A Comprehensive Study of Deep Learning Compiler Bugs

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

Haoyang Ma
College of Intelligence and Computing, Tianjin University
China
haoyang_9804@tju.edu.cn

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

Yongqiang Tian
Cheriton School of Computer Science, University of Waterloo
Canada
y258tian@uwaterloo.ca

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

Xiang Chen
School of Information Science and Technology, Nantong University
China
xchencs@ntu.edu.cn

ABSTRACT

There are increasing uses of deep learning (DL) compilers to generate optimized code, boosting the runtime performance of DL models on specific hardware. Like their traditional counterparts, DL compilers can generate incorrect code, resulting in unexpected model behaviors that may cause catastrophic consequences in mission-critical systems. On the other hand, the DL models processed by DL compilers differ fundamentally from imperative programs in that the program logic in DL models is implicit. As such, various characteristics of the bugs arising from traditional compilers need to be revisited in the context of DL compilers.

In this paper, we present the first systematic study of DL compiler bugs by analyzing 603 bugs arising in three popular DL compilers (i.e., TVM from Apache, Glow from Facebook, and nGraph from Intel). We analyzed these bugs according to their root causes, symptoms, and the stages where they occur during compilation. We obtain 12 findings, and provide a series of valuable guidelines for future work on DL compiler bug detection and debugging. For example, a large portion (nearly 20%) of DL compiler bugs are related to types, especially tensor types. The analysis of these bugs helps design new mutation operators (e.g., adding type cast for a tensor to promote implicit type conversion in subsequent tensor computations) to facilitate type-related bug detection. Further, we developed TVMfuzz as a proof-of-concept application of our findings to test the TVM DL compiler. It generates new tests based on TVM's original test suite. They expose 8 TVM bugs that are missed by the original test suite. The result demonstrates the usefulness of our findings.

*Junjie Chen is the corresponding author.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Software defect analysis**; • **Computing methodologies** → *Neural networks.*

KEYWORDS

Deep Learning, Deep Learning Compiler Bug, Empirical Study, Compiler Testing

ACM Reference Format:
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468591

1 INTRODUCTION

Deep learning (DL) has emerged as a promising computational paradigm to solve problems in various domains, such as autonomous driving cars [10], face recognition [61], aircraft collision avoidance systems [42], and software engineering [9, 20, 68, 70]. Various DL frameworks (e.g., TensorFlow [5], PyTorch [4], and Keras [2]) are developed to facilitate the implementation of DL models (e.g., convolutional neural network (CNN) [46], recurrent neural network (RNN) [57], and generative adversarial network (GAN) [30]). Specific hardware has also been designed to accelerate the execution of these DL models. Examples include Google TPU [41], Intel NNP [36], and Apple Bionic [43]. Driven by immense demands, there are many DL frameworks and hardware products on the market. Coding efficient DL models to cater to these frameworks and hardware products is challenging [33, 47].

To alleviate the burden, DL compilers were proposed [47]. They take a DL model programmed on a DL framework as input and generate hardware-optimized code as output. Multiple DL compilers are now available in the market. The dominant ones are TVM [6] from Apache, Glow [1] from Facebook, and nGraph [3] from Intel. Like other software systems, DL compilers are subject to bugs. Buggy DL compilers can be devastating when their incorrectly generated codes are deployed for mission-critical DL applications.

Besides, DL compiler bugs complicate the fault diagnosis of anomalous behaviors in DL applications.

DL compilers may not share the same characteristics as traditional compilers [47]. On one hand, the processed subjects are very different. The DL models processed by DL compilers do not have explicit logical structures like those in the imperative programs processed by traditional compilers. On the other hand, DL compilers have their own multi-level IR (Intermediate Representation) and a large number of DL-specific optimizations (such as operator fusion). Therefore, existing test generation and bug localization techniques for traditional compilers may not work for DL compilers.

This motivates us to conduct the first systematic study on DL compiler bugs to facilitate the understanding of DL compiler bugs. In particular, we investigated the *root causes* of DL compiler bugs, the *symptoms* that bugs exhibit, and the *stages* of a DL compiler in which bugs occur. Our study is based on three most popular DL compilers, including TVM [6] from Apache, Glow [1] from Facebook, and nGraph [3] from Intel, as experimental subjects. These three compilers are diverse in nature, e.g., the dynamic tensor shape is supported by TVM and nGraph but not Glow, and auto-tuning is supported by TVM but not the other two. The diversity facilitates the generalizability of our findings. We studied 603 bugs that were collected and labeled manually according to a systematic process (to be presented in Section 3.2).

From our manual analysis on these DL compiler bugs, we identified 12 root causes and 6 bug symptoms, and obtained 12 major findings. Based on these findings, we provided a series of guidelines for DL compiler bug detection and debugging in the future. In particular, we further made a preliminary proof-of-concept application of our findings by designing a simple but effective method **TVMfuzz** for testing TVM. The design of TVMfuzz is inspired by some findings in our study. It can generate new tests based on TVM's original test suite. We ran **TVMfuzz** for two days. It detected 8 TVM bugs that cannot be detected by TVM's original test suite. The result demonstrates the usefulness of our findings.

To sum up, we make four major contributions.

- We conduct the first systematic study on DL compiler bugs based on 603 bugs from three popular and diverse DL compilers.
- We provide a classification of root causes of DL compiler bugs and symptoms that DL compiler bugs exhibit.
- We discuss and provide a series of guidelines for DL compiler bug detection and debugging in the future.
- We conduct a preliminary proof-of-concept application of our findings by designing a testing tool for the TVM compiler. It exposes 8 TVM bugs that cannot be detected by TVM's original test suite.

2 DEEP LEARNING COMPILERS

DL models are popularly programmed on top of DL frameworks (such as TensorFlow [5], PyTorch [4], and Keras [2]). Various kinds of hardware (e.g., Google TPU [41], Intel NNP [36], and Apple Bionic [43]) are also designed to accelerate the execution of these models. To attain the acceleration, a DL model based on a specific framework is deployed with code that is compiled for optimized execution on the deployed hardware. Various DL compilers have been developed to take a DL model as input and generate hardware-optimized code as output.

