is straightforward. A label in CompCert is a positive identification number, so we translate a label from Machblock to Asmblock by applying the identity function.

The translation of Machblock basic instructions give Asmblock basic instructions. Indeed, if a Machblock instruction does not modify the control, then there is no reason for the corresponding Asmblock instructions to do so. By "modifying the control" we mean here: not taking any branch, and going to the "next" instruction in line instead. This is done by the `transl_ basic_code` and `transl_instr_basic` with the following signatures (the Boolean is there for an optimization explained in the next section):

```
transl_basic_code:
    Machblock.function → list Machblock.basic_inst → bool → list Asmblock.basic_inst
transl_instr_basic: Machblock.function → Machblock.basic_inst → bool → list Asmblock.basic_inst
```

The translation of Machblock control-flow instructions is less trivial. It gives a list of Asmblock basic instructions (possibly empty), followed by a single Asmblock control-flow instruction.

For example, the translation of a `MBreturn` cannot be just a `ret` assembly instruction, because other operations such as restoring the stack pointer must be handled in the function epilogue.

Another example is the `(MBcond cond args lbl)` instruction which evaluates the arguments `args` with the condition `cond`, and jumps to `lbl` if this evaluation is `true`. Evaluating a comparison of a register to `0` can be done in one instruction in the Kalray assembly. However, comparing two registers must be done by at least two instructions: one `(compd rd rs1 rs2)` instruction comparing the two registers `rs1` and `rs2`, writing the Boolean result to `rd`, then a `cb` (Conditional Branch) instruction using the value of `rd` to decide of the branch.

The translation of control-flow instructions is done by a `transl_instr_control` function of signature:

```
transl_instr_control:
    Machblock.function → option Machblock.control_flow_inst → list Asmblock.instruction
```

If we note $(L;\ lb;\ c)$ a Machblock basic block, the translation then consists in generating the Asmblock basic block $(L; t_b(lb) \mathbin{+\!\!+} l';\ c')$ where t_b is `transl_basic_code` , and l', c' are defined such that $t_c(c) = l' \mathbin{+\!\!+} (c' :: \texttt{nil})$, with t_c being `transl_instr_control`.

B.1.2 The argument pointer code optimization. There are many possible ways to pass the arguments to a function in assembly code. In order to ensure a compatibility between the different compilers as well as hand-written assembly code, each architecture defines an ABI (Application Binary Interface), a specification of how arguments should be handled.

The arguments are usually first passed into registers, and then onto the stack if there isn't enough place in the registers. In the case of the Kalray Coolidge ABI (but it is also the case for some other ABIs like the Risc-V), when there are too many arguments to be contained by the registers, the caller pushes the extra arguments on the stack. Since each stack-frame is adjacent, the callee can then access the arguments directly via its SP (Stack Pointer) register. For instance, if the stack size of the callee is 24 bytes long, and the stack is in decreasing order (parents are in higher addresses), then the first extra argument will be at address `(SP+32)`.

This poses an issue within the memory model of CompCert. A CompCert address is a couple (b, off), where b is an integer representing the memory block, and off is the offset within that memory block. In this model, memory blocks are not contiguous. If we take the above example, `(SP+32)` would actually be pointing to an invalid memory. Consequently, in CompCert, function arguments cannot be accessed with the SP register directly.

The solution adopted by CompCert backends, is to keep the value of the "old SP " somewhere on the stack. It is then part of the specifications that the function prologue should be saving SP before modifying SP. This is done by the `Pallocframe` pseudo-instruction, for which we give below the specification:

```
| Pallocframe sz pos ⇒
    let (mw, stk) ≔ Mem.alloc mr 0 sz in
    let sp ≔ (Vptr stk Ptrofs.zero) in
    match Mem.storev Mptr mw (Val.offset_ptr sp pos) rsr#SP with
    | None ⇒ Stuck
```

```
    | Some mw ⇒ Next (rsw #FP ← (rsr SP) #SP ← sp #RTMP ← Vundef) mw
    end
```

This pseudo-instruction is doing two actions in regards to the old value of SP:

- Storing it at (Val.offset_ptr sp pos)
- Copying it at the register FP, a register arbitrarily chosen among the general purpose registers that can be used for any operation.

