Portland State UNIVERSITY

An Equivalence Checking Framework for Agile Hardware Design

Yanzhao Wang[1], Fei Xie[1],
Zhenkun Yang[2], Pasquale Cocchini[2], Jin Yang[2]
[1]Portland State University, Portland, OR, 97201,
[2]Intel Labs, Hillsboro, OR, 97124

Outline

- ***Motivation and Background***
- Equivalence Checking Framework
  - Naïve approach
  - Challenges
  - Proposed solutions
  - Optimized equivalence checking framework
  - Integration with HalideIR-based Agile Hardware Design Frameworks
- Evaluations
- Summary & Future Work

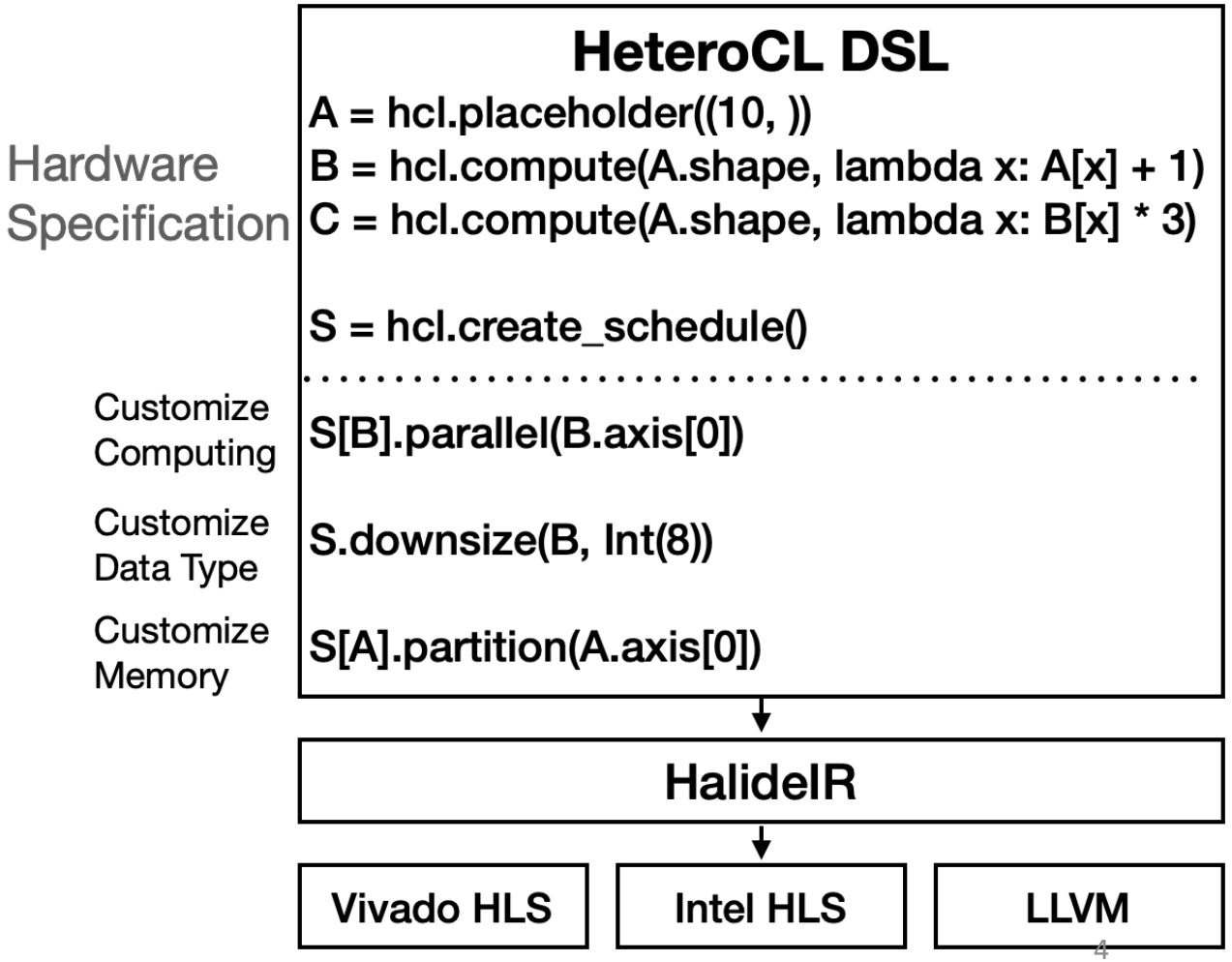
Agile Hardware Design

- ***Design agility***
  - Designers can experiment at a higher level of abstraction to explore design space optimizations

- ***Implementation agility***
  - Designers can generate various platform specific implementations of designs quickly

Agile Hardware Design with HalideIR

- **HalideIR** is a popular IR in image processing and deep-learning
- HalideIR enables agile design because it separates the specification of an algorithm from its execution schedule
- HeteroCL is an agile hardware design framework that utilizes HalideIR. We use it as our target

