Fuzzing Deep Learning Compilers with HirGen

Haoyang Ma
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
China
haoyang.ma@connect.ust.hk

Qingchao Shen
College of Intelligence and Computing, Tianjin University
China
qingchao@tju.edu.cn

Yongqiang Tian
University of Waterloo
Canada
The Hong Kong University of Science and Technology
China
yongqiang.tian@uwaterloo.ca

Junjie Chen
College of Intelligence and Computing, Tianjin University
China
junjiechen@tju.edu.cn

Shing-Chi Cheung*
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
China
scc@cse.ust.hk

ABSTRACT

Deep Learning (DL) compilers are widely adopted to optimize advanced DL models for efficient deployment on diverse hardware. Their quality has a profound effect on the quality of compiled DL models. A recent bug study shows that the optimization of high-level *intermediate representations* (IRs) is the most error-prone compilation stage and bugs in this stage account for 44.92% of the whole collected ones. However, existing testing techniques do not consider the features related to high-level optimization (e.g., the high-level IR), and are therefore weak in exposing bugs at this stage. To bridge this gap, we propose HirGen, an automated testing technique that effectively exposes coding mistakes in the optimization of high-level IRs. The design of HirGen includes 1) three coverage criteria to generate diverse and valid computational graphs; 2) the use of the high-level IR's language features to generate diverse IRs; 3) three test oracles of which two are inspired by metamorphic testing and differential testing. HirGen has successfully detected 21 bugs that occur at TVM, with 17 bugs confirmed and 12 fixed. Further, we construct four baselines using state-of-the-art DL compiler fuzzers that can cover the high-level optimization stage. Our experiment results show that HirGen can detect 10 crashes and inconsistencies that cannot be detected by the baselines in 48 hours. We also evaluate the usefulness of our proposed coverage criteria and test oracles.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

*Shing-Chi Cheung is the corresponding author.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598053

KEYWORDS

Deep Learning Compiler, Software Testing

ACM Reference Format:
Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing Deep Learning Compilers with HirGen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598053

1 INTRODUCTION

Deep learning (DL) compilers, such as TVM [15], Glow [30], XLA [1] and nGraph [17], have shown effectiveness in optimizing advanced DL models for efficient model deployment at diverse devices [23]. They take as input a DL model, extract its computational graph, and re-represent the DL model using intermediate representations (IRs) [23]. DL compilers consist of multiple compilation stages, which include high-level and low-level optimizations. DL compilers arrange these two optimizations in order, with high-level optimization followed by low-level optimization. The optimizations aim to compile deep learning models into binary executables that can run efficiently on target hardware devices.

Like conventional compilers[12, 33, 34, 39], DL compilers are prone to bugs. These bugs can cause undesired compiler behaviors, such as crashes, unexpected wrong behavior, and poor performance [31]. These undesired behaviors could result in catastrophic effects on the correctness and reliability of mission-critical DL applications (*e.g.*, autonomous driving cars [13] and aircraft collision avoidance systems [19]).

Techniques have recently been proposed to detect bugs in DL compilers, including TZER [26], TVMFuzz [2], MT-DLComp [36] and NNSmith [25]. Despite preliminary reported success in bug detection for TVM, they are inefficient in revealing bugs that occur in high-level optimization, which account for 44.92% of the bugs found in DL compilers [31]. TZER and TVMFuzz [2, 26] are proposed to detect low-level optimization bugs in a DL compiler with generated low-level IRs. Since these two techniques test DL compilers by mutating low-level IRs, and low-level IRs cannot be used by high-level optimization, they theoretically cannot detect bugs in the

high-level optimization stage. MT-DLComp [36] tests a DL compiler by constructing mutated DL models. Since its mutation strategies only insert operators that yield zero, the kinds of operators and the available places to insert these operators are limited. Therefore, it cannot generate models of diverse computational graphs to cover corner high-level optimization cases. In test oracle design, MT-DLComp does not take advantage of the language features of the high-level IR and high-level optimizations. As a result, it cannot effectively detect bugs in high-level optimizations as demonstrated in Section 5. NNSmith [25] mainly focuses on revealing the hidden defects in DL compilers, such as arithmetic problems, fragile type systems, and poor support for specific data layouts, by generating computational graphs and inputs. Since its generation process and design of test oracle do not consider language features of the high-level IR in DL model generation, and do not consider high-level optimizations in test oracle design, it cannot efficiently detect bugs related to high-level optimization. In Section 5, we will show NNSmith is orthogonal to HirGen regarding bug detection ability.

HirGen. To bridge the gap, we propose the first DL compiler fuzzing technique that focuses on high-level optimization, HirGen. HirGen is designed to satisfy the following four objectives: 1) the satisfaction of integrity constraints, such as type match and tensor shape match, that govern high-level IRs to avoid an early crash before invoking optimization, 2) the exploration of the diversity of computational graphs, 3) the utilization of the high-level IR's language features for the construction of diverse high-level IRs, 4) the capability of detecting multiple types of optimization bugs. To achieve the first objective, HirGen performs type checking and shape checking in each insertion of the operator node by leveraging the information of each existing node, including its type, shape, and connections. After insertion, HirGen also updates the information of the new node for future use. To meet the second objective, HirGen is coverage-guided in input space to explore diverse operator nodes, operator edges, and the combination of operator type and data type. To meet the third objective, HirGen can construct diverse high-level IRs from a single computational graph to achieve full use of the IR's language features. To meet the fourth objective, HirGen incorporates three test oracles, two of which are designed purposely for DL compilers. An example of test oracles is that a model should not make a different prediction after optimization. Besides functional correctness, HirGen can also test the robustness of DL compilers. Specifically, HirGen provides an option of generating invalid computational graphs that violate type constraints and shape constraints [37]. This option tests whether DL compilers can catch such invalid computational graphs and throw the expected exceptions. This way, HirGen can also detect bugs caused by incorrect exception handling.

Following prior works on DL compiler testing, we evaluate the performance of HirGen on TVM, which is the most popular DL compiler. For baseline selection, we choose 1) TVMfuzz (with lowercase f), a preliminary proof-of-concept application from a bug study [31]. The tool is chosen because it is the only testing technique that focuses on detecting bugs arising from high-level optimizations in DL compilers; 2) MT-DLComp [36], a metamorphic testing framework that can cover the high-level optimization stage; 3) LEMON [35], a fuzzing technique for DL libraries (e.g., Tensorflow [11], PyTorch [28]) testing; and 4) NNSmith [25], a generation-based DL compiler fuzzer. We repeated the comparison experiment ten times to mitigate the influence of randomness. Our experimental results show that 1) HirGen can detect around ten distinct crashes that are not detectable by other techniques in a two-day execution; 2) TVMfuzz, MT-DLComp, and LEMON are all inefficient in detecting bugs, they found about three crashes in total; 3) NNSmith is also efficient in bug detection, detecting nine distinct crashes. But except for one crash, all the crashes that they found are orthogonal to the crashes found by HirGen. We will elaborate on the experimental results in Section 5.1. In addition to this comparison experiment, we examine the usefulness of the coverage-guided strategy in generating diverse computational graphs by an ablation study in Section 5.5.

Contribution. This study makes three major contributions.

- This work introduces a new focus on testing the most bug-prone stage, high-level optimization, of DL compilers. We propose a computational graph generation algorithm and three test oracles to detect bugs of diverse root causes in high-level optimizations.
- We implement HirGen, a fuzzing technique targeting at TVM. It has detected 21 bugs, of which 17 have been confirmed, 12 have been fixed, and 14 bugs were previously unknown. Among the 17 confirmed bugs, 14 are highly related to high-level optimizations, and others are about low-level optimization and deployable code generation. Furthermore, our extensive evaluation shows that HirGen outperform the state-of-the-art testing techniques.
- We release HirGen, the details of detected bugs and experiment data at https://github.com/haoyang9804/HirGen.

