Compiler verification, micro-architecture verification: What next?

Xavier Leroy

INRIA Paris-Rocquencourt

TTVSI, 2008-03-26
Apply formal methods to the source code to mathematically establish safety and security properties of the program.

Spectacular progress towards practically-usable verification tools in the last 10 years.
In reality . . .

Development tools

- Scade
- Matlab
- C
- Assembly
- Machine code
- code generation
- compilation
- assembling, linking

Micro-processors

- Machine code
- translation
- Internal mach.code
- micro-architecture
- Logical units
- electronics
- Circuits

X. Leroy (INRIA)  Compiler verification  TTVSI, 2008-03-26
In reality ...

Development tools

- Scade
- Matlab
- C
- Assembly
- Machine code

Formal methods

code generation

compilation

assembling, linking

Machine code

Micro-processors

- Machine code
- Internal mach.code
- Logical units
- Circuits

translation

micro-architecture

electronics

Testing
Formally verified programs deserve formally verified execution paths

Can formal methods (esp. theorem proving) be applied to development tools and micro-processors to provide “end-to-end” correctness guarantees?

In this talk: two examples,

- Formal verification of compilers
- Formal verification of micro-architectures
1 Introduction

2 Formally verified compilers

3 Verification of micro-architectures

4 Perspectives
An example of optimizing compilation

double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}

Compiled for the Alpha processor and manually decompiled back to C...
double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
    L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
        f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
        f12 = a[4]; f16 = f18 * f16;
        f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
        f11 += f17; r1 += 4; f10 += f15;
        f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
        f1 += f16; dp += f19; b += 4;
        if (r1 < r2) goto L17;
    L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
        f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
        f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
        a += 4; b += 4; f14 = a[8]; f15 = b[8];
        f11 += f22; f1 += f21; dp += f24;
    L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
        f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
        a += 4; f28 = f29 * f28; b += 4;
        f10 += f14; f11 += f12; f1 += f26;
        dp += f28; dp += f1; dp += f10; dp += f11;
        if (r1 >= n) goto L5;
    L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
        dp += f18;
        if (r1 < n) goto L19;
    L5: return dp;
    L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
    L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
    L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
    L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28; dp += f1; dp += f10; dp += f11;
    if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
    dp += f18;
    if (r1 < n) goto L19;
L5: return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
Apply formal methods to the compiler itself to prove that it preserves the semantics of the source code:

**Theorem**

*For all source codes S, if the compiler generates machine code C from source S, without reporting a compilation error, and if S executes correctly with observable behavior b, then C executes correctly with the same observable behavior b.*

Note 1: compilers are allowed to fail (ill-formed source code, or capacity exceeded).

Note 2: compilers are allowed to do anything if the source code goes wrong (has semantically undefined behavior).
Let $Spec$ be a functional specification for an application, expressed in terms of observable behavior.

**Corollary**

*If the compiler generates machine code $C$ from source $S$, without reporting a compilation error, and if $S$ satisfies $Spec$, then $C$ satisfies $Spec$.***
The Compcert experiment

(X.Leroy, Y.Bertot, S.Blazy, Z.Dargaye, P.Letouzey, T.Moniot, L.Rideau, B.Serpette)

Develop and prove correct a realistic compiler, targeted to critical embedded software.

- Source language: a subset of C.
- Target language: PowerPC assembly.
- Generates reasonably compact and fast code
  ⇒ some optimizations.

This is “software-proof codesign” (as opposed to proving an existing compiler).

The proof of semantic preservation is mechanized using the Coq proof assistant.
The subset of C supported

Supported:

- Types: integers, floats, arrays, pointers, struct, union.
- Operators: arithmetic, pointer arithmetic.
- Structured control: if/then/else, loops, simple switch.
- Functions, recursive functions, function pointers.

Not supported:

- The long long and long double types.
- goto, unstructured switch, longjmp/setjmp.
- Variable-arity functions.
The formally verified part of the compiler

- Clight
  - initial translation

- C#minor
  - stack allocation
  - instruction selection

- Cminor
  - CFG construction
  - decomposition of expressions

- LTLin
  - linearization of the CFG
  - spilling, reloading
  - calling conventions

- LTL
  - register allocation

- RTL
  - Optimizations:
    - constant propagation,
    - common subexpressions

- Linear
  - layout of stack frames

- Mach
  - PowerPC code generation

- PPC
The whole CompCert compiler

- C source
- construct AST
- type-checking (CIL)
- parsing
- simplifications
- AST C
- Type reconstruction
- Graph coloring
- AST Clight
- verified validators
- Executable
- assembling
- linking
- Assembly
- printing of asm syntax
- AST PPC
- Not proved
- Proved in Coq

X. Leroy (INRIA)
The correctness proof (semantic preservation) for the compiler is entirely machine-checked, using the Coq proof assistant (40000 lines).

Theorem transf_c_program_correct:
forall prog tprog behavior,
transf_c_program prog = Some tprog ->
Csem.exec_program prog behavior ->
PPC.exec_program tprog behavior.

Observable behaviors are either
- Termination, with a finite trace of input-output events (system calls) and the integer returned by the main function (exit code).
- Divergence, with a finite or infinite trace of input-output events.
All verified parts of the compiler are programmed directly within Coq’s specification language, in pure functional style.

- Uses monads to deal with errors and state.
- Purely functional (persistent) data structures.

(4500 lines of Coq + 1500 lines of non-verified Caml code.)

Coq’s extraction mechanism produces executable Caml code from these specifications.

Probably the biggest program ever extracted from a Coq development.
Performances of the generated code

Compilation times: within a factor of 2 of gcc -01.
Lessons learned

It can be done! for realistic source and target languages, within the limits of today’s provers.

Much room for improvement: handling goto, more optimizations, . . .

