

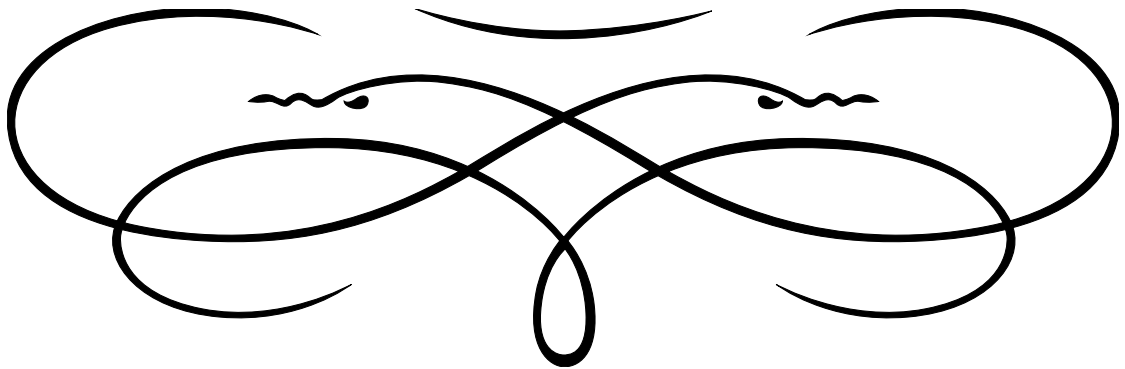
ÉCOLE NORMALE SUPÉRIEURE DE LYON

---

**Optimisations en compilation certifiée**  
— Encadré par David Monniaux, VERIMAG —

---

NICOLAS NARDINO



## Abstract

We try to develop a prepass instruction scheduling oracle for COMPCERT that is aware of register pressure, in order to minimise spills, while retaining an efficient schedule.

We start from the existing `list_scheduler`, a greedy algorithm. We modify it to track register usage, and when the pressure reaches a set threshold, we issue an instruction which decreases it the most.

We try different methods of dealing with edge-cases, and benchmark them to choose which one to keep.

## Remerciements

Je tiens à remercier David Monniaux, mon maître de stage, pour les idées, pour l'aide apportée, et simplement pour la possibilité de faire ce stage ; Sylvain Boulmé, co-encadrant, pour les mêmes raisons ; ainsi que l'ensemble de l'équipe VERIMAG pour l'accueil, et le café.

## Contents

<b>1. Introduction</b>	<b>3</b>
1.1. COMPCERT . . . . .	3
1.2. Prepass scheduling, RTL and LTL . . . . .	3
1.3. Current oracles . . . . .	4
1.4. Challenging the current oracles . . . . .	5
<b>2. The algorithm</b>	<b>6</b>
2.1. Skeleton . . . . .	7
2.2. Pre-computations . . . . .	7
2.3. Tracking liveness . . . . .	8
2.4. Edge cases . . . . .	8
<b>3. A variant of <code>list_scheduler</code> with another priority function</b>	<b>9</b>
<b>4. Benchmarks</b>	<b>9</b>
4.1. Empirically determining an optimal threshold . . . . .	9
4.2. In number of spills . . . . .	10
4.3. Output code performance . . . . .	11
<b>5. Conclusion</b>	<b>14</b>
<b>Appendix A. Source code</b>	<b>16</b>
<b>Annexe B. Contexte institutionnel</b>	<b>29</b>

# 1. Introduction

## 1.1. COMPCERT

The correctness of compilers is a key part of producing correct programs. Even if the source code of a program has a formal proof of correctness, there may be a bug in the compiler, and we have no reason to believe that the output binaries do what we want.

Several studies have found bugs in many compilers ([ER08], [YCER11]). These bugs are insignificant in everyday use, but when dealing with critical software, upon which highly sensitive information, or even lives may depend—*i.e.* when a proof of correctness of the program is expected—then these issues need to be solved.

The COMPCERT<sup>1</sup> compiler is an answer to this issue, as it ensures that the output program “behaves as expected”: the following theorem has been formally proven.

### Theorem 1.1 (Semantic preservation)

For all source programs  $S$  and compiler-generated code  $C$ , if the compiler, applied to the source  $S$ , produces the code  $C$ , without reporting a compile-time error, then the observable behavior of  $C$  is one of the possible observable behaviors of  $S$ .

Compilation in COMPCERT is done in several passes, from one Intermediate Language (IL) to the next, and for each of these steps, there is a proof of the semantic preservation theorem, written with the COQ proof assistant.

VERIMAG is maintaining its own fork of the COMPCERT compiler, in which some optimisations, rather than being written in COQ, use oracle/validator pairs: the oracle, in OCaml, outputs a possible reduction of the input code, and the validator verifies that the symbolic execution of both yields the same behaviours.

In the following, COMPCERT designates the fork of the COMPCERT compiler maintained by VERIMAG.

## 1.2. Prepass scheduling, RTL and LTL

Instruction scheduling is an optimisation pass found in all optimising compilers, when compiling towards fully pipelined, *in-order* architectures, such as ARM/Aarch64. Its purpose is to re-order the instructions to minimise stalling, without breaking any data dependency.

In COMPCERT, the main instruction scheduling pass is done before register allocation, allowing for more flexibility as the only data dependencies present are true dependencies. However, the current oracles are unaware of register pressure, thus can produce a schedule with more live registers than the architecture allows for, resulting in more spills than necessary, and lower performances.

Register allocation happens between two ILs of COMPCERT: RTL and LTL. Both are generic assembly languages, in which the code is represented by *Control Flow Graphs* (CFG), separated into *superblocks*. The main difference—there are others but they don’t matter to us—is that RTL works with an infinite number of pseudo-registers, while LTL uses the actual registers of the architecture we are compiling towards.

---

<sup>1</sup><https://compcert.org>

Prepass instruction scheduling is the very last optimisation pass of the RTL step, just before register allocation.

We are trying to develop a prepass scheduling oracle that is aware of register pressure, in order to reduce unnecessary spills. Or more precisely, register pressures, since most architectures have several *classes* of registers—usually, *general-purpose registers*, to store address and integer data, and *floating-point registers* specialised towards floating-point data and operations, usually bigger than general-purpose registers.

### 1.3. Current oracles

There are several oracles currently implemented in COMPCERT, but the main one that is actually used is a greedy algorithm (`list_scheduler`). When benchmarked, the code produced by the other ones was almost never more efficient, and when it was, it was only by a small margin.