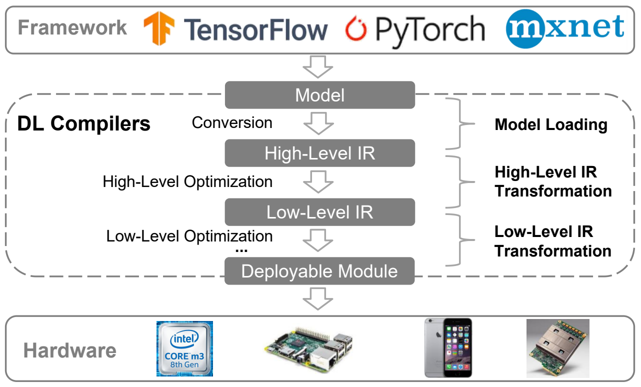


Figure 1: The Architecture of DL compilers

Although DL compilers are built with functional components (e.g., front end and back end) similar to traditional compilers, these two types of compilers have different characteristics. Firstly, DL compilers take DL models as input while traditional compilers take programs as input. There are fundamental differences in the control- and data-flow structures between DL models and programs. Secondly, DL compilers have their own multi-level IR and DL-specific optimizations (such as operator fusion and layout transformation). Figure 1 shows the general architecture of DL compilers, which contain the following three major stages:

- **Model Loading** is a stage responsible to load a DL model and transform it into a computation graph representation (i.e., high-level IR). High-level IR, which aims to construct the control flow and the dependency between data and operators, is hardware-independent.
- **High-Level IR Transformation** is a stage responsible to conduct hardware-independent optimizations on high-level IR to reduce redundancy and improve efficiency. It generates an optimized computation graph for the next stage. The optimizations can be general-purpose or DL-specific. They include node-level optimizations (e.g., zero-dim-tensor elimination), block-level optimizations (e.g., operator fusion), and dataflow-level optimizations (e.g., layout transformation).
- **Low-Level IR Transformation** is a stage responsible to transform high-level IR to low-level IR, which is sufficiently fine-grained to capture hardware characteristics. The transformation also involves hardware-specific optimizations (such as hardware intrinsic mapping and memory latency hiding) on the low-level IR. Then, the optimized low-level IR is compiled to generate code deployed on the specific hardware.

We selected three popular DL compilers (i.e., TVM [6] from Apache, Glow [1] from Facebook, and nGraph [3] from Intel) as experimental subjects in our study. All the three compilers are built with the architecture in Figure 1, but they are also enough diverse, especially in the low-level IR transformation stage. Specifically, TVM adopts machine learning methods to automatically determine the optimal optimizations for a specific hardware. Glow designs

Table 1: Statistical Information on Datasets

Compiler	Duration	#PR	#Bug
TVM	2019-07-30 ~ 2020-11-08	845	318
Glow	2019-07-12 ~ 2020-10-21	271	145
nGraph	2019-04-25 ~ 2020-07-25	245	140
Total	–	1,361	603

a lowering phase in this stage to reduce the operator space by transforming each operator into a sequence of low-level primitives. nGraph does not have its own low-level IR. Its low-level IR transformation is implemented by integrating existing kernel libraries (e.g., cuDNN [22]) or coordinating with PlainML [73].

3 METHODOLOGY AND CLASSIFICATION

3.1 Data Collection

In the study, we used the three most popular DL compilers as subjects, including TVM [6] from Apache, Glow [56] from Facebook, and nGraph [24] from Intel. Since our study aims to investigate the characteristics of DL compiler bugs, we collected closed and merged pull requests that are responsible to fix bugs from their GitHub repositories following the existing work [29, 53]. The reasons why using such pull requests are twofold: 1) Bugs involved in such pull requests have been accepted and fixed by developers; 2) Such pull requests contain more comprehensive information (e.g., code changes, links to related issues, and discussions among developers) about the involved bugs, which facilitates to understand these bugs. In fact, there are several categories of pull requests with different purposes (such as bug fixing, refactoring, and adding new features). Hence, we need to identify the pull requests with the purpose of bug-fixing among the collected pull requests for our study. Specifically, following the existing study [29], we collected the pull requests whose tags or titles contain at least one bug-relevant keywords (i.e., fix, defect, error, bug, issue, mistake, incorrect, fault, and flaw), and call such pull requests *bug-fixing pull requests.*

Table 1 presents the detailed information on our datasets. Since we need to manually analyze each bug, it is unaffordable for us to collect all bugs for manual inspection. Therefore, we collected the bugs for a given duration (i.e., over 15 months) shown in the second column in Table 1. The third column presents the number of collected bug-fixing pull requests for the given duration. After obtaining bug-fixing pull requests, the first two authors manually analyzed and labeled them independently (the process will be described in Section 3.2), and the last column presents the number of bugs identified from these bug-fixing pull requests. In total, we collected 1,361 bug-fixing pull requests and identified 603 bugs, including 318 TVM bugs, 145 Glow bugs, and 140 nGraph bugs. The dataset can be found at our project homepage[1].

3.2 Classification and Labeling Process

In the study, we investigated each bug from three aspects: 1) the **root causes** of bugs, 2) the **symptoms** that bugs exhibit, and 3) the **stages** of a DL compiler in which bugs occur. To label the root cause and symptom of each bug, we adopted the taxonomies of root causes and symptoms from the existing work [29, 37, 38, 58, 62, 63] as the initial taxonomies, and then adapted them to DL compiler bugs through an open-coding scheme following the existing work [38]. Regarding the stages of DL compilers, they have been introduced in Section 2. More specifically, one author first went through all the pull requests to determine the root-cause and symptom categories of our collected DL compiler bugs based on the initial general taxonomy, including adding DL compiler specific categories (e.g., node type problem) and removing irrelevant categories. Then, the first two authors independently labeled these pull requests. Following the existing work [38], we measured the inter-rater agreement among them via the Cohen's Kappa coefficient [64]. In particular, the Cohen's Kappa coefficient was nearly 30% for the first 5% of labeling results, and thus we conducted a training session about labeling. After that, the first two authors labeled 10% of pull requests (including the previous 5% of pull requests), and the Cohen's Kappa coefficient achieved 85%. After further discussion and investigation for these disagreements, the Cohen's Kappa coefficient was always more than 95% in subsequently labeling studies (i.e., labeling 20%~100% of pull requests with the interval of 10%). In each labeling study, the first two authors discussed the disagreements with the third author, and finally all the bugs were labeled consistently.

During the labeling process, we filtered out the pull requests that are actually irrelevant to bug fixing. Some pull requests fixed more than one bug, and we labeled each of them as an individual bug following the existing work [29].

3.3 Root Causes of DL Compiler Bugs

Based on the above classification and labeling process, all the root causes of DL compiler bugs are presented as follows.