The problem of trusting the compiler hasn’t fully disappeared, but is reduced to the problem of trusting the semantics for the source (Clight) and target (PPC) languages.

A side benefit: better semantic understanding of classic compilation algorithms.
Example: CFG linearization

Input (left): machine-level instructions, arranged in a control-flow graph.
Output (right): the same instructions, arranged in a linear list + branches and conditional branches + labels.
Textbook descriptions of linearization

Tend to focus on clever heuristics to linearize loops in the best possible way. At best, pseudocode is given that looks like a depth-first traversal:

```
emit(i):
    if node i was already emitted:
        generate ‘goto Li’
    else:
        mark i as already emitted
        if i has one successor j:
            generate ’label Li’; generate instruction i; emit(j)
        if i is a conditional branch with successors j1, j2:
            generate ’label Li’
            generate ‘if(cond) goto j1’; emit(j2); emit(j1)
            or generate ‘if(not cond) goto j2’; emit(j1); emit(j2)
```

This transformation is difficult to prove correct directly, if only because it is very imperative.
Reformulating linearization to simplify the proof

Pass 1: heuristically enumerate the reachable nodes of the CFG. → an ordered list $i_1; \ldots; i_n$ of CFG nodes.

Pass 2: for each node $i$ in the order given by this list,
if $i$ has one successor $j$,
    emit label $L_i$; instr($i$); goto $L_j$
if $i$ is a conditional branch with successors $j_1, j_2$,
    emit label $L_i$; if($cond$) goto $L_{j_1}$; goto $L_{j_2}$
or label $L_i$; if($not \ cond$) goto $L_{j_2}$; goto $L_{j_1}$

Pass 3: remove goto to immediately following labels:
goto $L_i$; label $L_i$ → label $L_i$
The semantic preservation argument

Based on the following simulation diagram:

\[
Pc, R, M \xrightarrow{c = \text{label } Lpc; k} c, R, M
\]

\[
\xrightarrow{\vdots \quad t \quad \vdots \quad + \quad t}
\]

\[
Pc', R, M' \xrightarrow{c' = \text{label } Lpc'; k'} c', R', M'
\]

Left: execution of one transition in the CFG.
Right: execution of one or several transitions in the linearized code.
Semantic preservation can be proved under surprisingly weak hypotheses about the enumeration \( L \) of CFG nodes produced by pass 1:

1. \( L \) must contain all reachable nodes of the CFG
2. \( L \) must not contain any duplicates.

These conditions are easy to check for simple enumeration strategies like DFS.

For more sophisticated strategies, the enumeration can be computed by untrusted, unverified, hand-written Caml code, then validated a posteriori by proved Coq code that checks conditions (1) and (2).
1. Introduction

2. Formally verified compilers

3. Verification of micro-architectures

4. Perspectives
Formal verification of microprocessors

At the circuit level: a strong tradition of formal synthesis and verification, esp. using model checking.

At the architectural level (instruction set, memory model, …): very few publically available formal specifications.

Some academic projects focusing on “end-to-end” semantic preservation (for various values of “end”):

- From Piton assembly language to NDL netlist
  (J. Strother Moore et al, 1996)

- From ARM machine code to ARM6 micro-architecture

- The Verisoft project
  (Wolfgang Paul et al, Germany, 2005–2007)
The ARM6 micro-architecture

Figure 3: The ARM6 Data Path.
The formal verification

In HOL:

- Specification of almost all the ARM instruction set architecture (semantics of the machine language).
- Specification of the micro-architecture (pipeline, etc).
- Correctness proof of the M.A. with respect to the I.S.A.
Pipelined execution

Difficulty: several instructions are “in flight” at any given time.

Redeeming feature: synchrony. The M.A. state is determined as a function of time and the initial state.
Lessons learned

It can be done! for a realistic, widely used ISA and micro-architecture.

Side result of the highest value: a formal specification of the ARM ISA, validated by the proof of correctness of the MA.

Many possible extensions:
- to more complex, out-of-order micro-architectures;
- to asynchronous behaviors (interrupts, multi-processors).
Outline

1. Introduction
2. Formally verified compilers
3. Verification of micro-architectures
4. Perspectives
Towards a verified development and verification environment?

Much verification remains to be done:

On components of the execution path:
- Code generators, e.g. Scade $\rightarrow$ C
- Compiler front-ends for other languages, e.g. Java, ML.
- Assembler, linker, code loader.
- More verification of microprocessor architectures.

On the verification tools themselves:
- Static analyzers
- Model checkers
- Program provers (VCgen, etc)
Almost done: a verified translator mini-ML $\rightarrow$ Cminor.
(Z. Dargaye)

In progress: proof of a memory allocator and GC, written in Cminor, and interfaced with the compiler proof.
(T. Ramananandro, A. Tolmach)

In progress: a verified extraction Coq $\rightarrow$ mini-ML.
(P. Letouzey, S. Glondu)

Grand challenge: a “bootstrap” of CompCert and of the critical parts of Coq (kernel proof-checker, extraction)?
Agreeing on shared formal semantics?

Semantics of (a reasonable subset of) C and of machine code are crucial interfaces in such verification efforts. But no agreement yet on what these semantics are, exactly.
A challenge: shared-memory parallelism

Shared-memory parallelism is well known as a challenge for program verification.

It’s even more of a challenge for verifying compilers and program transformations:

- Loss of atomicity
  (Many more interleavings are possible for the generated code than for the source)
- Weakly-consistent hardware memory models.

No hope to prove semantic preservation for “racy” programs.

A glimpse of hope for race-free programs, as characterized e.g. by concurrent separation logic.
To finish...

The formal verification of compilers and other programming and verification tools

... is quite a challenge,

... is definitely exciting,

... and showcases nicely the power of theorem proving.