MetaMapper: Automatic Rewrite Rule Synthesis and Instruction Selection

Ross Daly & Jackson Melchert

Stanford University

Motivation

1 Every new instruction set architecture (ISA) must be accompanied by a set of rewrite rules used for code generation

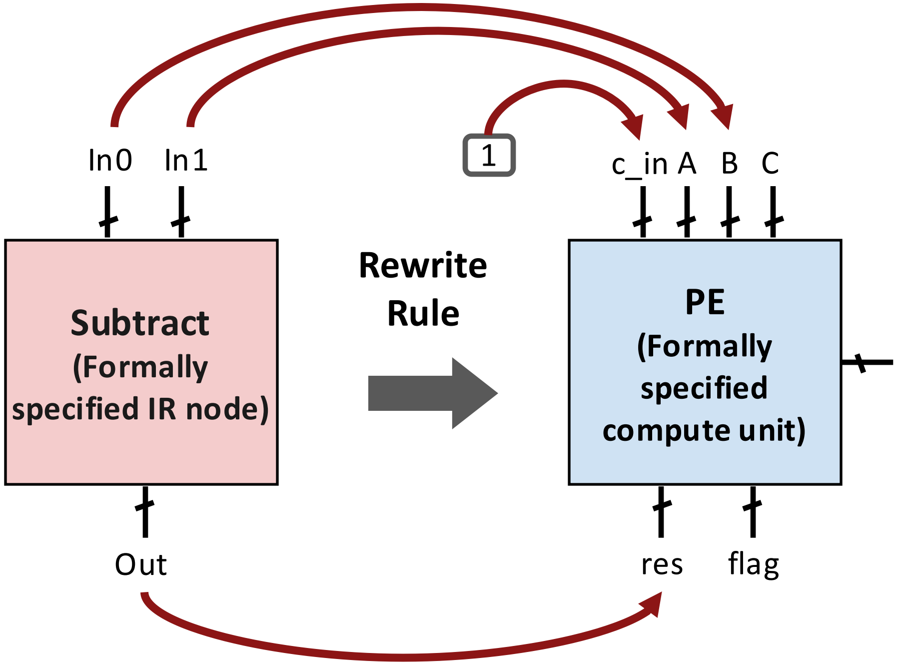
2 Crafting these rules by hand is time consuming and error prone

3 This leads to a world where there are few ISAs and design space exploration is difficult

Contributions

- A methodology for efficiently encoding and solving the rewrite rule synthesis problem using SMT
- A technique for supporting parametric rewrite rules
- A method for abstracting operations whose semantics are either unknown or too complex to model efficiently

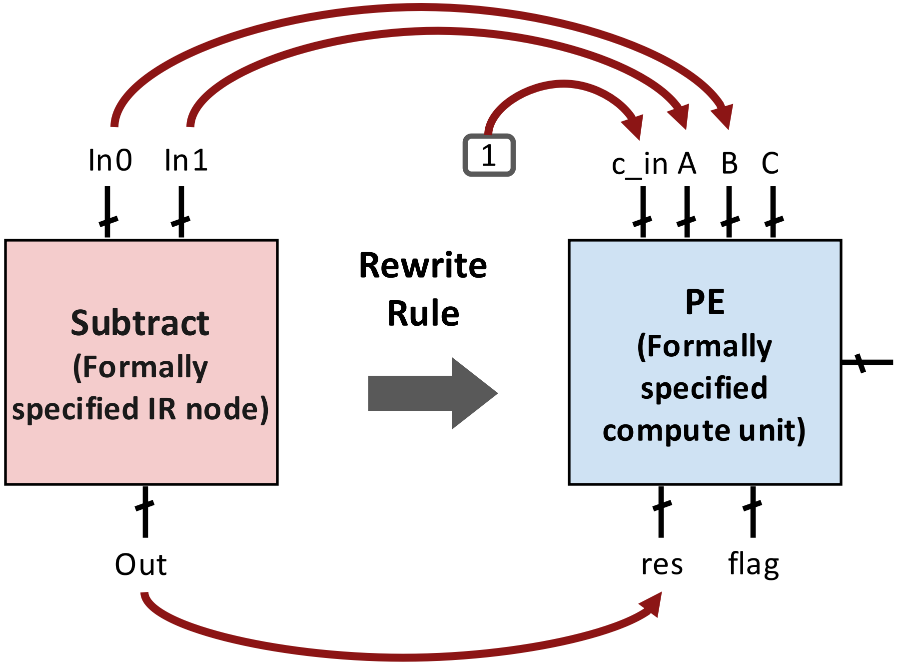
Automatic Rewrite Rule Synthesis Using SMT