These oracles are at the core, algorithms for finding a topological order in a directed graph, with a few quirks:

- There are time constraints between instructions
- If the resources allow it, multiple instructions can be scheduled at the same time

`list_scheduler` works as follows:

---

#### Algorithm 1 `list_scheduler`

---

**Require:** A list of constraints between instructions

**Ensure:** A correct schedule of said instructions

Compute the longest paths from each instruction to the exit

Compute a maximum time bound for the problem, from constraints

Create a set of *ready* instructions: instructions which can be scheduled now

**repeat**

    Find an instruction with the longest path to the exit in the *ready* set

**if** None can be scheduled **then**

        Advance time

        Update resources

*(\* Go to the next cycle \*)*

**else**

        Schedule it at current time

        Update the *ready* set

        Update resources

**until** The time bound is reached

**if** All instructions have been scheduled **then**

**return** The schedule

**else**

**return** None

*(\* The original order will be kept \*)*

---

In short, the algorithm prioritises instructions with long paths to the exit. The resulting schedules

usually start with a long sequence of initialisations, leading to registers being live for longer, and potentially, spills.

The same algorithm is also used in an other way: the problem is reversed, then a schedule is computed, and then reversed (`rev_list_scheduler`). This leads to initialisations being scheduled as late as possible, leading to maybe less efficient schedules, but fewer spills.

However, these are just general observations from benchmarks, and since these oracles are insensitive to register pressure, there may exist examples where the schedule produced by them is less efficient than the original order, because of spills created.

#### 1.4. Challenging the current oracles

Finding such examples can be a first step in developing new oracles. All tests were done with Aarch64 as the target architecture, with 29 available general-purpose registers, and 32 floating-point registers.

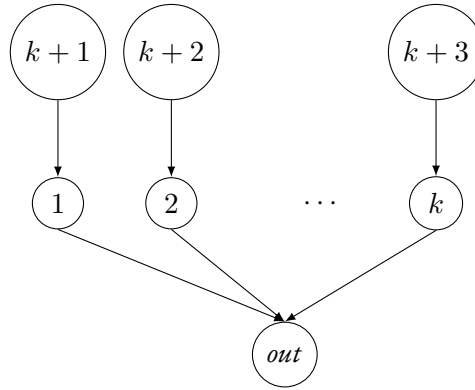
##### Challenging `rev_list_scheduler`

```
int f(int k) {
    int a1 = k;
    int b1 = 2*a1;
    int c = a1;
    /* ... */
    int a26 = k+25;
    int b26 = 2*a26;
    c += a26;
    return b13 + /* ... */ b1 +
        b14 + /* ... */ b26 + c;
}
```

This was the first example we found, and the one which was the most surprising at first. Indeed, it does not spill when scheduled in the original order, or when using `list_scheduler`. However, spills appear when scheduling using `rev_list_scheduler`, which seems to contradict the observations mentioned earlier.

But this can be explained by taking a closer look at the way `rev_list_scheduler` schedules:

Figure 1: Scheduling of a dependency graph by `rev_list_scheduler`



Each node is an instruction, and its label is its position in the output schedule of `revlist`, in reverse order, starting at the end. In short, if all paths have same length, it schedules in a breadth-first order, starting at the output. Therefore, in our example, where we have a dependency graph of the sort, all the arithmetic operations will be scheduled last, and all the initialisations, at the beginning, leading to too many live registers at once, and thus, spills.

### Challenging `list_scheduler`

Surprisingly few changes were needed to find an example that makes `list_scheduler` spill:

```

int f(int k) {
    float a1 = (float) k;
    float b1 = 2.*a1;
    float c = a1;
    /* ... */
    float a30 = (float) k+29;
    float b30 = 2.*a30;
    c += a30;
    return c + b1 + /* ... */ b30;
}

```

There is however a small caveat: this example works because of architecture resources limitations. Indeed, `list_scheduler` will schedule the first load (`a1`), then the first floating-point operation (`b1`) (because the necessary resources are free), but then, since float operations take a long time, it will schedule the next load (`a2`), then the operation on it (`b2`). All the additions on `c` will be delayed to the end, so we need to keep the values of `a<i>` in live registers, or to spill them.

## 2. The algorithm

The oracle is largely inspired by [GH88]

## 2.1. Skeleton

The main skeleton of the algorithm is based on `list_scheduler`, with a few changes:

- we track register liveness during the whole execution (see 2.3)
- when the pressure gets too high, we switch to another mode, prioritising instructions that reduce it

There are also a few pre-computations needed, due to the current representation of superblocks in COMPCERT, which will be detailed later (see 2.2).

---

### Algorithm 2 `reg_pres_scheduler`

---

**Require:** A list of constraints between instructions, live entry registers for the superblock, information on register usage by each instruction

**Ensure:** A correct schedule of said instructions

Compute the longest paths from each instruction to the exit

Compute a maximum time bound for the problem, from constraints

Create a set of *ready* instructions: instructions which can be scheduled now

Initialise necessary data structures (see 2.3)

**repeat**

**if** The pressure for register class  $t$  is above a set threshold (see 4.1) **then**

        Find an instruction which decreases pressure along this class the most

**if** No instruction can reduce it **then**

            See 2.4

**else**

        Find an instruction with the longest path to the exit in the *ready* set

**if** None can be scheduled **then**

        Advance time

        Update resources & pressures

*(\* Go to the next cycle \*)*

**else**

        Schedule it at current time

        Update the *ready* set

        Update resources & pressures

**until** The time bound is reached

**if** All instructions have been scheduled **then**

**return** The schedule

**else**

**return** None

*(\* The original order will be kept \*)*

---

## 2.2. Pre-computations

Since previous scheduling oracles in COMPCERT work only from a list of constraints, we need to add more information to the problem type (oracles have type `problem -> solution option`).

- The list of live input registers of the superblock

- For each instruction, the list of argument and destination registers
- Typing information for each register (already present in the `superblock` type)
- For each register, a count of the number of times it's referenced as an argument (we use reference counting to track down when registers are freed)

### Getting live input registers

In the `superblock` type in `COMP CERT`, we have the information of live output registers, therefore, we only need to go back the sequence of instructions, removing destination registers from the list, and adding back argument registers.

### Reference counting

In a traversal of the sequence of instructions, for each instruction, we add for each argument register a binding in a hash table, along with its class, to the number of time it has been encountered so far. In the same traversal, we can build our lists of argument and destination registers.

### 2.3. Tracking liveness

To track pressure, we use an array of size *the number of register classes*, with initial values *the number of available register for each class for the target architecture*. Each time a new register becomes live, we decrement the value corresponding to its class, and increment it when a register is used as an argument for the last time. To track this moment, we use reference counting, as previously mentioned: the number of times a given register is referenced as an argument has been pre-computed, so when we issue an instruction, we iterate on its list of arguments (pre-computed), and decrement the reference count of each argument register. We use another hash table to track which registers are live (in order to know when new register become live).