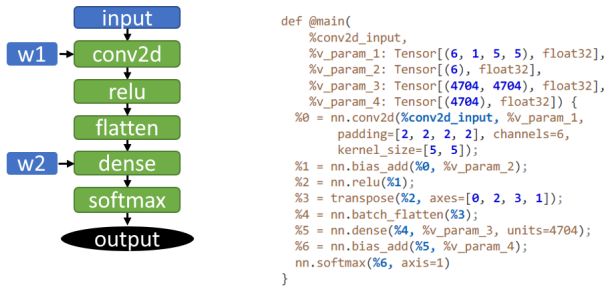
2 BACKGROUND

2.1 DL Compilers

DL compiler takes as input DL models. These models can be constructed with the help of DL frameworks, such as Tensorflow[11] and PyTorch[28]. After interpreting the computational graph of input model, DL compiler converts it into a high-level IR. Each node in the computational graph is represented by one or several IR expressions. For instance, a *conv2d* (two-dimensional convolution) node is represented by *nn.conv2d* in the high-level IR of TVM. DL compilers then optimize the computational graph at the high-level IR. For instance, a static subgraph independent of inputs can be optimized through constant folding at the IR. After optimization, the high-level IR is translated into the low-level IR for further optimization. In this step, a high-level IR expression (e.g., `nn.conv2d`) is expanded into a nested loop of low-level computation instructions. Subsequently, low-level optimizations are performed to improve efficiency. For instance, loop tiling can be performed at low-level optimization to accelerate the computation of *conv2d* on a specified hardware device. Finally, the low-level IR is translated into deployable code for diverse hardware using traditional compilers and platforms.

2.2 Computational Graph and High-Level IR

A computational graph is a directed graph that expresses the data flow in computation. High-level IR, also known as graph-level IR,



(a) Computational Graph (b) High-level IR

Figure 1: An Example of a Computational Graph and the Corresponding High-level IR

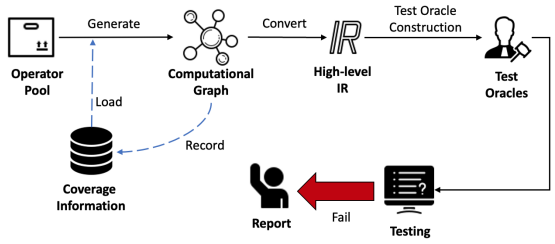


Figure 2: Workflow of HirGen

is an intermediate representation to express the computational graph. In DL community, high-level IRs are widely used to describe computational graphs by DL compilers (e.g., TVM) and also frameworks (e.g., ONNX). Figure 1a illustrates the computational graph of a 2-dimensional convolutional neural network, where variable/-constant nodes and operator nodes are colored blue and green, respectively. A black ellipse denotes the end of a graph. Arrows in this graph represent data flows. Specifically, variable and constant nodes are the starting points of a data flow, passing their data to the following nodes. In contrast, operator nodes work as a relay to extend the data flow, passing the results they calculate. Figure 1b shows the corresponding Relay IR, the high-level IR in TVM, of the computational graph in Figure 1a. This IR is not unique as the same semantics can be represented in other ways by utilizing the language features of IR. For instance, Relay offers first-class functions to separate the main function into multiple functions with complex call chains. ONNX also plans to support this feature in the future [7].

3 APPROACH

This section presents the design and underlying methodology of HirGen. Figure 2 shows the workflow of HirGen. HirGen maintains a pool of 58 operators that can be expressed by high-level IRs in popular high-level frameworks (e.g. Relay [29], ONNX [4]). HirGen first loads existing coverage information, generates a computational graph based on it and updates coverage information from the newest graph. Then, HirGen leverages a high-level framework such as Relay or ONNX, to convert the graph into a high-level IR and feeds it into the DL compiler. To efficiently capture the defects in the target DL compiler, besides the commonly-used oracle crash, HirGen constructs two test oracles from the spirits of metamorphic testing and differential testing. Any test case that violates the oracles is regarded as a witness to a bug of the compiler and will be reported to developers. The remainder of this section is divided into three parts. Section 3.1 elaborates on the details of our computational graph generation algorithm. Section 3.2 presents how to utilize the language features of the high-level IR and convert computational graphs into high-level IRs. In Section 3.3, we will introduce the design of the test oracles proposed by us.

3.1 Computational Graph Generation

3.1.1 Overview. We consider the generation of a computational graph as a process of continuously inserting various operator nodes into the initially empty graph $CG = \{\}$ until the number of operator nodes reaches a threshold. In each iteration, HirGen selects an operator from the operator pool, loads the operator into CG as node nd, and constructs a connection between nd and other existing nodes. Meanwhile, HirGen maintains the information of each node, including data type, tensor shape, and connection information. Furthermore, HirGen also involves three coverage criteria to improve the diversity of the graph.

With these prerequisites, HirGen provides two generation modes. To generate valid computational graphs, HirGen utilizes the aforementioned node information for strict type checking and shape checking. In this way, each insertion strictly satisfies the constraints. We call this mode *strict generation*. However, strictly following the constraints may miss the opportunity to test the exception handling ability of DL compilers when constraints are violated. Therefore, HirGen also provides *disruptive generation* to deliberately break type constraints and shape constraints. We will first elaborate on the strict generation and then the disruptive generation.

3.1.2 Node Information. Inserting a node to CG requires node information of all existing nodes to perform type-checking and shape-checking. Node information describes the typical features of a node, and such information is essential to ensure the correctness of each insertion. For instance, in the insertion of an operator named add that sums two nodes, HirGen first checks all available nodes (including operator nodes, variable nodes, and constant nodes) and selects two nodes n_a and n_b from available nodes, such that n_a and n_b have compatible tensor shapes and data types, and their data types are acceptable by the operator add. Each type of node has its own node information, as detailed in Table 1.

Table 1: Node Information

Node Type	Node Information
variable	$dataType$, $tensorShape$
constant	$dataType$, $tensorShape$, $value$
operator	$dataType$, $parentNodes$, $tensorShape$ = INFERENCE($parentNodes$)

Specifically, HirGen considers the following three types of nodes.

(1) *Variable* node. It involves data type $dataType$ and tensor shape $tensorShape$ describing the details of the tensor wrapped in this node. $dataType$ corresponds to the data type of all elements in this tensor, such as $int64$ and $float32$. $tensorShape$ is a vector of the scale of all dimensions in the tensor.

(2) *Constant* node. Besides the $dataType$ and $tensorShape$, *constant* node includes the value of tensor *value* as a part of its information.
(3) *Operator* node. Operators require parameter(s); thus, they are all connected with other nodes in the graph. To document this connection information for each operator node, besides $dataType$, HirGen records its parent node(s) $parentNodes$ to which this node connects and records its tensor shape inferred from the parent node(s).

3.1.3 Coverage Guidance. To explore diverse data types, shapes, and operators in computational graph generation, we design three coverage criteria.

Operator-datatype Coverage. Let op_i be the i_{th} operator in the operator pool. Let $dtype_j$ be the j_{th} data type in the collection of data types. $Cov(op_i, dtype_j)$ is 1 when op_i has once been inserted into the graph as a node with data type $dtype_j$. Otherwise, it is 0. This coverage encourages HirGen to 1) involve different operators in the graph and 2) utilize diverse data types since the data type problem is a big concern for DL compilers [31].

