Verified Agile Hardware

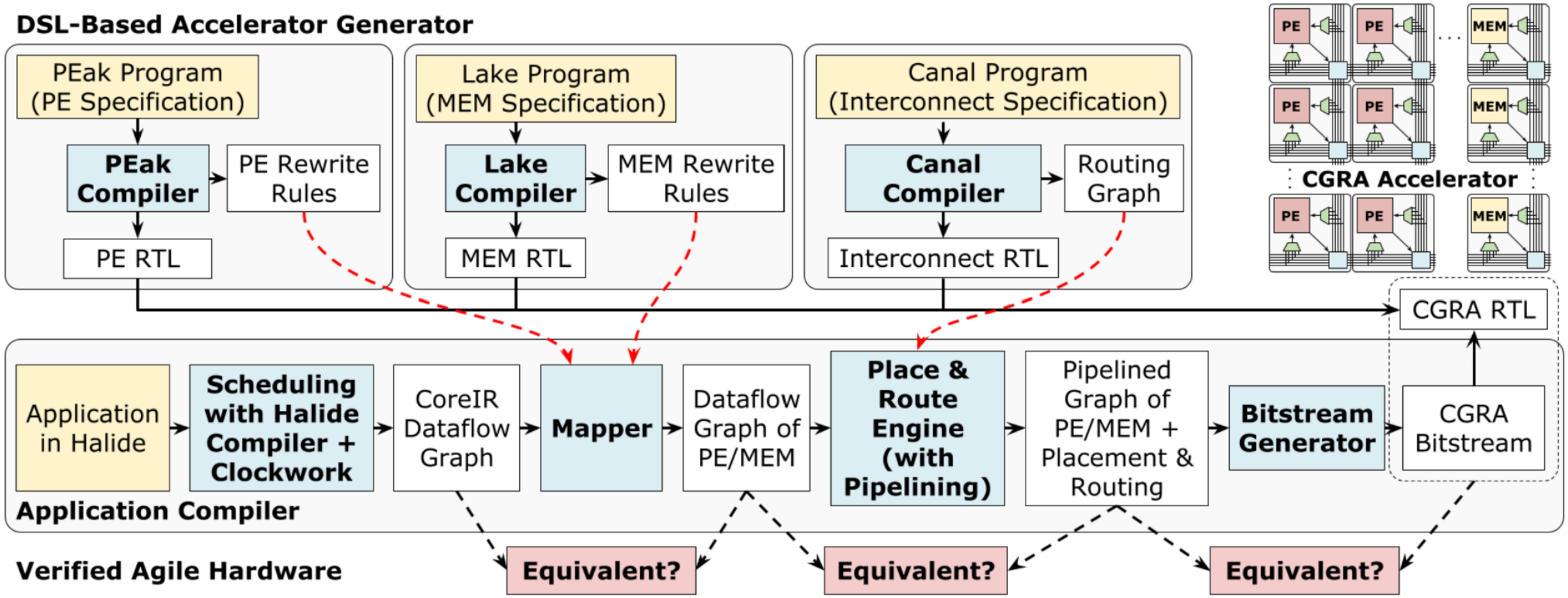
JACK MELCHERT

Motivation

- PEak, Lake, and Canal have enabled easy creation of our CGRA and its compiler
  - Any change in our hardware is reflected in our compiler
- The verification and security story has largely been missing so far
- We expect to be able to automate the verification of the CGRA and the compiler

