Testing a Formally Verified Compiler

David Monniaux

VERIMAG

July 15, 2023

CompCert

Formally verified C compiler, effort led by Xavier Leroy

"If compilation succeeds, then the assembly program matches the C program."

Formally verified: compiler written in Coq
+ correctness theorem proved in Coq, a proof assistant
(mathematical proof, machine-checked)

Rationale for CompCert

Certain industries (avionics, nuclear...) must demonstrate that the object code is equivalent to the source.

Conventional approach

Disable optimizations
"Human" comparisons
"This compiler worked in other safety-critical projects"

CompCert

Use the mathematical proof

Versions under discussion

“Official” releases

`https://github.com/AbsInt/CompCert`

“Chamois” branch

`https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert`

Correctness theorem

execution = trace of "externally visible events" (calls to external functions, volatile variables accesses)

The trace at assembly matches the C trace.

Obtained by "forward simulation" (assembly simulates C) through "match" relations

Forward simulations

Lockstep "One step of the program before transformation maps to one step after transformation."
$\sigma_1 \rightarrow_e \sigma_2$ and $m(\sigma_1, \sigma_1')$ then there exists σ_2' such that $\sigma_1' \rightarrow_e \sigma_2'$ and $m(\sigma_3, \sigma_2')$
$e =$ "observable events"
e.g. "replace $x \times y$ by a move from a register already containing that expression"

Plus "One step maps to several steps."
e.g. function call from one instruction to many (move operands to registers / stack etc.)

Star "Several steps map to several steps."

Formally Verified Defensive Programming

(see Sylvain Boulmé's habilitation for many relevant examples)

Total correctness

"This Coq procedure always terminates and produces a result according to specification

Partial correctness

"This procedure, if it terminates in an OK condition (not Error), produces a result according to specification"

Why partial correctness

Not proving termination

Termination often annoying to prove in Coq.
Just put a maximum number of iterations and answer Error if reached.

Untrusted oracle

Call an oracle (often written in OCaml)
(The oracle possibly logs information such as invariants)
Verify the oracle's result using formally verified code (OK or Error)

Trusted Computing Base

(See Monniaux & Boulmé, *The Trusted Computing Base of the CompCert Verified Compiler*, ESOP 2022)

- logical axioms, Coq, Coq to OCaml extractor, OCaml runtime (not a serious issue)
- C semantics (possible discrepancies with standard / programmers' expectations)
- assembly semantics, including macro instructions (there have been some bugs here)

Example of a C semantic discrepancy

```
#include <stdio.h>

struct stuff { int xx : 3; };

int main() {
  struct stuff v = { 1 };
  printf("%d\n", v.xx);
}
```

```
Stuck state: in function main, expression <loc v>
Stuck subexpression: <loc v> = 1
ERROR: Undefined behavior
```

What happened

Initialization / non-initialized (undefined value) appreciated at word level.

Field initialization initializes one field, so word still uninitialized as a whole.

Reading field from uninitialized word yields undefined.

Printing an uninitialized value has undefined behavior.

What could have happened worse

Code with local bitfields still compiles ok

But aggressive optimizations could replace undefined behavior by other defined behavior.
e.g. replace this function by e.g.

```
printf("%d\n", 2);
```

General comment

Deliberately, some static analyses and optimizations in CompCert

- are not as precise as they could
- do not take advantage of undefined behavior

C compilers taking advantage aggressively of undefined behavior = source of trouble

and a lot of C programs (dragged over decades) rely on some specific behaviors

Abstract assembly

CompCert's `Asm.v` defines a somewhat abstract view of assembly.

- values are in a "sum type" integers + longs + floats + doubles + "undefined" (in reality: bitstrings)
- addresses are (b, o), b is a block identifier, o an offset
- macro instructions (memory copy, stackframe allocation, stackframe destruction, access to global symbols...) expanded by trusted OCaml code

A few bugs found in trusted macro expansion.