Operator-shape Coverage. Let op_i be the i_{th} operator in the operator pool. Let *shape* be the shape of the output tensor of this operator node after being inserted into the graph. Let $Cov(op_i, shape)$ be 1 if op_i has once been inserted into the graph as a node with tensor shape *shape*, and 0 otherwise. With operator-shape coverage, HirGen tries various calculations with diverse tensor shapes, thus increasing the probability of encountering calculation problems, such as the poor implementation of some operators in special shapes or different calculation results on different platforms.

Operator-edge Coverage. Let op_i and op_j be the i_{th} and j_{th} operators in the operator pool. Let $Cov(op_i, op_j)$ be 1 if there exists one edge from op_i to op_j, and 0 otherwise. This coverage guides HirGen to connect the new operator node to the existing operator nodes instead of variable and constant nodes. This way, the generated computational graph contains a more complex and deep data flow instead of a parallel connection of several simple data flows. In the implementation, we can easily extend operator-edge coverage to operator-path coverage with three or more operator nodes included. In this way, we can explore a more diverse and complicated computational graph, but at greater time costs.

The design of the first two coverage criteria is motivated by the fact that type problem and shape problem are the two major root causes of DL compiler bugs [31]. The design of the third one tries to complicate the data flow of the computational graph since the third coverage encourages HirGen to interleave different operators in a computational graph. Specifically, With these three coverage criteria, HirGen is prevented from producing previously generated subgraphs in computational graph generation. Since high-level optimizations often involve identifying, annotating, re-constructing, and shrinking optimizable subgraphs, duplicate subgraphs can only find duplicate bugs. For instance, *operator fusion* is a high-level optimization that can fuse operators in a high-level expression into a larger operator. Fusing different operators could encounter different situations and cover different codes. With coverage criteria, a group of operators will unlikely appear in a high-level expression in the previous order. Therefore, these three coverage criteria facilitate HirGen to find new bugs more efficiently. Bug_2 in Table 2 was detected by HirGen in three days but was never detected by HirGen$_r$ (HirGen without coverage guidance) in our 2-week experiment.

Algorithm 1 Computational Graph Generation

```
1:  procedure GENERATION(rOpNum)
2:      CG ← {}
3:      opnum ← 0
4:      opPool ← {add, subtract, multiply, divide, ...}
5:      dataTypeSet ← {int64, int32, int16, int8, uint64, uint32, ...}
6:      repeat
7:          opNode ← SELECT(opPool)
8:          dataType ← SELECT(dataTypeSet)
9:          connection, shape, CG ← PREINSERT(opNode, dataType, CG)
10:         coverage ← GETCOVERAGE(opNode, dataType, connection, shape)
11:         if NEWCOVERAGE(coverage) then
12:             opnum ← opnum + 1
13:             UPDATECOVERAGE(coverage)
14:             opNode.info ← (connection, shape, dataType)
15:             CG ← CG ⋃ {node}
16:         end if
17:     until opnum = rOpNum
18:     return CG
19: end procedure
20: procedure PREINSERT(opNode, dataType, CG)
21:     availableNodes ← TYPECHECK(CG, dataType)
22:     nodeGroup1, nodeGroup2, ... ← SHAPECHECK(availableNodes)
23:     paramNodes ← SELECT (nodeGroup1, nodeGroup2, ...)
24:     if NODENOTENOUGH(paramNodes) then
25:         node1, node2, ... ← CREATE(dataType, paramNodes)
26:         CG ← CG ⋃ {node1, node2, ...}
27:         paramNodes ← paramNodes ⋃ {node1, node2, ...}
28:     end if
29:     for node IN paramNodes do
30:         connection ← (opNode, node)
31:     end for
32:     shape ← INFERENCE(connection)
33:     return connection, shape, CG
34: end procedure
```

3.1.4 Constraint-Awareness Graph Generation. Algorithm 1 presents two procedures used by HirGen to generate computational graphs that strictly follow the type and shape constraints of the high-level IR. GENERATION is the main procedure. It takes as input the required number of operators $rOpNum$ to be contained in the output computational graph. PREINSERT is an auxiliary procedure that enforces type-checking and shape-checking in generation. GENERATION procedure includes the following two main parts.

Initialization. HirGen performs initialization from Line 2 to Line 5. Specifically, an empty computational graph *CG* is initialized and the number of operators *opnum* in *CG* is set to 0. The operator pool and data type set are both initialized for future use.

Generation Loop. In each iteration (Lines 6-17), HirGen generates an operator node, updates its node information, and finally inserts it into *CG* if new coverage is explored. Specifically, HirGen first randomly selects an operator and data type (Line 7 and 8). Then it seeks for connection from *CG* and infers the tensor shape of the operator node *opNode* built from the newly selected operator (Line 9). Subsequently, HirGen calculates coverage and performs update and insertion if new coverage is explored (Line 10-16) i.e., there is any increment of the three coverages defined in section 3.1.3. During the update, HirGen adds *opnum* by 1, updates coverage

and node information of *opNode*. When *opnum* equals *rOpNum*, HirGen stops the generation loop and returns *CG*.

Procedure preInsert shows the details of building connection(s) between *opNode* and existing nodes of *CG* and shape inference. In type-checking, HirGen absorbs type constraints in the target DL compiler and uses them to avoid type mismatch (e.g., float32 and int64 for *add* operator). In shape-checking, HirGen includes shape rules in the target DL compiler, such as broadcasting rule, which specifies that if the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with the ones on its leading side. Shape-checking avoids shape mismatch. With type-checking (Line 21) and shape-checking (Line 22), HirGen sorts out several node groups from *CG*. All nodes of each node group are mutually shape-compatible and nodes from these groups are all type-compatible with *opNode*. Then HirGen selects a node group and dumps all its nodes into *paramNodes* (Line 23). The number of required parameter nodes is fixed for each kind of operator. In the implementation, HirGen has a certain probability of connecting one node to an operator node multiple times, such as connecting one variable node to add a node twice, meaning adding the node to itself. But to complicate the data flow, HirGen discourages this behavior in favor of connecting the required number of different nodes. Therefore, if the number of parameter nodes in *paramNodes* is insufficient for *opNode*, HirGen has a large probability of creating variable nodes or constant nodes that are shape-compatible with all parameter nodes and type-compatible with *opNode*, inserts them into *CG* and updates *paramNode* (Line 24-28). Finally, HirGen creates connection information (Line 29-31), infers the tensor shape of *opNode* and returns them both plus the possibly updated *CG*.

3.1.5 Disruptive Generation Algorithm. Disruptive generation is similar to constraint-awareness generation. It also needs coverage to memorize what type constraints and shape constraints have been broken. In addition, node information is also required. Since it contains the data type and tensor shape of each node, which is necessary for breaking constraints. Specifically, during disruptive generation, HirGen purposely 1) connects the operator node to other node(s) with the data type(s) it cannot accept in TVM (e.g., `add` operator with `bool` data type), and 2) connects nodes that are type-incompatible or shape-incompatible (e.g., add two nodes of which the shapes are `[3,4]` and `[2,3]` respectively).

3.2 High-Level IR Generation

High-level IR generation is simple with the help of existing high-level frameworks, such as Relay and ONNX. Taking Relay as an example, it provides ample APIs for receiving node information of various types of nodes and diverse operator nodes. For instance, `relay.var` takes as inputs its name, data type, and tensor shape, and `relay.add` takes as input only its connection information. These APIs contain strict type constraints and shape constraints, and it is easy to crash early before optimization if the computational graph contains an error.