Instruction (
op = Add,
invert_A = 1,
...
)

Synthesizing Instruction Selection Rewrite Rules from RTL Using SMT. Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. Formal Methods in Computer-Aided Design (FMCAD), 2022

Automatic Rewrite Rule Synthesis Using SMT



Instruction (
  op = Add,
  invert_A = 1,
  …
)

∃inst ∀inputs : Subtract(inputs) == PE(inst,inputs)

Solve using an SMT solver

Synthesizing Instruction Selection Rewrite Rules from RTL Using SMT. Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. Formal Methods in Computer-Aided Design (FMCAD), 2022

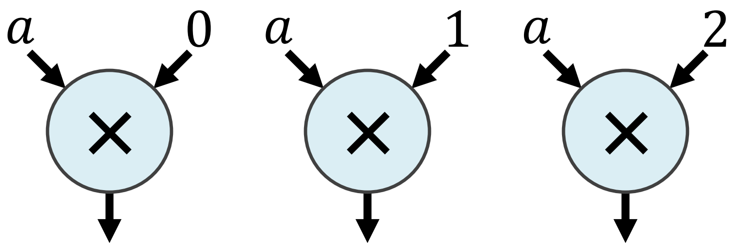
Automatic Rewrite Rule Synthesis Using SMT

- More generally, we solve this formula for every IR operation we are interested in:

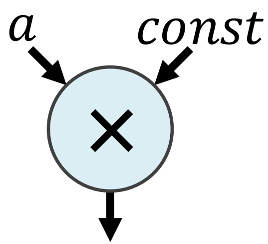
$$\exists inst\ \forall inputs : PE(inst, inputs) == IR(inputs)$$

Synthesizing Parametric Rewrite Rules

- Sometimes we are interested in *parameterized* rules



Instead of these rewrite rules



We have this single rule

Synthesizing Parametric Rewrite Rules

- To solve for parametric rules, we solve the following formula:

$$\exists inst\ \forall inputs, const : PE(inst(const), inputs) == IR(const, inputs)$$

- inst(const) is a function from the constant value to instructions

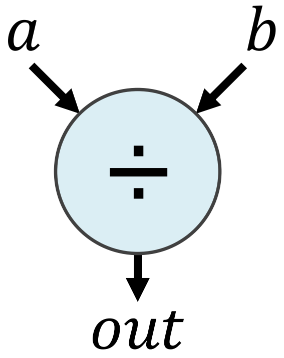
Abstracting Complex Instructions

- Complex instructions like floating point arithmetic pose a challenge
  - Some complex operations cannot be represented well in SMT
  - For example, PEs might include encrypted Verilog for performing floating point arithmetic

- It's often the case that there are identical complex operations in the IR and the architecture
  - We can replace these complex operations with uninterpreted functions, or black boxes