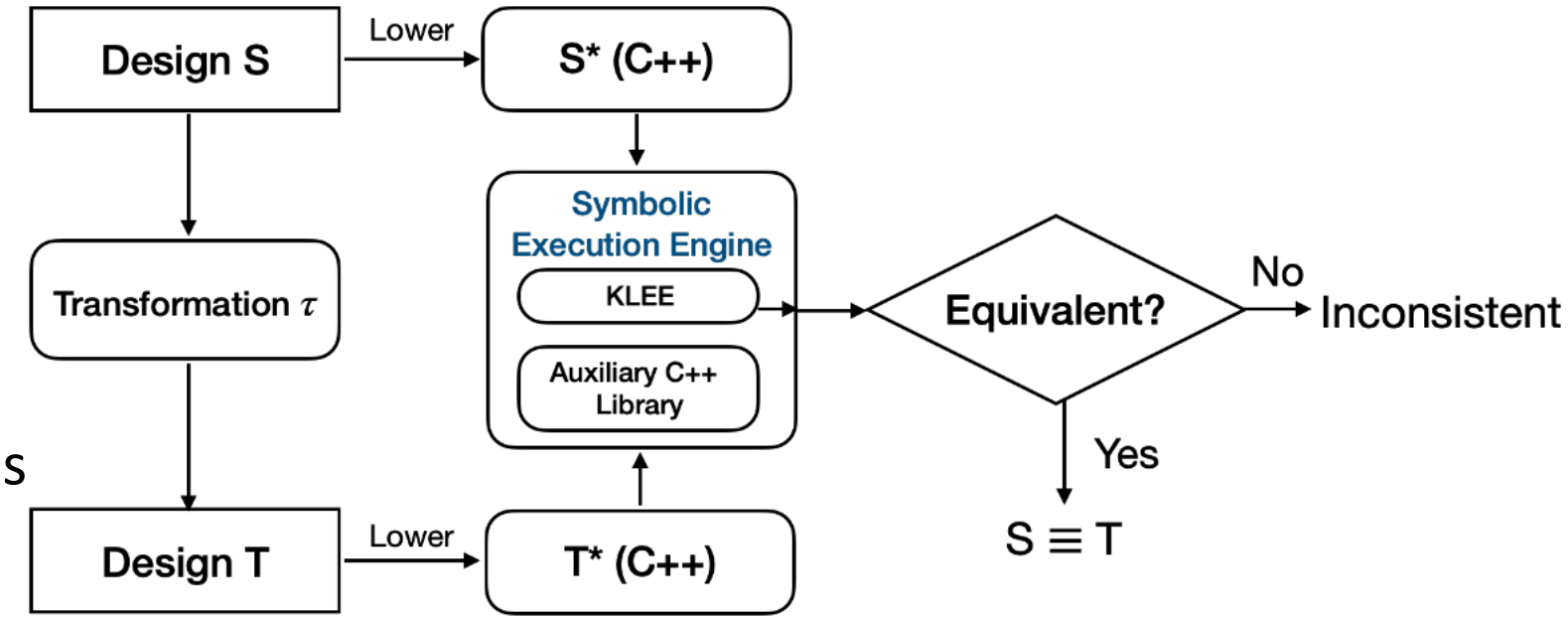


Outline

- Motivation and Background
- ***Equivalence Checking Framework***
  - Naïve approach
  - Challenges
  - Proposed solutions
  - Optimized equivalence checking framework
  - Integration with HalideIR-based Agile Hardware Design Frameworks
- Evaluations
- Summary & Future Work

Naïve Approach

- Given two designs S and T, we lower them into C++
- Then use KLEE to check if two C++ are equivalent.
- Otherwise, we identify the reasons for the divergence



Challenges

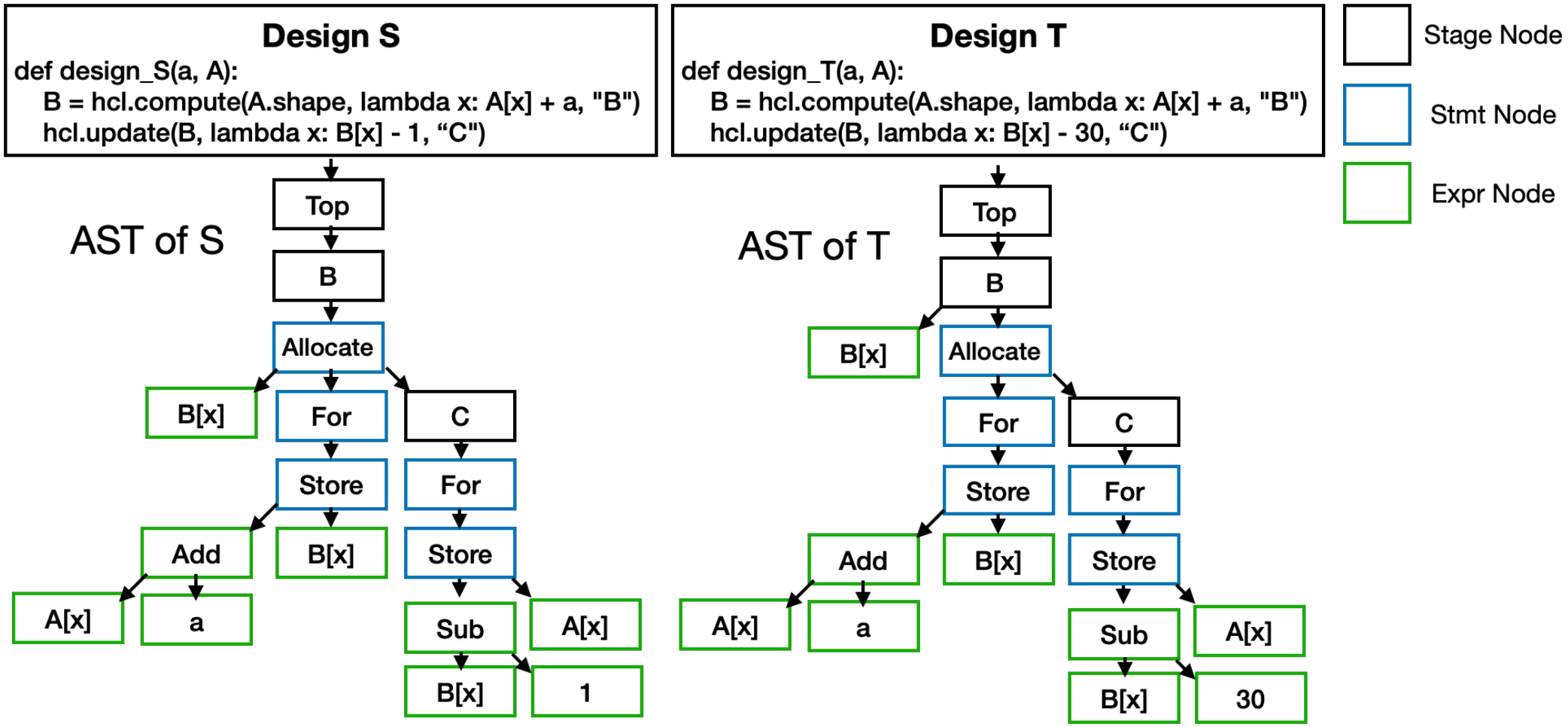
- ***Checking entire designs is not scalable***
  - A direct comparison between the entire states of two designs can easily lead to path explosion for non-trivial designs and does not scale to complex designs
  - When comparing entire design states, the points of divergence between the designs compared cannot be easily located, making debug very challenging