Goals

1. Compute and memory mapping verification
2. Place and route verification
3. Bitstream generation verification

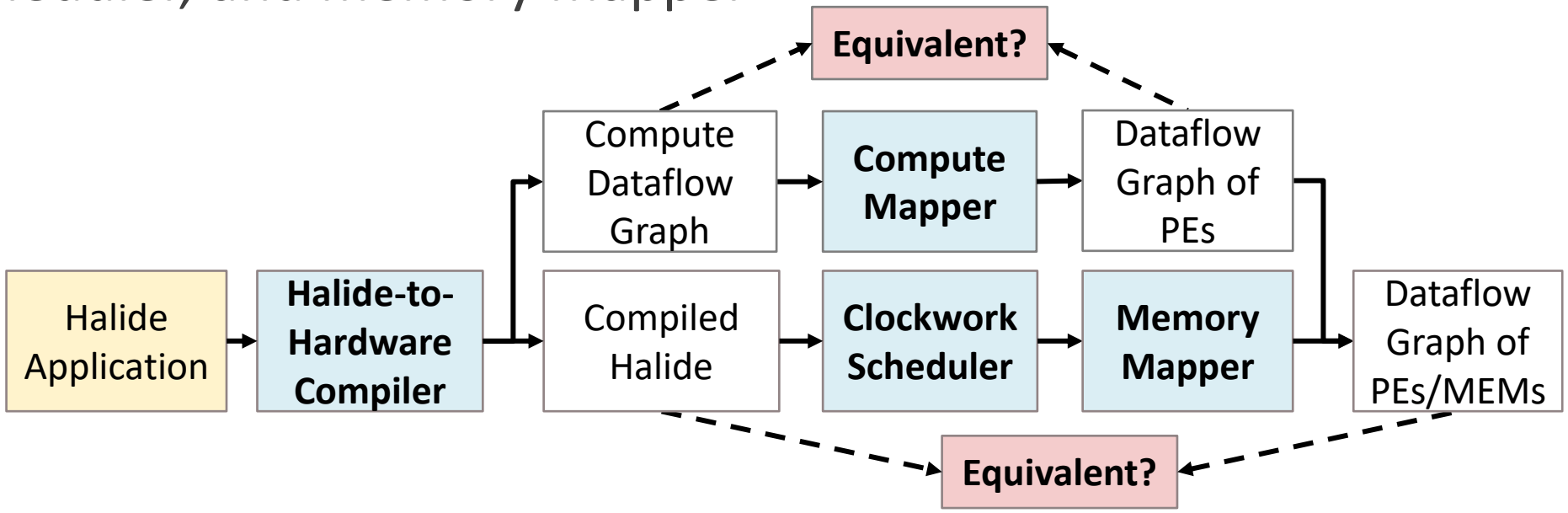


Background

- Translation validation
  - Technique for formally proving the equivalence of a source and target program after a software compiler pass
  - Requires the ability to symbolically represent the two programs
- SMT – Satisfiability Modulo Theories
  - Generalization of Boolean SAT to more complex formulas
  - SMT solvers (cvc5, bitwuzla) and model checkers (pono) are used for translation validation

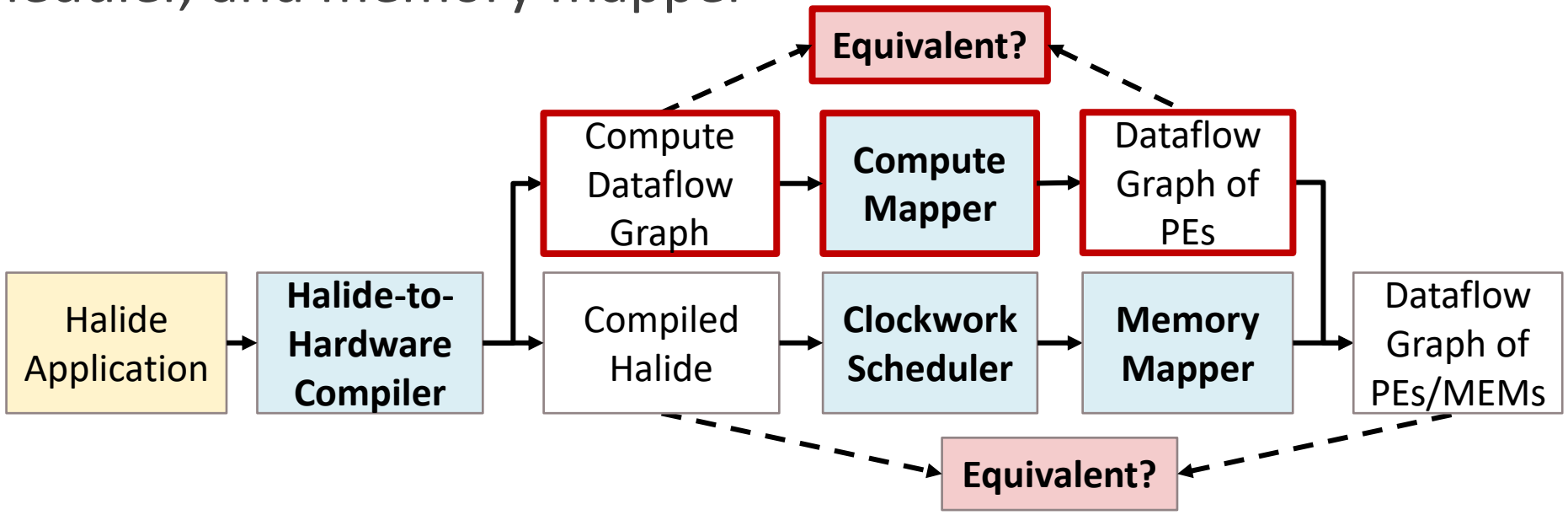
Mapping Verification

- Mapping verification includes verifying the compute mapper, scheduler, and memory mapper



