CASM - Optimized Compilation of Abstract State Machines [1]

Roland Lezuo, Philipp Paulweber and Andreas Krall

Institute of Computer Languages, Vienna University of Technology

Jun 12, 2014

[1] This 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.

Abstract State Machines

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(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 ⇒ **CASM**

- statically typed
- compilation to C

Parallel and Sequential execution semantics

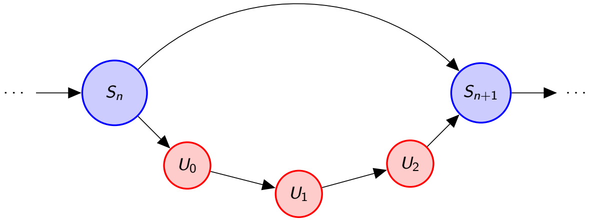
Swap

```
{
        x  :=  y
        y  :=  x
}
```

parallel semantics

```
{|
        t  :=  x
        x  :=  y
        y  :=  t
|}
```

sequential semantics



$\Rightarrow$ concise modeling parallelism in pipelines / VLIW

PAR/SEQ nesting

program

```
{
| {|
| | x := 23
| | y := 42
| | {
| | |    x := y
| | |    y := x
| | }
| |}
| // ...
```

updates	state	update-set
	x=undef, y=undef	{}
x=23	x=23, y=undef	{ x_1=23 }
y=42	x=23, y=42	{ x_1=23, y_1=42 }
x=42		{ x_2=42, y_1=42 }
y=23		{ x_2=42, y_2=23 }
	x=42, y=23	
	x=undef, y=undef	{ x_1=42, y_1=23 }

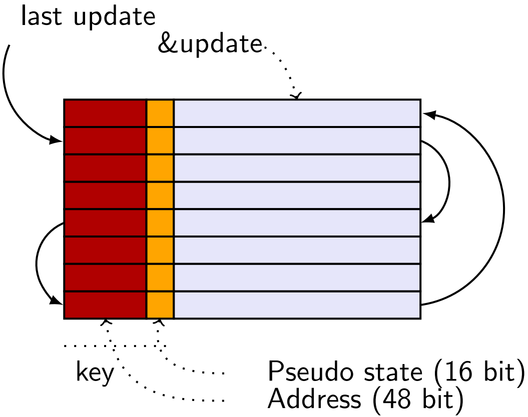
- sequential execution of parallel blocks
- updates collected into sets
- on leaving a block: merge updates into surrounding one

Idea: no intermediate states, *overlay* update-set

- run-time stack of update-sets

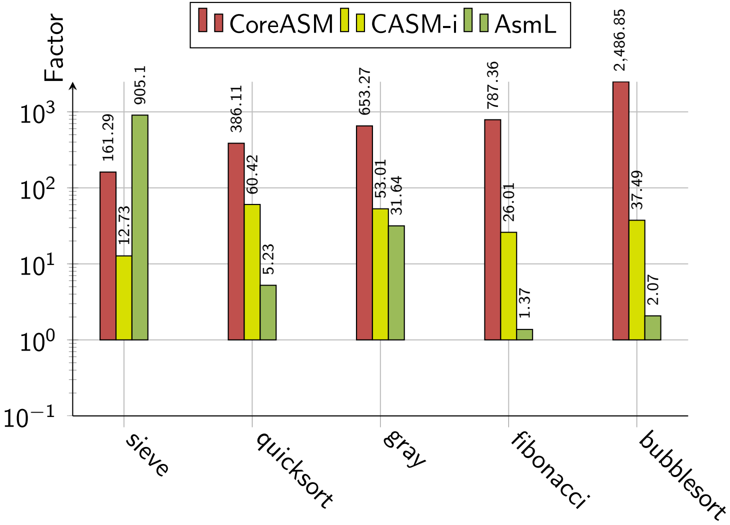
Linked Hash-Map

lookup: $\mathcal{O}(\#ps)$, merge: $\mathcal{O}(\#updates)$, insert: $\mathcal{O}(1)$



⇒ most expensive run-time operations

CASM baseline compiler



Redundant Lookup and its Elimination

```
{

  if X(3) = 3 then
    skip
  if X(3) = 4 then
    skip
}
```

```
{
  local X_3 = X(3) in
    if X_3 = 3 then
      skip
    if X_3 = 4 then
      skip
}
```

Optimization Patterns 2/3

Preceded Lookup and its Elimination

```
{|
  X(4) := foo
  if X(4) > 0 then
    skip
|}
```

```
local L_1 = foo in
  {|
    X(4) := L_1
    if L_1 > 0 then
      skip
  |}
```

Optimization Patterns 3/3

Redundant Update and its Elimination

```
{|
  X(5) := foo
  X(5) := bar
|}
```

```
{|

  X(5) := bar
|}
```

Patterns in Compiled Simulation (simplified)

```
rule basicblock = {|
  call fetch(0x8000)
  call execute
  call step

  call fetch(0x8001)
  call execute
  call step

  // ...

  call fetch(0x8023)
  call execute
  call step
|}
```

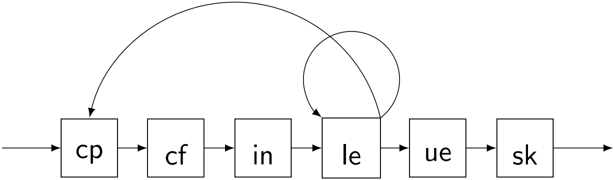
- redundant update
- preceded lookup
- redundant lookups

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

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

rule fetch(r : Int) =
  pipeline(S1) := PROGMEM(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)
}
```

similar for register file

Supporting Optimizations

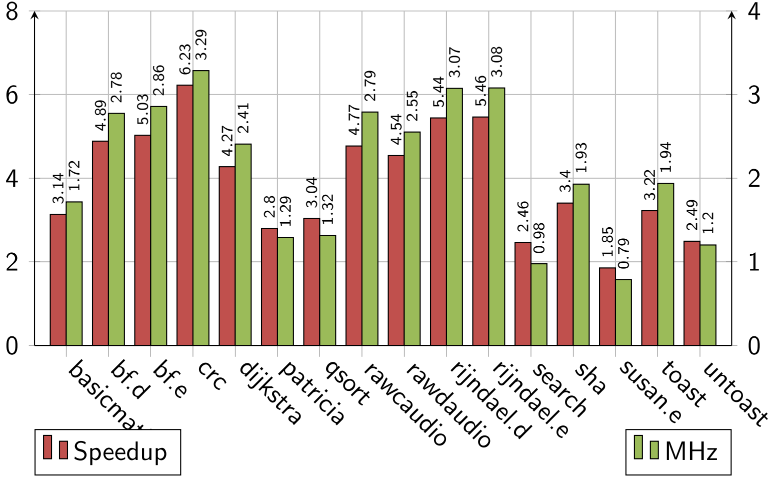
Key: statical analysis of *locations*



- 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 optimziations