Some industrial applications I've been involved in

David Monniaux

CNRS / VERIMAG

July 11, 2019

VERIMAG is a joint research unit of CNRS, a national research organization, Université Grenoble Alpes and Grenoble-INP (school of engineering).

Me

Co-head of PACSS group at VERIMAG (public research laboratory)

PACSS = safety and security

- decision procedures
- assisted proofs
- certified compilation
- attacker models
- concolic execution
- abstract interpretation, convex polyhedra etc.

Contents

Astrée

(Involvement: 2001–2007)

Automatic static analysis tool for inferring invariants and proving

- absence of undefined behaviors / **runtime errors**
- assertions

Input: C source
Outputs: warnings, optionally invariants of the execution

Undefined behaviors in C

MISRA-C 2004, Rule 1.2 (required): *No reliance shall be placed on undefined or unspecified behaviour.*

Undefined behaviors include:

- Array access out of bounds
- Bad pointers
- Signed arithmetic overflow
- Arithmetic conversion overflows
- ...

In general these are **undecidable properties**.

Arbitrary properties

```
int *p = NULL, x;
if (stuff()) p = &x;
*p = 5;
```

Arbitrary properties

```
int *p = NULL, x;
if (stuff()) p = &x;
*p = 5;

int count = 0;
while(true) {
  if (stuff()) {
    count++;
  } else {
    count = 0;
  }
}
```

Undecidable?

"There is no algorithm that, given the source code of a program with unbounded memory, can say whether it terminates or not."
(see in theoretical model by Turing and others)

MISRA-C 2012 now flags properties as "undecidable" or not.
Distinguish hard properties from properties checkable on program syntax.

Take-home message: no static analysis tool can flag exactly undefined behaviors in a C program. It must have at least one of:

- **false positives**: warnings about nonexistent problems
- **false negatives**: missing existent problems

Interval analysis

```
int x, y, z;
assume(x >= 0 && x <= 1000);
assume(y >= 0 && y <= 1000);
z = x+y;
```

Proves that $0 \leq x + y \leq 2000$ and thus cannot overflow.

Interval analysis may be imprecise

```
int x, y, z;
assume(x >= 0 && x <= 1);
y = 1-x;
z = 1000/(x+y);
```

$x \in [0, 1]$, $y \in [0, 1]$, $x + y \in [0, 2]$
flags possible division by zero!

Loops

```
int x = 0;
while(true) {
  x++;
  if (x==1000) x=0;
}
```

Depending how it's done: $0 \leq x \leq 1000$ at head of loop, $0 \leq x$ only...

Relations

```
int x = 0, y;
assume(0 <= y && y <= 1);
while (test()) {
  x++;
  y++;
}
z = y-x;
```

Interval analysis: cannot prove $z \in [0, 1]$
Relational analyses (convex polyhedra, “octagons”: $z \in [0, 1]$)

Second-order filter

$$y_n = \alpha_0 x_n + \alpha_1 x_{n-1} + \alpha_2 x_{n-2} + \beta_1 y_{n-1} + \beta_2 y_{n-2}$$

Cannot be bounded by interval analysis

- enclose (y_n, y_{n-1}) in an ellipsoid?
- or approaches based on Z-transform

Pointers

For every pointer, track to what it may point.
This can be hard!

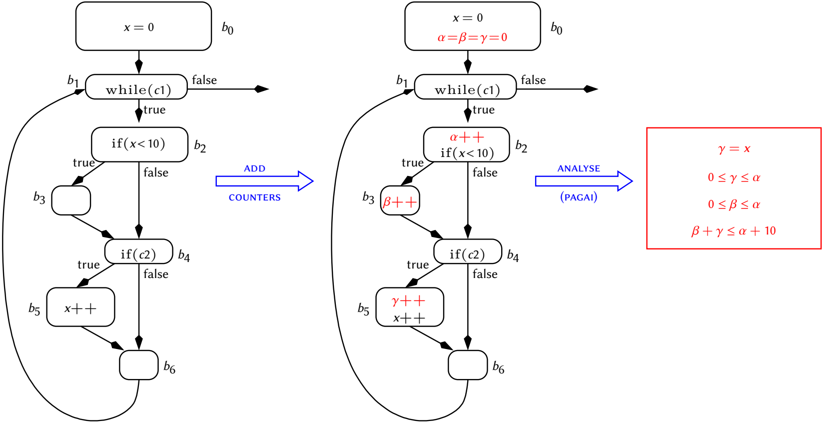
```
int x = 0, y = 0;
int *p = stuff() ? &x : &y;
(*p) ++;
(*p) --;
assert(x==0);
assert(y==0);
```

Depending how it’s done, we can prove the assertions…or not.

