Correct Compilers for Correct Processors

Andreas Krall and Roland Lezuo
{andi,rlezuo}@complang.tuwien.ac.at

Vienna University of Technology, Institute of Computer Languages

January 21st 2014

This work is supported in part by the Austrian Research Promotion Agency (FFG) and by Catena DSP GmbH
Overview

1. Motivation
2. The Modeling Language CASM
3. Compiler Verification
4. Processor Simulation
5. Preliminary Results and Summary
Motivation

Computer controlled safety critical applications

Potentially can harm human life

- Aerospace, automotive, railway, industrial plants, medical equipment
- Require highest quality standards
- Often formal verification of correctness required
- Extreme high cost of quality audits
Design Constraints

Constraints:

- Performance
- Power
- Device costs
- Development costs
- Maintenance costs

Constraints
Often require application specific processors
Safety critical applications

Software
Safety critical software is written in C, and (partially) verified

Hardware
Verified hardware designs are used for safety critical applications

Compiler
A traditional compiler breaks the chain of trust

Untrusted machine code
Again requires expensive verification
Need for a verified compiler

- First attempts of verification go back to 1967
- Until now no fully verified optimizing compiler
- Most advanced compiler: CompCert
  (http://compcert.inria.fr/)
- More than 50k lines of Coq (interactive theorem prover)
- Supports a very large subset of C
Verification Techniques

- Verifix was a compiler for the Alpha architecture
- Part of Verifix was verified using abstract state machines (ASM)
- ASM have been used to specify the semantics of programming languages,
- Mature tools exists
- CoreASM and AsmL (Microsoft Research)
- We were not satisfied with the performance of existing implementations
CASM Language

Formal foundations
Based on Gurevich’s abstract state machine (ASM) method.

Well suited to model cycled circuits
- A synchronous parallel execution model (hardware is inherently parallel)
- Also allows to express sequential computation as a single atomic step (allows to express what happens during a clock cycle)
Parallel and Sequential Composition

Parallel execution mode

rule swap = {
    x := y
    y := x
}

Semantic: create update set swaping values of x and y

Sequential execution mode

rule swap =
    let temp = x in
    seqblock
    x := y
    y := temp
    endseqblock

Caller can not distinguish which swap was executed

Semantic: merge swap’s update set with the one of caller
rule andi(addr : Int) =
let rs = PARG(addr, FV_RS) in
let rt = PARG(addr, FV_RT) in
let imm = PARG(addr, FV_IMM) in
  if rt != 0 then
    GPR(rt) := BVand(32, GPR(rs), BVZeroExtend(imm, 16, 32)))
enum PipelineStages = { ID, EX, MEM, WB }
enum PipelinePhases = { begin, end }
function Pipeline : PipelineStages -> RuleRef

rule execute_pipeline =
seqblock
  forall s in PipelineStages do //begin
    let op = Pipeline(s) in //begin
      if op != undef then //begin
          call (POP(op))(op, s, begin) //begin
      end //end
  end //end
endseqblock
Motivation

The Modeling Language CASM
Compiler Verification
Processor Simulation
Preliminary Results and Summary

Modeling an Instruction

```plaintext
rule andi(addr:Int, stage:Int, phase:Int) = {
    if stage = ID and phase = end then
        let rs = PARG(addr, FV_RS) in
        let rt = PARG(addr, FV_RT) in
        let imm = PARG(addr, FV_IMM) in {
            call(ID_READ_OP1)(rs)
            IDOP2 := BVZeroExtend(imm, 16, 32)
            IDRESREG := rt
        }
    if stage = EX and phase = begin then
        EXRES := BVAnd(32, EXOP1, EXOP2)

    if stage = WB and phase = begin then
        call(WRITE_REGISTER)(WBRESREG, WBRES)
}
```

HiPEAC 2014 Correct Compilers for Correct Processors
CASM Implementations

Three different implementations have been developed:

- Interpreter
- Optimizing source to source compiler C/C++
- Interpreter for symbolic execution (traces in TPTP format)
Compiler Verification

Multiple techniques used

- Verified compiler (frontend and analyses)
- Translation validation (backend)
- Cooperative compiler
Hydra analysis and transformation specification language

