Certified and Efficient Instruction Scheduling

Application to Interlocked VLIW Processors

CYRIL SIX, Kalray S.A., France and Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, France

SYLVAIN BOULMÉ, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, France

DAVID MONNIAUX, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, France

COMP CERT 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 COMP CERT backend for a VLIW core (*i.e.* with explicit parallelism at the instruction level), the first COMP CERT 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.

ACM Reference Format:

Cyril Six, Sylvain Boulmé, and David Monniaux. 2020. Certified and Efficient Instruction Scheduling: Application to Interlocked VLIW Processors. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 129 (November 2020), 29 pages. <https://doi.org/10.1145/3428197>

1 INTRODUCTION

The COMP CERT 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.

COMP CERT 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

Authors' addresses: Cyril Six, Kalray S.A., 180 avenue de l'Europe, Montbonnot, 38330, France, Cyril.Six@kalray.eu, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, 700 avenue Centrale, Saint Martin d'Hères, 38401, France, Cyril.Six@univ-grenoble-alpes.fr; Sylvain Boulmé, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, 700 avenue Centrale, Saint Martin d'Hères, 38401, France, Sylvain.Boulme@univ-grenoble-alpes.fr; David Monniaux, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, 700 avenue Centrale, Saint Martin d'Hères, 38401, France, David.Monniaux@univ-grenoble-alpes.fr.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2020 Copyright held by the owner/author(s).

2475-1421/2020/11-ART129

<https://doi.org/10.1145/3428197>

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 [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 K VX 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 K VX VLIW Core

The Kalray K VX 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 K VX 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 K VX, 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

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

sooner or later (e.g., an addition takes fewer stages than a multiplication). If, during the RR stage², 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

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	

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

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.

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* (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*.

Fig. 2. The Intermediate Languages of COMPCERT

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

³When 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 K VX, where these aspects are just matters of code efficiency, not correctness.

⁴For 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.

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 K VX 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 K VX 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:

- 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

⁵A *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).

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

- A generic certified scheduling checker, written in Coq, with a proof of semantic preservation, implementing two independent checks: (1) 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. (2) 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 `COMP CERT` for the K VX, including our scheduler⁷ and compared their execution time to that of the same software compiled with the reference compiler for the K VX (versions of the GNU C Compiler supplied by the chip designers).

1.5 Related Works

Our `COMP CERT` 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 `COMP CERT` 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 `COMP CERT` 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 `COMP CERT`. Finally, Section 8 presents experimental evaluations of our backend.

⁷Our full source code is available on <https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/comp-cert-kvx>.

Fig. 3. Architecture of Our Solution in COMPCERT (and of This Paper)

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, and further detailed in Appendix.

2 COMPCERT BACKEND SEMANTICS

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 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 included in those of P_1 , as formalized by Leroy [2009a,b]. 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: (1) the initial (resp. final) states of P_1 match those of P_2 ; (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.

Lock-step simulation

“Plus” simulation

Fig. 4. Examples of Simulation Diagrams in COMPCERT

2.2 The Asm IR

COMP CERT 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 COMP CERT 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 COMP CERT. 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 KVM 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 main inductive types: `basic` for basic instructions and `control` for 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 one.

```
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 $(\text{Next } rs \ m)$ 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 $(\text{bstep } b \ rsr \ rsw \ mr \ mw)$ 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 $(\text{estep } f \ ext \ sz \ rsr \ rsw \ mw)$ 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 $(\text{parexec_wio } \dots \ rsr \ mr)$ is the composition of the basic and control steps.

For example, the bundle “ $R_0 := R_1; R_1 := R_0; \text{jump}@toto$ ” 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) “ $\text{jump } @toto$ ” leads to $rsw_3 = rsw_2[PC \leftarrow @toto]$

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

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

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

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 $rs, m \leftarrow e_1$ IN e_2 ” is a notation for:

“match e_1 with Next $rs\ m \Rightarrow e_2$ | _ $\Rightarrow$ Stuck end”

```
Fixpoint parexec_wio_body bdy rsr rsw mr mw : outcome :=
match bdy with
| nil  $\Rightarrow$  Next rsw mw
| bi::bdy'  $\Rightarrow$  NEXT rsw', mw'  $\leftarrow$  bstep bi rsr rsw mr mw IN parexec_wio_body bdy' rsr rsw' mr mw'
end
```

The in-order parallel execution of a block (defined 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.

Fig. 5. Parallel In-Order Blockstep

Here, f is the current function and sz is the offset by which PC is incremented in the absence of a control-flow instruction.

