An Automated Verification Framework for HalideIR-Based Compiler Transformations

Yanzhao Wang and Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97201, USA
{wyanzhao, xie}@pdx.edu

Zhenkun Yang, Jeremy Casas, Pasquale Cocchini and Jin Yang
Strategic CAD Labs
Intel Corporation
Hillsboro, OR 97124, USA
{zhenkun.yang, jeremy.casas, pasquale.cocchini, jin.yang}@intel.com

***Abstract*—HalideIR is a popular intermediate representation for compilers in domains such as deep learning, image processing, and hardware design. In this paper, we present an automated verification framework for HalideIR-based compiler transformations. The framework conducts verification using symbolic execution in two steps. Given a compiler transformation, our automated verification framework first uses symbolic execution to enumerate the compiler transformation's paths, and then utilizes symbolic execution to verify if the output program for each transformation path is equivalent to its source. We have successfully applied this framework to verify 46 transformations from the three most-starred HalideIR-based compilers on GitHub and detected 4 transformation bugs undetected by manually crafted unit tests.**

I. Introduction

HalideIR [17] is a popular intermediate representation (IR) widely used in deep-learning [5], image processing [3], and hardware designs [2], [7], [10], [16]. HalideIR separates the specification of an algorithm from its execution schedule and provides various transformations, such as unrolling loops, parallelizing, loop nesting, and vector operations, for optimizing execution schedules. This feature allows designers to efficiently experiment with various optimizing transformations for an algorithm because they can change an algorithm's execution schedule without modifying the algorithm itself. Therefore, HalideIR enables fast algorithm optimization iterations. Besides built-in transformations, e.g., unrolling, and loop nesting, HalideIR also provides interfaces for developers to implement their own transformations, further extending HalideIR's scalability.

Despite intensive testing, bugs in HalideIR-based compiler transformations do happen and can cause incorrect target code generated from correct source programs. It is impractical to fully prove the correctness of HalideIR-based compilers due to their large code bases. A weaker formal technique - the translation validation approach, introduced by [15], has been widely used in compiler verification [11], [13], [13], [20], [20]. This approach checks each compilation result against the source program and guarantees the transformation's correctness. However, it is insufficient for proving the correctness of a transformation because it does not verify all applications of the transformation.

In this paper, we introduce an automated verification framework for HalideIR-based compiler transformations. This framework features a novel two-step symbolic execution approach: it first uses symbolic execution to enumerate all execution paths for the given transformation and then utilizes symbolic execution conduct translation verification on each symbolic path explored. If our framework can certify the equivalence between all outputs of a given transformation and its corresponding symbolic input, it can certify the correctness of the transformation.

We have applied this framework to the three most-starred HalideIR-based compilers from GitHub: HeteroCL [7], Halide-HLS [16], and Stanford AHA's Halide-To-Hardware [2]. In total, we have verified 46 transformations from those compilers and detected four transformation bugs during verification, all of which were undetected by the unit tests accompanying these compilers.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the methodology and architecture of our verification framework. Section 4 and 5 provides implementation details for this architecture. Section 6 presents experimental results and discusses the limitations of our approach. Section 7 summarizes our verification approach and test results.

II. Related Work

Formal compiler certification and translation validation are the two mainstream formal approaches for verifying compiler transformations.

Formal Compiler Certification: This approach formally verifies if every transformation of a compiler preserves the semantics of the input program, e.g., CompCert [9] is a compiler for C that is formalized and verified in Coq. However, compiler certification is a highly complex and labor-intensive process, and every compiler revision requires reproofing. These drawbacks hinder future compiler improvements. Newcomb et al. [14] present an automatic verification tool for soundness and termination of HalideIR's rewriting system. This tool only supports checking the Halide $Simplify$ transformation. In addition, this tool only checks the correctness of HalideIR rewriting, and it cannot detect runtime errors introduced by illegal code generation. In comparison, our approach supports more types of transformations. Furthermore, since our approach verifies runtime code generated from HalideIR, it can detect code bugs and runtime errors such as overflows.