### 2.4. Edge cases

Several solutions for dealing with the case *the pressure is above the threshold, but no instruction that can be scheduled decrease it* were tested. The first one, which was a temporary solution, was to use a waiting window, of at most 5 cycles, to see if such an instruction becomes available. If we are still in the same situation after 5 cycles, we schedule an instruction as if the pressure was below the threshold.

This temporary solution was used for the first benchmarks, and was the most effective when it comes to reducing the number of spills (see 4.2). However, stalling the processor is not ideal when trying to produce efficient code. The next idea, was to then schedule an instruction with the shortest path to the exit, with the assumption that when its execution path ends, registers will be freed [GH88]. However, this lead to a significant increase in spills, compared to `list_scheduler`, which we could not explain.

What worked in the end was to do the opposite: schedule instructions with the longest path to the exit, which do not increase register pressure.

This still leaves the case: all schedulable instructions increase register pressure. For this situation, we used a waiting window.



### 3. A variant of `list_scheduler` with another priority function

As previously mentioned, `list_scheduler` is at its core, an algorithm for finding a topological order in a directed graph. For choosing the next instruction to schedule when several are available, we use a total order on instructions: a lexicographical order on  $(-c, i)$ , where  $c$  is the length of a critical path, and  $i$  the original position of the instruction. It is a strict and total order on instructions, and we then choose the next instruction to be minimum for it, and *issuable* (*i.e.* it is ready, and the resources are free).

Another idea for a scheduler that is sensitive to register pressure was to always choose, at equal critical path lengths, the instruction with the smallest pressure delta. This can be implemented by changing this ordering function to a lexicographical order on  $(-c, \delta, i)$ , with  $\delta$  the register pressure delta of the instruction. However, this ordering function is not static, it changes along the execution of the algorithm. For example, given two instructions:

```
A: X0 = 237;  
B: X0 = 612;
```

Assuming X0 was not already alive, if A has already been scheduled, then B has a delta of 0, but of +1 otherwise.

This also means we don't differentiate between an instruction which frees a register of one class, and captures a new register of another, and an instruction with a null delta on all classes.

This variant proved to be very ineffective at reducing spills when benchmarked (see 4.2).

### 4. Benchmarks

Benchmarks were done on a set of 150 programs generated by YARPGEN<sup>2</sup>.

Performance benchmarks were executed on a Raspberry Pi with an ARM Cortex-A53 (Aarch64)

There are two significant values we are measuring here through benchmarks: the number of spills produced by a particular schedule, and the number of processor cycles elapsed during its execution.

For the first, we count the number of matches of pattern

```
local\([0-9]+\,[a-z]+\)= (X|D)[0-9]+
```

in the LTL code generated during compilation.

It should be noted that among the 150 programs generated by YARPGEN, only about a third had spills with `list_scheduler`.

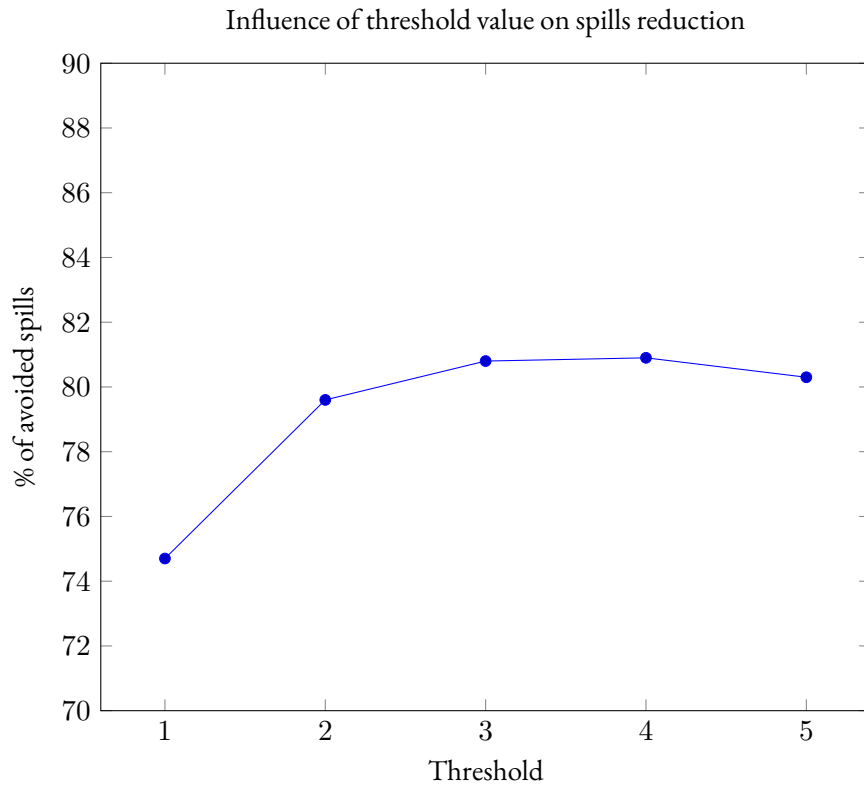
For the second, we simply execute the program, while locked on a single core, and measure the difference between the clock time before and after execution.

#### 4.1. Empirically determining an optimal threshold

Determining an optimal threshold on the number of free registers below which our scheduler changes mode was done through benchmarks.