This pseudo-instruction does not correspond to any particular assembly instruction: it is expended into a sequence of assembly instructions by not-formally-verified (but trusted) part of CompCert[26].

The copy into FP is part of a small optimization done during code generation. Loading a value from memory is expensive in most architectures, and having to load this "old SP " for each access of a parameter would result in a particularly inefficient code. In order to alleviate this, CompCert backends remember during the code generation whether they previously loaded the old SP or not. Then, when translating a MBgetparam, additional code is inserted if the old SP isn't already loaded in FP. For instance, translating two consecutive MBgetparam writing a parameter in a different register than FP, will result in at most one FP reload operation.

In order to remember whether FP is loaded, a Boolean value parameter ep is added to each translation function. The translation starts with ep=true (the prologue initially loads the old SP into FP) , then throughout the translation sets ep to true if it has been reloaded, or to false if it might have been destroyed. For instance, ep is set to false every time a label is encountered, since it is not possible to know where did the control originate from.

More precisely, the next value of ep (outside of labels, handled separately) is given by this function (in the case of Kalray backend, but other backends also have a similar function):

```
Definition fp_is_parent (before: bool) (i: Machblock.basic_inst) : bool :=
  match i with
  | MBgetstack ofs ty dst ⇒ before && negb (mreg_eq dst MFP)
  | MBsetstack src ofs ty ⇒ before
  | MBgetparam ofs ty dst ⇒ negb (mreg_eq dst MFP)
  | MBop op args res ⇒ before && negb (mreg_eq res MFP)
  | MBload chunk addr args dst ⇒ before && negb (mreg_eq dst MFP)
  | MBstore chunk addr args res ⇒ before
  end
```

MFP stands for the Mach register corresponding to FP. In the case of an operation whose result is to be stored in FP, we must invalidate FP by setting ep to false, hence the (negb (mreg_eq dst MFP)) used in most cases of the match.

B.1.3 Constraints of an Asmblock basic block. There are two main constraints to verify when forming an Asmblock basic block. These are enforced by a wf_bblock predicate inside a bblock record.

The first constraint is that the bblock should never be empty. Indeed, by our semantics, PC is incremented by the size of the block in the end of its execution. If there are three instructions

[26]It can be a source of bugs to perform such a critical operation in the uncertified part of CompCert, especially when taking into account the complex support for variadic arguments. Yet, as of now, all backends adopt that approach, and careful testing can rule out these bugs.

in the bblock, then, at the end of the bblock step, PC is incremented by three. If the bblock was allowed to be empty, then PC would not get modified, we would then execute the same bblock, which would cause an infinite stuttering.

The second constraint comes from the Pbuiltin special instructions. A Pbuiltin instruction represents a *builtin* instruction, that is, an instruction that is inserted directly from the C code to the assembly code, without any translation from the compiler.

Here is an example of a C code containing such a builtin:

```
/* Invalidating data cache */
__builtin_k1_dinval();

int v = tab[40];
printf("%d\n", v+3);
```

The __builtin_k1_dinval builtin produces directly the instruction dinval, which is used to invalidate the data cache, before accessing a value on the heap. Such instructions are specially handled by compilers.

In terms of CompCert assembly semantics, they are treated as external function calls (generating a trace), however they pose the problem that, in assembly, we do not know exactly what they are made of. Their expansion lies in the uncertified part of CompCert, with the only restriction that a builtin cannot create new functions or sections. Each builtin instruction could potentially be generating an entire block of code, with its own labels and more than one control-flow instructions.