Besides the plain conversion by loading each node into its corresponding high-level expressions and assembling them into a high-level IR, we can also utilize the primitive features of these high-level frameworks. Take Relay for example, to improve expressivity, it allows using a function to wrap a subgraph and call the function in other ones. ONNX also plans to support this feature by supporting Function API. To better utilize these features, we also consider extracting a subgraph from the generated computational graph and wrapping it with a high-level function. In this way, we can better test how DL compilers tackle the situation where functions are included.

Algorithm 2 Conversion from a Computational Graph to a High-level IR

```
1:  procedure Conversion(CG)
2:      Functions ← {}
3:      Expressions ← {}
4:      for node in CG do
5:          Expressions ← load(node.info, Functions, Expressions)
6:          if roll() == func then
7:              inputNodes, outputNodes ← analyze(Expressions)
8:              function ← composeFunction (inputNodes, outputNodes)
9:              Functions ← Functions ⋃ function
10:             Expressions ← {}
11:         end if
12:     end for
13:     return Expressions ⋃ Functions
14: end procedure
15: procedure load(node.info, Functions, Expressions)
16:     expression ← ConstructExpression (node.info)
17:     if parentInFunction(node.info) then
18:         functions ← find(node.info)
19:         callExprs ← createCallExpression(functions)
20:         Expressions ← Expressions ⋃ {callExprs}
21:     end if
22:     Expressions ← Expressions ⋃ {expression}
23:     return Expressions
24: end procedure
```

The overall algorithm of converting a computational graph into a high-level IR is shown in Algorithm 2. Conversion procedure takes as input a computational graph *CG* and outputs its corresponding high-level IR. During initialization, HirGen creates two empty sets, named *Functions* and *Expressions* respectively (Line 2, 3). They represent the collection of functions and high-level expressions, respectively. In the for loop (Line 4-12), HirGen traverses all nodes in *CG*, loads each node into high-level expression and update *Expressions* (Line 5). It randomly selects a set of high-level expressions and wraps them with a function (Line 6-11). To compose a function, HirGen first analyzes the input nodes and output nodes of the underlying subgraph of the expressions (Line 7). It composes a function using these nodes (Line 8. Finally, HirGen updates *Functions* and *Expressions* (Line 9-10). Conversion procedure returns the union of *Expressions* and *Functions* as the high-level IR. load procedure presents the detail of loading a node into high-level expression. During loading, HirGen takes care of connection information by inquiring whether the node connects to other nodes wrapped in function(s) (Line 17), if it is the case, then a call expression is created (Line 19). This procedure returns *Expressions* after the update.

3.3 Test Oracles

Test oracles determine if a test passes or fails. In this paper, we consider three test oracles to detect different types of failures. Any failed test case determined by these oracles will be reported.

3.3.1 $Oracle_1$: Crash. Crash is widely used in test oracle construction to decide whether the testing fails [26]. Besides, according to

the statistics in a compiler bug study [31], the number of bugs with the crash symptom occupy 59.37% of all collected 603 bugs. This huge proportion shows an urgent need to take crashes seriously. As for crash bugs detected when type-checking and shape-checking are turned off, we only report the bug if the crash is a segmentation fault because other crashes with detailed bug traces are primarily due to explicit violations of constraints in the computational graph. As for other crash bugs, we report them all since the generated computational graph under checking strictly follows all constraints in TVM and the crash is largely due to the poor implementation of TVM.

3.3.2 $Oracle_2$: Result Inconsistency among the Original High-Level IR, the Optimized High-Level IR and the Mutated High-Level IR. Intuitively, high-level optimization only relates to performance boosts such as calculation acceleration and memory cost saving, but can not change results. In addition to involving high-level optimization, we also design a mutation strategy named function rewrite to generate the mutated high-level IRs that have the same output as the original high-level IR given the same input. This mutation strategy is inspired by Relay's support for functional programming features. By function rewriting, we can better utilize Relay's expressions and better test TVM with richer high-level IRs. Specifically, this mutation strategy can rewrite function expressions in the high-level IR in the following ways.

- Turn a global function f into the local closure of another newly created global function g. g has the same parameters as f and its returned value is a call to f with these parameters. After tuning, this mutation also substitutes all calls to f with calls to g.
- Wrap a function f with an empty function g which returns f and also change all calls to f to calls to the call to g.
- Call a function f and return the call in another function g, then substitute all calls to f with calls to g.

The mutated high-level IR only differs from the original high-level IR in the function call chain. Therefore, it is expected that the three high-level IRs produce the same calculation results given the same input. This metamorphic relation inspires us to form this oracle. In addition to different calculation results, if the original high-level IR passes compilation and runtime but the optimized or mutated one fails in one of these two processes, we also count it as the result inconsistency.

3.3.3 $Oracle_3$: Result Inconsistency across Hardware Devices. To maintain the same predictive capability of a DL model on different supported hardware devices, TVM should promise to output the same results on diverse hardware given the same input to a DL model. And similar to $Oracle_2$, inconsistent execution status (e.g., crash on CPU but execute well on GPU) is also counted as result inconsistency. Following this common sense, we build $Oracle_3$ with the spirit of different testing. Given any high-level IR, after compiling it with multiple provided compilation approaches, feeding input and executing it on CPU and GPU, it is reasonable to expect the same calculation results.

4 EXPERIMENT SETUP

4.1 Research Questions

In this study, we aim to answer the following research questions:

RQ1 How effective is HirGen in detecting bugs of TVM?
RQ2 Are all the test oracles effective in detecting bugs?
RQ3 Are bugs found by HirGen highly related to high-level optimization?
RQ4 Is disruptive generation useful in finding exception-handling bugs?
RQ5 Can coverage-guided generation benefit the diversity of the computational graph?

4.2 HirGen Implementation

We implement HirGen in C++ with around 3K lines of code. Our implementation involves 58 operators [6] to generate computational graphs, 25 high-level optimizations for catch optimization bugs and four compilation methods to conduct testing.

4.2.1 Operators. In total, HirGen includes 58 operators supported by TVM, including 23 binary operators and 35 unary operators. And it is easy to extend HirGen with other operators.

4.2.2 Optimization and Compilation Methods. We select in total 25 high-level optimizations supported by TVM [5]. The main reason for choosing these high-level optimizations in TVM is our generated computational graphs can trigger them. Besides these high-level optimizations, it is easy to extend HirGen with other optimizations. Besides collecting these high-level optimizations, we also utilize different compilation methods provided by TVM. Different compilation methods deal with different scenarios and include different optimization sequences. Overall, HirGen supports the following four compilation methods.

(1) `relay.build()`
(2) `relay.build_module.create_executor('debug')`
(3) `relay.build_module.create_executor('graph')`
(4) `relay.build_module.create_executor('vm')`

4.3 Bug Report

For each bug we have found, we report it in one of the three channels: 1) upload the bug-triggered script and experiment environment on TVM Community [9]; 2) report the bug on Github Issue [10] with a reproducible script, experimental environment, and most importantly, our analysis on the reason for triggering it; 3) create a pull request with the elaboration of this bug and our code patch. We choose our reporting channels primarily based on our expertise in the problem. For the least familiar bug, we submit it on TVM Community in the form of a question to get rid of misdiagnosis. Then, we wait for an official fix or some comments from developers on this problem. For the most familiar one, we directly fix it, and we succeed in creating two pull requests and fixing two bugs. For other situations, we choose the second way and leave some comments on how to fix the bug.

4.4 Baseline Selection

We selected four baselines from the literature.

TVMfuzz. TVMfuzz is a preliminary proof-of-concept application for fuzzing TVM [31]. It can learn TVM API call chains from unit test scripts, then re-order and mutate them. By learning from

high-level IRs and optimization-related unit test scripts, TVMfuzz can cover this stage.