- ***Writing test harness requires major manual efforts***
  - Checking synthesizable C++ code by symbolic execution requires time-consuming and error-prone manual work in creating test harnesses that include symbolic inputs, outputs, and wrapper code

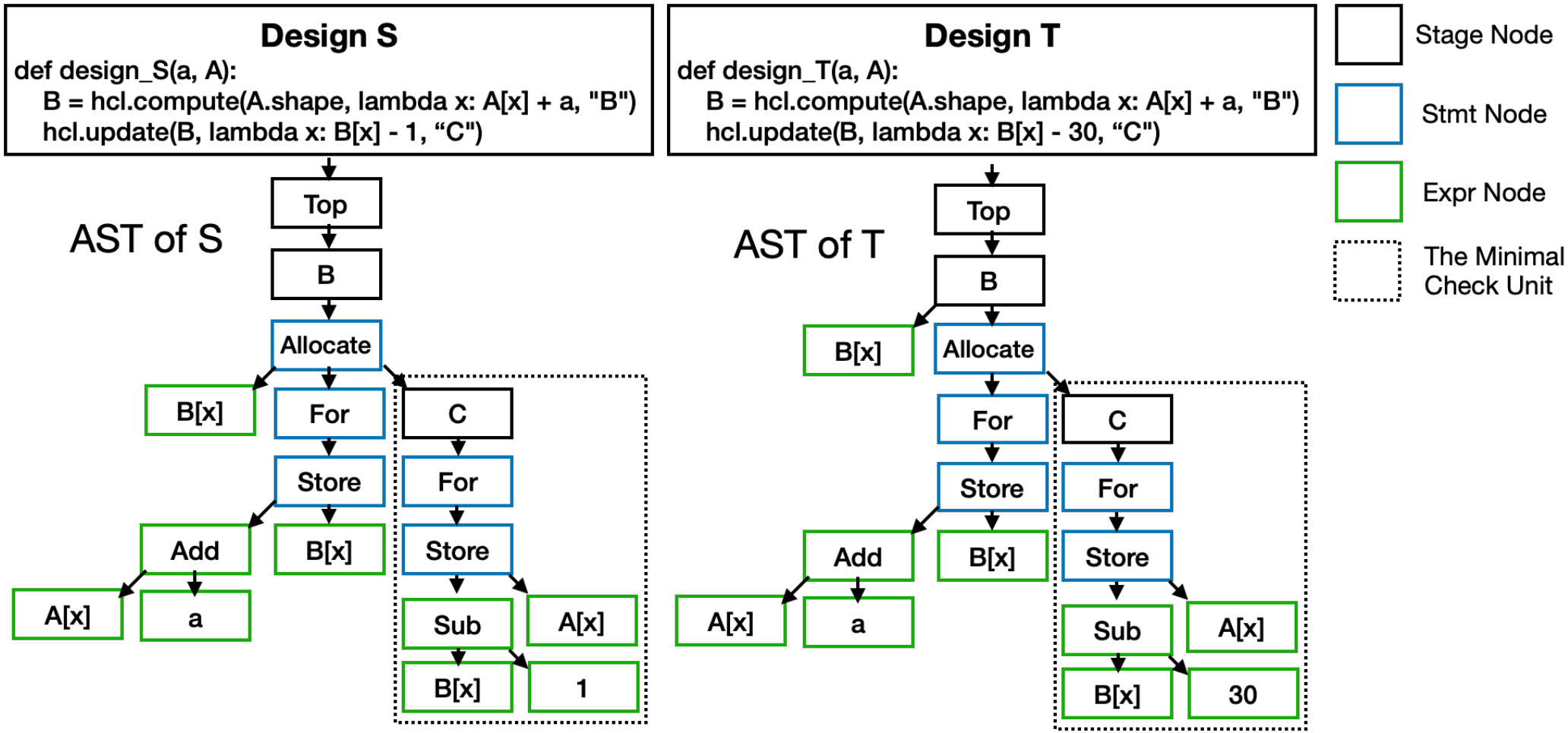
Proposed Solutions

- ***Identification of minimal check units***
  - Identifying HalideIR Stage as a minimal check unit
  - Using a stage as a check unit to locate the specific operations that cause divergences in design behaviors
- Automatic uninterpreted function optimization
  - For certified sub-stages of a stage, replace sub-stages with equivalent uninterpreted functions.
  - For each certified minimal check unit, use KLEE to check that their input variables are equivalent and remove all nodes in minimal check units
- Automatic test harness generation
  - Identifying input and output variables to minimal check units