We could treat them the same way we treat function call instructions, but then the uncertified scheduler could return us a schedule where the builtin is to be bundled with another basic instruction, which could lead to absurd code if the builtin is eventually expanded to several instructions and/or labels.

We are hitting here the limits of the builtin specification inside CompCert. In order to prevent the above case, we chose to isolate each builtin, with one desired property: if an Asmblock basic block has a builtin instruction, then that instruction must be the only instruction of the basic block.

This ensures that the uncertified expansion should have no effect on the rest of the generated code - though this is not proven formally, by lack of precise builtin semantics. Just like the function prologue and epilogue, test benchmarks are used to remedy this lack of formal certification.

Let us also recall a third constraint, which is not specific to a bblock but rather a direct definition of our Asmblock semantics: we cannot branch in the middle of a bblock, *i.e.* a Stuck state is induced when the PC register does not point to the beginning of a bblock. This third constraint is necessary to define a blockstep semantics.

B.1.4 Function prototypes for implementation. We describe here the actual functions used for implementing the translation.

For each function, linearly, each basic block is translated via a transl_blocks function, starting with ep=true:

```
Definition transl_function (f: Machblock.function) :=
  do lb ← transl_blocks f f.(Machblock.fn_code) true;
```

```
  OK (mkfunction f.(Machblock.fn_sig) (make_prologue f lb))
```

The `make_prologue` consists in inserting the `Pallocframe` pseudo-instruction, as well as instructions making sure the return address is stored on the stack, in order to have a correct linking when executing the `ret` instruction of the function epilogue.

The `transl_blocks` goes through each basic block of the list, translates it, then propagates `ep` if the block does not have any label.

```
Fixpoint transl_blocks (f: Machblock.function) (lmb: list Machblock.bblock) (ep: bool) :=
  match lmb with
  | nil ⇒ OK nil
  | mb :: lmb ⇒
      do lb ← transl_block f mb (if Machblock.header mb then ep else false);
      do lb' ← transl_blocks f lmb false;
      OK (lb @@ lb')
  end
```

`transl_block` is split into three functions: `transl_basic_code` which translates linearly each basic instruction of the Machblock basic block, then `transl_exit_code` which translates the optional control-flow instruction, and finally, `gen_bblocks` which makes one or several basic blocks based on the results of the two last functions.

```
Definition transl_block f fb ep : res (list bblock) :=
  do c ← transl_basic_code f fb.(Machblock.body) ep;
  do ctl ← transl_instr_control f fb.(Machblock.exit);
  OK (gen_bblocks fb.(Machblock.header) c ctl)
```

The `gen_bblocks` function ensures that we generate `bblocks` that satisfy the two earlier described constraints:

- a `bblock` cannot be empty: while we believe this should never happen, we must nevertheless tackle what happens in the case that it does to ensure a forward simulation. We have chosen to generate a `bblock` with a single `nop` instruction if we ever come across an empty Machblock basic block.
- a builtin instruction must be alone in its `bblock`. When we encounter one, we split the basic block into two `bblocks`.

```
Program Definition gen_bblocks (hd: list label) (c: list basic) (ctl: list instruction) :=
match (extract_ctl ctl) with
| None ⇒ match c with
  | nil ⇒ {| header := hd; body := Pnop::nil; exit := None |} :: nil
  | i::c ⇒ {| header := hd; body := ((i::c) ++ extract_basic ctl); exit := None |} :: nil
  end
| Some (PExpand (Pbuiltin ef args res)) ⇒ match c with
  | nil ⇒ {| header := hd; body := nil; exit := Some (PExpand (Pbuiltin ef args res)) |} :: nil
  | _ ⇒ {| header := hd; body := c; exit := None |} ::
         {| header := nil; body := nil; exit := Some (PExpand (Pbuiltin ef args res)) |} :: nil
  end
| Some ex ⇒ {| header := hd; body := (c ++ extract_basic ctl); exit := Some ex |} :: nil
end
```

Finally, the `extract_basic` and `extract_ctl` functions (not detailed here) extract respectively the basic and the control-flow instructions of a list of instructions.