MT-DLComp. MT-DLComp is an automated testing framework for DL compilers [36]. It mutates existing DL models to generate equivalent models and test DL compilers by three oracles. Though this technique is not specially created for detecting bugs in high-level optimization, it can cover this bug-prone stage. Therefore, we also include it as a baseline.

LEMON. LEMON is a testing technique for deep learning frameworks [35]. It generates Keras [3] models by mutating existing models. By setting different backends of Keras, LEMON detects prediction differences incurred by these backends. Though LEMON is not for testing DL compilers, we can retrofit it to barely achieve the goal. In short, we remain the mutation part to generate new models and test DL compilers by two test oracles: 1) crash and 2) above-threshold prediction difference between original Keras models and compiled Keras models.

NNSmith. NNSmith is a generation-based fuzzer for DL compilers [25]. During generation, it generates diverse computational graphs, converts them into DL models using different DL frameworks, and uses gradient-guided search to generate inputs. During testing, it conducts differential testing among several DL compilers. In the testing process, NNSmith captures all prediction differences and crashes.

4.5 Metrics

We mainly target *bug counting* for evaluation. To evaluate HirGen, we count bugs based on independent fixes and developers' confirmation in Section 5.1. In Section 5.1 and 5.5, we studied five baselines in total and obtained a number of crashes/inconsistencies. Since many of them are duplicates, reporting them to the TVM community for bug confirmation can be time-consuming and possibly receives no reply according to our interaction with the developers. Therefore, we use the proximity of bug counting in the experiment of these two sections. In particular, manually-deduplicated bugs in Section 5.1.1 have totally different stack traces, and thus comparing the number of crashes/inconsistencies with distinct stack traces is reasonable. The proximity of bug counting is also used in the evaluation of other works [27, 38].

4.6 Miscellaneous

Timeout Setting. There are two comparison experiments involving timeout. The first is comparing HirGen with the four baselines. The second is comparing HirGen with HirGen$_r$. We executed each of the involved techniques separately for two days, and each execution was conducted ten times to mitigate the influence of randomness. Since all the studied techniques do not find distinct crashes/inconsistencies after 26 hours, it indicates that our 2-day timeout is reasonable to a large extent.

Platform. We conducted experiments on a server with Intel Xeon CPU, NVIDIA GeForce GTX1080Ti GPU, and 128 RAM, coordinated with 64-bit Ubuntu 16.04 OS.

5 EVALUATION

5.1 RQ1: Bug Detection Capability of HirGen

Running three months under the strict mode and one week under the disruptive mode, HirGen has found 21 bugs, of which 17 have been confirmed. 12 out of 17 confirmed bugs have been fixed. Moreover, 10 bugs are previously unknown and 5 fixed bugs were previously unknown. Table 2 presents the details of all the confirmed bugs discovered by HirGen, including their symptoms, root causes, the test oracles detecting them, the fixing status, whether they are previously unknown, by which generation mode were they detected, were they also found by other techniques (blanks mean no other techniques detected the bugs in experiments) and whether they are high-level optimization bugs. Symptom includes crash and inconsistency. The former means that TVM terminates unexpectedly while the latter means that different results or statuses are caught in testing. We also manually investigate the root cause of each bug adopting the taxonomy of a recent bug study [31]. Specifically, We carefully compare these bugs with the collected historical bugs and assign each of them a root cause.

There are five root causes resulting in these bugs.

- Type Problem. This category of bugs is triggered by data type-related problems, including incorrect type inference, incomplete implementation of an operator on one data type, etc.
- Incorrect Exception Handling. This category of bugs occurs when TVM lacks rich and readable warning messages or even has no handling of some extreme situations. This kind of bug is related to the robustness of TVM.
- Incorrect Numerical Computation. This root cause involves incorrect numerical computations, values, or usages.
- Internal API Incompatibility. This category of bugs is triggered because TVM can not handle the combination of some APIs correctly. For instance, unexpected refusal of one combination of several high-level optimizations is counted as this kind of bug.
- Memory Allocation Problem. This root cause refers to poor or incorrect memory allocation.

Check marks in *Previously Unknown* column in Table 2 indicate that the corresponding bug was unknown before we reported it. Since TVM was evolving fast, we found some cases early in the experiment that crashed on the version we tested (TVM v0.9, commit id: 124813f) but worked fine on the latest version. These bugs have been actually fixed before being reported and thus marked as previously known bugs.

Comparison with State-of-the-art Techniques. On average, HirGen detected 11.8 distinct crashes/inconsistencies. The variance of the number of them in the 10 repeated experiments is 0.36. We also conducted a manual inspection of these crashes/inconsistencies by two experienced researchers. We observed that the average number of crashes/inconsistencies related to high-level optimization is 8.8, and the variance of the number is 0.36. TVMfuzz detected 3.7 distinct crashes on average, of which 1.4 crashes are related to high-level optimizations. By Mann-Whitney U Test [20], p-value of the difference between HirGen and TVMfuzz is $0.00018 < 0.01$, which implies the result that HirGen outperforms TVMfuzz in DL compiler bug detection has statistical significance. MT-DLComp and LEMON do not detect any bugs. As for NNSmith, it detected

Table 2: Confirmed Bugs found by HirGen

Bug ID	Symptom	Root Cause	Test Oracle	Status	Previously Unknown	Generation Mode	Found By	High-level Optimization
1	Crash	Incorrect Numerical Computation	$Oracle_1$	Fixed	✓	strict	NNSmith	
2	Inconsistency	Incorrect Exception Handling	$Oracle_2$	Confirmed	✓	strict		✓
3	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed	✓	strict		✓
4	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed		strict		
5	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed	✓	strict		
6	Inconsistency	Incorrect Exception Handling	$Oracle_2$	Fixed	✓	strict		✓
7	Inconsistency	Type Problem	$Oracle_2$	Fixed		strict		✓
8	Inconsistency	Type Problem	$Oracle_2$	Fixed		strict		✓
9	Inconsistency	Internal API Incompatibility	$Oracle_2$	Fixed	✓	strict		✓
10	Inconsistency	Incorrect Exception Handling	$Oracle_2$	Fixed		strict		✓
11	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed		disruptive		✓
12	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed		disruptive		✓
13	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed		disruptive		✓
14	Crash	Memory Allocation Problem	$Oracle_1$	Confirmed	✓	strict		✓
15	Inconsistency	Incorrect Numerical Computation	$Oracle_3$	Confirmed	✓	strict		
16	Inconsistency	Incorrect Numerical Computation	$Oracle_3$	Confirmed	✓	strict		
17	Inconsistency	Incorrect Numerical Computation	$Oracle_3$	Confirmed	✓	strict		

Empty cells in column **Previously Unknown** refer to the bugs that are previously known; empty cells in column **Found by** refer to the bugs that are not found by other techniques; empty cells in column **High-level Optimization** refer to the bugs that are not relevant to high-level optimization.

10 distinct crashes/inconsistencies on average. Among these crashes/inconsistencies, data layout problems and data type problems are predominant, altogether accounting for 52.2% of all crashes/inconsistencies. They are captured with bug messages such as *"WCHN layout is not supported"* or *"TVM cannot support type matching between int32 and int64"*. Among other crashes/inconsistencies, on average 3.5 crashes/inconsistencies are related to high-level optimization, and the variance is 1.45, showing that NNSmith is unstable in detecting high-level optimization bugs. The p-value of the high-level optimization crashes detection difference between HirGen and NNSmith is also $0.00018 < 0.01$, implying the result that HirGen outperforms NNSmith in high-level optimization bug detection has statistical significance. During the manual inspection, we only found one overlapping crash detected by both HirGen and NNSmith, showing that these two techniques have almost complementary bug detection abilities. We will discuss the reason in Section 7.1.