---

<sup>2</sup><https://github.com/intel/yarpgen>

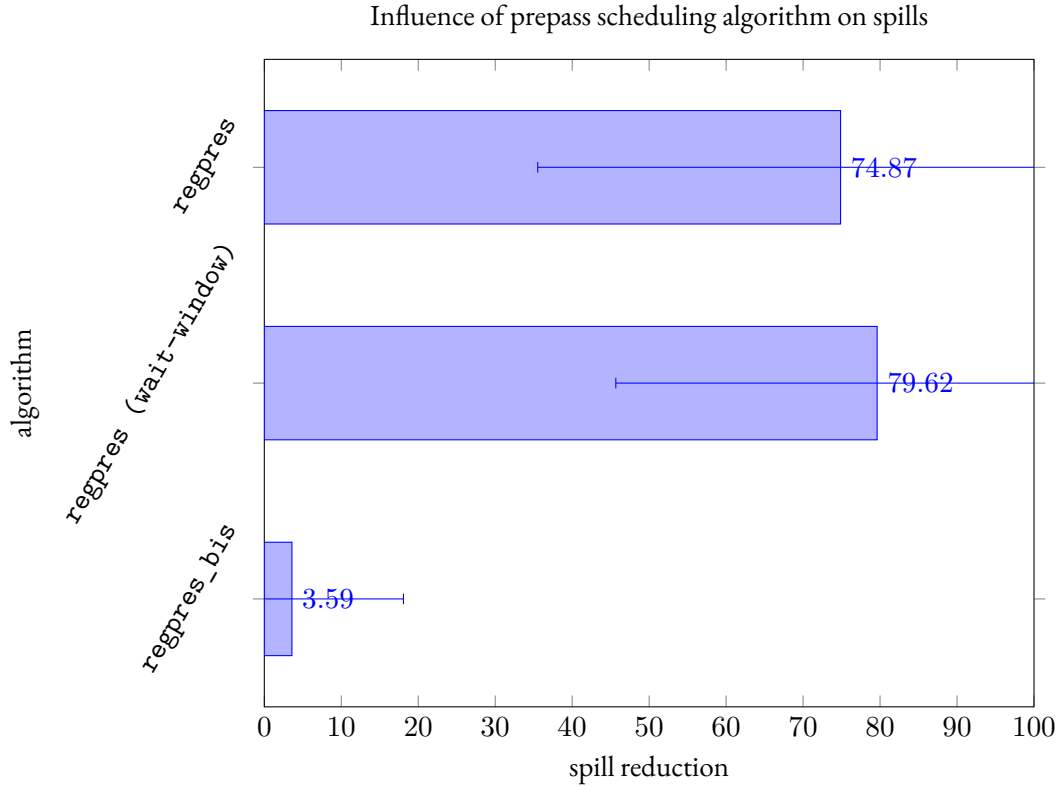


Tests for higher threshold values showed no improvement, the proportion of avoided spills stabilises around 80%. Therefore, 2 was chosen as a default value.

#### 4.2. In number of spills

The first thing that should be noted, is that out of the 150 programs that were used for benchmarks, spills appeared with `list_scheduler` in only 54. Given the way our algorithm is designed, in all the others, the schedules should be identical (which was verified).

The next graph shows the performances of our different variants, in terms of spill reduction compared to `list_scheduler`.



The error bars show standard deviation.

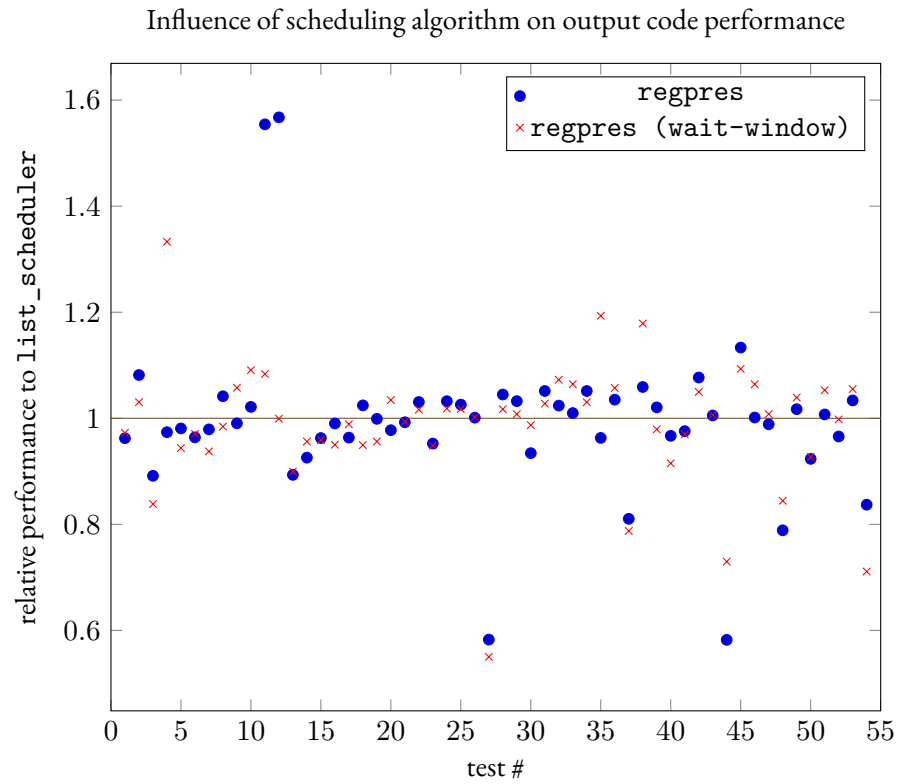
As previously mentioned, the `wait-window` option performs slightly better when it comes to the number of spills, and `regpres_bis`, the variant described in 3 is almost useless.

We can also see that for the two versions of `regpres`, the standard deviation is consequential. Indeed, when looking at the raw numbers, while for most of the test programs, the spills were completely eliminated (or close to), there were some that showed little to no improvement, and even a few cases of regression (*e.g.* 6 spills with `list_scheduler`, and 8 with `regpres`).

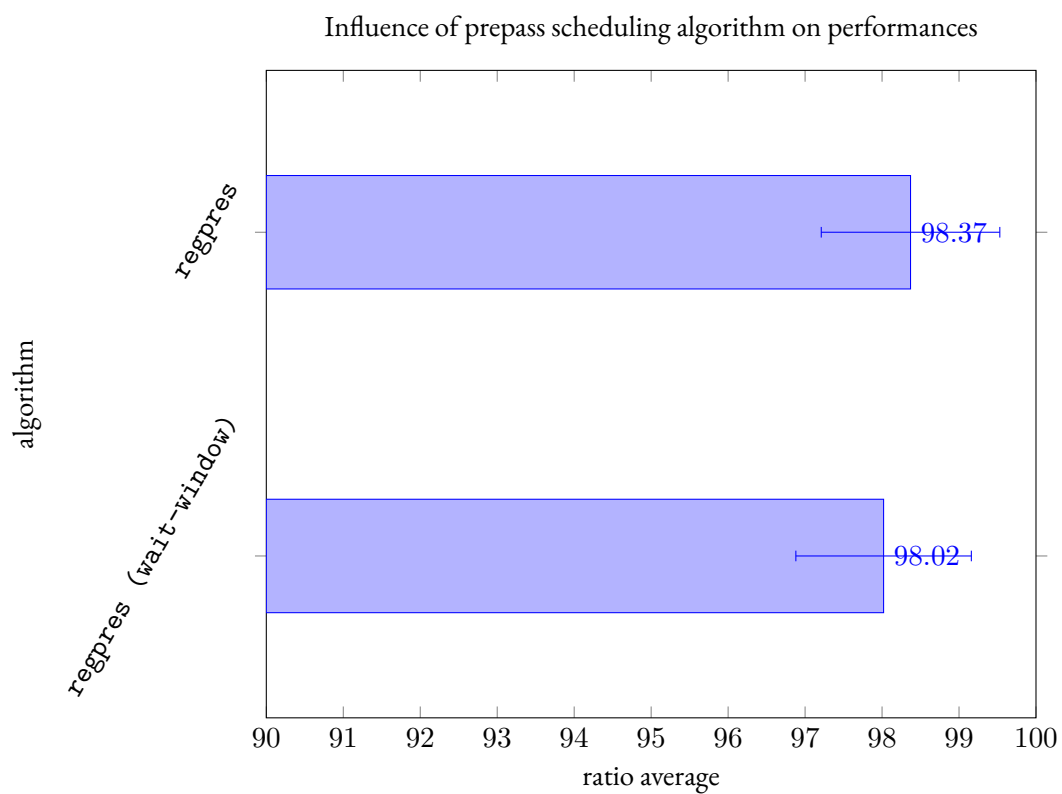
It would be tempting to say that the `wait-window` option should be kept as default, since it appears to be the most efficient at eliminating spills. However, we have to remember that it can produce significantly longer schedules, as it can introduce unnecessary bubbles to wait for a “more suitable” instruction. We therefore have to benchmark the output schedules on execution time.