```
Definition parexec_wio f bdy ext sz rs m :=
NEXT rsw', mw'  $\leftarrow$  parexec_wio_body bdy rs rsw 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 $(\text{parexec_bblock } f \ b \ rs \ m \ o)$, 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 :=
 $\exists$  bdy1 bdy2, Sorting.Permutation (bdy1 ++ bdy2) b.(body)
 $\wedge$  o = (NEXT rsw', mw'  $\leftarrow$  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 :=  $\forall$  o, parexec_bblock f b rs m o  $\rightarrow$  o = Next rs' m'
```

Given (rs', m') , the above holds only if all possible outcomes o satisfying $(\text{parexec_bblock } f \ b \ rs \ m \ o)$ are exactly $(\text{Next } rs' \ m')$; 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  $\Rightarrow$  Next rs m
  | bi::bdy'  $\Rightarrow$  NEXT rs', m'  $\leftarrow$  bstep bi rs rs m m IN exec_body bdy' rs' m'
end
Definition exec_bblock f b rs m: outcome :=
  NEXT rs', m'  $\leftarrow$  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.

In Coq, the scheduler is declared as a function⁹ 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  $\rightarrow$  (list (list basic))*(option control)

```

⁹The scheduler is declared as a pure function like other COMPCERT oracles.

¹⁰It would be unsound to declare `schedule` returning directly a value of type “`list bblock`”, since the “correct” proof field of `bblock` does not exist for the OCAML oracle.

Fig. 6. Certified Scheduling from Untrusted Oracles

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 1b 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 1b 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 1b 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).

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 “ $r_1 := \text{add}[r_2, r_3]$ ”;
- (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 “ $r_1 := (\text{load } ofs)[m, r_2]$ ”;
- (3) the store of register r_1 into memory address “ $ofs[r_2]$ ” is written “ $m := (\text{store } ofs)[m, r_1, r_2]$ ”.

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 assignment 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 summarize how we used AbstractBasicBlock to certify the scheduling of Asmblock/AsmVLIW. See Section 5 for a more detailed presentation.

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

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](#).

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` →
`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, `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' := $\forall$ 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`: $\forall$ ge f B lb,
`(verified_schedule B) = (OK lb) →  $\exists$  tb, is_concat tb lb  $\wedge$  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,

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

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) $r_1 := r_1 + r_2$; $r_3 := \text{load}[m, r_2]$; $r_1 := r_1 + r_3$

(B_2) $r_3 := \text{load}[m, r_2]$; $r_1 := r_1 + r_2$; $r_1 := r_1 + r_3$

They are both equivalent to this parallel assignment:

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

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 bisimulable¹³ as pictured on Fig 7.

Fig. 7. Diagram of `bblock_simu` Correctness

Example 4.3 (Simulation on symbolic memories). Consider:

(B_1) $r_1 := r_1 + r_2$; $r_3 := \text{load}[m, r_2]$; $r_3 := r_1$; $r_1 := r_1 + r_3$

(B_1) $r_3 := r_1 + r_2$; $r_1 := r_3 + r_3$

Both B_1 and B_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, 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 "`load[m, r2]`" 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.

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.

¹³A "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.

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: $f(t_1, \dots, t_m)$ and $g(u_1, \dots, u_n)$ 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 “ $\leadsto$ ” is the may-return relation of the IMPURE library: “`phys_eq x y  $\leadsto$  true`” means that “true” is a *possible* result of “`phys_eq x y`”. In other words, even if “`phys_eq x y  $\leadsto$  true`” and “`phys_eq x y  $\leadsto$  b`”, then we cannot conclude that “`b=true`” (we could also have “`b=false`”). The IMPURE library does not even assume that “ $\forall x, \text{phys_eq } x \ x \leadsto \text{true}$ ”.

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

```
Axiom phys_eq:  $\forall \{A\}, A \rightarrow A \rightarrow ??\text{bool}$ 
Extract Constant phys_eq  $\Rightarrow$  “==”
Axiom phys_eq_true:  $\forall A (x \ y : A),$ 
  phys_eq x y  $\leadsto$  true  $\rightarrow x = y$ 
```

¹⁴Hence, if we model pointer equality (OCAML’s `==`) as an infix function “`phys_eq :  $\forall \{A\}, A \rightarrow A \rightarrow \text{bool}$ ”`, then we are able to prove this wrong proposition (when x and y are structurally equals but are distinct pointers):

$$y = x \wedge (\text{phys_eq } x \ y) = \text{false} \wedge (\text{phys_eq } x \ x) = \text{true} \rightarrow \text{false} = \text{true}$$

¹⁵In 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.