5.2 RQ2: Effectiveness of Test Oracles

To demonstrate the effectiveness of our test oracles, we conduct a case study of several representative and confirmed bugs detected by each test oracle.

5.2.1 $Oracle_1$: Crash. $Oracle_1$ caught the most bugs among all test oracles. In total, it finds eight bugs with three root causes, including *Incorrect Numerical Computation*, *Incorrect Exception Handling*, *Memory Allocation Problem*.

Incorrect Numerical Computation. Take Bug_1 as an example. In the computational graph that triggers this bug, a `divide` operator first calculates the result of dividing a constant by a variable and then passes the calculation result R to `floor_mod` as a dividend. All involved variable nodes and constant nodes are of data type *uint* and this type finally flows into `floor_mod`. However, TVM pre-calculates the possible value range of R and detects it could probably be 0. Therefore, TVM incorrectly throws an exception and terminates even before we give values to $var1$ and $var2$, This bug only happens when the data type is *uint* and is caused by incorrect value range estimation. After developers confirmed this bug and fixed `const_int_bound` analyzer, this numerical computation-related bug was fixed.

Incorrect Exception Handling. Bug_{11}, Bug_{12} and Bug_{13} are three bugs of Incorrect Exception Handling. They are detected under disruptive generation. To trigger these bugs, HirGen must generate computational graphs containing obvious breaks of constraints. For example, in Bug_{11}, the bug-triggering computational graph includes a constant node of type $int16$, a `tan` operator node and the connection between these two nodes. The constant node passes its $int16$ data to the operator node. In this graph, HirGen purposely breaks the constraint that `tan` only accepts $float$ data type defined in TVM and receives a segmentation fault during compilation. This is because TVM does not have exception handling for this operator and its unacceptable data types.

Memory Allocation Problem. Bug_{14} is the only bug of this root cause. Specifically, when HirGen leverages `relay.shape_of` to infer the tensor shape of the variable node with static tensor shape $(1, 2)$, an unexpected crash happens with warning message `Cannot allocate memory symbolic tensor shape [?, ?]`. The question mark here refers to a dynamic shape.

5.2.2 $Oracle_2$: Result Inconsistency among the Original High-Level IR, the Optimized High-Level IR, and the Mutated High-Level IR. $Oracle_2$ caught a total of six confirmed bugs, and five of them have been fixed. These bugs are caused by three different root causes, including *Incorrect Exception Handling*, *Type Problem*, and *Internal API Incompatibility*.

Incorrect Exception Handling. Take Bug_{10} as an example. HirGen catches this bug because it finds that a high-level IR passes compilation while its optimized version fails. Specifically, HirGen

places `FirstOrderGradient` before `FuseOps` in an optimization sequence and detects that TVM cannot successfully handle this optimization sequence. This is because exception handling is too strict. Concretely, TVM performs a traversal on the high-level IR after `FirstOrderGradient` for conducting `FuseOps`. When visiting a constant node, TVM finds this node is not scalar because `FirstOrderGradient` has rewritten this attribute. Therefore, TVM throws an exception and the compilation terminates. However, this check about scalar attributes is too strict and does not consider data type. A fix for this bug completes this exception handling and makes the optimized version successfully passes compilation. Besides, Bug_6 is also a representative, detected by our effort in utilizing the high-level IR's language features. HirGen takes advantage of first-citizen functions in Relay IR and tries to return a function in another function. Since TVM v0.9 cannot well support the lowering of this high-level language feature into a low-level counterpart, a segmentation fault is thrown. The effort in utilizing the high-level IR's language features also helps us find Bug_5, Bug_7, and Bug_8.

Type Problem. Take Bug_8 for instance. This bug is detected by function rewrite mutation. Specifically, after changing a global function f into the local closure of another empty global function g and returning f in g, TVM can not infer the type of g. This is because after successfully inferring the type of f, this type information is lost when TVM begins to infer the type of g.

Internal API Incompatibility. The bug Bug_9 is detected because `relay.build_module.create_executor('vm')` fails, but compilation in other ways runs smoothly. Specifically, after HirGen transforms a high-level IR into the A norm form. Compilation with the virtual machine cannot figure out the bound relation between x_{91} and a global function. However, other compilation ways do not encounter this problem.

5.2.3 $Oracle_3$: Result Inconsistency across Hardware Devices. $Oracle_3$ caught a total of three confirmed bugs, but none of them has been fixed. This is because the difference between computation results on CPU and GPU is caused by platform-specific differences. More specifically, LLVM and CUDA have different implementations of the same operator, while TVM lacks full specification about this operator or lacks a complete warning message about using this operator. Developers responded with a confirmation of this deficiency but they consider it unnecessary to remedy it without it violating the effectiveness of TVM seriously.

Take Bug_{15} as an example. HirGen creates a simple computational graph containing a `right_shift` operator node. This operator node takes as input two other variable nodes. Subsequently, HirGen first generates the corresponding high-level IR, then compiles the IR with `relay.build` to generate the runtime model, and finally creates the input and runs the runtime model on CPU and GPU to get two computation results. When the second variable is larger than the first one, the results are inconsistent. This is because this situation incurs a poison value in LLVM and the use of it in an operator is undefined. Although this confirmed bug does not come from a bad implementation of TVM but from an external compiler issue, it still confuses users when their DL model triggers this inconsistency. The refinement of the exception handling system could be a compromise approach for this ill situation.

5.3 RQ3: Bugs Related to High-Level Optimization

As a DL compiler fuzzer focusing on high-level optimization, HirGen is capable of detecting bugs in high-level optimization or bugs highly related to this stage. In this subsection, we manually study the code patch of each fixed bug detected by HirGen and analyze their relationship with high-level optimization and how the detection of them improves this stage.

Bug_2, Bug_8, Bug_9, Bug_{10} are bugs detected in high-level optimization. Bug-triggered patterns for these four bugs are similar: after high-level optimizations, HirGen detects a violation of $Oracle_2$. These bugs show the inability to optimize the structure that several high-level optimizations should have optimized, and incompatibility among several optimizations. For instance, Bug_8 shows that after performing `InferType` on one function, the solved types cannot be passed to the next function and thus triggers a type problem. Bug_{10} shows `FuseOp` can not be well performed after performing `FirstOrderGradient`. Fixing these bugs directly improves the performance of the optimization and facilitates the possibility of multiple optimization combinations.

Besides, HirGen finds eight bugs with crash symptoms and all of them were triggered during compilation. Among them, Bug_3, Bug_{14} are directly related to high-level optimization. To improve efficiency, TVM calls `OptimizeImpl` during compilation and invokes 11 high-level optimizations implicitly. These optimizations work by one or several passes on the high-level IR, which performs a rewrite at any optimizable expression. All expressions in the high-level IR are visited in each pass, and assertions embedded in TVM check each expression. Bugs in this process may prevent high-level optimizations from being well executed or even result in a crash to stop the optimization. Fixes for these bugs are indirect fixes for the required IR passes needed by high-level optimizations. Besides, Bug_{11}, Bug_{12}, and Bug_{13} are in the high-level IR construction. Since construction happens before optimization, these bugs also prevent high-level optimizations.

Although our approach is proposed for high-level optimization, the test cases generated by our approach can also execute low-level optimizations and deployable code generation. Thus, it has the side effect of testing the other stages. The results also confirm it. Bug_{15}, Bug_{16}, and Bug_{17} are all related to the low-level part and code generation of TVM. They are detected due to inconsistent calculation results on different backends (i.e., LLVM and CUDA) given the same inputs. These bugs show the need to couple TVM with these backends better. Bug_1 and Bug_5 are arithmetic problems at low-level. HirGen can detect them because the generated computational graphs contain error-triggered computational logic.