Translation Validation: The translation validation approach was first introduced by [15]; it introduces a weaker formal tech-

978-3-9819263-7-8/DATE23/© 2023 EDAA

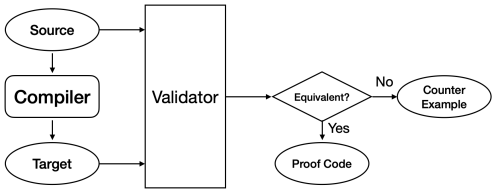

Fig. 1. Workflow of Translation Validation Approach

nique that effectively certifies the conformity of a compiler's individual executions to the compiler's specification. This approach compares a compiler's input and output for its specific application. Figure 1 depicts the workflow for the translation validation approach, where both the source and target programs are sent to a validator. If the validator finds the generated target program correctly implements the source program, it produces proof; otherwise, it produces a counterexample. Alive2 [11] implements a translation validation tool for LLVM, which verifies application instances of a compiler transformation, but cannot fully verify the transformation. In comparison, our approach can fully verify and certify compilers' transformation because it synthesizes symbolic execution with translation validation. Our framework first employs symbolic execution to enumerate all possible execution paths for a HalideIR-based compiler transformation, and then uses translation validation to certify the equivalence of the input and the outputs generated by those paths.

III. Proposed Methodology

Symbolic execution [6] is a powerful technique for exploring program execution paths; however, it may face challenges such as state-space explosions. Modern compilers, e.g., GCC [1], and LLVM [8], usually have unbounded loops and complicated structures in their transformations, rendering symbolic execution ineffective in exploring compiler transformation paths.

We observed that HalideIR provides a mechanism that can potentially reduce complexities for symbolic execution of a compiler transformation: Each transformation algorithm was further divided into mutators for different IR nodes. HalideIR uses pattern matching to apply the proper mutator to each node to optimize codes, while GCC and LLVM need unbounded loops to iterate through different IR nodes. The pattern matching mechanism adopted by HalideIR renders unbounded loops inside a transformation unnecessary, making it efficient and effective to apply symbolic execution to HalideIR-based compiler transformations. Figure 2 shows an example of a HalideIR transformation: the Simplify transformation that reduces a complex IR into a simpler form. This complex transformation is composed of mutators for different IR nodes such as $Cast$, $Ramp$, and Add. The input for the mutator in Figure 2 is an $Ramp$ node. Depending on different conditions, there are three possible output nodes: $Broadcast$, $Self$, or $Ramp$. For this example, if we can check the equivalence of all three output nodes ($Broadcast$, $Self$, and $Ramp$) with the input node $Ramp$, we can verify the correctness of this mutator. If all mutators pass verification, then the transformation is correct.

Figure 3 illustrates our automated transformation verification framework for HalideIR-based compiler transformations. Our framework has two parts: the symbolic enumeration of transformation paths and the translation validation. First, our framework collects all mutators from a HalideIR transformation. Then, to overcome the drawback that the translation validation approach cannot fully verify compiler transformations, our framework makes each mutator's input node fully symbolic and use our symbolic execution engine to explore all execution paths of a mutator under test. The framework collects all symbolically transformed IR nodes generated by the mutator, then lowers the source IR node and all symbolically transformed IR nodes to LLVM bitcode, executable by KLEE [4]. If KLEE asserts all the target bitcode is equivalent to the source bitcode, the mutator is certified; otherwise, our framework reports a counterexample. If all mutators of a transformation are certified, our framework certifies the HalideIR transformation. Our approach makes a HalideIR node fully symbolic and guarantees full coverage when exploring the transformation paths of the mutators.

IV. Enumeration of Transformation Paths

In this section, we discuss how to symbolically enumerate the HalideIR transformation paths. Figure 4 shows an overview of HalideIR syntax. There are two HalideIR node types: $Expr$ and $Stmt$. $Expr$ represents certain value and has a data type, such as Max, Add, and Div. $Stmt$ is a program statement such as $IfThenElse$. Because HalideIR entities are implemented using C++ class, e.g., all $Stmt$ nodes such as $AssertStmt$ and $IfThenElse$ inherit from the base class $Stmt$. This brings challenges for existing symbolic execution tools like KLEE since these tools do not readily support C++ classes. Therefore, we implement a HalideIR object support library to encode HalideIR C++ objects symbolically.

```
1  class Broadcast {
2    Expr value;
3    int lanes;
4  }
5
6  void visit(const Broadcast *op, const Expr &self) {
7     Expr value = mutate(op->value);
8     if (value.as<Max> && op->lanes > 1)
9        expr = value;
10    else if(value.as<Min>)
11       expr = Min::make(value, op->lanes);
12    else
13       expr = Add::make(value, op->lanes);
14 }
```