B.2 Forward simulation proof

B.2.1 Issues of using the usual Mach to Asm diagram. The usual diagram consists in starting from two states s_1 and s_2 for which a certain `match_states` relation holds, executing one Mach

instruction, then executing the translated Asm instruction and proving that the two obtained states s'_1 and s'_2 also verify the `match_states` relation, that we give below for our backend: [27]

```
Inductive match_states: Machblock.state → Asmvliw.state → Prop :=
  | match_states_intro:
      ∀ s fb sp c ep ms m m' rs f tf tc
        (STACKS: match_stack ge s)
        (FIND: Genv.find_funct_ptr ge fb = Some (Internal f))
        (MEXT: Mem.extends m m')
        (AT: transl_code_at_pc ge (rs PC) fb f c ep tf tc)
        (AG: agree ms sp rs)
        (DXP: ep = true → rs#FP = parent_sp s),
      match_states (Machblock.State s fb sp c ms m)
                   (Asmvliw.State rs m')
```

This relation ensures several things, among which:

- `AG` and `MEXT` ensures the value of registers and memory match those of the Machblock state
- `AT` ensures that, in the memory pointed by PC register, the Machblock code `c` is translated into the Asmblock code `tc` with the Boolean parameter `ep`.
- `DXP` ensures that if `ep` is `true`, then FP must contain the value of the old SP.

In the other CompCert backends, the `match_states` relation holds between each execution of a Mach instruction. For each possible Mach instruction, lemmas are then proved to ensure that executing on one hand the Mach instruction, on the other hand the translated Asm instructions, lead to the same result.

In our case, using the above `match_states` directly is tricky: instead of executing one Mach instruction and reasoning on its translation, we execute an entire basic block of Machblock instructions, and must reason on the translation of the whole block.

A possibility for us could be to define a finer grain `step` relation, which would execute a single instruction of a basic block on the Machblock part. In Machblock, executing one instruction could be as simple as executing its effects, then removing it from the current basic block. On the Asmblock side, one could suggest to increment PC instruction by instruction instead of doing it all at the end of the `bblock`, however that would not be compatible with the constraint that we must not branch in the middle of a `bblock`.

B.2.2 A new diagram to prove Machblock to Asmblock. In the lights of this difficulty, we chose to introduce a new state definition, used for reasoning at a finer grain. We call it a Codestate, and define it as follows:

```
Record codestate :=
  Codestate { pstate: state;         (* projection to Asmblock.state *)
              pheader: list label;   (* list of label *)
              pbody1: list basic;    (* list of basics coming from the Machblock body *)
              pbody2: list basic;    (* list of basics coming from the Machblock exit *)
              pctl: option control;  (* exit instruction, coming from the Machblock exit *)
              ep: bool;              (* reflects the [ep] variable used in the translation *)
              rem: list AB.bblock;   (* remaining bblocks to execute *)
              cur: bblock            (* current bblock to execute - useful to increment PC *)
            }
```

A Codestate augments a state of Asmblock by also including the instructions to execute, much like Machblock. It also includes the value of `ep` used when translating the first instruction of Codestate.

[27] We are here only describing the particular case of executing code within an internal function

With this new Codestate, we can decompose the `match_states` relation into two relations.

