Verified compilation: the case of CompCert

David Monniaux

To cite this version:

David Monniaux. Verified compilation: the case of CompCert. Doctoral. En ligne, Russia. 2022. hal-04828059

HAL Id: hal-04828059

https://hal.science/hal-04828059v1

Submitted on 9 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Verified compilation: the case of CompCert

David Monniaux

VERIMAG — CNRS

February 16, 2022

Contents

Compiler safety

Scheduling

Fun stuff in compiling

Usual compiler

source code

target code (assembler source, object code, bytecode…)

tens of thousands of bugs in gcc's bug tracker

Why compiler bugs are nasty

A compiler bug may disappear

- if optimization levels are changed to ease debugging
- if a different compiler is used
- if debugging code is added to the program (even just `printf`)

A compiler bug is most often at first **undistinguishable from reliance on undefined behavior** in the program.

CompCert

Formally verified C compiler
project led by Xavier Leroy, then at INRIA, now at Collège de France

Non-commercial `https://github.com/AbsInt/CompCert`
Commercial `https://www.absint.com/compcert/index.htm`

trace of execution = sequence of external calls, volatile read/writes

valid trace of execution at C level

same trace of execution at assembly level

Trace of execution

A compiler optimizer may reorganize everything internally...
but must preserve all **interactions** with the outside world and their **ordering**

- calls to external functions (system calls, I/O, GUI...)
- read/write to volatile variables (for system-level programming)

Use case: traceability

Safe-critical systems (e.g. avionics)

Obligation to match object code to source

Conventional method: `-O0` and some manual inspection

CompCert replaces this by mathematical proofs.
Can **use optimization**.

Semantics and proofs in CompCert

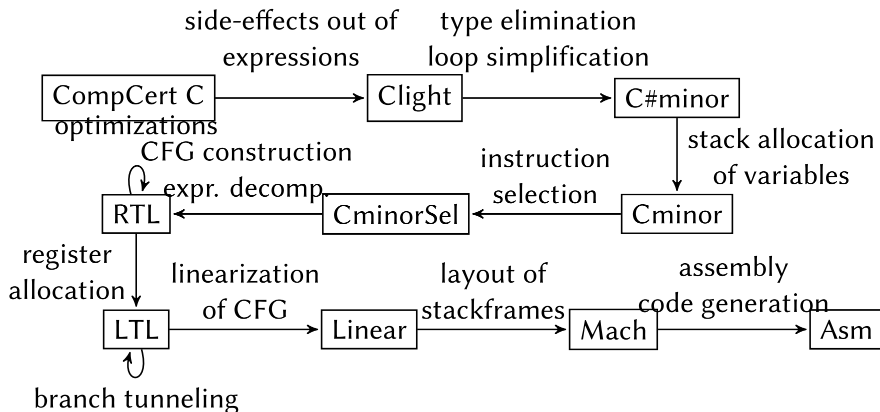
Each intermediate language comes with a **semantics** written in Coq. Gives a **mathematical meaning** to all constructs in the intermediate language.

e.g. at C level, gives a notion of environment of variables (local and global), defines + as addition on various data types depending on types of inputs

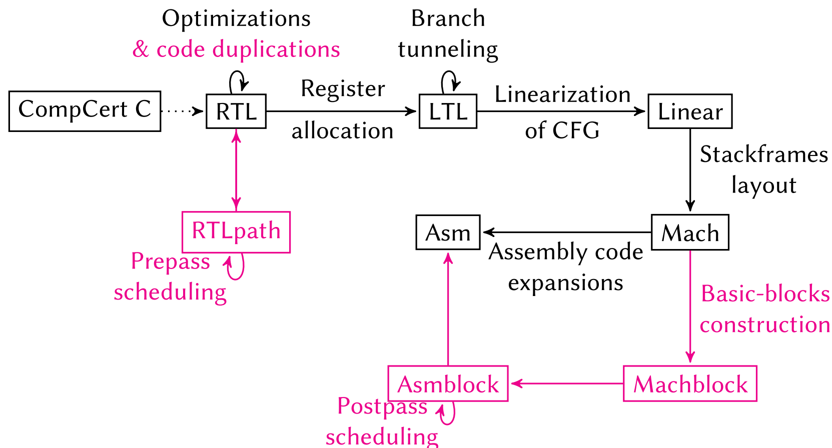
Optimization / transformation phases written in Coq.
(Can call external untrusted OCaml code.)