The code above shows the definition for $Broadcast$ node, and its sample corresponding mutator. In the mutator, it first checks if the $value$ is a Max node, and $lanes$ is larger than 1 (Line 8). If yes, it transforms $Broadcast$ node to the $value$ node. If the $value$ is a Min node, it transforms the input $Broadcast$ node to a new Min node. Otherwise, it transforms $Broadcast$ node to an Add node.

To verify this mutator, we need to encode all members of the $Broadcast$ node object symbolically: $value$, and $lanes$. The $lanes$ is a C-built-in integer, and $value$ has the $Expr$ type here, which can be any $Expr$ node types in Figure 4. For $lanes$, making it symbolic is trivial and we can directly call $klee\_make\_symbolic$. To make $value$ fully symbolic to verify transformations, intuitively, we can use KLEE to try

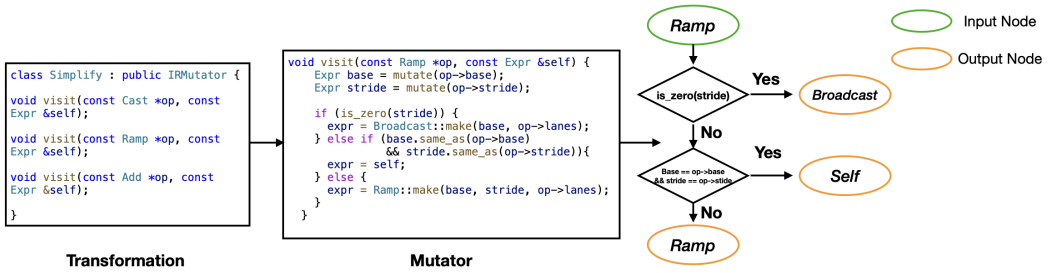

Fig. 2. An example of HalideIR transformation.

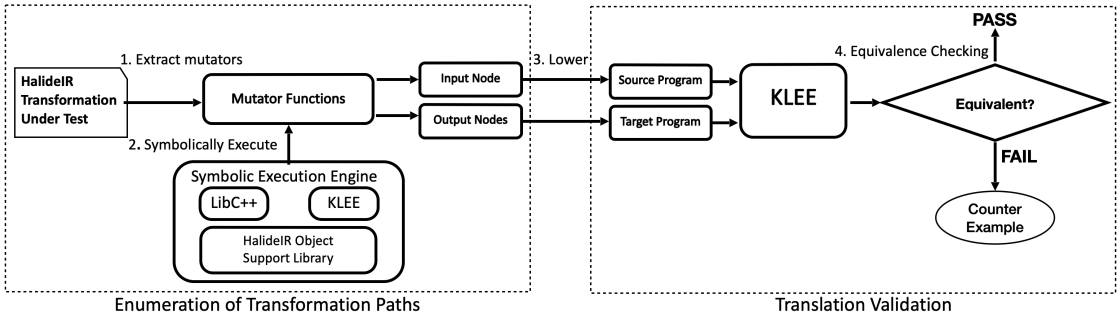

Fig. 3. Architecture of Automated Verification Framework for HalideIR-Based Compiler Transformations

```
Program ::= Body
Body ::= Stmt_list
Stmt_list ::= Stmt | Stmt_list
```

```
Stmt ::= Expr |
IfThenElse (Expr, Stmt, Stmt) |     // If Then Else
AssertStmt (Expr, Expr, Stmt) |     // Assertion
Store (Var, Expr, Expr, Expr)  |    // Store Data to Memory
...
```

```
IntImm ::=  i ∈ Z
UIntImm ::= I ∈ N
Var ::= x
Expr ::= IntImm | UIntImm  | + | - | x |  div | mod | ∨ | ∧ | ¬ | < | > | = | select | min | max | ...
```

Fig. 4. Example of HalideIR syntax

every possible $Expr$ node types, but this leads to a state-space explosion. Therefore, it is necessary to limit types for $value$ to the ones the mutator uses. For example, a HalideIR might have more than 37 types of node, but the mutator only uses two Max, and Min. Accordingly, KLEE only needs to try the two node types.