Summary

- Automatically infer properties on program variables
- These properties hold initially are stable by **induction** ("if true at loop iteration n then true at iteration $n + 1$)
- Thus they are **true at every iteration**.
- Can prove properties, or give information (e.g. ranges or relationships or alias relations)
- A lot of variation on cost and precision of approaches.

Counters for WCET



(At VERIMAG: done by Raymond, Maïza, Parent-Vigouroux et al. with PAGAI;
also experiments for WCET using SMT; ask me about it!)

Tools

Astrée

Designed for safety-critical fly-by-wire avionics systems

e.g. A340, A380

`http://www.astree.ens.fr/`

`https://www.absint.com/astree/index.htm`

Tools

Astrée

Designed for safety-critical fly-by-wire avionics systems
e.g. A340, A380
`http://www.astree.ens.fr/`
`https://www.absint.com/astree/index.htm`

Frama-C value analysis

`https://frama-c.com/value.html`

PAGAI

(research prototype)
`https://gricad-gitlab.univ-grenoble-alpes.fr/pagai/pagai`

A remark on precision

Some tools advertise 98% precision
Meaning: out of 100 possible “undefined behaviour” warnings they prove 98% not to occur (GREEN)

A remark on precision

Some tools advertise 98% precision
Meaning: out of 100 possible "undefined behaviour" warnings they prove 98% not to occur (GREEN)

200,000-LOC source code ⇒ 4,000 warnings (ORANGE)

Astrée aimed at **0 or few warnings**
Astrée aimed at a **specific domain** (safety-critical control applications) and their classes of invariants.

Industrialization lessons learned on Astrée

High precision

Off the shelf tools will give poor precision — need tayloring
Researchers need the actual code to be analyzed or at least highly representative examples (same constructs, same kind of invariants).
Perhaps hear feedback on difficult-to-analyze constructs.

Scope

Eventually you end up supporting a very large subset of C. Code is seldom fully in a “reasonable” subset (e.g. “no pointer arithmetic”).

Don’t give up

“Static analysis does not work”
Many tools
Many approaches

Contents

Choosing an embedded processor

For speed?

- out-of-order superscalar
- fast clock
- multicore

Choosing an embedded processor

For speed?

- out-of-order superscalar
- fast clock
- multicore

For reliability?

- slow
- simple control

Choosing an embedded processor

For speed?

- out-of-order superscalar
- fast clock
- multicore

For reliability?

- slow
- simple control

For predictability? (WCET)

- simple, predictable cache
- in-order core

Choosing a compiler

For speed?

- agressive optimizations
- the resulting code does not resemble the source

Choosing a compiler

For speed?

- agressive optimizations
- the resulting code does not resemble the source

For qualification?

- assembly code follows C (side-by-side comparison, same C block always compiled the same)
- slow code

To summarize

For performance

- high performance out-of-order core
- agressive optimizations in compiler

For qualification

- predictable core
- no optimizations
- assembly/object code “visually” matches the source

CompCert

(Xavier Leroy et al.)

Mathematically defined semantics

- for source program
- for target code (assembly)

Proof that the semantics is preserved.

Example of how it works

Inside instruction selection: simplification of or-immediate:

```
Nondetfunction orimm (n1: int) (e2: expr) :=
  if Int.eq n1 Int.zero then e2
  else if Int.eq n1 Int.mone then Eop (Ointconst Int.mone) Enil
  else match e2 with
       | Eop (Ointconst n2) Enil ⇒ Eop (Ointconst (Int.or n1 n2)) Enil
       | Eop (Oorimm n2) (t2:::Enil) ⇒ Eop (Oorimm (Int.or n1 n2)) (t2:::Enil)
       | Eop Onot (t2:::Enil) ⇒ Eop (Oornimm n1) (t2:::Enil)
       | _ ⇒ Eop (Oorimm n1) (e2:::Enil)
  end
```

Theorems

Simple, local proofs of soundness:

```
Theorem eval_orimm:
  ∀ n, unary_constructor_sound (orimm n) (fun x ⇒ Val.or x (Vint n))
Proof
```

"Even after simplifications, || still means "or"!"

More involved theorems

```
Theorem match_state_codestate:
  ∀ mbs abs s fb sp bb c ms m,
  (∀ ef args res, MB.exit bb <> Some (MBbuiltin ef args res)) →
  (MB.body bb <> nil ∨ MB.exit bb <> None) →
  mbs = (Machblock.State s fb sp (bb::c) ms m) →
  match_states mbs abs →
  ∃ cs fb f tbb tc ep,
    match_codestate fb mbs cs ∧ match_asmstate fb cs abs
    ∧ Genv.find_funct_ptr ge fb = Some (Internal f)
    ∧ transl_blocks f (bb::c) ep = OK (tbb::tc)
    ∧ body tbb = pbody1 cs++pbody2 cs
    ∧ exit tbb = pctl cs
    ∧ cur cs = Some tbb ∧ rem cs = tc
    ∧ pstate cs = abs
```

"Given the mapping of the stack, the assembly code generated has the same semantics as that of the last intermediate representation."