Mapping Verification

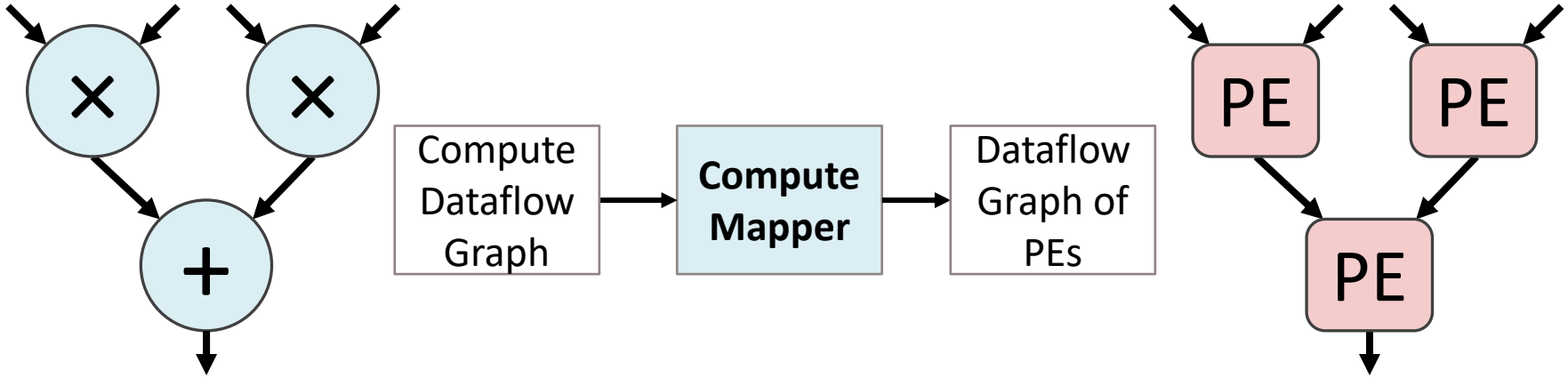
- Mapping verification includes verifying the compute mapper, scheduler, and memory mapper



- We'll start with verifying the compute mapping result

Compute Mapping

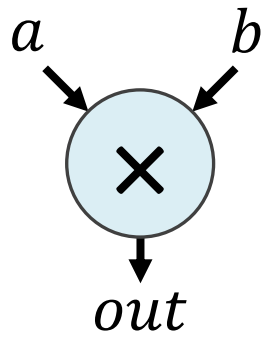
- Compute mapping transforms a dataflow graph of primitive operations into a dataflow graph of configured PEs
  - Multiply will be replaced with a PE configured to do a multiply



- Each node in the dataflow graphs has a PEak representation

Compute Mapping

- PEak programs can be interpreted in three ways:
  1. Python functional model
  2. RTL
  3. Formal representation in SMT



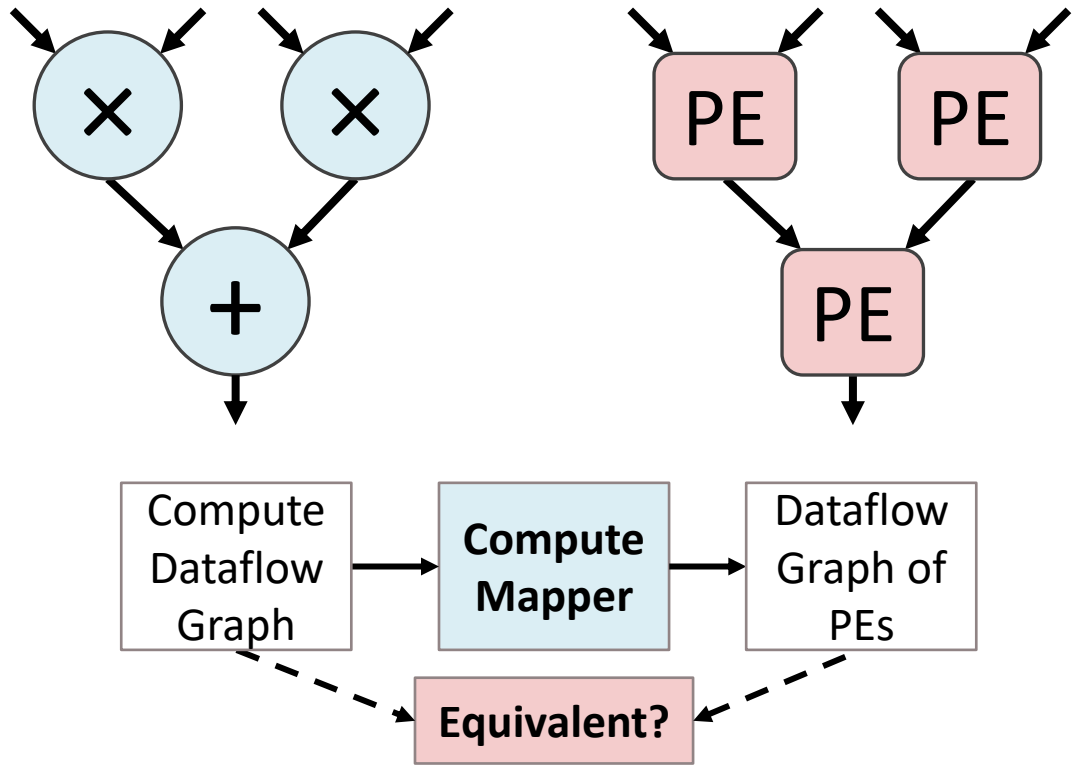
1.
```
def __call__(Data a, Data b):
    return a*b
```

2.
```
module mul(input[15:0] a,b, output [31:0] out);
    assign out = a*b;
endmodule
```