The first of these two relations is `match_codestate`, which ensures an agreement between a Machblock state and a Codestate, namely: the code residing in the Codestate must have been a result of a translation of a Machblock code, the memory and register states should correspond, and also the ep value of the Codestate should match with the one used in the translation.

```
Inductive match_codestate fb: Machblock.state → codestate → Prop :=
  | match_codestate_intro:
      ∀ s sp ms m rs0 m0 f tc ep c bb tbb tbc tbi
        (STACKS: match_stack ge s)
        (FIND: Genv.find_funct_ptr ge fb = Some (Internal f))
        (MEXT: Mem.extends m m0)
        (TBC: transl_basic_code f (MB.body bb) (if MB.header bb then ep else false) = OK tbc)
        (TIC: transl_instr_control f (MB.exit bb) = OK tbi)
        (TBLS: transl_blocks f c false = OK tc)
        (AG: agree ms sp rs0)
        (DXP: (if MB.header bb then ep else false) = true → rs0#FP = parent_sp s)
        ,
      match_codestate fb (Machblock.State s fb sp (bb::c) ms m)
        {|  pstate := (Asmvliw.State rs0 m0);
            pheader := (MB.header bb);
            pbody1 := tbc;
            pbody2 := extract_basic tbi;
            pctl := extract_ctl tbi;
            ep := ep;
            rem := tc;
            cur := tbb
        |}
```

The second relation is `match_asmstate` between a Codestate and an Asmblock state, ensuring that the code present in a Codestate actually resides in memory of the Asmblock state, at the address pointed by the PC register.

```
Inductive match_asmstate fb: codestate → Asmvliw.state → Prop :=
  | match_asmstate_some:
      ∀ rs f tf tc m tbb ofs ep tbdy tex lhd
        (FIND: Genv.find_funct_ptr ge fb = Some (Internal f))
        (TRANSF: transf_function f = OK tf)
        (PCeq: rs PC = Vptr fb ofs)
        (TAIL: code_tail (Ptrofs.unsigned ofs) (fn_blocks tf) (tbb::tc))
        ,
      match_asmstate fb
        {|  pstate := (Asmvliw.State rs m);
            pheader := lhd;
            pbody1 := tbdy;
            pbody2 := extract_basic tex;
            pctl := extract_ctl tex;
            ep := ep;
            rem := tc;
            cur := tbb |}
        (Asmvliw.State rs m)
```

Both relations take an extra `fb` parameter, which is the `function block`, an integer value identifying the current function.

The details of proving the Machblock to Asmblock pass with these two new `match` relations are very cumbersome (it is in general the case for all of the Mach to Asm proofs of the various backends - there are a lot of details and corner cases to consider). In particular, we are not covering here the case of a builtin instruction. However, we are giving below the general idea of simulating a `bblock` without builtin, assuming we already have an existing Mach to Asm proof to base ourselves on.

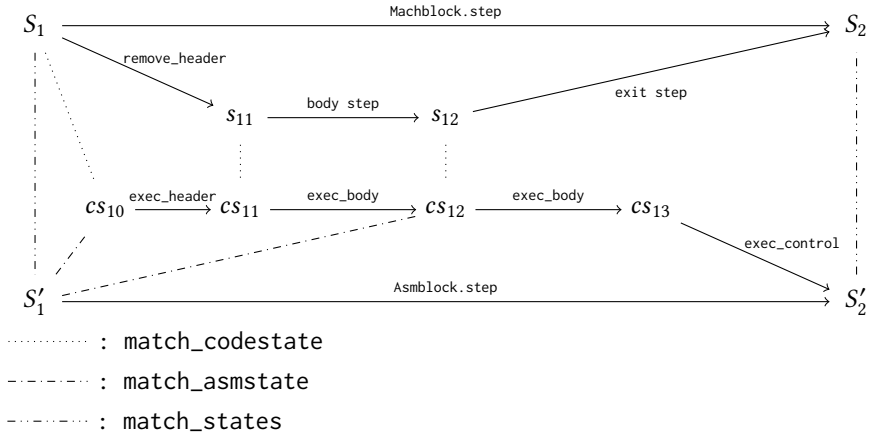

Fig. 21. Diagram of the simulation proof

Figure 21 depicts the diagram we went for. It can be decomposed in three theorems for simulating respectively the Machblock header, body, and exit. The `exec_header` is a predicate which removes the header from the `bblock`, and sets `ep` to false if there was a header:

```
Inductive exec_header: codestate → codestate → Prop :=
  | exec_header_cons: ∀ cs1,
      exec_header cs1 {| pstate := pstate cs1; pheader := nil; pbody1 := pbody1 cs1;
                         pbody2 := pbody2 cs1; pctl := pctl cs1;
                         ep := (if pheader cs1 then ep cs1 else false); rem := rem cs1;
                         cur := cur cs1 |}
```