Algorithm 1: hooked-as

Input: The type to test: T
Output: The instance of the input type: I

```
1  I ← Nil
2  pid ← fork-state()
3  if pid == 0 then
       // Call type T's constructor
4      I ← new-node(T)
5      symbolize-c-built-in-member(I)
6      return I
7  end
8  return Nil
```

To prune types of $Expr$ members for the node under test, we implement two algorithms in our framework: the **hooked-as** function in Algorithm 1 and Algorithm 2.

Algorithm 2: Prune types for node under test that has $Expr$ members

Input: The node under test: σ
Input: The mutator program under test: $Mutator$
Output: Pruned types of node for expr_members of σ: $SavedTypes$

```
1  SavedTypes ← ∅
2  symbolize-c-built-in-member(σ)
   // Execute the mutator symbolically
3  for i ← Mutator do
4      if as(type, expr_member) called then
5          I ← hooked-as(type)
6          if I then
7              SavedTypes ← ⟨type, expr_member⟩
8          end
9      end
10 end
11 foreach ε ∈ expr_members of σ do
12     SavedTypes ← ⟨OtherExpr, ε⟩
13 end
14 return SavedTypes
```

We first implement the **hooked-as** function to explore every type of $Expr$ that the mutator uses. As Algorithm 1 shows, this function takes the to-be-checked type from the original **as** function, T, as the input. The output, I, is an instance of the input type T.

This algorithm first forks the state of the program (Line 2). Subsequently, the algorithm calls the type T's constructor to construct a new instance in the child-process, where it symbolizes all its C-built-in members and returns this new instance (Line 3-6). Next, the algorithm returns the Nil in the parent process.

We then implement Algorithm 2 to prune types for node under test. The inputs for this algorithm are both the mutator program under test and the mutator's node σ. First, this algorithm symbolizes all C-built-in members of σ and executes the mutator symbolically. If the **as** function is called, the algorithm calls the **hooked-as** function. If the **hooked-as** returns a new instance, the Algorithm 2 saves the type and the $expr\_member$ of σ that calls the as function (Line 5-8). Otherwise, it continues to explore the as function's failed path. Because our framework has already forked the state in the **hooked-as** function, it can explore both paths.

In the end, for each $expr\_member$ of σ, Algorithm 2 adds the $OtherExpr$ node to its $SavedTypes$ (Line 12). $OtherExpr$ is a placeholder $Expr$ type that we added to HalideIR to cover the other case in the mutator, such as Line 12-13 in the $Broadcast$ mutator example. Using this algorithm, for the node under test with $Expr$ members, our framework can efficiently collect only the $Expr$ types that the mutator uses.

Take the $Broadcast$ mutator as an example. Our framework first symbolizes the $lanes$ of the $Broadcast$ node. When the first **as** is called (Line 8), Algorithm 2 saves the Max type for $value$ to $SavedTypes$. Then, this algorithm continues exploring the second **as** function and repeats this process. When the program ends, the algorithm saves $OtherExpr$. Consequently, the pruned types for the $value$ of $Broadcast$ node are Max, Min, and $OtherExpr$.

After pruning types of $Expr$ members for the node under test, the last step to fully explore the mutator paths is constructing the harness. This step is trivial: our framework directly symbolizes C-built-in members. And for $Expr$ members, our framework collects the pruned types, and then calls each type's constructor. For the newly constructed instances of the constructors, our framework recursively implements the same symbolization and construction process. It implements a similar process to $Stmt$ nodes.

V. Translation Validation for Mutator under Test

In translation validation, we contribute an equivalence checking algorithm using KLEE to verify HalideIR nodes. Halide provides the capability to generate C code directly. Therefore, unlike other translation validation approaches [11], [13], [19], [20] which need to encode compiler IR semantics in SMT and verify the SMT instances using a SMT solver, such as Z3 [12], we replace this encoding process by reusing Halide's existing C backend because KLEE already provides support for verifying C programs. In our translation validation stage, our framework lowers the input node of the mutator and the generated target nodes into C programs and compile them into LLVM bitcode, and then it checks their equivalence using KLEE.