- Set based language
- Specification of intermediate representations
- Specification of analyses (fix point iterations)
- Specification of transformations
- Concise specifications, easy to verify manually or semiautomatically
Translation Validation

Source Code -> PASS -> Compiler -> PASS -> Object Code

Check Check Check

Translation Validation
Example Pass: Instruction Selection

- Source Code
  - Lexing
  - Parsing
  - AST

Frontend

Middleend

- PASS
  - AST
  - PASS
  - AST
  - PASS

Backend

- Machine language
  - MOV R0,R1
  - ADD R1,R2
  - XOR R2,R0
  - Register-mapping

- Object Code

HiPEAC 2014
Correct Compilers for Correct Processors
Simulation Proofs

\[ p \in SL \xrightarrow{\text{compile}} C(p) \in TL \]

SL semantics (ASM) \[ [p]_{SL} \] symbolic execution predicates Theorem Prover

\[ \cong \] restricted simulation equality in compiler correctness

TL semantics (ASM) \[ [C(p)]_{TL} \] symbolic execution predicates

A compilation is correct iff \[ [p]_{SL} \cong [C(p)]_{TL} \]

HiPEAC 2014 Correct Compilers for Correct Processors
### Example

#### Program

```
MOV 0x03, R1
ADD R0, R1, R2
```

#### ASM model

```
seqblock
RESULT := ADD(3, OP(1))
endseqblock

seqblock
REG(1) := 3
REG(2) := ADD(REG(0), REG(1))
endseqblock
```

#### Predicates

- `OP(1, sym0)`.
- `ADD(sym0, 3, sym1)`.
- `RESULT(sym1)`.
- `REG(1,3)`.
- `REG(0, sym2)`.
- `ADD(sym2, 3, sym3)`.
- `REG(2, sym3)`. 
Correctness of Instruction Selection

AST

$$\begin{array}{l}
\text{ADD} \\
3 \\
\text{OP1}
\end{array}$$

Machine language

$$\begin{array}{l}
\text{MOV} \ 0x03, \ R1 \\
\text{ADD} \ R0, \ R1, \ R2
\end{array}$$

Register mapping

$$\begin{array}{l}
R0: \ \text{OP1} \\
R1: \ \text{int\_const} \ 3 \\
R2: \ \text{AST result}
\end{array}$$

Correctness (simplified)

$$\forall \text{op} : \text{regmap}(\text{op}) \equiv \text{op} \implies \text{regmap}(\text{result}_{\text{AST}}) \equiv \text{result}$$
Symbolic execution of ASM models

Goal

Generate (first-order) predicates of semantic transformation.

- Common semantic vocabulary modeled as external functions
- Update $f(l) := u \xrightarrow{\text{predicate}} f(l, u)$.
- Invocation of external function $f(a) \xrightarrow{\text{predicate}} f(a, r)$.
- If $f(l) \equiv \text{undef}$:
  - create symbol $s$, change $f(l) := s \xrightarrow{\text{predicate}} f(l, s)$.
- Correctly handle explicitly set undef values

$\Rightarrow$ Set of predicates over common semantic vocabulary and symbolic values
Example Proof

From symbolic execution:

OP(1, sym0).
ADD(sym0, 3, sym1).
RESULT(sym1).

REG(1,3).
REG(0, sym2).
ADD(sym2, 3, sym3).
REG(2, sym3).

From Registermapping:

sym0 = sym2.

To prove:

sym1 = sym3?

Proof done by theorem prover (Vampire, VanHelsing)
Processor simulation

Instruction set simulator:

Interpreting simulator
- Low start up time (loading of application program)
- High simulation time

Compiling simulator
- Each application program is compiled to a specialized simulator
- High start up time (compilation of application and specialized simulator)
- Low simulation time
Interpreting simulator

- Compile CASM to C++
- Link with C++ runtime
- Interface MIPS syscall with host C library
- Link simulatee with C library stubs (using syscall)
Simulator verification (by symbolic execution)

- Execution models
- Proofs are trivial
- Additionally check pipeline and execution models
- Can also proof operand forwarding to be correct