3.3.1 API Misuse. This category of bugs occur due to misunderstanding of APIs, including 1) **wrong API**: developers use a wrong API or wrong arguments in an API; 2) **condition missing/redundancy**: developers miss to use (or redundantly use) a condition check for an API; 3) **API missing/redundancy**: developers miss to use (or redundantly use) an API.

3.3.2 Incompatibility. This category of bugs occur under the three scenarios: 1) there are API compatibility issues within a DL compiler caused by API evolution, called **internal (API) incompatibility**; 2) there are API compatibility issues between a DL compiler and third-party libraries (e.g., TensorFlow and NumPy), called **external (API) incompatibility**; 3) there are compatibility issues between a DL compiler and external resources, called **resource incompatibility**, for example, the used image is too large to be incompatible with the DL compiler or some characteristics of the target device are incompatible with the DL compiler.

3.3.3 Type Problem. This category of bugs involves type-related problems, such as type conversion and type inference. According to the characteristics of DL compilers, we further divide this category into three subcategories: 1) **node type problem**: a DL compiler works on a computational graph, where nodes represent the atomic DL operators (such as convolution and pooling) and edges represent the tensors [47]. Each node takes zero or more tensors as

[1] https://github.com/ShenQingchao/DLCstudy.

input and produces a tensor as output. This subcategory refers to the problem involving the types of nodes; 2) **tensor type problem**: this subcategory refers to the problem involving the types of tensors. Specifically, a tensor is a multi-dimensional matrix containing elements of a single data type. 3) **conventional data type problem**: besides nodes and tensors, there are also conventional variables widely used in the development of traditional software systems. This subcategory refers to the problem involving the types of conventional variables.

3.3.4 Tensor Shape Problem. This category of bugs is related to tensor shape or layout. Specifically, Tensor shape describes the number of elements in each dimension. Layout describes how the tensor is represented in memory, which plays an important role in model performance. These bugs are triggered during the operation of tensor shape matching, tensor shape transformation, tensor shape inference, data layout transformation, etc.

3.3.5 Incorrect Numerical Computation. This root cause involves incorrect numerical computations, values, or usages, such as using incorrect operators or operands, dividing by 0, missing operands, or redundant operands.

3.3.6 Incorrect Assignment. This root cause involves that a variable is incorrectly initialized or assigned, or a variable lacks initialization.

3.3.7 Incorrect Exception Handling. This category of bugs occurs due to incorrect exception handling, for example, a DL compiler 1) does not throw an exception when it should, 2) throws an exception when it should not, and 3) provides incorrect/imprecise exception messages.

3.3.8 Misconfiguration. This category of bugs is caused by incorrect configurations in a DL compiler, such as misconfigurations in the file `CMake`.

3.3.9 Concurrency. This root cause involves incorrect operations on concurrency-oriented structures (e.g., locks, threads, and critical regions).

3.3.10 Incorrect Code Logic. Code logic refers to the implementation logic of an algorithm (e.g., an optimization on the computational graph). This category of bugs involves a number of statements or blocks. According to the components where bugs occur, this category of bugs are divided into two subcategories:

- **Incorrect Optimization Code Logic.** Optimization is an important functionality of a DL compiler. As presented in Section 2, a DL compiler tends to contain a number of optimizations (including high-level optimizations and low-level optimizations). This subcategory of bugs occurs at the component of various optimizations.
- **Incorrect Non-optimization Code Logic.** This subcategory of bugs occurs at the other components except for DL compiler optimizations.

3.3.11 Typo. This root cause is due to the carelessness of developers, e.g., “default” is mistakenly written as “defualt”.

3.3.12 Others. Each case in this root cause occurs very infrequently and does not belong to any other root causes.

3.4 Symptoms of DL Compiler Bugs

Based on the above classification and labeling process, all the symptoms of DL compiler bugs are presented as follows:

3.4.1 Crash. Crash means that a DL compiler terminates unexpectedly during compilation, which usually produces an error message.

3.4.2 Wrong Code. Wrong code means that a DL compiler behaves in an unexpected way without a crash, which produces a wrong result or middle result (e.g., the non-equivalent IR after an optimization).

3.4.3 Bad Performance. This symptom means that the time cost or memory consumption spent by a DL compiler is much larger than developers'/users' expectation (e.g., the performance achieved on the previous version during regression testing or the performance required by specific hardware).

3.4.4 Hang. This symptom means that a DL compiler keeps running for a long period of time without producing the expected result.

3.4.5 Build Failure. This symptom means that the installation of a DL compiler terminates unexpectedly.

3.4.6 Unreported. There are also DL compiler bugs whose symptoms cannot be identified by analyzing the corresponding pull requests (including code changes, discussions, and related issues).

3.5 Research Questions

Our study aims to address the following five research questions (RQs):

RQ1: What is the occurrence frequency of different root causes of DL compiler bugs? The root causes facilitate the understanding of the nature of DL compiler bugs, which is helpful to detect, localize, and fix bugs. Moreover, it is interesting to explore the root causes specific to DL compiler bugs and also investigate whether the conclusions on common root causes between DL compiler bugs and traditional software bugs are consistent or not.

RQ2: What is the occurrence frequency of different symptoms of DL compiler bugs? The symptoms facilitate the understanding of the consequences of DL compiler bugs, which is helpful to triage them and assess their impacts. Also, according to the bug symptoms, it is helpful to design DL compiler testing methods with effective test oracles.

RQ3: What is the relationship between root causes and symptoms of DL compiler bugs? Understanding root causes in RQ1 and symptoms in RQ2 is the first step to investigate DL compiler bugs. By mapping them (i.e., understanding which root cause is more likely to produce a specific bug symptom), it is helpful to provide more information to deal with bugs.

RQ4: Which stages in DL compilers are more fragile to bugs? DL compilers contain three major stages and the fragility of different stages may be different. Identifying bug-prone stages is helpful to guide developers to allocate their efforts during the process of DL compiler testing and maintenance.

