

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

Verified Compiler

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  
}
```

Sequential execution mode

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

Semantic: create **update set** swapping values of x and y

```
rule caller =  
  call swap
```

Caller can not distinguish which swap was executed

Semantic: **merge** swap's update set with the one of caller

Specifying an Instruction

```
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))
```

Modeling a Pipeline

```
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
  forall s in PipelineStages do           //end
    let op = Pipeline(s) in              //end
      if op != undef then                 //end
        call (POP(op))(op, s, end)       //end
      end
    end
  end
endseqblock
```

Modeling an Instruction

```
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)  
}
```

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

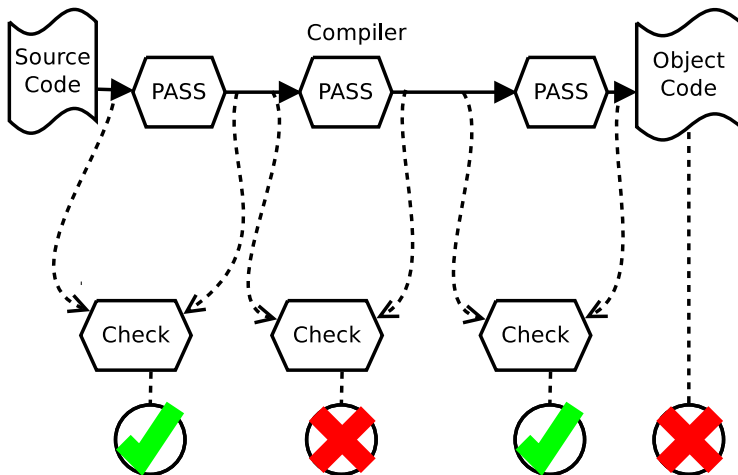
- Verified compiler (frontend and analyses)
- Translation validation (backend)
- Cooperative compiler

Frontend Verification

Hydra analysis and transformation specification language

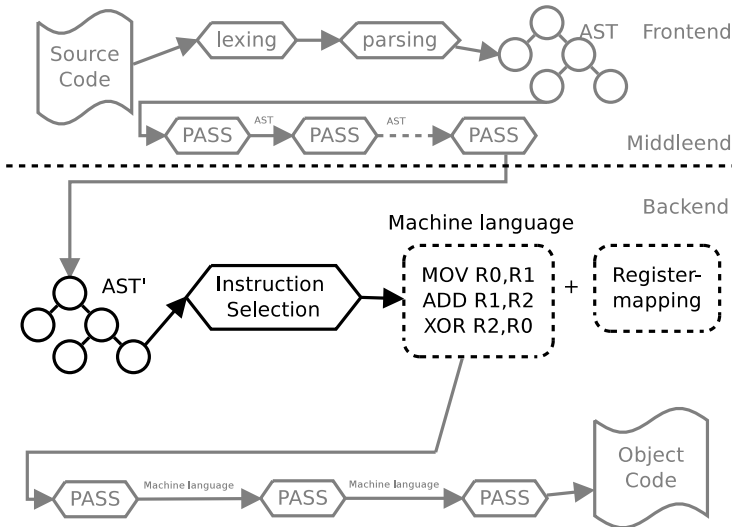
- Set based language
- Specification of intermediate representations
- Specification of analyses (fix point iterations)
- Specification of transformations
- Concise specifications, easy to verify manually or semiautomatically

Translation Validation

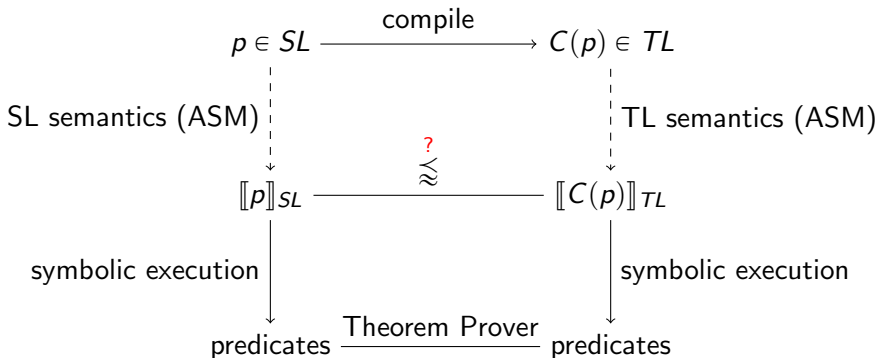


Translation Validation

Example Pass: Instruction Selection

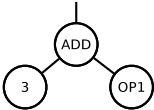


Simulation Proofs



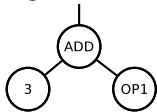
A compilation is correct *iff* $[[p]]_{SL} \approx [[C(p)]]_{TL}$
 (\approx : restricted simulation \triangleq equality in compiler correctness)

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



Machine language

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

Registermapping

