CASM - Optimized Compilation of Abstract State Machines

Roland Lezuo, Philipp Paulweber and Andreas Krall

Institute of Computer Languages, Vienna University of Technology

Jun 12, 2014

---

1This work is partially supported by the Austrian Research Promotion Agency (FFG) under contract 827485, Correct Compilers for Correct Application Specific Processors and Catena DSP GmbH.
ASM in a nutshell:
- well-founded (formal) method
- generalization of finite state machines
- changes interpretation of an algebra

pure functions calculate successor state in parallel

ASM is well suited for:
- programming language semantics
- clocked circuits (like micro-processors)
Motivation

We use ASM in:
- compiler verification
- formalization of instruction sets

**Rule addiu**

```plaintext
rule addiu(addr : Int) =
let rs = FIELD(addr, FV_RS) in
let rt = FIELD(addr, FV_RT) in
let imm = FIELD(addr, FV_IMM) in
if rt != 0 then
  GRP(rt) := BVadd(32, GPR(rs), BVse(16, 32, imm))
```

MIPS *addiu* (functional model)

Idea: re-use models for
- instruction set simulation (ISS)
- compiled simulation (CS)

Issue: existing ASM tools too slow \(\Rightarrow\) **CASM**
- statically typed
- compilation to C
Parallel and Sequential execution semantics

\begin{align*}
\{ x := y \\
y := x \}
\end{align*}

parallel semantics

\begin{align*}
\{ t := x \\
x := y \\
y := t \}
\end{align*}

sequential semantics

⇒ concise modeling parallelism in pipelines / VLIW
sequential execution of parallel blocks
updates collected into sets
on leaving a block: merge updates into surrounding one

Idea: no intermediate states, overlay update-set
run-time stack of update-sets
PAR/SEQ implementation

**Linked Hash-Map**

- lookup: $O(\#ps)$
- merge: $O(\#updates)$
- insert: $O(1)$

⇒ most expensive run-time operations
CASM baseline compiler

Factor

10^3

10^2

10^1

10^0

10^{-1}

sieve  
161.29  
905.1

quicksort  
12.73  
60.42

gray  
5.23  
31.64

fibonacci  
1.37  
2.60

bubblesort  
2.07  
37.49

CoreASM  
CASM-i  
AsmL

Roland Lezuo, Philipp Paulweber and Andreas Krall

CASM - Optimized Compilation of Abstract State Machines
Redundant Lookup and its Elimination

\begin{verbatim}
{ 
  if X(3) = 3 then
    skip
  if X(3) = 4 then
    skip
}
\end{verbatim}

\begin{verbatim}
{ 
  local X_3 = X(3) in
    if X_3 = 3 then
      skip
    if X_3 = 4 then
      skip
}
\end{verbatim}
Preceded Lookup and its Elimination

\[
\{|
\text{X(4) := foo}
\text{if X(4) > 0 then}
\text{skip}
\}
\]

\[
\text{local L_1 = foo in}
\{|
\text{X(4) := L_1}
\text{if L_1 > 0 then}
\text{skip}
\}
\]
Redundant Update and its Elimination

\[
\begin{align*}
\{ l & \\
X(5) & := \text{foo} \\
X(5) & := \text{bar} \\
l & \}
\end{align*}
\]

\[
\begin{align*}
\{ l & \\
X(5) & := \text{bar} \\
l & \}
\end{align*}
\]
Patterns in Compiled Simulation (simplified)

```plaintext
// pipeline stages
enum S = { S1, S2, S3 }

// phases (latch-in, latch-out)
enum P = { P1, P2 }

rule fetch(r : Int) =
    pipeline(S1) := PROGRAM(r)

rule execute =
    forall s in S do
        if pipeline(s) != undef then {|
            call (pipeline(s))(P1)
            call (pipeline(s))(P2)
        |
    }

rule step = {
    pipeline(S2) := pipeline(S1)
    pipeline(S3) := pipeline(S2)
}
```

- **redundant update**
- **preceded lookup**
- **redundant lookups**

similar for register file
Supporting Optimizations

Key: statical analysis of *locations*

- constant propagation
- constant folding
- inlining
- lookup elimination
- update elimination
- sinking

compilation to C ⇒ less complex code, better C optimization
Achieved Speedup and Performance

speedup depends on size of frequently executed basic blocks
Conclusion

- re-use of formal models
- baseline compiler order of magnitudes faster than other tools
- for CS application: optimizations yield factor 6
- current work: interprocedural analysis, new optimizations