As discussed in section 4, we need to check the equivalence for two types of Halide nodes: $Expr$ and $Stmt$. Consider a given mutator function τ, an input $Expr$ node E_I, and a set of target output nodes generated from the input: E_O. A set of C expressions P_I and P_O are generated from E_I and E_O using C backend. Then we use symbolic execution to assert their equivalence. The mutator τ is correct **iff**:

$\forall\langle \mathrm{E}_I, E_O\rangle \Rightarrow$ **sym-exe(assert(**P_I, P_O**))** $==$ **true**

In other words, since $Expr$ in HalideIR represents a value of a certain type, we can establish the correctness for the mutator τ when the expressions generated from τ's input $Expr$ node and output $Expr$ nodes are equivalent. We can use klee_assert to prove this directly.

For $Stmt$ nodes, the approach is different since $Stmt$ nodes are pieces of code, and we cannot check their equivalence directly. As shown in Figure 4, children of a $Stmt$ node could be $Expr$ nodes or another $Stmt$ nodes. As Algorithm 3 shows, when the input node S_I is a $Stmt$ node, our framework first checks if the output node S_O has the same type as the input node (Line 1) to establish the correctness for the mutator τ. If yes, the equivalence checking is trivial since the framework compares both nodes member by member. If the input node S_I has $Expr$ children, the framework traverses all $Expr$ nodes and conducts equivalence checking the same way using the $Expr$ checking algorithm (Line 2-8). For $Stmt$ members, the framework check those nodes by recursively calling this $Stmt$ checking function (Line 10-11). The HalideIR's mutator transformation guarantees that there will not have infinite recursion.

If the input node and the output $Stmt$ node have different types, we cannot check the equivalence by direct comparison. According to HalideIR's definition, $Stmt$ nodes produce side-effecting on the system state. To compare if two $Stmt$ nodes with different types are equivalent, we need to compare their final system state after symbolic execution. The system state for $Stmt$ nodes includes input and output variables. The input variables are variables within the $Stmt$ that are neither allocated nor written by $Allocate$ or $Store$ nodes. While the output variables are variables within the $Stmt$ that are written by the $Store$ nodes but not allocated by the $Allocate$ nodes. To check the equivalence for two $Stmt$ nodes whose types are different, first, our framework traverses the abstract syntax tree (AST) of S_I and S_O and collects the input variables for S_I and S_O: I_I and I_O, and output variables: O_I, and O_O (Line 14). Next we check if $I_I \equiv \mathrm{I}_O$. If not, two $Stmt$ nodes are not equivalent (Line 15-16). If yes, the framework compiles S_I and S_O using the C backend into P_I and P_O, and symbolically executes them (Line 18-19). Then it checks if the output variables O_I and O_O are equivalent (Line 20-22) to determine the equivalence of the $Stmt$ nodes.

With the above algorithms, our validation framework can conduct translation validation for HalideIR's transformation paths efficiently.

VI. Evaluation Results

This section evaluates the efficiency and effectiveness of our automated verification framework. We have applied this framework to the three most starred HalideIR-based compilers on Github: HeteroCL [7], Halide-HLS [16], and Stanford AHA's Halide-To-Hardware project [2]. We have verified 46 transformations from the three compilers and detected 4 transformation bugs undetected by manually crafted unit tests accompanying these compilers. We classify the compiler bugs identified into two categories: incorrect transformations and compiler crashes, e.g., compiler execution paths that cause core dumps or overflow. Incorrect transformations are tricky