3.
```
bvmul IVAR_V_0@0 IVAR_V_1@0
```

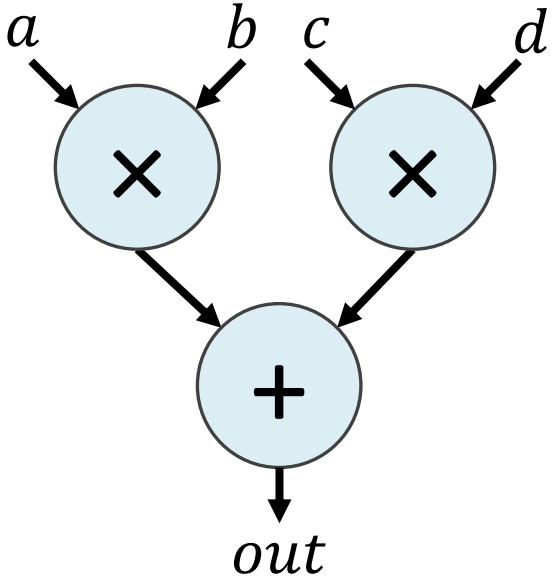
Compute Mapping Verification

- Problem:
  - Prove mapped application compute is equivalent to the pre-mapped application compute for all possible inputs
- Approach:
  - Construct formal representation of pre-mapped application compute and mapped application compute and prove equivalence using an SMT solver



Compute Mapping Verification

- We leverage the formal interpretation of PEak
  - Determine SMT representation of each node
  - Use structure of dataflow graph to compose full equation

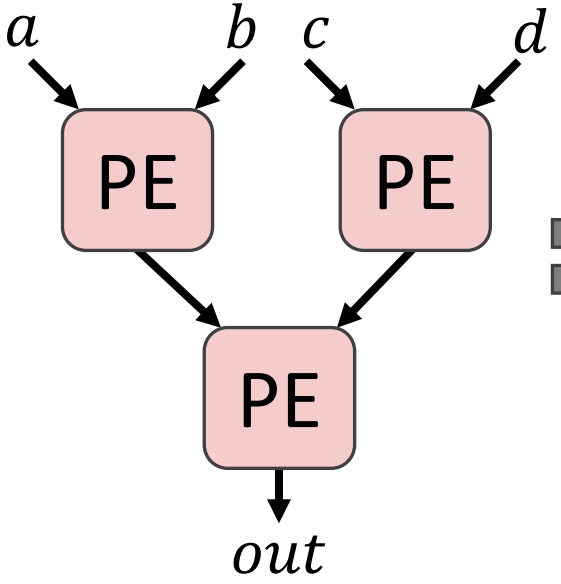


Application Dataflow Graph

$$out = (a \times b) + (c \times d)$$

Compute Mapping Verification

- We leverage the formal interpretation of PEak
  - Determine SMT representation of each node
  - Use structure of dataflow graph to compose full equation



Mapped Dataflow Graph

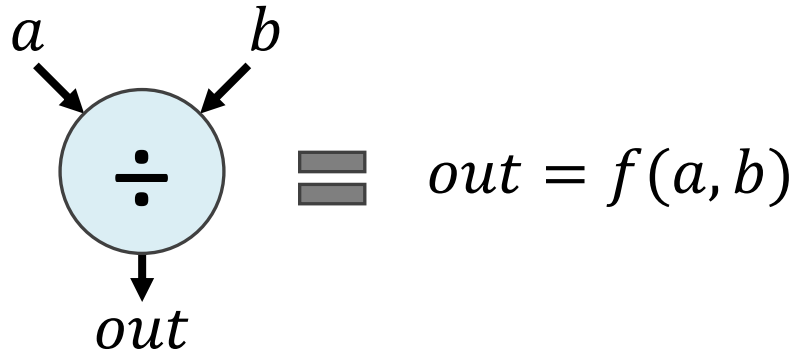
$$out = PE(add, PE(mul, a, b), PE(mul, c, d))$$

Compute Mapping Verification

- Compute equation:
  - $out_c = (a_c \times b_c) + (c_c \times d_c)$
  - $in_c = \{a_c, b_c, c_c, d_c\}$
- Mapped equation:
  - $out_m = PE(add, PE(mul, a_m, b_m), PE(mul, c_m, d_m))$
  - $in_m = \{a_m, b_m, c_m, d_m\}$
- To prove equivalence:
  - $\forall in_c, in_m : (in_c = in_m) \rightarrow (out_c = out_m)$

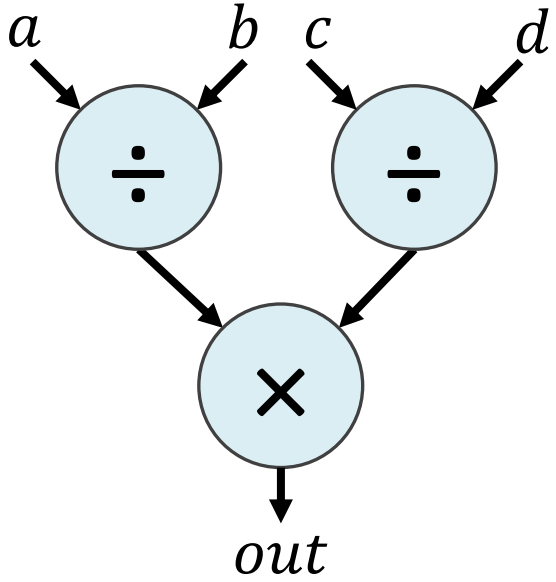
Compute Mapping Verification with Complex Ops