RQ5: Do the bugs of different DL compilers have commonality? Identifying the relationship among the bugs in different DL compilers is helpful to design general bug detection and localization

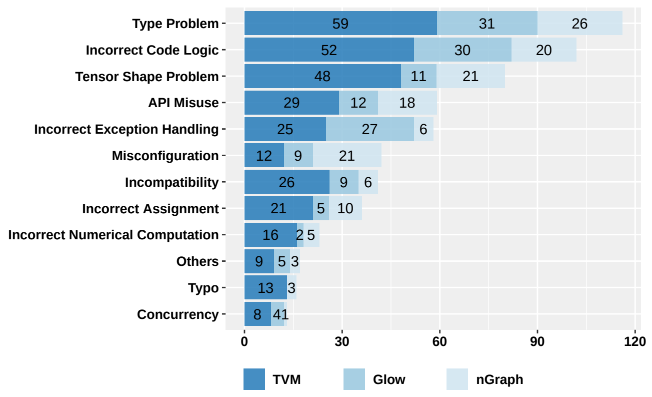


Figure 2: Bug Distribution by Root Causes

Table 2: Bug Distribution by Type Problem Subcategories

Compiler	Tensor Type	Conventional Type	Node Type
TVM	31	16	12
Glow	20	8	3
nGraph	11	10	5
Total	62	34	20

methods for DL compilers, even achieve the goal of cross-compiler bug detection.

4 RESULTS AND ANALYSIS

In this section, we present and discuss the experiment results for the five RQs.

4.1 RQ1: Root Causes

Figure 2 shows the number of DL compiler bugs by their root causes. From this figure, Type Problem is the most common root cause. It accounts for 116 bugs, including 59 in TVM, 31 in Glow, and 26 in nGraph. Table 2 further shows the bug distribution by subcategories. We find that Tensor Type Problem is the most common subcategory. Specifically, among 116 bugs in Type Problem, 62 bugs are caused by Tensor Type Problem, 34 bugs are caused by Conventional Data Type Problem, and 20 bugs are caused by Node Type Problem. This is because all the DL computations rely on one or more tensors and both IR and a large number of optimizations in DL compilers involve tensors. Moreover, tensor type is an important property in a tensor. Therefore, there are a large number of operations on type conversion (especially implicit type conversion) and type inference during compilation. The result indicates that handling types, especially tensor types, in DL compilers is very challenging and deserves more attention. Figure 3 shows an example of Tensor Type Problem bug[2], where PyTorch expects the output tensor type for the operator to be *float32*, but the corresponding Glow operator outputs *float16*, leading to type mismatch. Thus, the fix is to add an upcast operator to convert its output tensor type to *float32*.

[2]https://github.com/pytorch/glow/pull/4945.

```
-    return addValueMapping(outputs[0], EB->getResult());
+    if (is4Bit) {
+      auto *CT = F.createConvertTo(
+            "ConvertEmbeddingBag4BitRowwiseOffsetsOutput",
+            EB, ElemKind::FloatTy);
+       return addValueMapping(outputs[0], CT->getResult());
+    } else {
+       return addValueMapping(outputs[0], EB->getResult());
+    }
```

Figure 3: A *Tensor Type* Bug Example from Glow#4945

```
-    input_size = 1
-    for _, shape in enumerate(input_tensor_shape):
-        input_size *= shape
-    batch_size = int(input_size / weight_tensor_shape[1])
-    target_shape = tuple((batch_size, weight_tensor_shape[1]))
+    target_shape = tuple((-1, weight_tensor_shape[1]))
```

Figure 4: A *Tensor Shape* Bug Example from TVM#6038

Finding 1: Type Problem is the most common root cause, accounting for 19.23% of DL compiler bugs. Among these bugs, Tensor Type Problem bugs are the most common.

Incorrect Code Logic is the second most common root cause, accounting for 16.92% DL compiler bugs, including 55 bugs caused by Incorrect Optimization Code Logic and 47 bugs caused by Incorrect Non-optimization Code Logic. Due to the rapid development of deep learning, DL compilers are also frequently updated so as to incorporate the rapid advancement in DL algorithms and hardware. Moreover, for new features in DL frameworks and hardware, the implementation in DL compilers supporting the corresponding features tends to involve complex code logic. Therefore, that may lead to various technical debts occurring in the implementation of DL compilers. Optimization code tends to be more complex than the non-optimized one due to the complexity in the optimization algorithms and the handling of different kinds of hardware. As a result, the bugs arising from Incorrect Optimization Code outnumber those arising from Incorrect Non-optimization Code.

Finding 2: Code logic bugs are common due to the implementation complexity of DL compilers. These bugs are more often caused by Incorrect Optimization Code Logic than Incorrect Non-optimization Code Logic.

The third most common root cause is Tensor Shape Problem according to Figure 2. It accounts for 13.27% of bugs, including 48 in TVM, 11 in Glow, and 21 in nGraph. Figure 4 shows an example of Tensor Shape Problem bugs[3], where the *fully connected converter* used the shapes from the TFLite model to reshape the data tensor, but the TFLite model shapes do not reflect those provided by the *data_shape* parameter in *from_tflite()*. Thus, the fix is that

[3]https://github.com/apache/tvm/pull/6038.

Table 3: Distribution of API Misuse Bugs

Compiler	API M/R	Condition M/R	Wrong API			
			Receiver	Name	Args	*Total*
TVM	1	9	3	9	7	19
Glow	2	2	2	4	2	8
nGraph	2	5	1	5	5	11
Total	5	16	6	18	14	38

* M/R is short for Missing/Redundancy.

the reshape is always set to (-1, *n_units*) without needing to calculate a batch size for the particular operator. Similar to the Tensor Type Problem, this root cause also concerns the tensor computation in DL, and indeed tensor shape is also an important property in a tensor. The result indicates that tensor operations are error-prone regardless the operations are type-related or shape-related. Therefore, more care should be devoted to tensor operations in DL compilers.

Finding 3: Tensor Shape Problem is the third most common root cause, accounting for 13.27% of DL compiler bugs.

While the root causes (e.g., Tensor Type Problem, Tensor Shape Problem, and Incorrect Optimization Code Logic) specific to DL compilers are significant, some root causes common to DL compilers and traditional software systems are notable, e.g., API Misuse and Incompatibility. Therefore, we further analyzed whether they follow similar patterns or not by taking the two relatively frequent common root causes (i.e., API Misuse and Incompatibility) as the representatives.

Following the practice of existing work [8, 75], we divide the bugs caused by API Misuse into three subcategories as shown in Table 3. According to the existing studies on MuBench [7, 8], which is one of the most widely-studied benchmarks in the area of API misuse and contains 90 API misuses from real-world Java projects, API Missing/Redundancy is the most common cause but it is the least common in DL compilers. In DL compilers, the most common API misuses concern Wrong API, among which 15.79% are caused by wrong receivers, 47.37% are caused by wrong API names, and 36.84% are caused by wrong arguments. It suggests that developers may not often understand the correct API usage scenarios, especially when multiple similar APIs exist.