```
R0: OP1
R1: int_const 3
R2: AST result
```

Correctness (simplified)

$$\forall op : \text{regmap}(op) \equiv op \implies \text{regmap}(\text{result}_{AST}) \equiv \text{result}$$

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

⇒ 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:

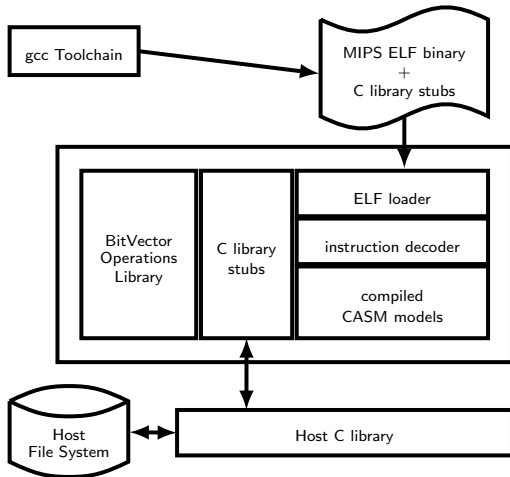
Interpreting simulator

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

Compiling simulator

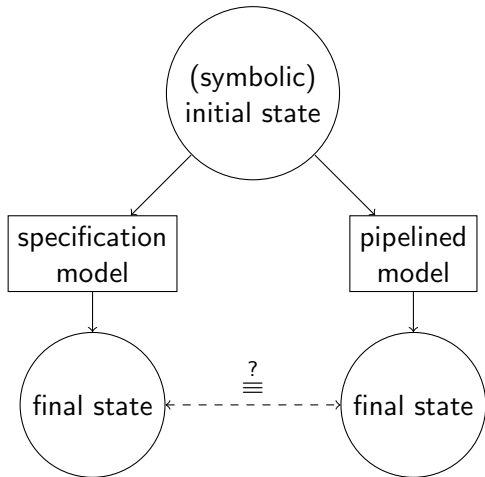
- Each application program is compiled to a specialized simulator
- High start up time (compilation of application and specialized simulator)
- Low simulation time

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

Evaluation of the MIPS CASM models

Specification Models

- 600 LOC for instructions
- 60 LOC for execution model
- 50 LOC for state, and memory access helpers
- Written in 2 days

Pipeline Models

- 1500 LOC for instructions
- 400 LOC for each pipeline model (forwarding and bubbling)
- 1 day to create forwarding model
- 15 min to derive bubbleline pipeline model

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

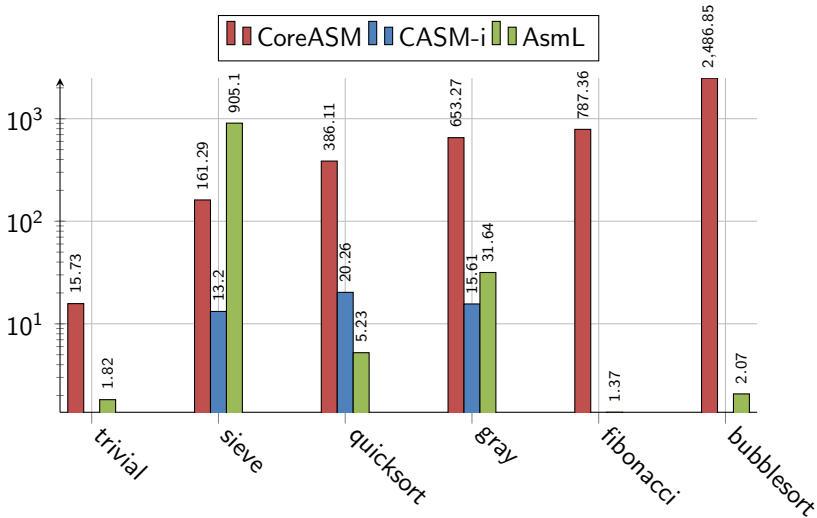
Proofs (LLVM and VLIW compiler):

- 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)



Performance of instruction selection verification

	IR CASM	ML CASM	Traces & Prover	Total
Files	1904	1904	2124	
Lines	747602	29597978	5887294	
Time	8.002 s	19.148 s	81.190 s	284.200 s

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)

Acknowledgements

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!