5.4 RQ4: Effectiveness of Disruptive Generation

During experiments, HirGen generated 170 computational graphs with different bug-triggering combinations of the operator, data type, and tensor shape. All these graphs can incur crashes of TVM with only "segmentation fault" information, showing the deficiency of exception handling ability. In the latest TVM version, all these bugs have been fixed. All these obvious breaks of constraints trigger crashes with detailed bug information now. By comparing the bug

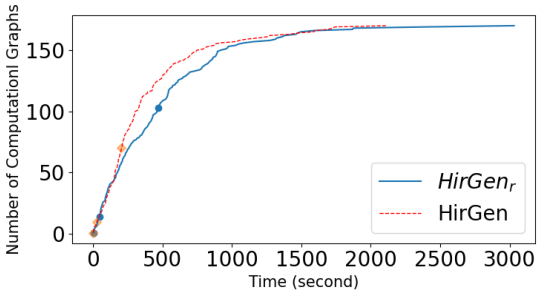


Figure 3: HirGen *vs.* HirGen$_r$ under Disruptive Generation

information of the latest TVM, we found there are three bugs in total triggered by these 170 graphs.

5.5 RQ5: Effectiveness of Coverage-Guided Generation

To generate diverse computational graphs with various data types, tensor shapes, and operators, we design three coverage criteria. To answer this RQ, We implement a simplified version of HirGen, saying HirGen$_r$. HirGen$_r$ is identical to HirGen except that HirGen$_r$ is not guided to generate computational graphs but selects operators, shapes, and types under the rule that all selections are valid, and connects the new operator to the existing operator(s) randomly under the rule that the connection is valid. We conducted two experiments for the comparison between these two techniques.

The first experiment is about the bug detection ability of HirGen and HirGen$_r$ under strict generation mode. In each round of the experiment, we executed these two techniques for two days independently. To mitigate the influence of randomness, our experiment includes ten rounds. On average, HirGen found 11.8 distinct crashes/inconsistencies. Among them, 8.8 crashes/inconsistencies are related to high-level optimization. The variance of the number of distinct crashes/inconsistencies and that of the number of distinct crashes/inconsistencies related to high-level optimizations are both 0.36. As for HirGen$_r$, it found 8.9 distinct crashes/inconsistencies. 6.7 crashes/inconsistencies are related to high-level optimizations. The variance of the number of distinct crashes/inconsistencies is 3.16 and the variance of the number of distinct crashes/inconsistencies related to high-level optimizations is 1.12. By Mann-Whitney U Test, the p-value of HirGen outperforming HirGen$_r$ in detecting distinct crashes/inconsistencies related to high-level optimizations is $0.0028 < 0.01$, and the $p - value$ of HirGen outperforming HirGen$_r$ in detecting distinct crashes/inconsistencies is $0.00512 < 0.01$, implying the result that HirGen outperforms HirGen$_r$ in detecting high-level optimization bugs has statistical significance. Besides, HirGen$_r$ has a bigger variance, showing that it's unstable in bug detection.

The second experiment is similar to the first one. In this experiment, we compare the bug detection ability of HirGen and HirGen$_r$ under the disruptive generation mode. Since disruptive generation promises that each insertion contains a violation of constraints and must trigger failure, there is no need to generate a multiple-operator graph. So we let them generate one-node graphs. Figure 3 presents the experiment results. HirGen and HirGen$_r$ both generated 170 bug-triggered computational graphs, each of which contains unique tuples of (*operator*, *tensor shape*, *data type*). In this figure, HirGen shows a more exploratory nature in the diversity of graphs and thus detects bugs faster. Further, two techniques both found 3 bugs using these 170 graphs. And the timestamps of bug detection are also marked in this figure, showing that HirGen found bugs faster than HirGen$_r$.

6 DISCUSSION

6.1 Limitation and Benefit of $Oracle_3$

In our experiment, $Oracle_3$ is less effective than others. It only detected three confirmed but not fixed bugs, implying that 1) it's not as efficient as other oracles in bug detection, and 2) fix of these bugs is of low priority from the perspective of developers. The reason for this phenomenon is that the floating-point precision settings differ across platforms [36]. And it is hard for developers to tell whether the differences are caused by bugs or inconsistent precision settings. Therefore, they are reluctant in checking the code or refining the exception-handling module to warn users of the large differences in calculation results among platforms when they use some special operators. Despite this limitation, $Oracle_3$ could still help find confusing scenarios and provide experience for users who do not have enough knowledge of special error-triggering operators in low-level platforms. Take Bug_{15} [8] as an example. If TVM compiles the computational graph construction including `right_shift`, then LLVM may skew the results. Since TVM offers no warning, this phenomenon is confusing. We posted such findings online and obtained such a response from a developer: *There is not a full specification for right_shift intrin in TVM.* Therefore, we counted it as a TVM bug of incomplete documentation and exception handling module.

6.2 Threats to Validity

The threat to *internal* validity mainly lies in the implementation of HirGen. To reduce this threat, two authors of this paper have carefully checked and tested the functionality of all components of HirGen.

The threat to *external* validity mainly lies in the DL compiler we chose in our study. Until now, TVM is one of the most popular and active open-source DL compilers, with 9K stars on GitHub. Though HirGen now mainly supports converting its generated computational graph into the high-level IR of TVM with Relay. The technical approach is also useful for testing other DL compilers with the help of ONNX[4]. ONNX is an open format to represent diverse DL models defined by various DL frameworks and is currently supported by popular DL compilers. Similar to Relay, we can use ONNX's APIs to easily convert a computational graph into a high-level IR of ONNX. This IR is transformable to high-level IRs of existing DL compilers. Adding more support for ONNX to test more DL compilers is also our future work.

The threat to *construct* validity mainly lies in randomness and settings. In computational graph generation, though with coverage guidance, the selection of operator and connection also involves randomness. To alleviate the negative impact of randomness, we 1) repeated all randomness-involved experiments 10 times and utilize average, variance, and Mann-Whitney U Test to promise the

results are statistically significant. The threshold setting for comparing different prediction results in $Oracle_2$ and $Oracle_3$ (mainly $Oracle_3$) is still an open problem in DL compiler testing. One existing work [36] has shown that different floating-point precision settings in different platforms may lead to false positives in bug detection. Therefore, in comparing prediction results, threshold setting is vital in reducing false positives. Since no systematic study on how to set threshold exists, our settings are mainly based on experience and expertise in testing TVM. Since our test oracles did not frequently detect prediction differences, we set the threshold to a tiny floating number, 10^{-3}, to not miss any minor difference, and thus not miss any new bug. In the experiment, HirGen did not find false positives due to the floating-point roundoff.