Must prove **simulation** for each phase

Intermediate languages (1)



Intermediate languages (2)



Simulation proofs

Lockstep

one step of program before the transformation

one matching step of program after the transformation

More complex simulations replace sequences of steps by sequences of steps.

Overall use

These proofs concern the compiler designer only (and those who file for qualification for safety-critical systems).

For most users, using CompCert is just like using `gcc` or `clang`.

```
$ ccomp hello_world.c -o hello_world
$ ./hello_world
```

Or to produce assembly code

```
$ ccomp -S hello_world.c
```

Assembly output

```
// File generated by CompCert 3.10
// Command line: -S hello_world.c
  .section  .rodata
  .balign 1
__stringlit_1:
  .ascii  "Hello, world!\012\000"
  .type  __stringlit_1, @object
  .size  __stringlit_1, . - __stringlit_1
  .text
  .balign 4
  .globl main
main:
  mov  x29, sp
  sub  sp, sp, #16
  str  x29, [sp, #0]
  str  x30, [sp, #8]
  adrp  x0, __stringlit_1
  add  x0, x0, #:lo12:__stringlit_1
  bl  printf
  movz  w0, #0, lsl #0
  ldr  x30, [sp, #8]
  add  sp, sp, #16
  ret  x30
  .type  main, @function
```

Alternate front-ends

Many high-level languages / domain-specific languages compile to C.

- bugs in the high-level language compiler
- "semantic mismatches" between the high-level compiler and the C compiler

Solution: compile directly to one of CompCert's early intermediate representations!
e.g. **Vélus** `https://velus.inria.fr/` compiles a subset of the synchronous data-flow Lustre / SCADE language used in avionics etc.

Contents

Compiler safety

Scheduling

Fun stuff in compiling

Scheduling

An example of an **optimization**, particularly for low-power / embedded cores.

Instructions produce outputs a number of clock cycles after receiving their inputs.
Take this into account to **schedule** instructions.

A menu

1. oysters
2. veal blanquette
   - 2.1 prepare blanquette
   - 2.2 cook it
3. millefeuille
   - 3.1 **puff pastry**
     - 3.1.1 fold 1, wait 30 minutes
     - 3.1.2 fold 2, wait 30 minutes
     - 3.1.3 fold 3, wait 30 minutes
     - 3.1.4 fold 4, wait 30 minutes
     - 3.1.5 fold 5
   - 3.2 cream

Scheduling

“Official” CompCert produces instructions roughly in the source ordering.

Not the best execution order in general!

Especially on in-order cores.

Our solution: **verified scheduling**

Superblock scheduling

1. Partition each function into **superblocks**: one entry point, possibly several exit points, no cycle
2. Possibly do some other reorganization: tail duplication, etc. to get bigger superblocks
3. Schedule the superblock (no proof needed)
4. Witness through symbolic execution that the original and scheduled superblocks have equivalent **semantics** (proof needed)

Before register allocation, on IR.

On Kalray KVX and AArch64: reschedule basic blocks on assembly instructions after register allocation.

Equivalent semantics

- Same order of exit branches in original and scheduled superblock
- All live pseudo registers and memory have the same value at same exit point (non-live registers can differ)
- Same (or smaller) list of instructions that may fail (division by zero, memory access) reached at same exit point

Obtained by **symbolic execution**: two registers are considered equal if computed by exactly the same symbolic terms

Example

$r_1 := a * b$
$r_3 := a - b$
$r_2 := r_1 + c$
branch($a > 0$, EXIT1)

$r_3 := a - b$
$r_4 := a * b$
$r_2 := r_4 + c$
branch($a > 0$, EXIT1)

r_1 and r_4 are both dead at EXIT1 and at final point.

These two blocks are **equivalent**: in both cases
$r_2 = (a * b) + c$ and $r_3 = a - b$

Acceptable refinement

$$r_1 := a * b$$
$$r_3 := a - b$$
$$r_2 := r_1 + c$$
$$r_5 := a/b$$
branch($a > 0$, EXIT1)

$$r_3 := a - b$$
$$r_4 := a * b$$
$$r_3 := r_4 + c$$
branch($a > 0$, EXIT1)
$$r_5 := a/b$$

r_5 dead on EXIT1.