### 4.3. Output code performance

The next graph shows, for each test program that features spills when scheduled with `list_scheduler`, the ratio between the number of cycles of its execution when compiled with the given prepass scheduling optimisation, and `list_scheduler`.

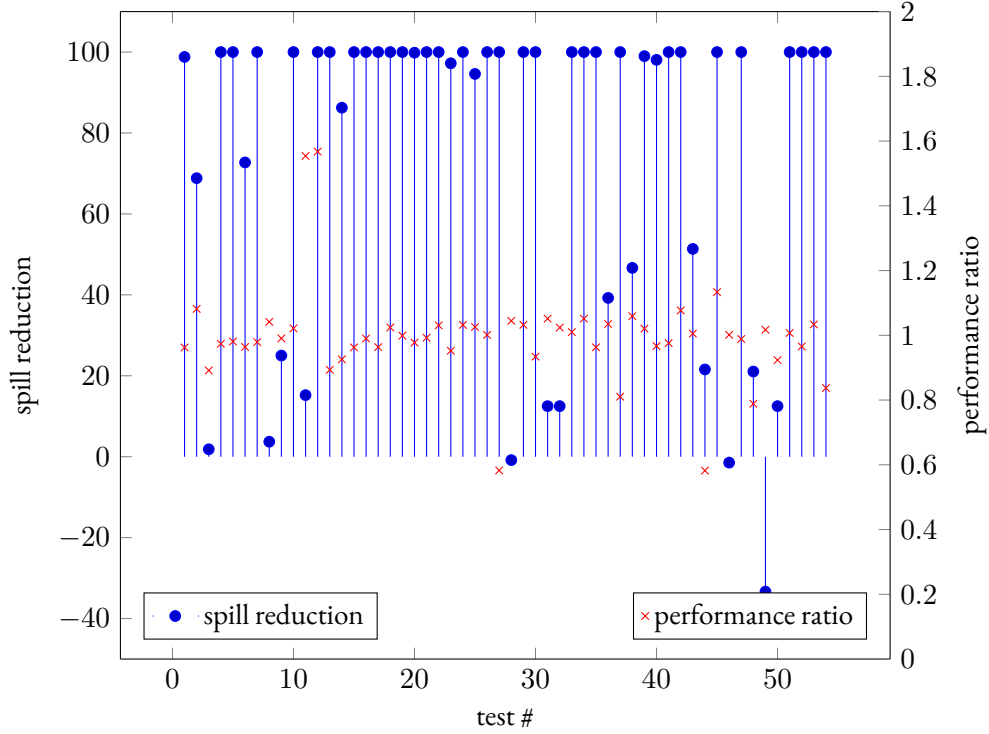


As we can see, appart from a few exceptions (in both directions), performances are pretty similar between the algorithms, and indeed, when computing the geometric means of these results:



It seems that both of them produce schedules that are, on average, slightly more efficient than `list_scheduler`, and that there is no significant difference between the two.

It could be interesting to see whether the tests that showed worse performances are those for which `regpres` couldn't reduce the number of spills.



But there doesn't seem to be any correlation.

We have to keep in mind that these test programs are not representative of real-world applications. YARPGEN is a tool designed to produce programs that trigger compilation bugs. We tried to compile GNU programs to benchmark compilation, but there were too many compilation issues, and we didn't have the time to solve them.

## 5. Conclusion

Our new algorithm appears to be very effective at reducing spills, but the resulting code does not appear to be significantly faster, even for the programs in which many spills were eliminated. This result was expected: `list_scheduler` is used in COMPCERT because most of the time, the schedules it produces are “very good”. As previously mentioned, there are other oracles that are implemented in COMPCERT, for example an oracle which uses integer linear programming to find a “better” solution, from an initial solution computed by `list_scheduler`. But in almost all cases, the ILP solver cannot find a better schedule.

However, the aim of `regpres` is not to produce significantly faster schedules in general, but to prevent situations, which may be rare (and the rarer the more registers the target architecture possesses), in which prepass scheduling introduces an unnecessarily high number of spills. This is also why `regpres` is designed to function identically to `list_scheduler` “most of the time” (*i.e.* as long as the pressure is below the threshold), and produces schedules which are therefore similar to those produced by `list_scheduler`.

## References

- [ER08] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, page 255–264, New York, NY, USA, 2008. Association for Computing Machinery.
- [GH88] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 2nd International Conference on Supercomputing*, ICS '88, page 442–452, New York, NY, USA, 1988. Association for Computing Machinery.
- [YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

## Appendix A. Source code

```

(** the useful one. Returns a hashtable with bindings of shape
** [(r,(t, n))], where [r] is a pseudo-register (Registers.reg),
** [t] is its class (according to [typing]), and [n] the number of
** times it's referenced as an argument in instructions of [seqa] ;
** and an array containg the list of regs referenced by each
** instruction, with a boolean to know whether it's as arg or dest *)
let reference_counting (seqa : (instruction * Regset.t) array)
  (out_regs : Registers.Regset.t) (typing : RTLtyping.regenv) :
  (Registers.reg, int * int) Hashtbl.t *
  (Registers.reg * bool) list array =
  let retl = Hashtbl.create 42 in
  let retri = Array.make (Array.length seqa) [] in
  (* retri.(i) : (r, b) -> (r', b') -> ...
  * where b = true if seen as arg, false if seen as dest
  *)
  List.iter (fun reg ->
    Hashtbl.add retl
      reg (Machregsaux.class_of_type (typing reg), 1)
  ) (Registers.Regset.elements out_regs);
  let add_reg reg =
    match Hashtbl.find_opt retl reg with
    | Some (t, n) -> Hashtbl.add retl reg (t, n+1)
    | None -> Hashtbl.add retl reg (Machregsaux.class_of_type
      (typing reg), 1)
  in
  let map_true = List.map (fun r -> r, true) in
  Array.iteri (fun i (ins, _) ->
    match ins with
    | Iop(_,args,dest,_) | Iload(_,_,_,args,dest,_) ->
      List.iter (add_reg) args;
      retri.(i) <- (dest, false)::(map_true args)
    | Icond(_,args,_,_,_) ->
      List.iter (add_reg) args;
      retri.(i) <- map_true args
    | Istore(_,_,args,src,_) ->
      List.iter (add_reg) args;
      add_reg src;
      retri.(i) <- (src, true)::(map_true args)
    | Icall(_,fn,args,dest,_) ->
      List.iter (add_reg) args;
      retri.(i) <- (match fn with
        | Datatypes.Coq_inl reg ->
          add_reg reg;
          (dest,false)::(reg, true)::(map_true args)
        | _ -> (dest,false)::(map_true args))
    | Itailcall(_,fn,args) ->