We start by the theorem `match_state_codestate`, allowing us to decompose (`match_states` S_1 S'_1) into a (`match_codestate` S_1 cs_{10}) and a (`match_asmstate` cs_{10} S'_1) with a fitting cs_{10}:

```
Theorem match_state_codestate:
  ∀ mbs abs s fb sp bb c ms m,
  (∀ ef args res, MB.exit bb <> Some (MBbuiltin ef args res)) →
  (MB.body bb <> nil ∨ MB.exit bb <> None) →
  mbs = (Machblock.State s fb sp (bb::c) ms m) →
  match_states mbs abs →
  ∃ cs fb f tbb tc ep,
    match_codestate fb mbs cs ∧ match_asmstate fb cs abs
    ∧ Genv.find_funct_ptr ge fb = Some (Internal f)
    ∧ transl_blocks f (bb::c) ep = OK (tbb::tc)
    ∧ body tbb = pbody1 cs ++ pbody2 cs
    ∧ exit tbb = pctl cs
    ∧ cur cs = tbb ∧ rem cs = tc
    ∧ pstate cs = abs
```

The Machblock header simulation is then straight forward, and is proven by the `step_simu_header` theorem below. After this theorem, the state s_{11} is free of any header.

```
Theorem step_simu_header:
  ∀ bb s fb sp c ms m rs1 m1 cs1,
  pstate cs1 = (State rs1 m1) →
  match_codestate fb (MB.State s fb sp (bb::c) ms m) cs1 →
  (∃ cs1',
       exec_header cs1 cs1'
    ∧ match_codestate fb (MB.State s fb sp (mb_remove_header bb::c) ms m) cs1')
```

The body simulation is then proven by induction on the list of basic instructions of s_{11}, each individual case is covered by adapting the old proofs of Mach to Asm for the basic instructions.

```
Theorem step_simu_body:
  ∀ bb s fb sp c ms m rs1 m1 ms' cs1 m',
  MB.header bb = nil →
  (∀ ef args res, MB.exit bb <> Some (MBbuiltin ef args res)) →
  body_step ge s fb sp (MB.body bb) ms m ms' m' →
  pstate cs1 = (State rs1 m1) →
  match_codestate fb (MB.State s fb sp (bb::c) ms m) cs1 →
  (∃ rs2 m2 cs2 ep,
       cs2 = {| pstate := (State rs2 m2); pheader := nil; pbody1 := nil; pbody2 := pbody2 cs1;
                pctl := pctl cs1; ep := ep; rem := rem cs1; cur := cur cs1 |}
    ∧ exec_body tge (pbody1 cs1) rs1 m1 = Next rs2 m2
    ∧ match_codestate fb (MB.State s fb sp
        ({| MB.header := nil; MB.body := nil; MB.exit := MB.exit bb |}::c) ms' m') cs2)
```

This theorem gives a state s_{12} without any body anymore, just the exit instruction. We can then use the last theorem, `step_simu_control`:

```
Theorem step_simu_control:
  ∀ bb' fb fn s sp c ms' m' rs2 m2 t S'' rs1 m1 tbb tbdy2 tex cs2,
  MB.body bb' = nil →
  (∀ ef args res, MB.exit bb' <> Some (MBbuiltin ef args res)) →
  Genv.find_funct_ptr tge fb = Some (Internal fn) →
  pstate cs2 = (Asmvliw.State rs2 m2) →
  pbody1 cs2 = nil → pbody2 cs2 = tbdy2 → pctl cs2 = tex →
  cur cs2 = tbb →
  match_codestate fb (MB.State s fb sp (bb'::c) ms' m') cs2 →
  match_asmstate fb cs2 (Asmvliw.State rs1 m1) →
  exit_step return_address_offset ge (MB.exit bb') (MB.State s fb sp (bb'::c) ms' m') t S'' →
  (∃ rs3 m3 rs4 m4,
      exec_body tge tbdy2 rs2 m2 = Next rs3 m3
  ∧  exec_control_rel tge fn tex tbb rs3 m3 rs4 m4
  ∧  match_states S'' (State rs4 m4))
```

