Chamois: agile development of CompCert extensions for optimization and security

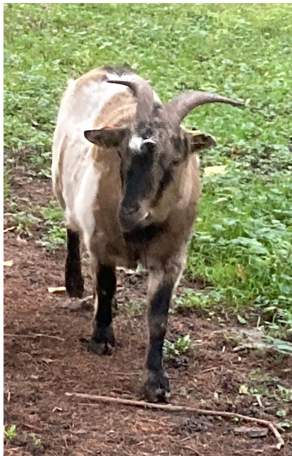
David Monniaux

CNRS / Verimag

February 1st, 2024

Joint work with Sylvain Boulmé (associate prof), Léo Gourdin, Cyril Six (PhD students), Benjamin Bonneau, Nicolas Nardino (interns)

Villetaneuse n'a qu'à bien se tenir



CHEVREX

Contents

CompCert

Optimizations

Code restructuring

Security

Conclusion & Perspectives

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

Our own “Chamois” branch

for agile development https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert

Targets

- x86 and x86-64 (not idiomatic)
- ARM
- AArch64
- RISC-V 32- and 64-bit
- PowerPC
- (Chamois only) Kalray KVX

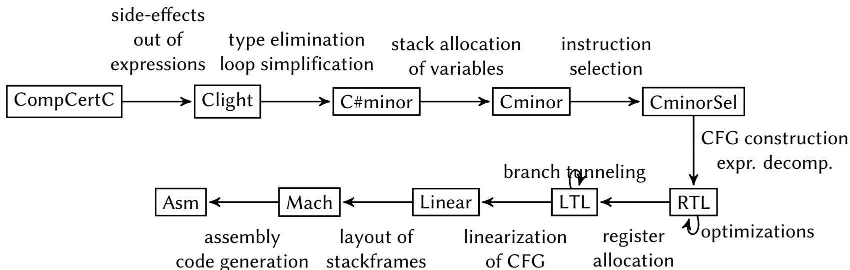
Correctness theorem

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

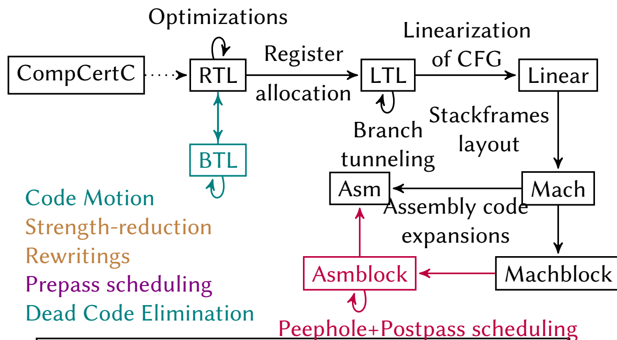
The trace at assembly matches the C trace.

Obtained by “**forward simulation**” (assembly simulates C) through “match” relations

CompCert's intermediate languages



Backend phases



The main difficulty

Optimizations etc. described quite informally in the literature.

Find good formalism definitions etc. to make proofs easier.

Same as with “formal mathematics” (Xena Project, Georges Gonthier…)

Contents

CompCert

Optimizations

Code restructuring

Security

Conclusion & Perspectives

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**

Contents

CompCert

Optimizations

Code restructuring

Security

Conclusion & Perspectives

Loop rotation

```
while(c) {
  body;
}
```

turned into

```
if (c) {
  do {
    body;
  } while (c);
}
```

Allows precomputing the condition inside the loop body, changes 2 branches per iteration into 1.

Loop peeling

```
while(c) {
  body;
}
```

turned into

```
if (c) {
  body
  while (c) {
    body;
  }
}
```

Makes sure that operations always executed inside the peeled loop body are "available" for the next loop body.
(Together with global subexpression elimination, performs loop-invariant code motion.)

cnrs UGA Université Grenoble Alpes Grenoble INP Verimag

Code morphisms

On control-flow graphs
Each node in the transformed program corresponds to a node in the original

Lockstep simulation = “the operations are the same on each side”

- loop unrolling
- loop peeling
- loop rotation
- factoring (= regroup CFG nodes according to equivalence/congruence/bisimulation)

Global common subexpression elimination

Goals

- Replaces computation by move if value available in same register on all incoming paths
- Replaces conditional branch by unconditional branch if condition statically known on all incoming paths

Lazy code motion

Hoist loop-invariant code out of loops.