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: `COMP CERT` 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 sequence of assignments* (ASA). A form of parallelism may occur in such a sequence through the special operator `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 $a = d_0$:

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

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 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](#)). 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` $\rightarrow$ `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  $\rightarrow$  mem  $\rightarrow$  Prop; post: R.t  $\rightarrow$  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  $\rightarrow$   
   $\forall$  m m', run ge p m = Some m'  $\leftrightarrow$  (d.pre) ge m  $\wedge$   $\forall$  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  $\wedge$   $\forall$  x, ht_eval ge (d.post) x) m  $\neq$  None  
Definition smem_simu (d1 d2: smem): Prop :=  $\forall$  ge m, smem_valid ge d1 m  $\rightarrow$   
  ( smem_valid ge d2 m  $\wedge$  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)  $\rightarrow$   
   $\forall$  ge m, (run ge p1 m)  $\neq$  None  $\rightarrow$  (run ge p2 m) = (run ge p1 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 w.r.t. 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 :=  
   $\forall$  ht, List.In ht hd.hpre  $\rightarrow$  ht_eval ge ht m  $\neq$  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  $\Rightarrow$  Some (m x) | Some ht  $\Rightarrow$  ht_eval ge ht m end  
(* The data-refinement relation *)  
Definition smem_model ge (d: smem) (hd:hsmem): Prop :=  
  ( $\forall$  m, hsmem_valid ge hd m  $\leftrightarrow$  smem_valid ge d m)  
   $\wedge$   $\forall$  m x, smem_valid ge d m  $\rightarrow$  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 _”

```

(* 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

and its bind operator noted “ $\text{DO } _ \leftarrow _ ; _$ ”. 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](#)) 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](#)), solved by an external tool (e.g., Gurobi).

$$\begin{array}{lcl}
 \text{Find } t : 0 \dots n \rightarrow \mathbb{N} \text{ satisfying:} & & \\
 \text{A resource usage constraint} & \left| \right. & \text{Latency constraints of the kind (where } j' > j\text{):} \\
 \forall i \sum_{j|t(j)=i} u(K(j)) \leq r & (1) & t(j') - t(j) \geq \delta \quad (2)
 \end{array}$$

Fig. 10. Our Family of Scheduling Problems

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 $t : 0 \dots n - 1 \rightarrow \mathbb{N}$ 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 $t(j') - t(j) \geq 0$.

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 K VX 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 $r \in \mathbb{N}^m$ and, for each instruction kind k , a vector $u(k) \in \mathbb{N}^m$. The constraint is that the sum of all $u(K(j))$ for all instructions j scheduled within the same bundle i should be coordinate-wise less than or equal to 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.

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 .

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

Algorithm 1: Sketch of our List Scheduler

```

 $i := 0 \parallel Q := \{0 \dots n - 1\}$ 
while  $Q \neq \emptyset$  do
   $R := \emptyset \parallel a := r$ 
  for  $j' \in Q$  do
     $ready := \text{true}$ 
    for  $j \xrightarrow{\delta} j' \in G$  do
      if  $t(j) > j' - \delta$  then  $ready := \text{false};$ 
    if  $ready$  then  $R := R \cup \{j'\};$ 
  for  $j \in R$  (in descending  $l(j, n)$  order) do
    if  $a \geq u(K(j))$  then
       $a := a - u(K(j)) \parallel Q :=$ 
       $Q \setminus \{j\} \parallel t(j) := i$ 
   $i := i + 1$ 

```

instructions j to schedule at time i (those for which $t(j) = i$). Our Algorithm 1 chains two ideas¹⁶:

Maximal Scheduling Sets Assume we have already chosen a set S of instructions to be scheduled at time i , such that $\sum_{j \in S} u(K(j)) \leq r$. Assume there is $j' \notin S$ such that all its operands are ready, and $\sum_{j \in S \cup \{j'\}} u(K(j)) \leq r$. 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 .

¹⁶This 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.


```

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

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).

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 [Appendix](#).

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.

¹⁷See the `igoto` instruction in Section 2.2.

Fig. 14. Diagram of the Simulation Proof

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](#) 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](#) 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 amount of readaptations. More details can be found in [Appendix](#).

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

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

8 EXPERIMENTAL EVALUATION

Our implementation⁷ 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 across, 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.

Fig. 15. Scheduling Times

8.2 Benchmarks Used

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

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

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.

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 instructions 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

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

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.

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 GCC²² compiler supplied by Kalray, adapted from version 7.5.0, at -O0...-O3 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.

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.

²¹But, 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.

²²The GNU Compiler Collection, <https://gcc.gnu.org/>

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).

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 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, Gergő Barany, Thomas Vandendorpe as well as the anonymous reviewers for their helpful remarks.

APPENDIX

A version of this paper, extended with appendices, is available on
<https://hal.archives-ouvertes.fr/hal-02185883>

REFERENCES

- Gergő 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.
- 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/castanedadacarlssonea-toplas-2019.html> Accepted for publication.
- Benoît Dupont de Dinechin. 2004. From Machine Scheduling to VLIW Instruction Scheduling. *ST Journal of Research* 1, 2 (September 2004), 1–35. <https://www.cri.enscm.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), Vol. 55)*, Martin Schoeberl (Ed.). 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, Vol. 18)*, Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm (Eds.). 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>
- 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/round-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-Baptiste 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.