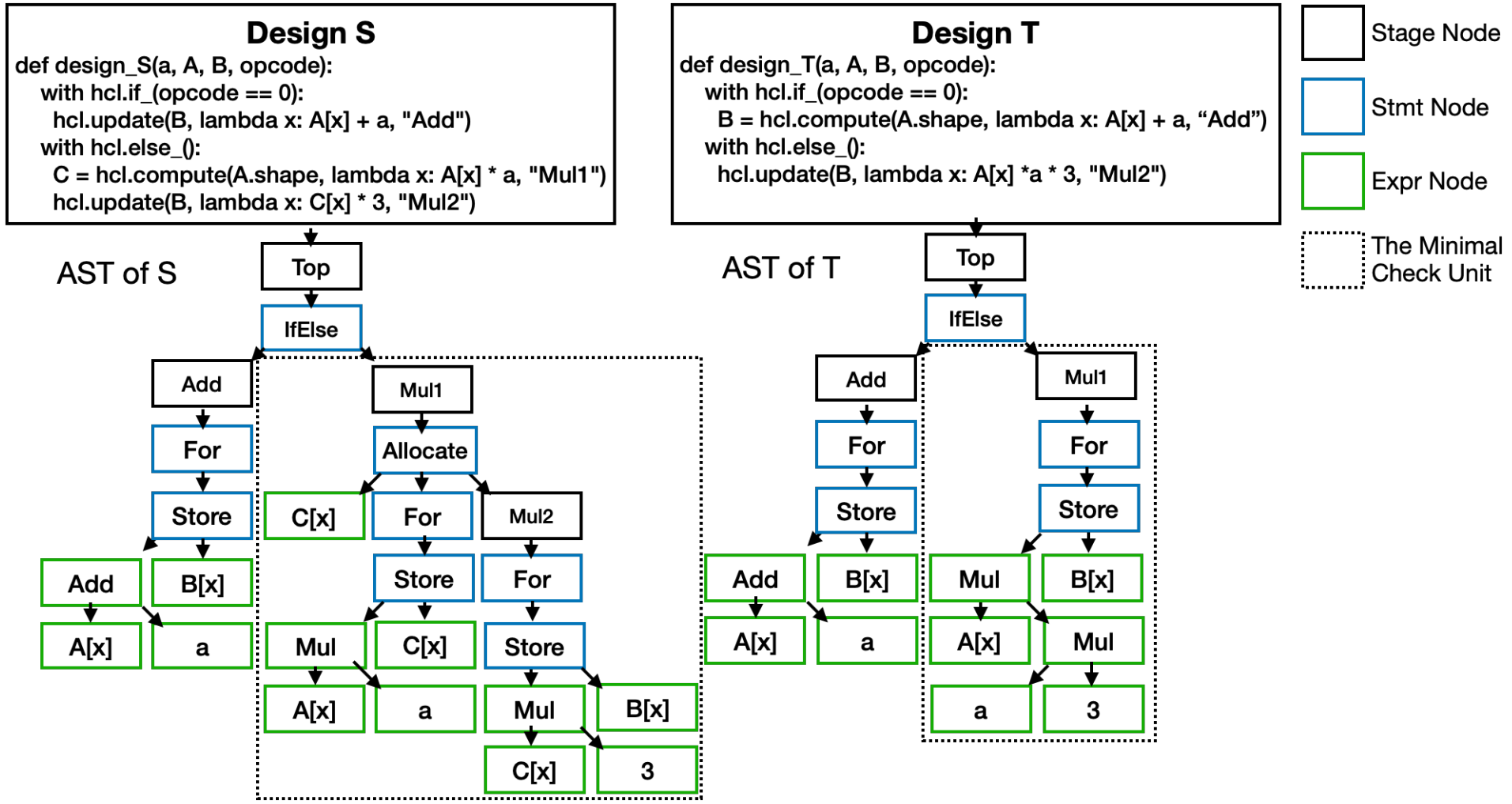
Example of HalideIR Stages



Identifying Minimal Check Units



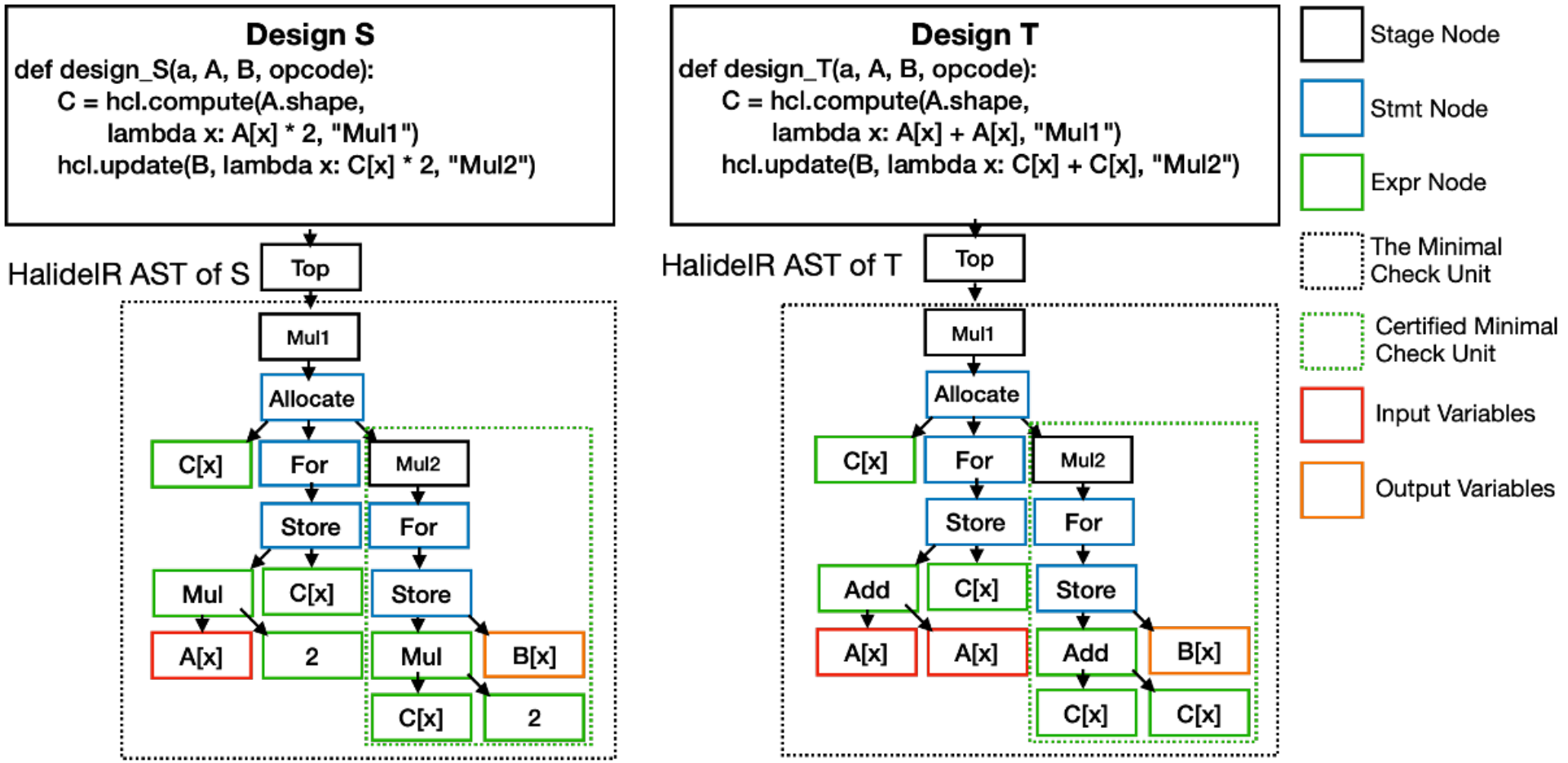
Designs with Significant IR Structure Differences