Table 4 presents the distribution of Incompatibility bugs by subcategories. The bug distribution differs significantly from that of DL program bugs [38]. For example, External Incompatibility (73.17%) accounts more often than Internal Incompatibility (14.63%) and Resource Incompatibility (12.2%) for DL compiler bugs, while Internal Incompatibility accounts most often for DL program bugs. The reason may be that the third-party libraries used in DL compilers are also frequently evolving (such as various DL frameworks, e.g., TensorFlow has nearly 50 commits per day on average and has already had 133 releases till February, 2021) and are relatively complex (such as various fundamental libraries relevant to systems, even hardware). Indeed, by analyzing the 30 External Incompatibility

Table 4: Distribution of Incompatibility Bugs

Compiler	Internal	External	Resource
TVM	3	19	4
Glow	2	6	1
nGraph	1	5	0
Total	6	30	5

Table 5: Bug Distribution by Symptoms

Compiler	Crash	WC	BP	Hang	BF	Unreported
TVM	207	76	6	4	13	12
Glow	85	35	2	1	14	8
nGraph	66	40	3	0	23	8
Total	358	151	11	5	50	28

*WC: Wrong Code *BP: Bad Performance *BF: Build Failure

bugs, 12 bugs are due to the incompatibility with DL frameworks and 7 bugs are due to the incompatibility with fundamental libraries relevant to systems.

Finding 4: API Misuse and Incompatibility bugs manifest in different ways between DL compilers and traditional software, calling for different bug detection strategies.

4.2 RQ2: Symptoms

Table 5 presents the distribution of DL compiler bugs by symptom categories. It shows that Crash is the most common symptom in all the three DL compilers. There are 207, 85, and 66 bugs that exhibit the Crash symptom in TVM, Glow, and nGraph, respectively. Detection of crashes does not require explicit test oracles. Therefore, the large percentage of crashes suggests the potentials of augmenting the existing test suite with the generated ones. Besides, it suggests the opportunities to design effective localization and de-duplicate methods based on the descriptive information collected from crashes.

Finding 5: Crash is the most common symptom of DL compiler bugs, accounting for 59.37% of bugs.

Wrong Code occurs when DL compilers generate incorrectly compiled code. It is a common symptom according to Table 5, accounting for 23.9% TVM bugs, 24.14% Glow bugs, and 28.57% nGraph bugs, respectively. The symptom is not as obvious as Crash. Specifically, the output of a DL compiler is an optimized model in some optimized IR. Determining the correctness of an optimized model or IR is hard due to its complexity. Therefore, testing such bugs is challenging because test oracles are difficult to define. Further, the adverse impact of this category of bugs is severe since the bugs could propagate to the models subsequently built using the buggy compiled code. To sum up, more attention should be paid to design effective testing methods for wrong code bugs from

```
-    operator int64_t() const { return static_cast<int64_t>(data_); }
+    operator int64_t() const {
+        return static_cast<int64_t>(fp16_ieee_to_fp32_value(data_));
+    }
     ......
-    operator int32_t() const { return static_cast<int32_t>(data_); }
+    operator int32_t() const {
+        return static_cast<int32_t>(fp16_ieee_to_fp32_value(data_));
+    }
```

Figure 5: A *Wong Code* Bug Example from Glow PR#4949

both academia and industry. Figure 5 shows a Wrong Code bug example[4], where Glow incorrectly calls `static_cast` on *float16* directly when casting *float16* to *int32* or *int64*, resulting in a wrong result as *float16* is defined using a proxy type *uint16_t*. Thus, the fix is that when casting *float16* to *int* type, it first converts it to *float32* and then conducts static cast.

> **Finding 6**: Wrong Code is the second most common symptom of DL compiler bugs, accounting for 25.04% of bugs. Generating high-quality test oracles for Wrong Code bugs is challenging.

According to Table 5, the symptoms of Hang and Bad Performance are uncommon, while the symptom of Build Failure is non-negligible. There are only 5 bugs with the Hang symptom and 11 bugs with Bad Performance symptom in total, while the percentage of bugs with the Build Failure symptom ranges from 4.09% to 16.43% across the three compilers. It indicates that configuring and building DL compilers require non-trivial efforts. In particular, Build Failure bugs occurs more often in nGraph (16.43%) than in TVM (4.09%) and Glow (9.66%). The reason may be that 1) nGraph (3,837) contains more LOC (lines of code) in configuration files than TVM (2,128) and Glow (3,222), and 2) both TVM and Glow provide more detailed configuration instructions than nGraph. Therefore, the provision of better-organized configuration options and more detailed configuration instructions in DL compilers, especially nGraph, is desired.

> **Finding 7**: Hang and Bad Performance are uncommon symptoms (0.83% and 1.82%), while Build Failure is a non-negligible symptom (8.29%) in DL compilers.

4.3 RQ3: Root Causes and Symptoms

Table 6 presents the number of DL compiler bugs caused by each root cause with each symptom. As the most common root cause, Type Problem occurs in almost all categories of symptoms (except Build Failure). Another root cause that can result in the same categories of symptoms is Incorrect Optimization Code Logic, which is also common. That is, the bugs caused by Type Problem and Incorrect Optimization Code Logic not only occur frequently, but also produce a wide variety of effects. Moreover, the two categories of bugs tend to be specific to DL compilers. Therefore, more attention from both DL compiler developers and researchers should be paid to detecting, localizing, and fixing these bugs.

The symptom of Build Failure is also non-negligible, but are not exhibited by the bugs induced by the common root causes. As shown in Table 6, Build Failure bugs are mainly caused by Incorrect Configuration and Incompatibility. These bugs are mostly caused by errors in configurations and dependencies. Our dataset of these bugs can facilitate the research on automated build failure fixing [35, 50, 51] and dependency conflict detection [25, 28] for DL compiler bugs.

> **Finding 8**: Type Problem and Incorrect Optimization Code Logic are common root causes of DL compiler bugs. They can induce all kinds of buggy symptoms except Build Failure, which is mostly exhibited by Incorrect Configuration and Incompatibility bugs.