Abstracting Complex Instructions

- Create uninterpreted function for every complex operation

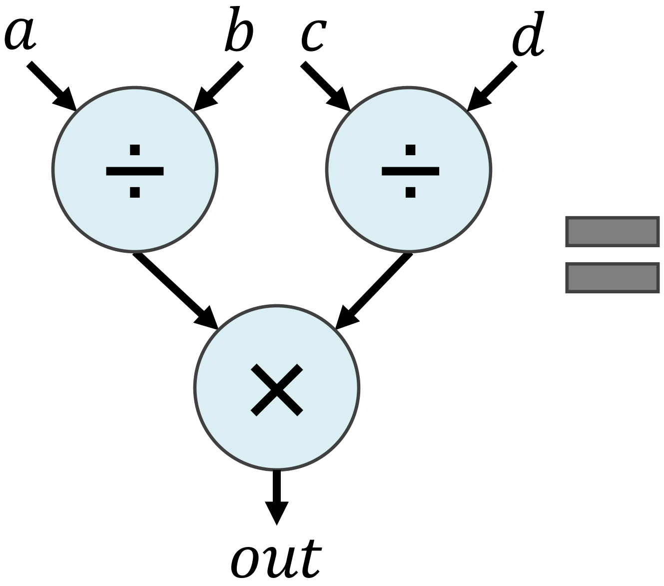


$$out = f(a, b)$$

- Division in this example is replaced with the function f

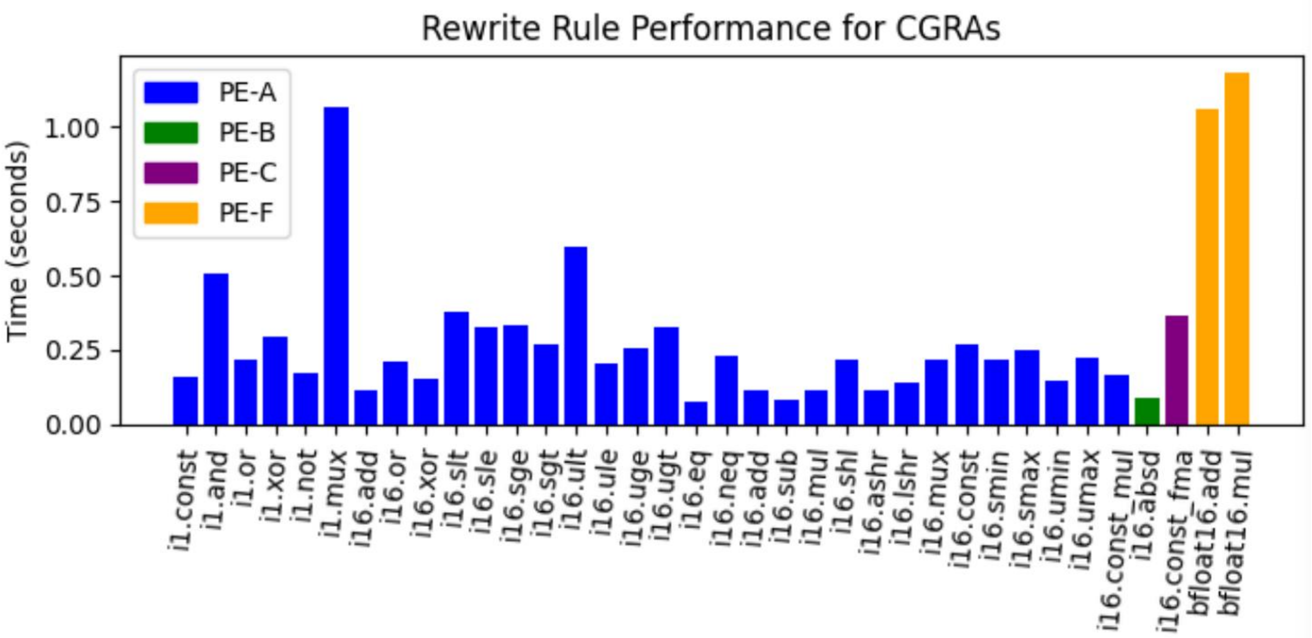
Abstracting Complex Instructions

- Reuse the same uninterpreted function for all occurrences of that operation in the PE and in the IR



$$out = g(f(a,b), f(c,d))$$

Rewrite Rule Synthesis Runtime



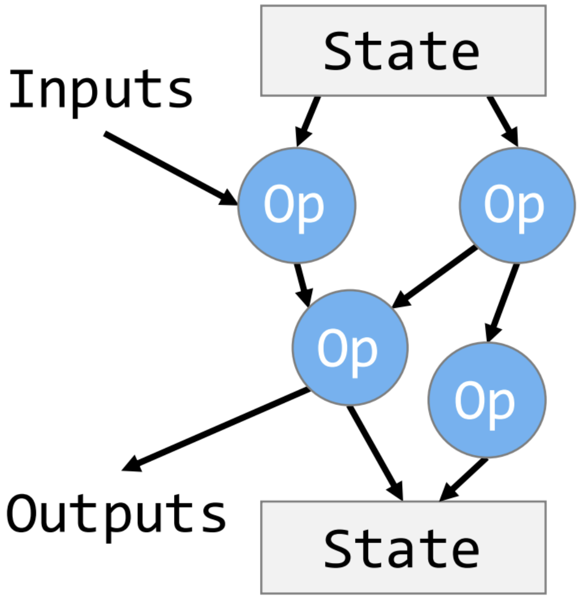
- Rewrite rule synthesis is fast and can be run during compilation
- Changes to architecture are easy to adapt to