Proposed Solutions

- Identification of minimal check units
  - Identifying HalideIR Stage as a minimal check unit
  - Using a stage as a check unit to locate the specific operations that cause divergences in design behaviors
- ***Automatic uninterpreted function optimization***
  - For certified sub-stages of a stage, replace sub-stages with equivalent uninterpreted functions.
  - For each certified minimal check unit, use KLEE to check that their input variables are equivalent and remove all nodes in minimal check units
- Automatic test harness generation
  - Identifying input and output variables to minimal check units

Example of Replacing Certified Stages with Uninterpreted Functions

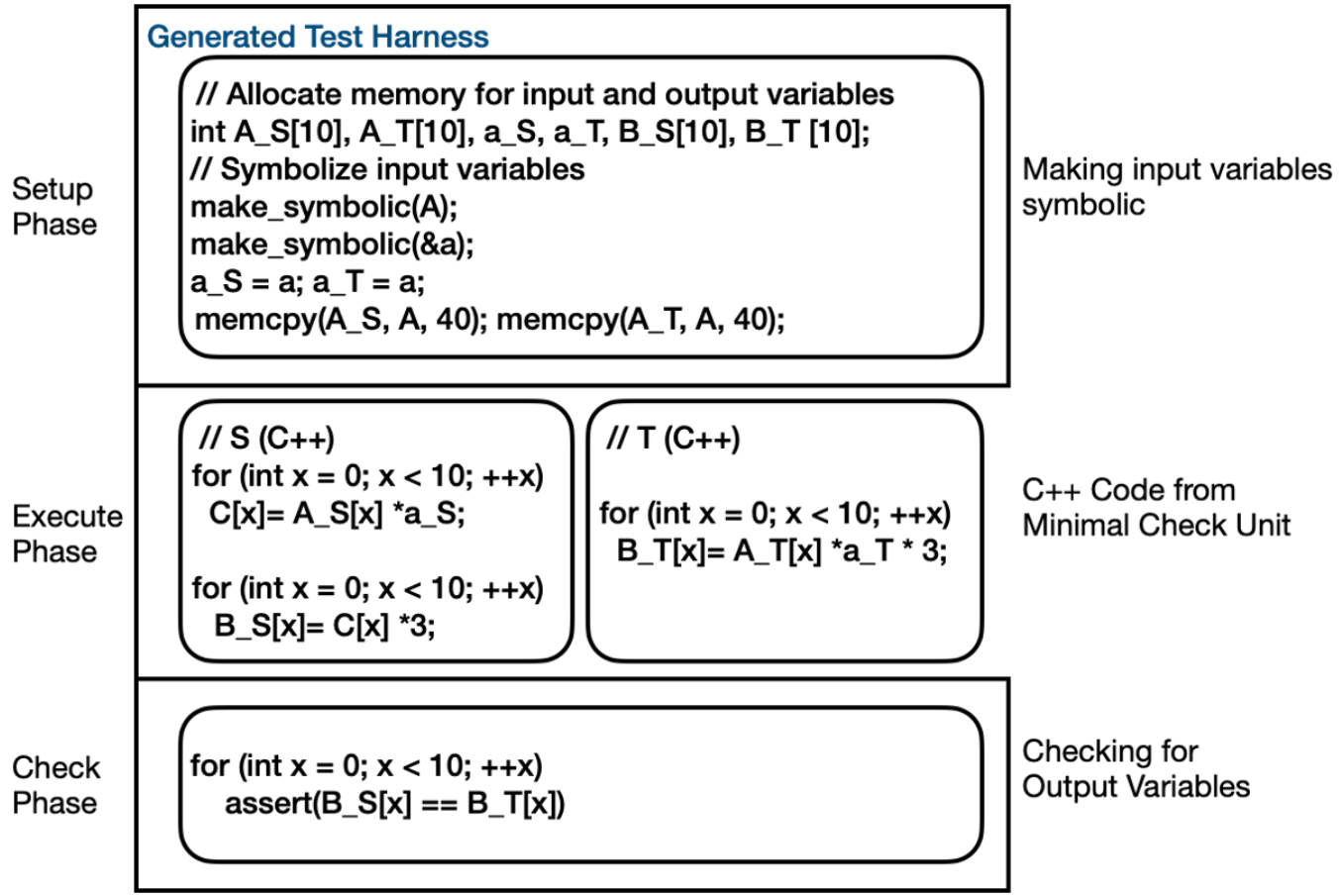


Proposed Solutions

- Identification of minimal check units
  - Identifying HalideIR Stage as a minimal check unit
  - Using a stage as a check unit to locate the specific operations that cause divergences in design behaviors
- Automatic uninterpreted function optimization
  - For certified sub-stages of a stage, replace sub-stages with equivalent uninterpreted functions.
  - For each certified minimal check unit, use KLEE to check that their input variables are equivalent and remove all nodes in minimal check units
- ***Automatic test harness generation***
  - Identifying input and output variables to minimal check units

Structure of Test Harness with Synthesized C++ Code of Minimal Check Units

- **Setup phase**
  - Making input variables symbolic
- **Execute phase**
  - C++ code from minimal check units
- **Check phase**
  - Equivalence checking for output variables