(Would need unit testing)

ABI

The definition of ABI in CompCert is not concentrated in a single place.

Need to test compatibility

- with various linker/OS features (e.g. thread-local storage)
- with other compilers

Example of (macro) assembly problems

In some rare cases of registers, the code for built-in memory copy was incorrect.

(My mistake) on Kalray KVX, some instruction was printed with the mnemonic of another.

What the correctness theorem says and does not say

If compilation succeeds, the formal assembly semantics match those of the source code.

Does not say

- that compilation succeeds (no "internal error")
- ...within acceptable time
- that the formal C semantics matches true C semantics
- that the formal assembly semantics matches true semantics
- that optimizations are not missed
- that generated code has good performance

Needs testing!

Our test suite (for continuous integration)

- test suite from Absint's Github CompCert repository
  - "compiles without failing"
  - compiles + runs and results matches reference result
- gcc 12.2.0 torture tests (compile + execute)
- Paranoia test suite (compile + execute)
- random generation by Csmith (compile + differential execute w/ timeout)
- random generation by YarpGen (compile + differential execute)
- random generation by CCG (compile)

Compilation success and timeout

Check if programs compile within acceptable time / memory.

(Disabled some gcc/LLVM tests that “tested” some ranges with unnatural programs)

What if we find an internal error

CompCert crashes with internal error message / OCaml uncaught exception (the exception generally points to the faulty pass, sometimes points to a later pass finding wrong input)

Reduce the test case using Creduce
Creduce = removes code / simplifies code as long as predicate is satisfied

Predicate = "CompCert crashes with the same error"

Differential testing

- compile with CompCert and gcc
- run both (possibly in QEMU)
- results must be equal
- (if necessary) run with carefully chosen timeout (both must return the same result, or both must timeout)

What if we find a discrepancy in differential testing

Outcomes with gcc and CompCert are different

Reduce the test case using Creduce
Problem: Creduce tends to produce code with undefined behavior

Predicate:

- compiles with gcc / CompCert / clang without warnings such as "undeclared function", implicit int types etc
- compiled with gcc runs in valgrind or with address sanitizer (ASan) / undefined behavior sanitizer (UBsan) without error
- still gives different outcome with CompCert

If bug in CompCert, then in its TCB (non-proved parts)

Performance evaluation

Test suite:

- LLVMtest
- Polybench
- MiBench (representative of embedded algorithms?)
- Tacle-Bench (benchmarks for WCET analysis)
- others (matrix computations, sorting algorithms, SAT solvers, OCaml interpreter, compressors, first-order solver)

Caveats for performance evaluation

Measure signal, not noise

- must run on target CPU or cycle-accurate simulator, not QEMU
- disable frequency scaling and migration between cores
- multiple runs, discard outliers, average

Code issues

Keep things comparable

- disable gcc-only in-code optimizations (e.g. `#ifdef __GNUC__` do something clever with gcc-specific code)
- disable floating-point contraction in gcc (`-ffp-contract=off`), which replaces `a*b+c` by `fma(a, b, c)`
  (semantic difference: $r(r(ab) + c)$ vs $r(ab + c)$)

Sample RISC-V experimental results

(compared to official releases; higher is better)

Setup	gcc -O1	Chamois
LLVMtest/fpconvert	+24.22%	+17.15%
LLVMtest/matmul	+15.9%	+144.11%
LLVMtest/nbench_bf	+74.58%	+24.51%
MiBench/jpeg	+27.75%	+24.75%
MiBench/sha	+92.43%	+51.73%
MiBench/stringsearch	+133.34%	-10.15%
PolyBench/*	+64.05%	+46.23%
TACLeBench/bsort	+49.04%	+33.16%
TACLeBench/deg2rad	+56.75%	+50.28%
TACLeBench/md5	+42.18%	+47.93%

Conclusion

- Formal proof protects you only in the formally verified part.
- Need to test hypotheses.
- (Possibly unit-test them, especially the formal semantics of instructions and pseudo-instructions.)