Algorithm 3: check-stmt

Input: The input $Stmt$ node S_I
Input: The output $Stmt$ node S_O
if ***same-type***$(S_I,\ S_O)$ **then**
  foreach $\varepsilon \in$ *Expr members of* S_I **do**
    $P_I, P_O \leftarrow$ **compile**$(\varepsilon\ of\ S_I, \varepsilon\ of\ S_O)$
    $eq \leftarrow$ **sym-exe(assert**(P_I, P_O)**)**
    if ***not*** eq **then**
      return false
    end
  end
  foreach $s \in$ *Stmt members of* S_I **do**
    check-stmt(s **of** S_I, s **of** S_O)
  end
end
else
  `/* Collect all system state vars */`
  $I_I, I_O, O_I, O_O \leftarrow$ **get-state-var**(S_I, S_O)
  if ***not*** $I_I \equiv I_O$ **then**
    return false
  end
  P_I,$P_O \leftarrow$ **compile**(S_I, S_O)
  sym-exe$(P_I,\ P_O,\ I_I,\ I_O,\ O_I,\ O_O)$
  if ***not*** $O_I \equiv O_O$ **then**
    return false
  end
end
return true

since this type of bug produces no warning throughout a compiler workflow. However, they produce incorrect results in the generated code, which are difficult to detect and debug for compiler users. Due to the complexity of HalideIR, a theorem-proving approach can only verify its limited parts. Even though earlier projects attempted to formalize HalideIR semantics and formally certify its rewriting system [14], [18], they could not detect bugs that we found.

This evaluation was conducted on a workstation with a 12-core AMD Ryzen 5900x CPU, 128 GB RAM, and Ubuntu 20.04 operating system. We set the maximum memory consumption for each transformation to 64GB. The evaluation results are summarized in Table I. In HeteroCL, we verified 7 transformations, and the total line of code is 8432. We found one new incorrect transformation in HeteroCL and four compiler crashes. The peak memory consumption is 25.3 GB. In Halide-HLS, we verified 9 transformations and detected 2 transformation bugs and 1 crash bug during the verification. The peak memory consumption is 28.6 GB. The total lines of code are 10518. In the Halide-To-Hardware project, we verified 30 transformations and detected one inconsistent transformation during the translation validation stage. And the peak memory consumption is 33.4 GB. The total lines of code are 8604. Next, we discuss some bugs we detected with our framework.

TABLE I
EVALUATION OF OUR FRAMEWORK ON HALIDEIR-BASED COMPILERS

Compiler	HeteroCL	Halide-HLS	Halide-To-Hardware
# of Transformation Verified	7	9	30
Memory Usage (GB)	25.3	28.6	33.4
# of Lines	8432	10518	8604
# of Incorrect Transformations	1	2	1
# of Compiler Crashes	4	1	0

A. Sample Bug from HeteroCL

In HeteroCL, we verified the compiler transformations such as **CodegenC**, **RemoveNoOp**, **LiftAllocateAttrs**, and **Simplify**. During the verification of the **Simplify** function, we detected an incorrect transformation in the $IfThenElse$ mutator:

```
1  const EQ *eq = next.as<EQ>();
2  const NE *ne = next.as<NE>();
3
4  if (eq && is_const(eq->b) && !or_chain) {
5  then_case = substitute(eq->a, eq->b, then_case); }
```

As illustrated by the above code fragment, in Line 4, if the $IfThenElse$'s condition is an Eq expression, the right-hand side (RHS) of the Eq expression is a constant, and the body of the $IfThenElse$ is not an Or expression, this transformation will replace all use occurrences of the left-hand side expression of the Eq expression in the Then case of $IfThenElse$ with the right-hand side expression of Eq (Line 5).

Our verification found counter-examples caused inconsistent behaviors during verification. As Figure 5 shows, the $Print$ in the illustrating C code on the left should print -1. On the right side is the correct IR. In the middle is the HeteroCL generated IR. The $Print$ function prints 0 instead of -1, because the transformation replaces all uses of x in the Then case with 0, which is the RHS expression of the Eq operator.

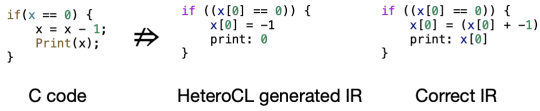

Fig. 5. A HeteroCL transformation bug.