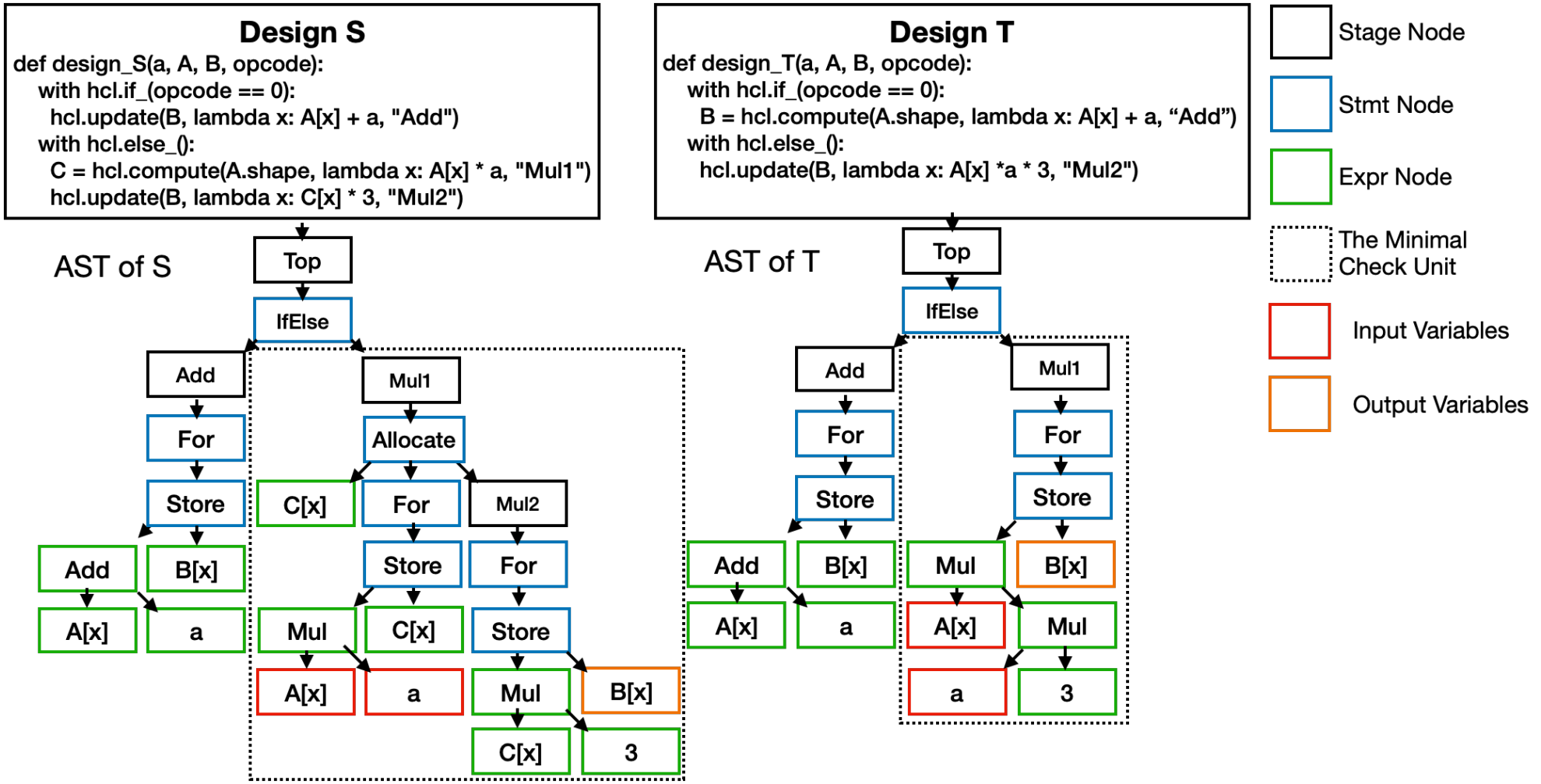


Identifying Input and Output Variables for Synthesized C++ Code

- **Identifying input variables**
  - Input variables are variables within the minimal check unit that are neither allocated nor written by ***Allocate*** or ***Store*** nodes

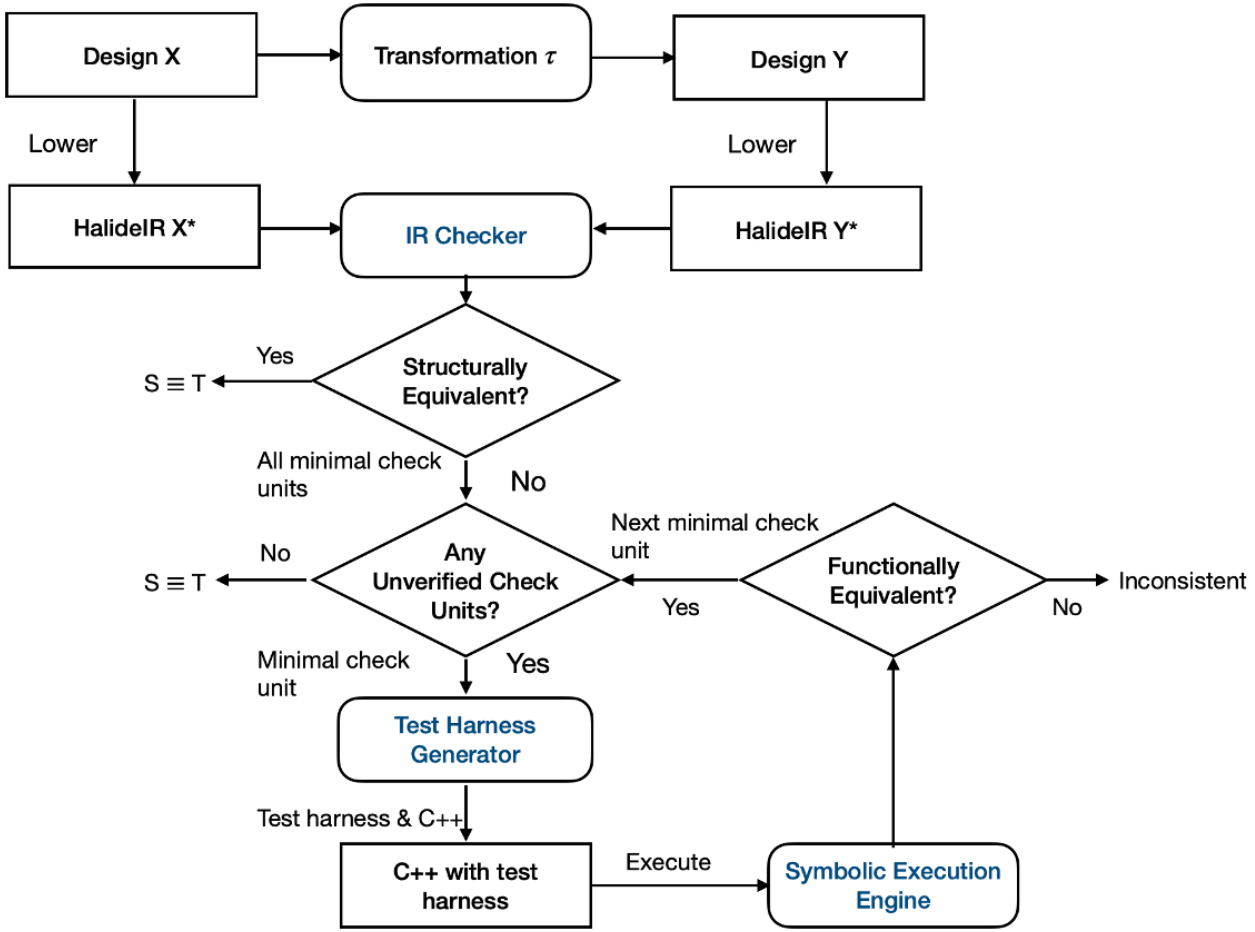
- **Identifying output variables**
  - Output variables are variables within the minimal check unit that are written by the unit's internal ***Store*** nodes, but not allocated by the ***Allocate*** nodes.