On x86, the division may fail:

- it's allowed to move it beyond the branch
- the converse is not allowed

What our CompCert does (roughly)

1. partition functions into superblocks
2. reorder instructions in each superblock
3. for each superblock, check that symbolic execution modulo live variable produces the same symbolic values for the original and transformed superblocks
4. (if the check fails, optimization fails; this does not happen in practice)

Simulation proof: if the check succeeds, a number of steps of the original program are simulated by a number of steps of the transformed program

Information needed

For all instructions

- **latency**: clock cycles between consuming operands and producing the value (or, more generally, a timetable of when each operand is consumed after the instruction is issued)
- **resource consumption**: CPU units in use that preclude other instructions being scheduled at the same time

Very difficult to find even for "open cores"!!!
(Reverse-engineer gcc and LLVM?)

Performance gain

CPU	Differences in cycles spent (%) compared to					
	no CSE3, no unroll			gcc -O2		
	avg	min	max	avg	min	max
Cortex-A53	-16	-63	+3	+10	-23	+87
Rocket	-10	-43	+1	+29	0	+184
Xeon	-21	-56	+4	+21	-3	+189
KV3	-11	-32	+3	+8	-13	+88

Verimag CNRS UGA Université Grenoble Alpes Grenoble INP

Reducing the proof workload

We split most of our optimizations into:

- an **untrusted** program transformation
- a **verifier**, which is proved to answer “yes” only if semantics is preserved

Advantages:

- reducing the proof effort
- easing changes to the transformation (no need to redo proofs)

Contents

Compiler safety

Scheduling

Fun stuff in compiling

Strength reduction

(Work in progress)
Rewrite costly instruction into cheaper instructions.

```
for(int i=0; i<n; i++) {
  tab[i*M] = 42;
}
```

into

```
data *p = tab;
for(int i=0; i<n; i++) {
  *p = 42;
  p += M;
}
```

Invariants

Being added to our compiler infrastructure for verifying optimizations.

Instead of just symbolic execution within the loop body, needs **invariants**.
Here, **at every loop iteration**, `p = &tab[i*M]`.
True at loop start, then holds by induction over the number of loop iterations.

Fun CPU feature: conditional move

For predictable hard real time code (fewer execution paths)

Branches are **bad** for worst-case execution time static analysis (Absint aIT, etc.)

Suggestion: add **conditional moves for integer and floating-point registers**
at least on in-order cores

Fun CPU feature: dismissible loads

An operation that may fail cannot be moved before a branch

$r_1 := a + i << 3$
branch($i > 3$, EXIT1)
$r_2 :=$ load(p)

$r_1 := a + i << 3$
$r_2 :=$ load$_s$(p)
branch($i > 3$, EXIT1)

Cannot be done if the load can fail.

Need special load returning a **default value** instead of trapping.

- easy without virtual memory
- needs OS collaboration with virtual memory

Dismissible load on KVX

8 cycles

```
L100:
 compw.ge $r32 = $r4, $r2
;;
 cb.wnez$r32? .L101
;;
 sxwd $r5 = $r4
 addw $r4 = $r4, 1
;;
 lws.xs $r3 = $r5[$r1]
;;
 addw $r0 = $r0, $r3
 goto .L100
;;
```

6 cycles

```
.L100:
 sxwd $r5 = $r4
 compw.ge $r32 = $r4, $r2
;;
 lws.s.xs $r3 = $r5[$r1]
;;
 cb.wnez $r32? .L101
;;
 addw $r0 = $r0, $r3
 addw $r4 = $r4, 1
 goto .L100
```

A general call for collaboration

Need collaboration between

- compiler writers
- architecture / core designers
- operating systems (low level)

Our version of CompCert with optimizations not found in the "official" releases + the KVX target:
`https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx`

Pre-pass scheduling: KVX; Cortex-A53/A35 (AArch64); Rocket, SweRV EH1 (Risc-V)
Post-pass scheduling: KVX; Cortex-A53/A35 (AArch64)