B. Sample Bug from Halide-To-Hardware

Since the two incorrect transformation bugs we found in the Halide-HLS project are similar to the one found in Halide-To-Hardware, we only present a detailed description of the Halide-To-Hardware bug. For Halide-To-Hardware, we have verified HalideIR transformations such as **Simplify**, **CodegenC**, and **RemoveDeadAllocations**. While verifying the $Call$ mutator in the $Simplify\_Call$ function, we found an incorrect transformation.

```
1  Input<Buffer<uint64_t>> input {"input",1};
2  Output<Buffer<uint64_t>> output {"output", 1};
3  void create_algorithm() {
4  hw_output(x) =  input(x) & (uint64_t)(0x3FFFFFFFFF);
5  output(x) = hw_output(x);
6  }
```

The above code is the counter-example that causes incorrect behavior. When executing the above code, the program is supposed to clear the 39th and 40th bit of the $Input$ buffer.

```
1  uint64_t _114 = (uint64_t)(63);
2  uint64_t _115 = _113 & _114;
3  int32_t _116 = _output_s0_x - _17;
```

In the C code generated by Halide-To-Hardware from the above code, where the $constant$ becomes 63, instead of the original 0x3FFFFFFFFF. The transformation produces incorrect results, and there are no compiler warnings during compilation. After discussions with the developers of Halide-To-Hardware, they confirmed this is a bug due to the combination of the $Call$ mutator and Halide-To-Hardware's backend $Codegen$ module.

```
1  else if (op->is_intrinsic(Call::bitwise_and)) {
2      Expr a = mutate(op->args[0]),
3      Expr b = mutate(op->args[1]);
4
5      uint64_t ub = 0;
6      int bits;
7
8  if (const_uint(b, &ub) &&
9        is_const_power_of_two_integer(make_const(a.type(),
10       ub + 1), &bits)) {
11     expr = Mod::make(a, make_const(a.type(), ub + 1));
12 }
```

As the above code from the $Call$ mutator in the $Simplify\_Call$ transformation shows, it converts the And operator to a Mod operator when the RHS value of the And operator plus one is the power of two (lines 9-11).

```
1  int bits;
2  if (is_const_power_of_two_integer(op->b, &bits)) {
3      ostringstream oss;
4      oss << print_expr(op->a)
5      << "&" << ((1 << bits)-1);
6      print_assignment(op->type, oss.str());
7  }
```

In the Mod mutator of $CodegenC$ shown above, when printing this Mod expression, the expression $((1 << bits)-1)$ has the type of 32 bits instead of the original 64 (in line 5). This downcast causes the compiler to produce incorrect behavior during transformation.

C. Limitations

Currently, we only support integer data types due to the limitation of KLEE. Our automated verification framework only supports bounded intra-mutator transformations, i.e., the execution of a mutator does not depend on other mutators. And if there is a loop in a mutator, it must be bounded.

VII. Summary

In this paper, we have presented an automated verification framework for HalideIR-based compiler transformations. This framework applies symbolic execution to (1) explore HalideIR transformation paths and then (2) check the equivalence of the symbolic input and outputs for each symbolic transformation path explored. Its effectiveness and efficiency have been demonstrated by successfully verifying the three most-starred HalideIR-based compilers on GitHub and detecting several transformation bugs that manually crafted unit tests failed to identify.

VIII. Acknowledgment

This research is partially supported by Semiconductor Research Corporation Contract: 2932.001 and a gift from Intel Corporation.

References

[1] Gcc. URL: https://gcc.gnu.org/. III

[2] Halide-to-hardware. URL: github.com/StanfordAHA/Halide-to-Hardware/. I, VI

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014. I

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. III

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018. I

[6] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. III

[7] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 242–251, 2019. I, VI

[8] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004. III

[9] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016. II

[10] Jiajie Li, Yuze Chi, and Jason Cong. Heterohalide: From image processing dsl to efficient fpga acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 51–57, 2020. I

[11] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021. I, II, V

[12] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. V

[13] George C Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 83–94, 2000. I, V

[14] Julie L Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. Verifying and improving halide's term rewriting system with program synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020. II, VI

[15] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, 1998. I, II

[16] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017. I, VI

[17] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013. I

[18] Alex Reinking, Gilbert Bernstein, and Jonathan Ragan-Kelley. *Formal semantics for the halide language*. PhD thesis, Master's thesis. EECS Department, University of California, Berkeley. http ..., 2020. VI

[19] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–27, 2008. V

[20] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. *ACM Sigplan Notices*, 44(6):316–326, 2009. I, V