Example of Identifying Input and Output Variables

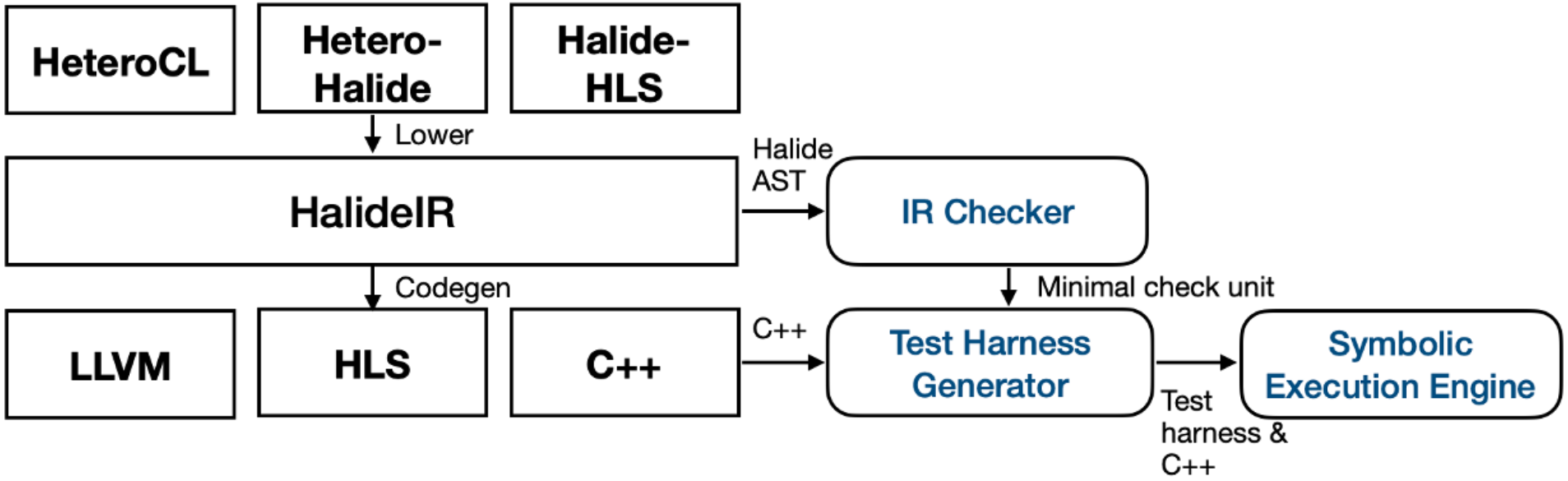


Optimized Equivalence Checking Framework

- Designs are first lowered to HalideIR
- IR checker determines if two IRs are structurally equivalent. If no, it produces minimal check units to the test harness generator
- Test harness generator wraps the code to an executable C++ program
- KLEE determines if two designs are behaviorally equivalent



Integration with HalideIR-based Agile Hardware Design Frameworks



Outline

- Motivation and Background
- Equivalence Checking Framework
  - Naïve approach
  - Challenges
  - Proposed solutions
  - Optimized equivalence checking framework
  - Integration with HalideIR-based Agile Hardware Design Frameworks
- ***Evaluations***
- Summary & Future Work

Evaluations Background

- We conduct verifications for two hardware designs from Intel that are implementations of an open-source deep-learning accelerator: VTA
- sVTA
  - A sequential model of of an open-source deep-learning accelerator: Versatile Tensor Accelerator (VTA)
- uVTA
  - A VTA model breaks down each of the 128-bit instructions into smaller micro-ops for potential parallelization.
- hVTA
  - A HeteroCL version of the VTA architecture strictly following its original structure

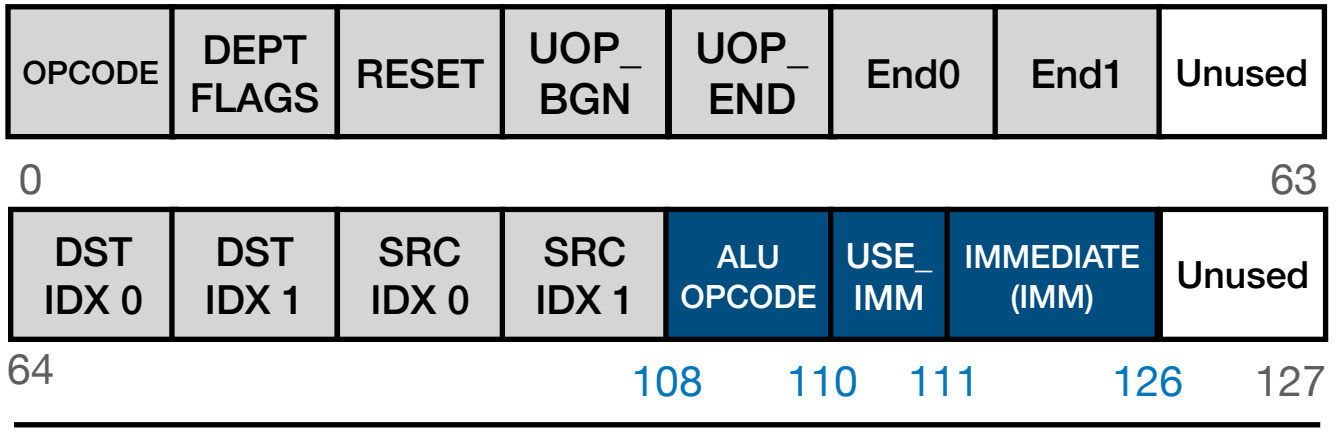
Evaluations

Design	LoC Python	LoC C++	Minimal Check Unit & Uninterpreted Function Optimization	Time (s)	Memory Consumption (MB)	# of Stages	# of Structural Inconsistencies	# of Behavioral Inconsistencies
sVTA-hVTA	296	560	No	Timeout	6781.73	No Data	No Data	No Data
sVTA-hVTA	296	560	Yes	65.39	128.37	211	8	2
uVTA-hVTA	195	1224	No	Timeout	7384.34	No Data	No Data	No Data
uVTA-hVTA	195	1224	Yes	1238.38	2384.98	301	84	3