```



```

List.iter (add_reg) args;
retr.(i) <- (match fn with
  | Datatypes.Coq_inl reg ->
    add_reg reg;
    (reg, true)::(map_true args)
  | _ -> map_true args)
| Ibuiltin(_,args,dest,_) ->
  let rec bar = function
    | AST.BA r -> add_reg r;
    retr.(i) <- (r, true)::retr.(i)
    | AST.BA_splitlong (hi, lo) | AST.BA_addptr (hi, lo) ->
      bar hi; bar lo
    | _ -> ()
  in
  List.iter (bar) args;
  let rec bad = function
    | AST.BR r -> retr.(i) <- (r, false)::retr.(i)
    | AST.BR_splitlong (hi, lo) ->
      bad hi; bad lo
    | _ -> ()
  in
  bad dest;
  | Ijumpable (reg,_) | Ireturn (Some reg) ->
    add_reg reg;
    retr.(i) <- [reg, true]
  | _ -> ()
) seqa;
(* print_string "mentions\n";
 * Array.iteri (fun i l ->
 *   print_int i;
 *   print_string ": [";
 *   List.iter (fun (r, b) ->
 *     print_int (Camlcoq.P.to_int r);
 *     print_string ":";
 *     print_string (if b then "a:" else "d");
 *     if b then print_int (snd (Hashtbl.find retl r));
 *     print_string ", "
 *   ) l;
 *   print_string "]\n";
 *   flush stdout;
 * ) retr; *)
retl, retr

let get_live_regs_entry (sb : superblock) code =
  (if !Clflags.option_debug_compcert > 6
   then debug_flag := true);
  debug "getting live regs for superblock:\n";
  print_superblock sb code;

```

```

debug "\n";
let seqa = Array.map (fun i ->
  (match PTree.get i code with
  | Some ii -> ii
  | None -> failwith "RTLpathScheduleraux.get_live_regs_entry"
  ),
  (match PTree.get i sb.liveins with
  | Some s -> s
  | None -> Regset.empty))
  sb.instructions in
let ret =
  Array.fold_right (fun (ins, liveins) regset_i ->
    let regset = Registers.Regset.union liveins regset_i in
    match ins with
    | Inop _ -> regset
    | Iop (_, args, dest, _) ->
    | Iload (_, _, _, args, dest, _) ->
      List.fold_left (fun set reg -> Registers.Regset.add reg set)
        (Registers.Regset.remove dest regset) args
    | Istore (_, _, args, src, _) ->
      List.fold_left (fun set reg -> Registers.Regset.add reg set)
        (Registers.Regset.add src regset) args
    | Icall (_, fn, args, dest, _) ->
      List.fold_left (fun set reg -> Registers.Regset.add reg set)
        ((match fn with
          | Datatypes.Coq_inl reg -> (Registers.Regset.add reg)
          | Datatypes.Coq_inr _ -> (fun x -> x))
          (Registers.Regset.remove dest regset))
          args
    | Itailcall (_, fn, args) ->
      List.fold_left (fun set reg -> Registers.Regset.add reg set)
        (match fn with
          | Datatypes.Coq_inl reg -> (Registers.Regset.add reg regset)
          | Datatypes.Coq_inr _ -> regset)
          args
    | Ibuiltin (_, args, dest, _) ->
      List.fold_left (fun set arg ->
        let rec add reg set =
          match reg with
          | AST.BA r -> Registers.Regset.add r set
          | AST.BA_splitlong (hi, lo)
          | AST.BA_addptr (hi, lo) -> add hi (add lo set)
          | _ -> set
        in add arg set)
        (let rec rem dest set =
          match dest with
          | AST.BR r -> Registers.Regset.remove r set
          | AST.BR_splitlong (hi, lo) -> rem hi (rem lo set)
          | _ -> set

```

```

        in rem dest regset)
        args
    | Icond (_, args, _, _, _) ->
        List.fold_left (fun set reg ->
            Registers.Regset.add reg set)
            regset args
    | Ijumpable (reg, _)
    | Ireturn (Some reg) ->
        Registers.Regset.add reg regset
    | _ -> regset
) seqa sb.s_output_regs
in debug "live in regs: ";
print_regset ret;
debug "\n";
debug_flag := false;
ret

(* A scheduler sensitive to register pressure *)
let reg_pres_scheduler (problem : problem) : solution option =
    (* DebugPrint.debug_flag := true; *)

    let nr_instructions = get_nr_instructions problem in

    if !Clflags.option_debug_compcert > 6 then
        (Printf.eprintf "\nSCHEDULING_SUPERBLOCK %d\n" nr_instructions;
         flush stderr);

    let successors = get_successors problem
    and predecessors = get_predecessors problem
    and times = Array.make (nr_instructions+1) (-1) in
    let live_regs_entry = problem.live_regs_entry in

    let available_regs = Array.copy Machregsaux.nr_regs in

    let nr_types_regs = Array.length available_regs in

    let thres = Array.fold_left (min)
        (max !(Clflags.option_regpres_threshold) 0)
        Machregsaux.nr_regs
    in

    let regs_thresholds = Array.make nr_types_regs thres in

    let class_r r =
        Machregsaux.class_of_type (problem.typing r) in

```

```

let live_regs = Hashtbl.create 42 in

List.iter (fun r -> let classe = Machregsaux.class_of_type
                    (problem.typing r) in
            available_regs.(classe)
            <- available_regs.(classe) - 1;
            Hashtbl.add live_regs r classe)
  (Registers.Regset.elements live_regs_entry);

let csr_b = ref false in

let counts, mentions =
  match problem.reference_counting with
  | Some (l, r) -> l, r
  | None -> assert false
in

let fold_delta i = (fun a (r, b) ->
  a +
  if class_r r <> i then 0 else
    (if b then
      if (Hashtbl.find counts r = (i, 1))
      then 1 else 0
    else
      match Hashtbl.find_opt live_regs r with
      | None -> -1
      | Some t -> 0
    )) in

let priorities = critical_paths successors in

let current_resources = Array.copy problem.resource_bounds in

let module InstrSet =
  struct
    module MSet =
      Set.Make (struct
        type t=int
        let compare x y =
          match priorities.(y) - priorities.(x) with
          | 0 -> x - y
          | z -> z
        end)
    end

    let empty = MSet.empty
    let is_empty = MSet.is_empty
    let add = MSet.add
    let remove = MSet.remove