From Table 6, Crash and Wrong Code are the most common symptoms of DL compiler bugs. Both can be induced by all categories of root causes except two (i.e., Incorrect Configuration and Typo for Wrong Code). Therefore, generating high-quality test oracles around the two symptoms can detect a wide variety of DL compiler bugs. As presented above, Crash has an obvious test oracle, while regarding Wrong Code, differential testing [32] may be a potentially promising direction, e.g., comparing the output results from the three DL compilers under the same set of test inputs.

> **Finding 9**: The symptoms of Crash and Wrong Code can be produced by various root causes of DL compiler bugs.

4.4 RQ4: Bug Prone Stages

Figure 6 presents the bug distribution among the three DL compilation stages. The model loading stage is relatively less buggy than the two IR transformation stages, which involve sophisticated IR optimization. Unlike TVM and Glow, the high-level IR transformation (73.28%) of nGraph is significantly more buggy than its low-level IR transformation (22.41%). This is because nGraph does not have its own low-level IR design. Instead, its low-level IR transformation integrates existing kernel libraries or coordinates with PlainML for hardware-specific optimizations as explained in Section 2. Through further analysis, Incorrect Code Logic (especially Incorrect Optimization Code Logic) is the most frequent root cause for both high-level IR transformation and low-level IR transformation stages. Due to a large number of DL compiler bugs induced by IR optimization, a variant of differential testing (i.e., Different Optimization Levels (DOL) method [14]) may be adapted to detect optimization-related bugs, which compares the results under different optimizations. Here, the characteristics of multi-level IR optimization in DL compilers should be considered.

[4]https://github.com/pytorch/glow/pull/4949.

Table 6: Relationship between Root Causes and Symptoms.

Root Causes \ Symptoms	Crash	Wrong Code	Bad Performance	Build Failure	Hang	Unreported	$Total_{symptom}$
Type Problem	75	29	1	3	1	7	116
Incorrect Code Logic	41	43	8	-	3	7	102
Tensor Shape Problem	54	24	1	-	-	1	80
API Misuse	38	19	1	-	-	1	59
Incorrect Exception Handling	51	6	-	-	1	-	58
Misconfiguration	1	-	-	40	-	1	42
Incompatibility	29	5	-	5	-	2	41
Incorrect Assignment	23	8	-	-	-	5	36
Incorrect Numerical Computation	11	11	-	-	-	1	23
Others	13	2	-	2	-	-	17
Typo	13	1	-	-	-	2	16
Concurrency	9	3	-	-	-	1	13
$Total_{causes}$	358	151	11	50	5	28	603

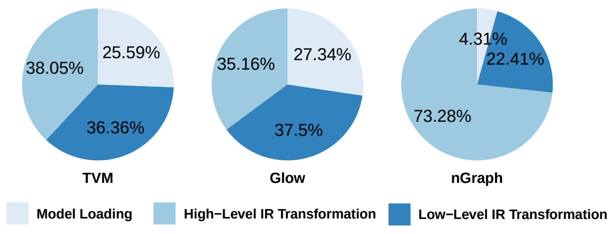


Figure 6: Bugs across Stages of the DL Compiler Pipeline

Finding 10: The high-level IR transformation stage (44.92%) and the low-level IR transformation stage (33.64%) are relatively more buggy than the model loading stage (21.44%) on average.

Figure 6 shows that a significant number of DL compiler bugs occur at the model loading stage, which plays a role analogous to the front end in traditional compilers. It differs from the prior findings [11, 14, 44, 60] that front-end bugs are rare in traditional compilers (e.g., GCC and LLVM). It is because the model loading stage, unlike traditional front ends, needs to handle inputs in multiple representations adopted by various DL frameworks. Moreover, the model loading stage needs to deal with various operators (e.g., convolution and pooling) supported by each DL framework when transforming an input model into high-level IR. In particular, different from stable front ends of traditional compilers, more and more DL operators can be proposed and developed, and thus DL compilers need to continually support them, which could incur new bugs at the model loading stage. As a result, the model loading stage requires handling a large number of cases arising from the data representations and operations supported by fast-evolving DL frameworks and libraries. We also analyze which DL frameworks are more often to induce bugs at the model loading stage. Specifically, through analyzing the front-end bugs in our dataset, there are 108 bugs for which we can identify the corresponding bug-inducing DL frameworks. We find that 29.63%, 31.48%, 21.3%, 4.63%, 4.63%, 3.7%, 2.78%, 1.85% of bugs arose from ONNX, PyTorch, TensorFlow, Keras, TFLite, MXNET, CoreML, and Caffe2 respectively. This indicates that it is important to consider inputs from different DL frameworks in designing tests for the model loading stage.

Finding 11: Unlike traditional compilers, a significant proportion of bugs (21.44% on average across the three DL compilers) occur at the model loading stage.

4.5 RQ5: Commonality

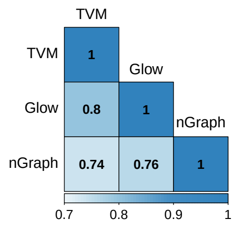
To measure the commonality across DL compilers, we calculated the Spearman correlation between each pair of DL compilers in terms of root cause distribution and symptom distribution. Spearman's correlation coefficient is a statistical measure of the strength of a monotonic relationship between two paired variables [72]. Figure 7 shows the correlation results. We find that all the correlation coefficients are larger than 0.74 in terms of root causes, and all the correlation coefficients are 1 in terms of symptoms. This demonstrates that the three DL compilers share a very high degree of commonality in the root causes and symptoms of bugs. It suggests the generalizability of the findings in this study as well as the feasibility to develop solutions that can be generalized to different DL compilers for the detection, localization, and fixing of DL compiler bugs.

Finding 12: The three DL compilers share significant commonality in the root causes and symptoms of bugs.

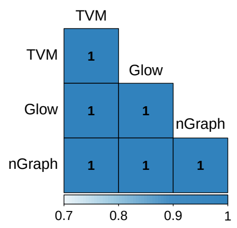
5 DISCUSSION

5.1 Implications

We discuss the implications on the detection and debugging of DL compiler bugs according to our findings.



(a) Root Causes Correlation



(b) Symptoms Correlation

Figure 7: Correlation among DL compilers

According to Findings 1 and 3, a large proportion of bugs are related to the type and shape of tensors in DL computation. Therefore, *new criteria can be defined to measure the test coverage of tensor type and tensor shape*. Existing mutation-based coverage [40] can be adapted to include representative mutation operators on tensor type and shape accordingly. For example, we find that DL compilers are more fragile in implicit type conversion, data layout transformation, and tensor shape matching. Therefore, new mutation operators should include 1) adding typecast for a tensor to promote implicit type conversion in tensor computations, 2) replacing a tensor with another one with a different layout to trigger data layout transformation, and 3) insert some new layers with diverse shapes to detect tensor shape mismatching. A large proportion of these bugs are manifested as crashes. These bugs can be detected if they are triggered. As such, new methodologies can be developed to search for effective tests based on the adapted coverage criteria.