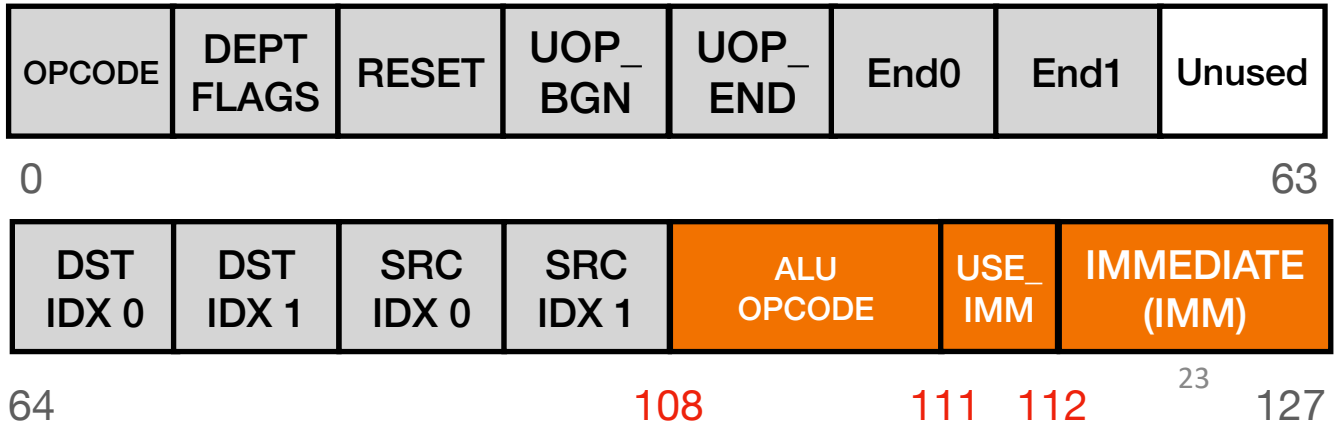
sVTA Inconsistency with hVTA in ALU Module

```
ALU_OPCODE = hcl.scalar(instr[111:108],
name="ALU_OPCODE") # extend to 3 bits

USE_IMM = hcl.scalar(instr[112:111],
name="USE_IMM",dtype=hcl.UInt(1))

IMM = hcl.scalar(instr[128:112], name="IMM")
src = hcl.select(USE_IMM.v == 1,
hcl.cast(hcl.Int(16), IMM),
      hcl.cast(hcl.Int(32), src_tensor[x][y]))

dst = hcl.cast(hcl.Int(32), dst_tensor[x][y])

with hcl.if_(ALU_OPCODE.v == VTA_ALU_OPCODE_MIN):
  dst_tensor[x][y] = hcl.select(dst <= src,
  dst_tensor[x][y], src)
```

VTA ALU Instruction



sVTA ALU Instruction



uVTA-hVTA Inconsistency in Load Module

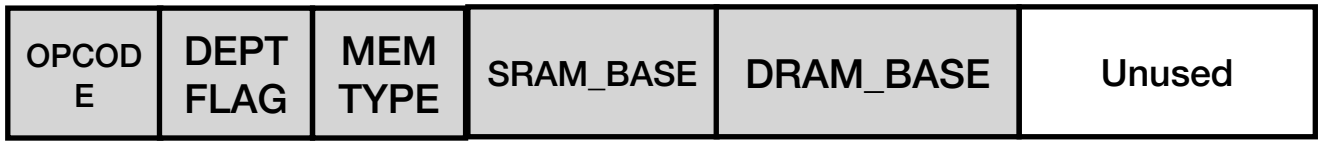
```
is_min_pad_value = hcl.scalar(instr[58:57],
name="is_pad_min_value")

pad_val = hcl.select(is_min_pad_value.v == 1,\
      hcl.cast(hcl.Int(16), 1 << (sram_bits - 1)), 0)

sram_idx = sram_base + x_tot * y + x

def clear(row, col):
    sram[sram_idx][row][col] = pad_val

hcl.mutate((nrows, ncols),clear,name='pad_clear')
```

uVTA Code

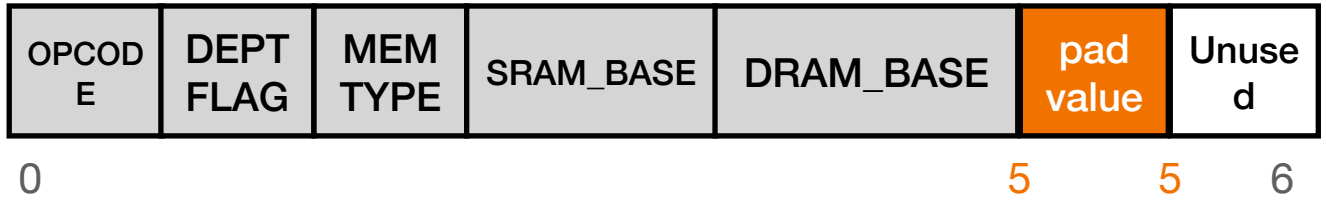
```
for (int i = 0; i < range; i++)
  for (int j = 0; j < MAT_AXI_RATIO; j++)
    mem[sram_idx++][j] = 0;
```

hVTA Code

VTA Store/Load Instruction



uVTA Store/Load Instruction



Outline

- Motivation and Background
- Equivalence Checking Framework
  - Naïve approach
  - Challenges
  - Proposed solutions
  - Optimized equivalence checking framework
  - Integration with HalideIR-based Agile Hardware Design Frameworks
- Evaluations
- ***Summary & Future Work***

Summary & Future Work

- Present a scalable equivalence checking framework for HalideIR
- Demonstrate the framework's effectiveness by performing equivalence checking on two practical deep-learning accelerator designs, sVTA, and uVTA
- Further optimize our framework for minimal check units with significant structural differences that may still post symbolic analysis challenges