MetaMapper

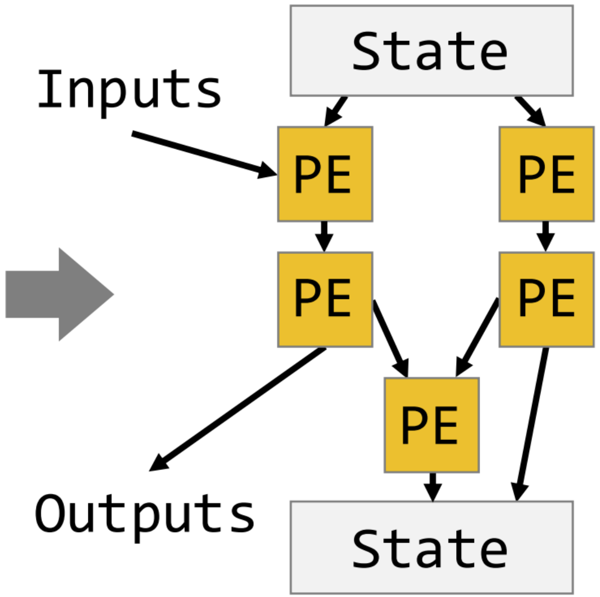
- MetaMapper integrates automatic rewrite rule synthesis with tools for instruction selection

- "Meta" mapper refers to the fact we are compiling a new compiler for each version of the hardware
  - Each invocation of MetaMapper with a new PEak PE will generate a set of rewrite rules

Instruction Selection



IR Dataflow Graph



Mapped PE Dataflow Graph

Demo

- In this demo, we will add an instruction to our existing processing element, generate a new rewrite rule for it, and then map an application that takes advantage of it
- First, we will map the camera pipeline app without any new operations
- `aha map apps/camera_pipeline_2x2`
- This uses 208 PEs

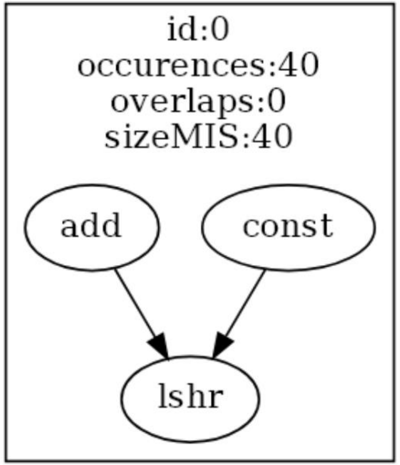
Demo

- One common complex operation that is present in this application is $(a + b) \gg const$
  - We will add this to our PE
- Open /aha/lassen/lassen/alu.py
- Add the following code:

`ADDSHR = 20` to line 29

`elif alu == ALU_t.ADDSHR:` to line 180

```
    res, res_p = adder_res >> UData(c), Bit(0)
```



Demo

- Now we need an IR representation of the operation:

```
(in1 + in0) >> in2
```

- Run: `mv /aha/addshr.py /aha/lassen/lassen/rewrite_rules/`

Demo

- Go to the /aha/lassen directory and run
- **`python scripts/solve_rewrite_rules.py`**
- This will generate the rewrite rule, or configuration for the PE such that it implements the add shift operation
- Now we can rerun **`aha map apps/camera_pipeline_2x2`** to see our reduction in number of PE needed to map this application

Conclusion

- MetaMapper allows for the compilation of a new compiler for any new PE design
    - It efficiently synthesizes rewrite rules for IR operations in your applications
    - Significantly reduces the amount of effort needed to compile to new design
    - It allows for design space exploration of interesting PE designs