That last theorem gives us a `match_states` between a S_2' and S_2 which is what we are looking for. The final theorem `step_simulation_bblock` then uses the four theorems to prove the `plus simulation`.

```
Theorem step_simulation_bblock:
  ∀ sf f sp bb ms m ms' m' S2 c,
  body_step ge sf f sp (Machblock.body bb) ms m ms' m' →
  (∀ ef args res, MB.exit bb <> Some (MBbuiltin ef args res)) →
  exit_step return_address_offset ge (Machblock.exit bb)
        (Machblock.State sf f sp (bb :: c) ms' m') E0 S2 →
  ∀ S1', match_states (Machblock.State sf f sp (bb :: c) ms m) S1' →
  ∃ S2' : state, plus step tge S1' E0 S2' ∧ match_states S2 S2'
```

C OVERVIEW OF THE ABSTRACTBASICBLOCK INTERMEDIATE REPRESENTATION

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

Because of this good feature, this appendix describing AbstractBasicBlock intends to be self-contained, and does not even require to read other parts of the paper in details (except to understand the applications motivating AbstractBasicBlock). In particular, this appendix reformulates entirely Sect. 5, while providing much more details.[28]

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

[28] Of course, this comes at the price of repetitions. Sorry for the careful reader of Sect. 5.

presents the *simulation test*, which checks that the sequential semantics of basic blocks is preserved by scheduling.

C.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 "$r1 \coloneqq \mathtt{add}[r2, r3]$". 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`". 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 "$(\mathtt{load}\ i)$". In other words, the syntax of AbstractBasicBlock provides an infinite number of operations "$(\mathtt{load}\ i)$" (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 "$\mathtt{PC} \coloneqq (\mathtt{incr}\ i)[\mathtt{PC}]$" 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:

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

It could equivalently be coded as the assignment "$\mathtt{PC} \coloneqq (\mathtt{lt}\ l)[(\mathtt{incr}\ i)[\mathtt{PC}], r]$". 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) modifies two registers: PC and RA (the returned address). Hence "call$f$" is translated as the *sequence* of 3 assignments in AbstractBasicBlock:

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

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 "$(\mathtt{Old}\ e)$" evaluates "e" in the initial state

of the surrounding AbstractBasicBlock instruction.[29] Hence, the parallel load of 128-bit words is given in terms of two loads of 64-bit words:[30]

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

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:

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

In conclusion, each instruction of our assembly is translated into a sequence of assignments, where some of these assignments modify several pseudo-registers in "parallel" thanks to the special `Old` operator. We speak about *atomic sequences of assignments*: these sequences represent atomic instructions which can themselves be combined either sequentially or in parallel.