```

```

let union = MSet.union
let iter = MSet.iter

let compare_regs i x y =
  let pyi = List.fold_left (fold_delta i) 0 mentions.(y) in
  (* print_int y;
   * print_string " ";
   * print_int pyi;
   * print_newline ();
   * flush stdout; *)
  let pxi = List.fold_left (fold_delta i) 0 mentions.(x) in
  match pyi - pxi with
  | 0 -> (match priorities.(y) - priorities.(x) with
          | 0 -> x - y
          | z -> z)
  | z -> z

(** t is the register class *)
let sched_CSR t ready usages =
  (* print_string "looking for max delta";
   * print_newline ();
   * flush stdout; *)
  let result = ref (-1) in
  iter (fun i ->
        if vector_less_equal usages.(i) current_resources
        then if !result = -1 || (compare_regs t !result i > 0)
        then result := i) ready;
  !result
end
in

let max_time = bound_max_time problem + 5*nr_instructions in
let ready = Array.make max_time InstrSet.empty in

Array.iteri (fun i preds ->
  if i < nr_instructions && preds = []
  then ready.(0) <- InstrSet.add i ready.(0)) predecessors;

let current_time = ref 0
and earliest_time i =
  try
    let time = ref (-1) in
    List.iter (fun (j, latency) ->
      if times.(j) < 0
      then raise Exit
      else let t = times.(j) + latency in
           if t > !time
           then time := t) predecessors.(i);
    assert (!time >= 0);

```

```

    !time
    with Exit -> -1
in
let advance_time () =
  (if !current_time < max_time-1
   then (
     Array.blit problem.resource_bounds 0 current_resources 0
       (Array.length current_resources);
     ready.(!current_time + 1) <-
       InstrSet.union (ready.(!current_time))
         (ready.(!current_time + 1));
     ready.(!current_time) <- InstrSet.empty));
  incr current_time
in
let cnt = ref 0 in
let attempt_scheduling ready usages =
  let result = ref (-1) in
  DebugPrint.debug "\n\nREADY: ";
  InstrSet.iter (fun i -> DebugPrint.debug "%d " i) ready;
  DebugPrint.debug "\n\n";
  try
    Array.iteri (fun i avlregs ->
      DebugPrint.debug "avlregs: %d %d\nlive regs: %d\n"
        i avlregs (Hashtbl.length live_regs);
      if !cnt < 5 && avlregs <= regs_thresholds.(i)
      then (
        csr_b := true;
        let maybe = InstrSet.sched_CSR i ready usages in
        DebugPrint.debug "maybe %d\n" maybe;
        (if maybe >= 0 &&
         let delta =
           List.fold_left (fold_delta i) 0 mentions.(maybe) in
         DebugPrint.debug "delta %d\n" delta;
         delta > 0
        then
          (vector_subtract usages.(maybe) current_resources;
           result := maybe)
        else
          if not !Clflags.option_regpres_wait_window
          then
            (InstrSet.iter (fun ins ->
              if vector_less_equal usages.(ins) current_resources
                && List.fold_left (fold_delta i) 0 mentions.(maybe) >= 0
              then (result := ins;
                   vector_subtract usages.(!result) current_resources;
                   raise Exit)
            )

```

```

        ) ready;
        if !result <> -1 then
            vector_subtract usages.(!result) current_resources
        else incr cnt)
    else
        (incr cnt)
    );
    raise Exit)) available_regs;
InstrSet.iter (fun i ->
    if vector_less_equal usages.(i) current_resources
    then (
        vector_subtract usages.(i) current_resources;
        result := i;
        raise Exit)) ready;
-1
with Exit ->
    !result in

while !current_time < max_time
do
    if (InstrSet.is_empty ready.(!current_time))
    then advance_time ()
    else
        match attempt_scheduling ready.(!current_time)
        problem.instruction_usages with
        | -1 -> advance_time()
        | i -> (assert(times.(i) < 0);
            (DebugPrint.debug "INSTR ISSUED: %d\n" i;
            if !csr_b && !Clflags.option_debug_compcert > 6 then
                (Printf.eprintf "REGPRES: high pres class %d\n" i;
                flush stderr);
            csr_b := false;
            (* if !Clflags.option_regpres_wait_window then *)
            cnt := 0;
            List.iter (fun (r,b) ->
                if b then
                    (match Hashtbl.find_opt counts r with
                     | None -> assert false
                     | Some (t, n) ->
                        Hashtbl.remove counts r;
                        if n = 1 then
                            (Hashtbl.remove live_regs r;
                            available_regs.(t)
                                <- available_regs.(t) + 1))
                    else
                        let t = class_r r in
                        match Hashtbl.find_opt live_regs r with
                        | None -> (Hashtbl.add live_regs r t;
                                available_regs.(t)

```

```

                                <- available_regs.(t) - 1)
        | Some i -> ()
    ) mentions.(i));
    times.(i) <- !current_time;
    ready.(!current_time)
    <- InstrSet.remove i (ready.(!current_time));
    List.iter (fun (instr_to, latency) ->
        if instr_to < nr_instructions then
            match earliest_time instr_to with
            | -1 -> ()
            | to_time ->
                ((* DebugPrint.debug "TO TIME %d : %d\n" to_time
                    * (Array.length ready); *)
                ready.(to_time)
                <- InstrSet.add instr_to ready.(to_time))
        ) successors.(i);
    successors.(i) <- []
)
done;

try
    let final_time = ref (-1) in
    for i = 0 to nr_instructions - 1 do
        DebugPrint.debug "%d " i;
        (if times.(i) < 0 then raise Exit);
        (if !final_time < times.(i) + 1 then final_time := times.(i) + 1)
    done;
    List.iter (fun (i, latency) ->
        let target_time = latency + times.(i) in
        if target_time > !final_time then
            final_time := target_time predecessors.(nr_instructions);
    times.(nr_instructions) <- !final_time;
    ((* DebugPrint.debug_flag := false; *)
    Some times
    with Exit ->
        ((* DebugPrint.debug_flag := true; *)
        DebugPrint.debug "reg_pres_sched failed\n";
        ((* DebugPrint.debug_flag := false; *)
        None
    ;;

    (*****)

let reg_pres_scheduler_bis (problem : problem) : solution option =
    ((* DebugPrint.debug_flag := true; *)
    DebugPrint.debug "\nNEW\n\n";
    let nr_instructions = get_nr_instructions problem in