Motivation
The Modeling Language CASM
Compiler Verification
Processor Simulation
Preliminary Results and Summary

Simulator verification (by symbolic execution)

- (symbolic) initial state
- specification model
- pipelined model
- final state
- final state

HiPEAC 2014 Correct Compilers for Correct Processors
### Evaluation of the MIPS CASM models

<table>
<thead>
<tr>
<th>Specification Models</th>
<th>Pipeline Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>600 LOC for instructions</td>
<td>1500 LOC for instructions</td>
</tr>
<tr>
<td>60 LOC for execution model</td>
<td>400 LOC for each pipeline model (forwarding and bubbling)</td>
</tr>
<tr>
<td>50 LOC for state, and memory access helpers</td>
<td>1 day to create forwarding model</td>
</tr>
<tr>
<td>Written in 2 days</td>
<td>15 min to derive bubbline pipeline model</td>
</tr>
</tbody>
</table>

Pipelined instruction models copy’n’paste error all caught by model verification
Preliminary Results

CASM language:
- Definition of the statically typed CASM language
- CASM compiler
- CASM interpreter
- CASM symbolic execution engine

Proofs (LLVM and VLIW compiler):
- Instruction selection
- Register allocation
- Instruction scheduling
- Software pipelining

Simulator:
- Processor models for MIPS and VLIW architectures
- Interpreting simulator
- Compiling simulator
Performance of CASM compiler (relative to compiler)

<table>
<thead>
<tr>
<th>Function</th>
<th>CoreASM</th>
<th>CASM-i</th>
<th>AsmL</th>
</tr>
</thead>
<tbody>
<tr>
<td>trivial</td>
<td>15.73</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sieve</td>
<td>1.82</td>
<td>161.29</td>
<td></td>
</tr>
<tr>
<td>quicksort</td>
<td>20.26</td>
<td>5.23</td>
<td></td>
</tr>
<tr>
<td>gray</td>
<td>15.61</td>
<td>31.64</td>
<td></td>
</tr>
<tr>
<td>fibonacci</td>
<td>1.37</td>
<td>2.486.85</td>
<td></td>
</tr>
<tr>
<td>bubblesort</td>
<td>2.07</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance of instruction selection verification

<table>
<thead>
<tr>
<th></th>
<th>IR CASM</th>
<th>ML CASM</th>
<th>Traces &amp; Prover</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td>1904</td>
<td>1904</td>
<td>2124</td>
<td>284.200 s</td>
</tr>
<tr>
<td>Lines</td>
<td>747602</td>
<td>29597978</td>
<td>5887294</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>8.002 s</td>
<td>19.148 s</td>
<td>81.190 s</td>
<td>284.200 s</td>
</tr>
</tbody>
</table>
Performance of instruction set simulator

- About 1 Mhz for interpreting simulator
- About 3 Mhz for compiling simulator
Summary

- ASM models for machine language and compiler IR
- Symbolically evaluate ASM models
- Use simulation proofs to show correctness of instruction selection
- Use very same models and an ASM to C++ compiler for fast cycle-accurate simulation
- Working on backend generation from ASM model
Conclusion

- Using different levels of abstraction, verification of compilers and processors can be done efficiently.
- It is often possible to improve already mature tools (sometimes by orders of magnitude).
- Formal modeling reduces design and evaluation time for application specific processors and the corresponding tool chain.
- Verification leads to better compilers and processors at lower cost.
- All compilers should be verified.
Literature

More detailed information

- CASM: Implementing an Abstract State Machine based Programming Language (ATPS’13)
- A Unified Processor Model for Compiler Verification and Simulation using ASM (ABCZ’12)
- Using the CASM Language for Simulator Synthesis and Model Verification (RAPIDO’13)
- CASM - Optimized Compilation of Abstract State Machines (LCTES’14)
This work would not have been possible without the contribution of

Dominik Inführ
Roland Lezuo
Philipp Paulweber
Richard Plangger
Dietmar Schreiner

This work was supported in part by the Austrian Research Promotion Agency (FFG) and by Catena DSP GmbH
Thanks

Thank you for your attention!