7 RELATED WORK

7.1 DL Compiler Testing

With the development of DL compiler, the importance of DL compiler testing has been noticed by more and more researchers. According to the testing focus, existing testing techniques can be divided into two categories. The first category aims at testing the whole workflow of DL compilers. Focusing on testing a single stage is another category. MT-DLComp and NNSmith are the former. MT-DLComp [36] can perform semantics-preserving mutation on seed DL models to generate new models with theoretically the same prediction capability. During testing, any prediction difference between mutated models and the seed model or any compilation failure will be captured. NNSmith [25] can generate computational graphs and their inputs/weights from scratch to test DL compilers. Although NNSmith and HirGen both generate computational graphs, HirGen are fundamentally different from NNSmith at least in terms of their testing purposes, their usage of the generated computational graphs in DL compiler testing, and the types of bugs detected. As for testing purposes, HirGen focuses on the most error-prone stage [31], i.e., the high-level optimization stage, while NNSmith has no testing preference for any compilation stage and focuses on validating the prediction correctness of the compiled models. Though HirGen can also cover low-level and codegen components, most of its technical details, including mutation strategies, use of high-level optimizations, and construction of $Oracle_2$, are all designed for the only stage. With different testing purposes, they utilize computational graphs differently. HirGen further generates multiple semantics-equivalent high-level IRs from the computational graphs and uses high-level optimizations to optimize them. NNSmith does not perform these steps. The utilization of high-level IRs and optimizations helps HirGen find much more high-level optimization bugs than NNSmith. NNSmith further finds a set of inputs and weights for the computational graphs such that the compiled DL models produce numerically valid outputs given such inputs and weights. Then NNSmith can validate prediction results and catch any prediction errors. Though HirGen could also cause several prediction errors, it's not as efficient as NNSmith. Because of these differences, HirGen and NNSmith has nearly orthogonal bug detection ability, as shown in section 5.1.

Different from MT-DLComp and NNSmith, several other techniques [2, 26, 31] focus on the testing of a single stage but not the whole workflow. Besides, they perform white-box testing to utilize knowledge gained from the codebase to achieve more efficient and effective testing results. For instance, TZER[26] collects low-level IR passes and mutates them to detect bugs in low-level optimizations, while TVMfuzz [31] focuses on high-level optimization by generating high-level API sequences.

7.2 Metamorphic Testing for Compiler

Metamorphic testing (MT) [16] is a popular approach to address the test oracle problem. Researchers proposed different metamorphic relations (MR) to construct test oracles for different systems under test based on the characteristics of the systems.

In compiler testing, the follow-up test inputs in MR are mostly programs equivalent to their seed programs [14]. Various semantics-preserving mutations have been proposed to generate equivalent programs for compiler testing [18, 21, 22, 32, 36]. MT-DLComp [36] conducts metamorphic testing on DL compilers via two semantics-preserving mutations on computational graphs. Specifically, it inserts always-yield-zero nodes into computational graphs to generate new graphs without skewing the calculation. GLFuzz [18] tests OpenGL by designing six semantics-preserving mutators. Unlike MT-DLComp, $Oracle_2$ of HirGen is for high-level IRs instead of computational graphs, and semantics-preserving mutations in HirGen are also for high-level IRs. Moreover, the mutations of HirGen focus on modifying the function call chain, which is orthogonal to all six mutations in GLFuzz. EMI [21], which stands for equivalence modulo inputs, is a methodology for constructing MRs [14]. The key insight is that given a seed program, a set of inputs can induce a collection of programs giving the same outputs as the seed one on these inputs. EMI has inspired several compiler testing techniques [21, 22, 24, 32]. The semantics-preserving mutation in EMI-based techniques requires profiling program executions before mutation. In addition, EMI-based mutants are constructed to be semantics-equivalent only under specific inputs. In contrast, HirGen requires no profiling, and the mutants generated are semantically equivalent to their seed high-level IRs under all inputs.

8 CONCLUSION

High-level optimization is the most bug-prone stage in the workflow of DL compilers. However, there is no systematic study on testing this stage. To fill this gap, we offer HirGen, a generation-based fuzzer with an effective computational graph generation approach and three test oracles. Different from existing works, HirGen can explore more complicated and valid high-level IRs and thus detect deeper bugs. Besides, three test oracles in HirGen also improve its capability of detecting bugs of various root causes. Our effort improved the robustness and functional correctness of high-level optimization and was recognized by the TVM community.

ACKNOWLEDGEMENT

This work is supported by the National Natural Science Foundation of China (Grant No. 61932021), Hong Kong Research Grant Council/General Research Fund (Grant No. 16205722), Hong Kong Research Grant Council/Research Impact Fund (Grant No. R5034-18), and National Natural Science Foundation of China under Grant Nos.62002256 and 62232001.

REFERENCES

[1] 2016. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla, last accessed on 5/25/2023.

[2] 2020. TVMFuzz. https://github.com/dpankratz/TVMFuzz, last accessed on 5/25/2023.

[3] 2022. Keras. https://keras.io/, last accessed on 5/25/2023.

[4] 2022. ONNX. https://onnx.ai/, last accessed on 5/25/2023.

[5] 2023. HirGen's Optimizations. https://github.com/haoyang9804/HirGen/blob/experiment/optimizations, last accessed on 5/25/2023.

[6] 2023. HirGen's README. https://github.com/haoyang9804/HirGen/blob/master/README.md, last accessed on 5/25/2023.

[7] 2023. Operator Schemas of ONNX. https://github.com/onnx/onnx/blob/main/docs/Operators.md, last accessed on 5/25/2023.

[8] 2023. TVM Bug. https://discuss.tvm.apache.org/t/operator-right-shift-obtains-different-results-in-different-devices/11939, last accessed on 5/25/2023.

[9] 2023. TVM Discuss. https://discuss.tvm.apache.org/, last accessed on 5/25/2023.

[10] 2023. TVM Issue. https://github.com/apache/tvm/issues, last accessed on 5/25/2023.

[11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[12] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (Oct 2021), 30 pages. https://doi.org/10.1145/3485500

[13] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. In *2015 IEEE International Conference on Computer Vision (ICCV)*. 2722–2730. https://doi.org/10.1109/ICCV.2015.312

[14] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (Feb 2020), 36 pages. https://doi.org/10.1145/3363562

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[16] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. arXiv:2002.12543 [cs.SE]

[17] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning.

[18] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct 2017), 29 pages. https://doi.org/10.1145/3133917

[19] Kyle D. Julian, Jessica Lopez, Jeffrey S. Brush, Michael P. Owen, and Mykel J. Kochenderfer. 2016. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. 1–10. https://doi.org/10.1109/DASC.2016.7778091

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[21] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[22] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

[23] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2021. The Deep Learning Compiler: A Comprehensive Survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (Mar 2021), 708–727. https://doi.org/10.1109/tpds.2020.3030548

[24] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. *SIGPLAN Not.* 50, 6 (Jun 2015), 65–76. https://doi.org/10.1145/2813885.2737986

[25] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. https://doi.org/10.1145/3575693.3575707

[26] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 73 (Apr 2022), 26 pages. https://doi.org/10.1145/3527317

[27] Weisi Luo, Dong Chai, Xiaoyue Run, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-Based Fuzz Testing for Deep Learning Inference Engines. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 288–299. https://doi.org/10.1109/ICSE43902.2021.00037

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Curran Associates Inc., Red Hook, NY, USA.

[29] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A New IR for Machine Learning Frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 58–68. https://doi.org/10.1145/3211346.3211348

[30] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2019. Glow: Graph Lowering Compiler Techniques for Neural Networks. arXiv:1805.00907 [cs.PL]

[31] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

[32] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

[33] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

[34] Ziyuan Wang, Dexin Bu, Aiyue Sun, Shanyi Gou, Yong Wang, and Lin Chen. 2022. An Empirical Study on Bugs in Python Interpreters. *IEEE Transactions on Reliability* 71, 2 (2022), 716–734. https://doi.org/10.1109/TR.2022.3159812

[35] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 788–799. https://doi.org/10.1145/3368089.3409761

[36] Dongwei Xiao, Zhibo LIU, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 15 (Feb 2022), 28 pages. https://doi.org/10.1145/3508035

[37] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 176–188. https://doi.org/10.1145/3533767.3534220

[38] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[39] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174

(2021), 110884. https://doi.org/10.1016/j.jss.2020.110884

Received 2023-02-16; accepted 2023-05-03