According to Findings 5, 6, and 9, Crash and Wrong Code are the two most common bug symptoms that can be exhibited by DL compiler bugs due to all kinds of root causes. Even though detecting crash bugs has an obvious test oracle, the crash messages reported by many crash bugs are ambiguous. The ambiguity can affect the localization and fixing of bugs. Analyzing the crash bugs in our dataset, we find the crash messages of nearly 11% of bugs were rephrased to provide more information after their fixes. Moreover, we find some users complained that crash messages are not informative in their issue reports using the words such as "the error message is confusing" and "unhelpful error message". Therefore, *it is necessary for developers to provide precise and informative error messages*, which could facilitate the understanding of crash bugs and the design of automated localization methods.

Regarding wrong code bugs, it is difficult to define test oracles to detect them. To facilitate their detection, *one potential direction is to transform some wrong code bugs to crash bugs by adding more checks (e.g., assertions) in DL compilers*. That is, it is important for researchers to explore *where to check* and *what to check* in DL compilers. Even though this solution could relieve the efforts of detecting wrong code bugs to some degree, the task of adding assertions manually is non-trivial. Therefore, the design of high-quality oracle to detect wrong code bugs is still a challenging problem to the quality assurance of DL compilers. This problem deserves more attention from the software engineering community.

Findings 2, 8 and 10 suggest that optimization-related bugs occur frequently and can lead to almost all kinds of symptoms. Therefore, *designing tests that can trigger various optimizations and their combinations is a promising research direction*. In particular, the mechanism of multi-level optimizations should be considered when designing tests. Moreover, a large number of optimizations can be involved during the compilation of a DL model, and thus it is also useful to design automated localization techniques for identifying the buggy optimizations to improve the quality of DL compilers.

Finding 4 discusses the differences in the common root causes (i.e., API Misuse and Incompatibility) between DL compilers and traditional software systems. The differences show that the two categories of DL compiler bugs manifest in different ways from those of traditional software systems. It suggests that *different bug detection strategies should be designed for DL compilers*. Specifically, API misuses in DL compilers are mainly due to invoking the wrong APIs (64.41% of API Misuse bugs), but existing API-misuse detection techniques for general software mainly focus on API-missing/redundancy. Thus, incorporating API-recommendation knowledge to API-misuse detection for DL compilers is useful, which could help distinguish usage scenarios of different APIs (even though they have very similar names) by mining code snippets. For Incompatibility bugs, 73.17% of them are caused by External Incompatibility, especially the incompatibility with various DL frameworks and fundamental system libraries. Therefore, adding version checks for these third-party libraries is helpful to avoid many incompatibility bugs. Moreover, our dataset of External Incompatibility bugs may facilitate the research on dependency conflict detection for DL compilers. Dependency conflict detection has been an important topic in the software engineering community [25, 28].

As shown in Finding 11, front-end bugs in traditional compilers are rare, but the number of DL compiler bugs in the model loading stage (analogous to the front end) is non-trivial due to the diverse input representations from various DL frameworks. Moreover, different DL frameworks can cause DL compilers to make mistakes at model loading, as presented in Section 4.4. It suggests that this stage also deserves attention during the process of DL compiler bug detection. Meanwhile, *it is necessary to consider various DL frameworks when constructing tests.*

5.2 Threats to Validity

Internal Threat. The internal threat to validity mainly lies in our manual classification and labeling of DL compiler bugs, which may have subjective bias or errors. To reduce this threat, we referred to the previous taxonomies of root causes and bug symptoms [29, 38, 58, 62, 63] as the initial taxonomies, and then adopted an open-coding scheme to update the taxonomies to fit DL compiler bugs. During the labeling process, the first two authors independently labeled DL compiler bugs with the supervision of Cohen's Kappa coefficient [64]. Any disagreement was discussed with the third author until a consensus is reached. More details about our classification and labeling process can be found in Section 3.2.

External Threat. The external threat to validity mainly lies in the datasets used in our study. To reduce this threat, we systematically collected DL compiler bugs as presented in Section 3.1. In particular, we only considered closed and merged pull requests that are responsible to fix bugs, since the bugs involved in these pull requests have been fixed and accepted by developers. To guarantee the generalizability of our study, we used three popular and diverse DL compilers as subjects and studied 603 DL compiler bugs in total by balancing the effort of manual analysis and the study scale. Moreover, we conducted the analysis about commonality as presented in Section 4.5, whose results demonstrate the significant commonality in the root causes and symptoms of bugs across all the three DL compilers.

6 A PRELIMINARY APPLICATION

In this section, we demonstrate the usefulness of our findings with a preliminary proof-of-concept application **TVMfuzz**[5], which aims to generate unit tests for the stage of high-level IR transformation in TVM based on existing TVM tests. In the preliminary application, we selected TVM due to its great popularity. Specifically, TVMfuzz leverages two findings: (a) bugs arising from tensor type and tensor shape are common and (b) high-level IR transformation is the most error-prone DL compiler stage. First, TVMfuzz constructs a directed graph for the calling relationship of those TVM APIs involved in a set of existing unit tests for *the high-level IR transformation stage*. Second, it randomly selects a subgraph, and then constructs a new unit test by (1) considering legitimate API call dependencies in this subgraph and (2) randomly mutating the *tensor type*, *tensor shape*, *node name*, and *the primitive value of tensor elements*. TVMfuzz repeatedly performs the second step until the testing process terminates.