Proved by glue invariants + symbolic execution.

```
void mul42(double *t, int n) {
  for(int i=0; i<n; i++)
    t[i] *= 42;
}
```

Move the constant 42 load out of the loop.

Strength reduction

```
for(int i=0; i<n; i++) {
    r += t[i];
}
```

Naive compilation: `t[i]` means multiplication/shift, add, load.
Yet the address differs only by a constant offset across iterations!

Example: complex sum-product

```
typedef struct { double re, im; } complex;

inline void sum_complex(complex *s, const complex *a,
    const complex *b) {
  double re = a->re + b->re;
  double im = a->im + b->im;
  s->re = re;
  s->im = im;
}

inline void mul_complex(complex *s, const complex *a,
    const complex *b) {
  double re = a->re * b->re - a->im*b->im;
  double im = a->re * b->im + a->im*b->re;
  s->re = re;
```

cnrs UGA Université Grenoble Alpes Grenoble INP Verimag

Example: complex sum-product

```
void sumproduct_complex_array(complex *s, int n,
    complex *a, complex *b) {
  complex r = {0., 0.}, p;
  for(int i=0; i<n; i++) {
    mul_complex(&p, a+i, b+i);
    sum_complex(&r, &r, &p);
  }
  s->re = r.re;
  s->im = r.im;
}
```

Compiled complex sum-product main loop

```
.L102:
        fld     f29, 0(x12)
        fld     f12, 0(x13)
        fld     f14, 8(x12)
        fld     f11, 8(x13)
        fmul.d  f30, f29, f12
        fmul.d  f2, f14, f12
        fmul.d  f28, f14, f11
        fmul.d  f5, f29, f11
        addi    x14, x14, 1
        addi    x13, x13, 16
        addi    x12, x12, 16
        fsub.d  f3, f30, f28
        fadd.d  f0, f5, f2
        fadd.d  f4, f4, f3
        fadd.d  f1, f1, f0
        blt     x14, x5, .L102
```

Contents

CompCert

Optimizations

Code restructuring

Security

Conclusion & Perspectives

Stack canaries

On functions with local arrays etc. put a canary (special data) at the end.
If buffer overrun ("stack smashing"), abort execution.
Blocks attempts at corrupting the return address.

```
void swap(int n, int *a, int *b) {
  int tmp[10];
  for(int i=0; i<n; i++) tmp[i]=a[i];
  for(int i=0; i<n; i++) a[i]=b[i];
  for(int i=0; i<n; i++) b[i]=tmp[i];
}

int main() {
  static int ka[15] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
  static int kb[15] = {31,32,33,34,35,36,37,38,39,20,21,22,23,24,25};
  swap(15, ka, kb);
}
```

```
$ ./localcopy
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Pointer authentication

(AArch64)

Return address mangled with a kind of hash depending on a secret key and the stack pointer

Proof: axiomatization that de-mangling composed with mangling is identity

Bouquetin: work in progress

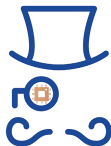
Countermeasures against hardware attacks (and some software attacks).

- redundant loads / redundant operations
- control-flow integrity by passing extra "magic numbers"

Proofs of

Correctness if no attack occurs, execution undisturbed

Adequacy if an attack occurs (within some constraints), execution either is undisturbed or aborts



PEPR Cybersécurité "Arsene"

Contents

CompCert

Optimizations

Code restructuring

Security

Conclusion & Perspectives

How to prove things

- Need to rework analyses.
- Think carefully about invariants, what needs to be proved, what needn't.
- Possibly split complex optimizations into distinct phases with simple specification.
- Possibly split thing into **an oracle and a checker**.
- **Any unclear / badly designed / delicate aspect of semantics will bite you.**

Rust

Verified compilation of Rust?

Thesis in progress on verified borrow-checker

Gratuitous advertisement

“gcc (at least quite a bit of it) but verified”

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