Main theorem

```
Theorem transf_c_program_correct:
  ∀ p tp,
  transf_c_program p = OK tp →
  backward_simulation (Csem.semantics p) (Asm.semantics tp)
```

Scientific challenge

The compiler designer must have a very clear idea of

- all semantics
- all invariants
- all properties of intermediate representations

to write the proofs!

Some simplification: do not prove the transformation, prove a checker verifying the transformation.

In practice

Just like gcc or clang.

e.g. compiling the GNU Linear Programming Toolkit

```
cd glpk-4.65
CC="ccomp␣-fall" ./configure --disable-reentrant --disable-shared
make
make install
```

Current involvement

Joint work with Cyril Six & Sylvain Boulmé

MPPA3

- Development of a backend for the Kalray MPPA3 (K1C core).
- Optimized VLIW instruction scheduling.

Secure processor

- Development of a backend for a processor with secure features (control flow integrity, encryption of code...)

Focus: local scheduling

Each CPU instruction i is a task, results available after L_i cycles

Each instruction uses a vector v_i of resources (LSU, ALU...), sum of resources of instructions at same cycle $\leq B$

Need to respect **dependencies**:

- compute/load a result **before** it's needed
- don't overwrite results before they're read

Solve local scheduling problems, reduce makespan

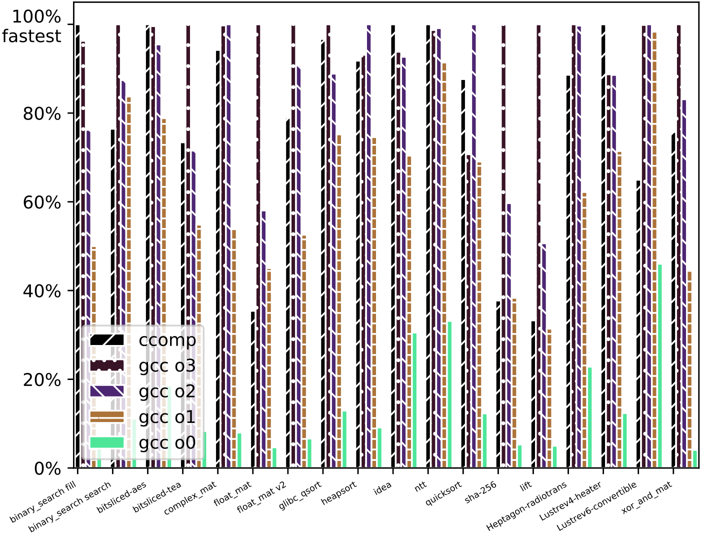
A remark on WCET: if-conversion

```
if (f) {
  x = a*b;
} else {
  x = a+b;
}
...
```

```
  mulw  $r3 = $r1, $r2
  addw  $r4 = $r1, $r2
  ld    $r16 = 8[$r12] # (following)
;;
  cmoved.weqz $r0? $r3 = $r4
```

Less branching = better for WCET

Performance



Performance indications

On Kalray K1c:
Our CompCert usually

- 2 to 17 times faster than gcc -O0
- 20% to 30% slower than gcc -O3, sometimes faster
- faster than gcc -O1

Highly dependent on the kind of code (thus the kind of optimizations we miss).

Recall gcc -O2 etc. lose traceability between source and object code and cannot be qualified for certain applications.

Future

We need **your** input!

- High-level optimizations? (e.g. loop rescheduling, software pipelining?)
- Direct compilation for high-level languages? (e.g. Scade)
- Semantics for concurrency? OpenMP?
- Alias analysis and related optimizations
- Exotic targets?
- Help for WCET?

Need to be driven by examples.

Lessons

Need proper documentation

Need optimized code generation for a new core? Give the documentation and a simulator.

Push-button?

CompCert, for the end user, is just like any other compiler.
Nearly full C99 support (no variable length arrays, no complex, no Duff's device)
A lot of code contains non-portable constructs (GNUisms etc.)

Difference with proving no undefined behaviors

Analysis: 98% green: 2% possible undefined behavior, bad
Compiling: 98% optimizations activated, very good

(NB: Absint's CompCert connected to aIT)

Questions ?

`http://www-verimag.imag.fr/~monniaux/`

- **static analysis**
- decision procedures, concolic execution
- **certified compilation**