- Some complex operations don't have PEak SMT definitions
  - Bfloat16 operations are defined in encrypted Verilog files
- PEak will replace these operations with black boxes
  - Black boxes are uninterpreted functions; they have inputs and outputs but no specification for what the function does



Compute Mapping Verification with Complex Ops

- Reuse the same uninterpreted function for every occurrence of the operation



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

- This ensures each occurrence is functionally equivalent
- Also reuse uninterpreted functions for occurrences within mapped graph

Compute Mapping with Pipelining

- While the application graphs have no state, the mapped PEs do
  - We use compute pipelining techniques which turn on pipelining registers within the PEs and may add pipelining registers to the mapped graph

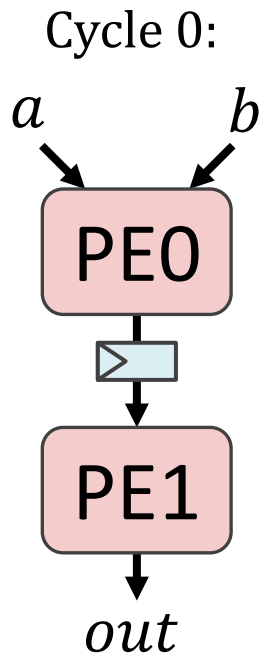
1. Create a functional transition system
2. Create state variables for every register in the mapped graph
3. Substitute each register output with the state variables
4. Set the next state value for each state variable
5. Unroll the functional transition system by the number of cycles it takes a piece of data to get from the input to the output of the mapped graph

Compute Mapping with Pipelining

- Unrolling a functional transition system:

$$reg\_in_0 = PE0(a, b)$$

$$out = PE1(reg\_out_0)$$

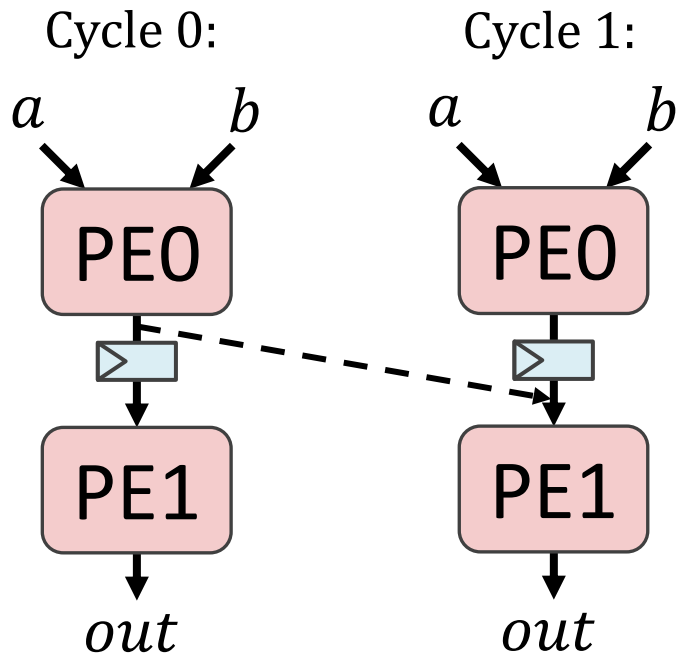


Compute Mapping with Pipelining

- Unrolling a functional transition system:

$reg\_in_0 = PE0(a, b)$

$out = PE1(reg\_out_0)$



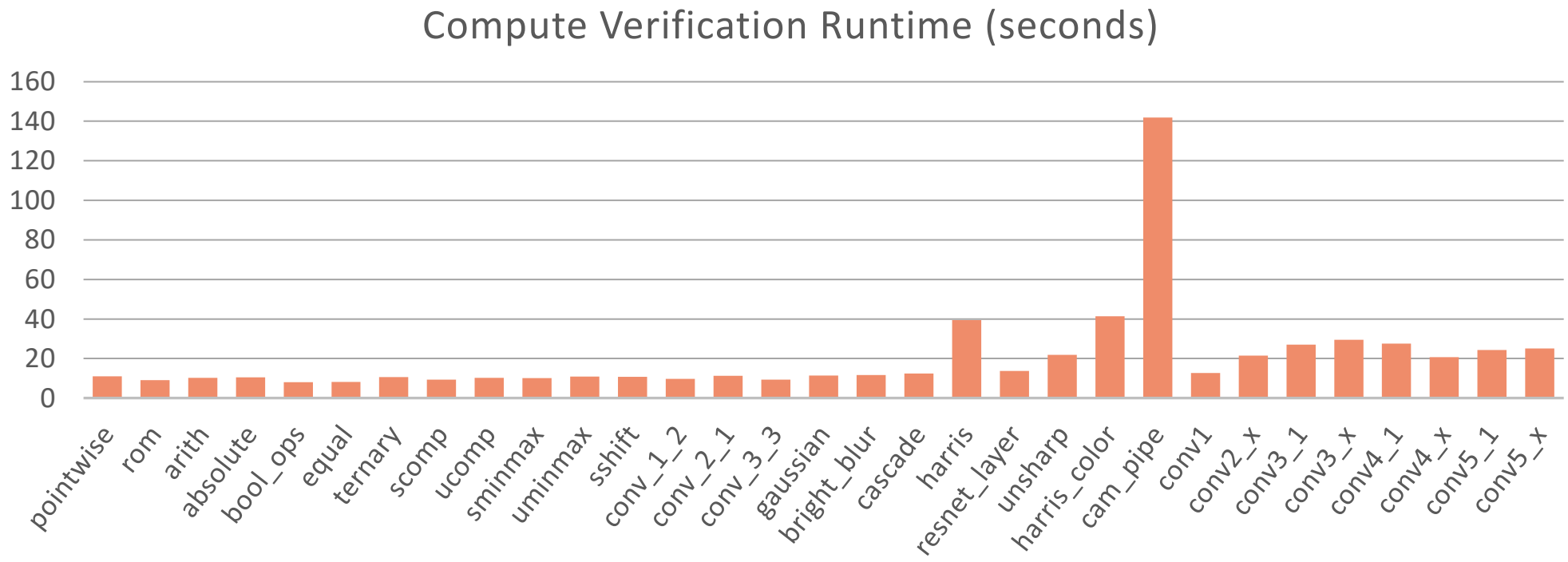
$reg\_out_1 = reg\_in_0$

$reg\_in_1 = PE0(a, b)$

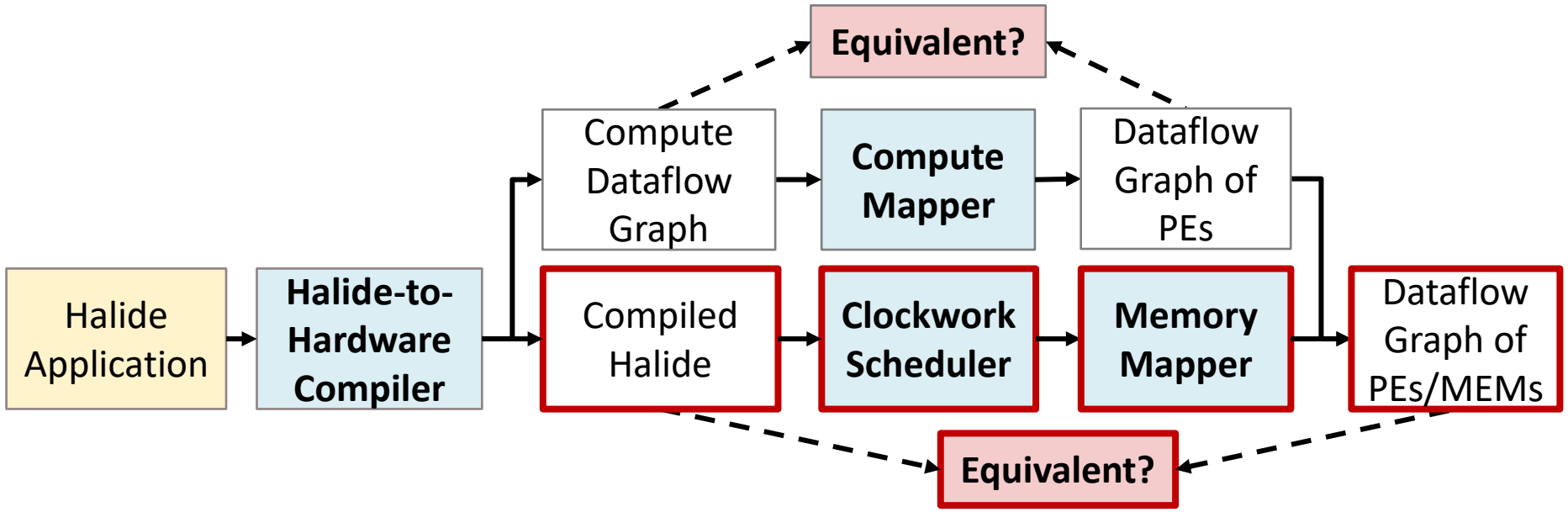
$out = PE1(reg\_out_1)$

Compute Mapping Results

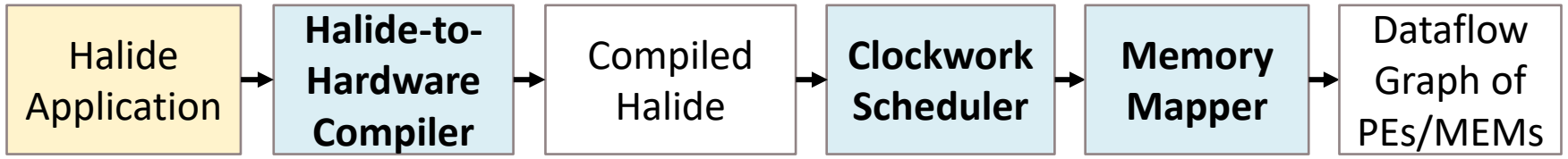
- Successfully verified all compute kernels from every application in our benchmark suite