C.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). (* inst = atomic 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
```

[29] 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].

[30] 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 (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.

C.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[31] 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 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.

C.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.

[31] 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 4.3, 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 4.3 below.

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

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

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:
$r_1 := r_1 + r_2;\ r_3 := \texttt{load}[m, r_2];\ r_3 := r_1;\ r_1 := r_1 + r_3 \quad (p_1) \qquad r_3 := r_1 + r_2;\ r_1 := r_3 + r_3 \quad (p_2)$
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 "$\texttt{load}[m, r_2]$" as a potential failure.

Our formal development is decomposed into two parts using a *data-refinement* style. In a first part, presented in Section C.4.1, we define a model of the symbolic execution and the simulation test. In a second part, sketched by Section C.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 C.4.2.

C.4.1 A Model of our Simulation Test. The principle of *symbolic execution* has been first introduced by King [1976]. "Symbolic execution" refers to *how to* compute "symbolic memories" (and not to *what* they are) : mimicking the concrete execution while replacing operations on "concrete memories" by operations on "symbolic memories".

In this analogy, "values" are replaced by "*symbolic values*", which are actually terms evaluated in the *initial memory*. Hence, our 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 our 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 the symbolic memory 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 m x, smem_valid ge d1 m → term_eval ge (d1.(post) x) m = term_eval ge (d2.(post) x) m)

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

Internally, as coined in the name of "symbolic execution", `bblock_smem` mimics `run`, by replacing operations on memories of type `mem` by operations of type `smem`: these operations on the symbolic memory are given in Fig. 22. 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 (i.e. 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. 22. Basic operations of the symbolic execution in the abstract model

C.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 "`⇝`" of type `?? A → A → Prop`. Intuitively, when "`A`" is seen as "$\mathcal{P}$`(A)`", then "`⇝`" 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 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.[32]

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`).

[32]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.

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

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.[33] 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` $x \leftsquigarrow e_1$ `;;` 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 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) :=...
```

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

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 `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.[34] 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` (Fig. 23). 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.

[34] 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. 23. Memoizing functions for hash-consing of terms (and list of terms)

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.

```
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).

C.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 C.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 C.4.2, and is thus itself parametrized by the memoizing functions `hC_term` and `hC_list_term` defined in Figure 23. 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 24 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 24 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 C.4.1. Figure 25 provides an implementation of the operations of Figure 22 that preserves the data-refinement relation. The smart constructors building hash-consed terms are invoked by the `exp_hterm` (i.e., the evaluation of expressions

```
(* 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. 24. Data-refinement of symbolic memories, with handling of hash-consed terms

on symbolic memories). The `hsmem_set` implementation (Fig. 25) is an intermediate refinement toward the actual implementation, improving on two points. First, in some specific cases—i.e., when `ht` is an input or a constant, we know that `ht` cannot fail. In these cases, we avoid adding it to `hd.(hpre)`. Second, when `ht` is structurally equal to `(Input x)`, the 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. There is much room for future work on improving the `hsmem_set` operation by, e.g., applying rewriting rules on terms.[35]

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 25.

```
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[36]. First, it creates two memoizing functions `hC_term` and `hC_list_term` (Fig. 23) from the generic factory `hCons` (see Section C.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 Fig. 26. 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

[35] Our implementation of `hsmem_set` is actually able to apply some rewriting rules. But, this feature is still not used by our verified scheduler.

[36] 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.

```
(* 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. 25. Refinement of the operations of Figure 22 for symbolic memories with hash-consing

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 Fig. 26 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. 26. Formal specification of the two auxiliary functions used by the simulation test

D OPTIMAL ILP SCHEDULER

We provide a solver based on Integer Linear Programming (ILP), that *optimally* solves scheduling problem of Section 6.2. 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 6.3 actually computes an optimal solution (for a counter-example, see Fig. 12 vs Fig. 13).

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 6.3: 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.

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 time slot, thus all these variables are exclusive, as expressed by Equation (3) of Fig. 27. 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 $\mathbf{r}$.

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

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

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

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

Fig. 27. Turning the scheduling problem into one of ILP

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 this optimization, requiring one ILP call, would be more efficient than a sequence of decision problems, 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.

Experimental evaluation of the (sub)optimality of our list scheduler. We compiled our benchmarks with the ILP scheduling described above. The ILP scheduler outperformed our list scheduler of

Section 6.3 in only 8 of 26,161 basic blocks: for 1 block, removing 2 cycles (from 106 to 104); in others, removing 1.

As detailed above, 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.