```



```

let successors = get_successors problem
and predecessors = get_predecessors problem
and times = Array.make (nr_instructions+1) (-1) in
let live_regs_entry = problem.live_regs_entry in

(* let available_regs = Array.copy Machregsaux.nr_regs in *)

let class_r r =
  Machregsaux.class_of_type (problem.typing r) in

let live_regs = Hashtbl.create 42 in

List.iter (fun r -> let classe = Machregsaux.class_of_type
                      (problem.typing r) in
              (* available_regs.(classe)
               * <- available_regs.(classe) - 1; *)
              Hashtbl.add live_regs r classe)
  (Registers.Regset.elements live_regs_entry);

let counts, mentions =
  match problem.reference_counting with
  | Some (l, r) -> l, r
  | None -> assert false
in

let fold_delta a (r, b) =
  a + (if b then
        match Hashtbl.find_opt counts r with
        | Some (_, 1) -> 1
        | _ -> 0
      else
        match Hashtbl.find_opt live_regs r with
        | None -> -1
        | Some t -> 0
      ) in

let priorities = critical_paths successors in

let current_resources = Array.copy problem.resource_bounds in

let compare_pres x y =
  let pdy = List.fold_left (fold_delta) 0 mentions.(y) in
  let pdx = List.fold_left (fold_delta) 0 mentions.(x) in
  match pdy - pdx with
  | 0 -> x - y
  | z -> z
in

```

```

let module InstrSet =
  Set.Make (struct
    type t = int
    let compare x y =
      match priorities.(y) - priorities.(x) with
      | 0 -> x - y
      | z -> z
    end) in

let max_time = bound_max_time problem (* + 5*nr_instructions *) in
let ready = Array.make max_time InstrSet.empty in

Array.iteri (fun i preds ->
  if i < nr_instructions && preds = []
  then ready.(0) <- InstrSet.add i ready.(0)) predecessors;

let current_time = ref 0
and earliest_time i =
  try
    let time = ref (-1) in
    List.iter (fun (j, latency) ->
      if times.(j) < 0
      then raise Exit
      else let t = times.(j) + latency in
        if t > !time
        then time := t) predecessors.(i);
    assert (!time >= 0);
    !time
  with Exit -> -1
in

let advance_time () =
  (* Printf.printf "ADV\n";
  * flush stdout; *)
  (if !current_time < max_time-1
  then (
    Array.blit problem.resource_bounds 0 current_resources 0
      (Array.length current_resources);
    ready.(!current_time + 1) <-
      InstrSet.union (ready.(!current_time))
        (ready.(!current_time + 1));
    ready.(!current_time) <- InstrSet.empty));
  incr current_time
in

let attempt_scheduling ready usages =
  let result = ref [] in
  try

```

```

InstrSet.iter (fun i ->
  if vector_less_equal usages.(i) current_resources
  then
    if !result = [] || priorities.(i) = priorities.(List.hd (!result))
    then
      result := i::(!result)
    else raise Exit
  ) ready;
if !result <> [] then raise Exit;
-1
with
Exit ->
let mini = List.fold_left (fun a b ->
  if a = -1 || compare_pres a b > 0
  then b else a
) (-1) !result in
vector_subtract usages.(mini) current_resources;
mini
in

while !current_time < max_time
do
if (InstrSet.is_empty ready.(!current_time))
then advance_time ()
else
  match attempt_scheduling ready.(!current_time)
  with problem.instruction_usages with
  | -1 -> advance_time()
  | i -> (
    DebugPrint.debug "ISSUED: %d\nREADY: " i;
    InstrSet.iter (fun i -> DebugPrint.debug "%d " i)
      ready.(!current_time);
    DebugPrint.debug "\nSUCC: ";
    List.iter (fun (i, l) -> DebugPrint.debug "%d " i)
      successors.(i);
    DebugPrint.debug "\n\n";
    assert(times.(i) < 0);
    times.(i) <- !current_time;
    ready.(!current_time)
    <- InstrSet.remove i (ready.(!current_time));
    (List.iter (fun (r,b) ->
      if b then
        (match Hashtbl.find_opt counts r with
         | None -> assert false
         | Some (t, n) ->
           Hashtbl.remove counts r;
           if n = 1 then
             (Hashtbl.remove live_regs r;
              (* available_regs.(t)

```

```

        * <- available_regs.(t) + 1 *)))
    else
        let t = class_r r in
        match Hashtbl.find_opt live_regs r with
        | None -> (Hashtbl.add live_regs r t;
            (* available_regs.(t)
            * <- available_regs.(t) - 1 *)))
        | Some i -> ()
    ) mentions.(i));
List.iter (fun (instr_to, latency) ->
    if instr_to < nr_instructions then
        match earliest_time instr_to with
        | -1 -> ()
        | to_time ->
            ((* DebugPrint.debug "TO TIME %d : %d\n" to_time
            * (Array.length ready); *)
            ready.(to_time)
            <- InstrSet.add instr_to ready.(to_time))
        ) successors.(i);
    successors.(i) <- []
)
done;

try
    let final_time = ref (-1) in
    for i = 0 to nr_instructions - 1 do
        (* print_int i;
        * flush stdout; *)
        (if times.(i) < 0 then raise Exit);
        (if !final_time < times.(i) + 1 then final_time := times.(i) + 1)
    done;
    List.iter (fun (i, latency) ->
        let target_time = latency + times.(i) in
        if target_time > !final_time then
            final_time := target_time) predecessors.(nr_instructions);
    times.(nr_instructions) <- !final_time;
    (* DebugPrint.debug_flag := false; *)
    Some times
with Exit ->
    DebugPrint.debug "reg_pres_sched failed\n";
    (* DebugPrint.debug_flag := false; *)
    None
;;

```

## Annexe B. Contexte institutionnel

Compte tenu du contexte sanitaire, j’ai effectué mon stage principalement à distance, et n’étais au sein du laboratoire qu’un jour par semaine. Je n’ai donc interragi qu’avec peu de personnes de l’équipe : mon maître de stage David Monniaux (directeur du laboratoire), mon co-encadrant Sylvain Boulmé, ainsi que les autres stagiaires. L’essentiel de la communication et des échanges se faisaient donc par mail, ou via un chat mis en place par l’UGA. J’ai en revanche pû assister — à distance — à un séminaire sur des idées d’améliorations de `CoQ` et `proof-general`.

VERIMAG<sup>3</sup> est un laboratoire commun Université Grenoble-Alpes, CNRS, et Grenoble INP, spécialisé dans les systèmes embarqués et critiques. Comme déjà évoqué, l’équipe développe sa propre « variante » de `COMP CERT`, et plus généralement travaille sur les domaines de la vérification, de la preuve formelle, de l’analyse de code...

---

<sup>3</sup><https://www-verimag.imag.fr/?lang=fr>