Memory Mapping Verification

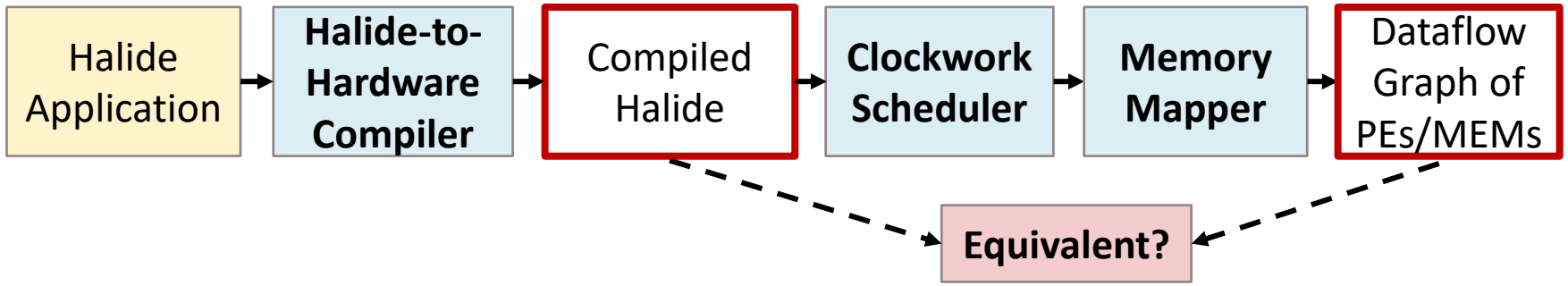


Scheduling and Memory Mapping



- Clockwork is a polyhedral scheduling tool that transforms the compiled Halide into memory accesses
- The memory mapper maps the memory access onto the global buffer, memory tiles, and register files

Memory Mapping Verification



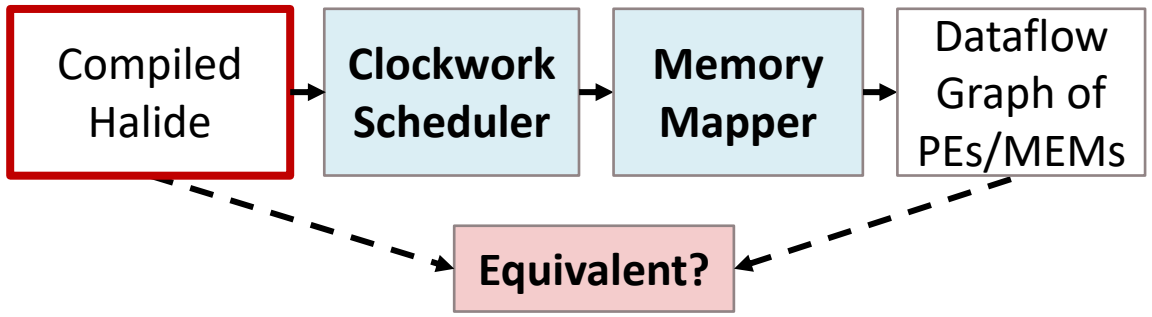
- Need to construct SMT representation of application before and after scheduling and memory mapping
  - Pre- and post-mapped compute can leverage PEak for formal definitions
  - Lake doesn’t have a SMT interpretation

Representing Halide Source in SMT

Basile Clément and Albert Cohen, “End-to-End Translation Validation for the Halide Language,” OOPSLA ’22

- Aims to verify that the Halide specification matches the low-level generated code
- Presents a formalization of the Halide specification and the generated imperative representation

Representing Halide Source in SMT



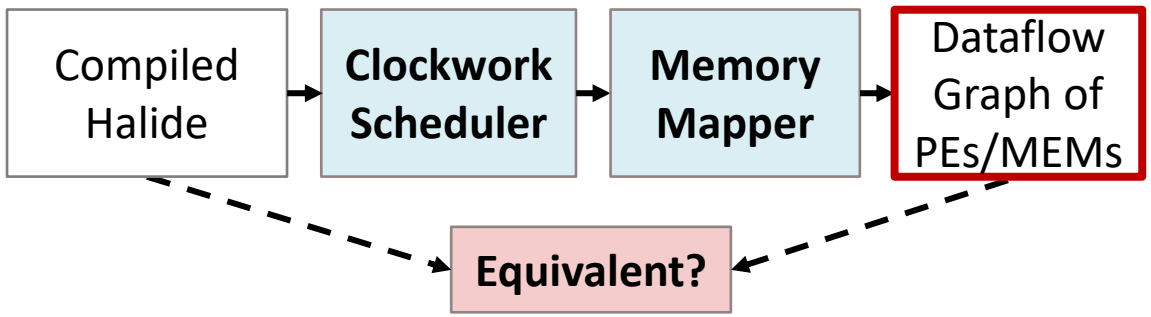
1. Produce the formalization of the compiled Halide from the Halide-to-Hardware compiler
2. Translate the LLVM IR to SMT
   - Existing tools (llvm2smt) can do this automatically

Representing Memories in SMT

N. Tsiskaridze et al., "Automating System Configuration," FMCAD ’21

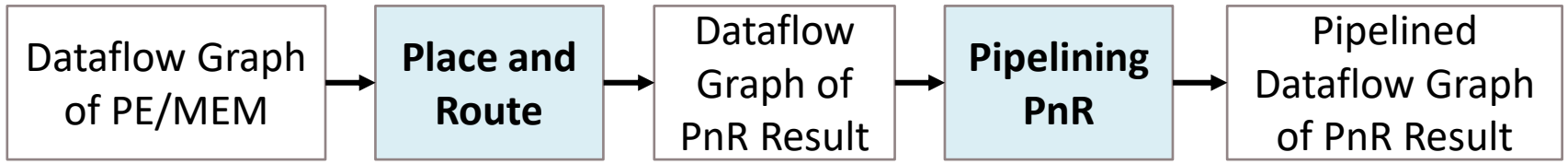
- Presents a framework for automated configuration of systems that can be represented as state machines
- Given a symbolic transition system S and input/output relationship P, find a configuration C such that S satisfies P
  - In the AHA flow, this was used to generate configurations for memory tile

Representing Memories in SMT



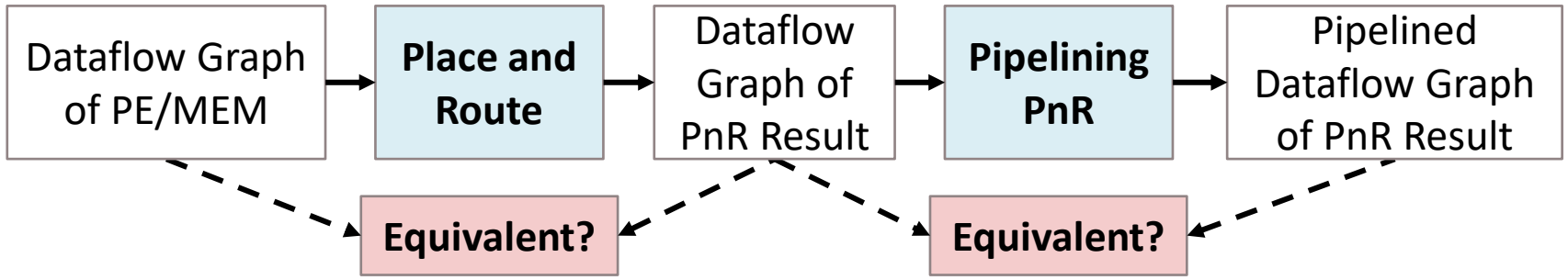
- The symbolic transition system was created from the memory tile hardware specification
  - Verilog to SMT was done using Yosys

Place and Route



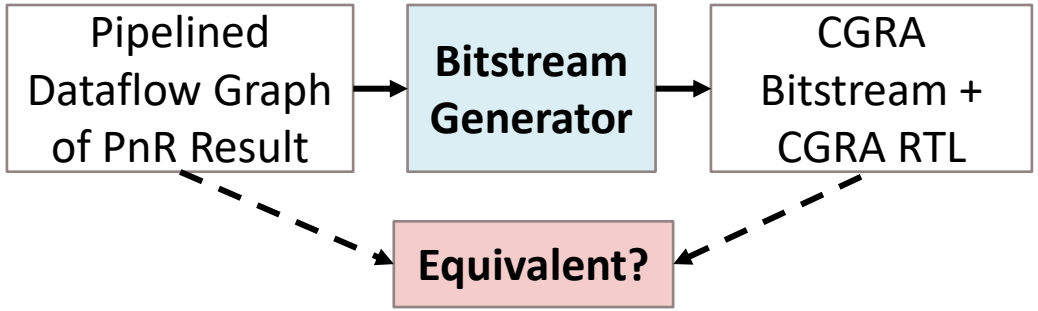
- Place and route takes the fully mapped application and places the tiles at physical locations on the CGRA
  - Routes on the interconnect are created for the connections in the mapped dataflow graph
- Pipelining the PnR result involves calculating the critical path of the application and inserting pipelining registers

Place and Route Verification



- Need to construct SMT formula for the PnR result
  - PnR dataflow graph has a limited set of node types, most of which just pass data from inputs to outputs
  - Complex nodes (like PEs and MEMs) already have SMT definitions
  - Can handwrite SMT representations of each of the remaining simple nodes

Configured RTL Verification



- Bitstream generation takes the pipelined PnR result and generates the configuration used to execute the application on the accelerator
- Reuse the SMT representation of pipelined PnR result
- Use Yosys to generate SMT representation of configured CGRA RTL

Next Steps

- Memory mapping verification:
  1. Representing Halide source in SMT
     - Leverage previous Halide translation validation techniques
     - Or use LLVM IR to SMT tools
  2. Representing memory tiles and fully mapped dataflow graph in SMT
     - Leverage Yosys RTL to SMT tools