We applied TVMfuzz to TVM via differential testing and the test oracle is whether a test produces consistent results between two TVM versions. We conducted experiments to evaluate the effectiveness of TVMfuzz by producing new tests based on the existing tests in TVM v0.7, which is the latest release version of TVM. We deployed TVMfuzz on TVM v0.7 and v0.8dev (the latest version under development) for two days, which include the time to produce and execute the new tests. Finally, TVMfuzz detected 8 crash bugs (further confirming Finding 5 in our study), including 5 bugs in TVM v0.7 and 3 bugs in TVM v0.8dev. Regarding the 5 crash bugs in TVM v0.7, all of them cannot be detected by the existing tests in TVM v0.7. The 5 bugs have been fixed in the latest version under development v0.8dev. This demonstrates that the generated tests by TVMfuzz are able to effectively augment the existing test suite. Regarding the 3 crash bugs in TVM v0.8dev, after our manual inspection, we have submitted them to the TVM developers and are awaiting their responses. Please note that the false positives of TVMfuzz are rare, i.e., there is only one false positive in our experiment via our manual inspection and communication with TVM developers. Table 7 presents the details of the detected bugs by TVMfuzz. We found that 3 bugs are caused by Type Problem, 2 bugs are caused by Tensor Shape Problem, and 3 bugs are caused by Incorrect Exception Handling.

Although this is a preliminary application (a simple testing tool with short testing time), several bugs have been detected by TVMfuzz and these bugs cannot be detected by the original test suite in the corresponding TVM version. The results demonstrate the usefulness of our findings. Please note that the effectiveness of TVMfuzz-generated unit tests depends on the adequacy of existing unit tests. It is independent of data dimension and complexity as data are handled by the same API calls in existing unit tests.

[5]The implementation and results of TVMfuzz can also be found at our project homepage: https://github.com/ShenQingchao/DLCstudy.

Table 7: Bugs Detected by TVMfuzz

Version	Root Cause	Status
v0.7	Type Problem	Fixed
v0.7	Type Problem	Fixed
v0.7	Type Problem	Fixed
v0.7	Tensor Shape Problem	Fixed
v0.7	Tensor Shape Problem	Fixed
v0.8dev	Incorrect Exception Handling	Awaiting
v0.8dev	Incorrect Exception Handling	Awaiting
v0.8dev	Incorrect Exception Handling	Awaiting

7 RELATED WORK

Empirical Studies on Bugs. There are a number of empirical studies on bugs in the literature [26, 29, 34, 37–39, 48, 52, 60, 62, 65, 76, 78]. The most related ones to ours are the empirical studies on *traditional compiler bugs*. For example, Sun et al. [60] conducted an empirical study to analyze the duration, priority, fixes, test cases, and locations of GCC and LLVM bugs. Zhou et al. [78] conducted an empirical study to investigate the optimization bugs in GCC and LLVM. Also, there are some empirical studies on *DL program bugs*. For example, Islam et al. [38] and Zhang et al. [76] conducted empirical studies on DL program bugs, such as TensorFlow program bugs. Jia et al. [39] investigated the bugs inside the TensorFlow framework. Garcia et al. [29] conducted an empirical study on the bugs of autonomous vehicles (a kind of important DL-based applications). Humbatova et al. [37] studied the taxonomy of DL programs bugs, which includes layer-related and training-related categories. In addition, there are some work on DL testing, including DL model testing [19, 67], DL program testing [69, 77], and DL library testing [54, 66].

Furthermore, there are also many studies on bugs in other software systems. For example, Lu et al. [52] conducted an empirical study on concurrency bugs. Franco et al. [26] conducted an empirical study on numerical bugs. Han and Yu [34] studied the performance bugs of highly configurable software systems, and Wan et al. [65] characterized the bugs of blockchain systems.

Different from the previous studies, we target DL compiler bugs. As presented in Section 2, DL compilers have different characteristics from traditional compilers (e.g., GCC and LLVM). Moreover, DL compilers are one of the most important infrastructures for deep learning, which have different roles and characteristics with DL programs and DL frameworks. Specifically, DL compilers take trained DL models as inputs. Our taxonomy includes DL-compiler specific categories, e.g., node-type problem (DL compilers need determine node type before many operations) and incorrect optimization code logic (DL compilers enact specific optimizations customized to various hardware). To our best knowledge, we are the first to characterize DL compiler bugs.

Traditional Compiler Bugs. Besides the above mentioned empirical studies on traditional compiler bugs, there are a number of studies on testing and localizing traditional compiler bugs in the literature [12–18, 21, 23, 27, 31, 44, 45, 49, 55, 59, 71, 74] For example, Yang et al. [71] developed Csmith, the most widely-used C test program generation tool, to test C compilers. Le et al. [44] proposed EMI, which relies on a pair of equivalent test programs under a set of inputs, to test C compilers. Chen et al. [13] proposed DiWi to localize the buggy compiler file by generating a set of effective witness test programs. Also, Regehr et al. [55] proposed CReduce to reduce a bug-triggering test program to a minimal one that is still able to trigger the compiler bug.

Different from them, our work focuses on DL compilers rather than traditional compilers. Due to their significant differences (presented in Section 2), the existing testing and debugging methods in traditional compilers cannot be directly applied to DL compilers. That is, there is no work directly targeting DL compiler testing and debugging. Therefore, as the first empirical study on DL compiler bugs, we believe that our work is helpful to design effective testing and debugging methods specific to DL compilers. In particular, we have conducted the first attempt to design a simple TVM testing tool *TVMfuzz* according to our findings, and our preliminary study demonstrates the effectiveness of TVMfuzz by detecting 8 crash bugs that cannot be detected by existing tests in the corresponding TVM version.

8 CONCLUSION

DL compilers can significantly alleviate the burden of deploying and optimizing DL models programmed on top of various DL frameworks to various hardware, and have been become one of the most fundamental and important software infrastructures in DL. However, it is inevitable for them to have bugs, like traditional compilers. Their bugs could be propagated to DL models compiled by them and produce unexpected, even dangerous, model behaviors during the real model usage. Therefore, it is necessary to guarantee the quality of DL compilers. In this paper, we conducted the first comprehensive study to understand DL compiler bugs so as to promote the design of effective bug detection and debugging techniques. Specifically, we manually studied 603 bugs from three popular and diverse DL compilers (i.e., TVM, Glow, and nGraph), identified 12 root causes and 6 bug symptoms, and obtained 12 major findings. Based on these findings, we provide a series of guidelines for DL compiler bug detection and debugging in the future, and make the first attempt to design a simple but effective TVM testing tool (i.e., TVMfuzz). During the testing period of two days, it generates new tests based on existing tests in TVM and detects 8 TVM bugs that cannot be detected by TVM's original test suite, which can demonstrate the practical potentials of our findings.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant No. 62002256, 61932021), the National Key Research and Development Program of China (Grant No. 2019YFE0198100), Hong Kong RGC/RIF (Grant No. R5034-18), Hong Kong ITF (Grant No: MHP/055/19), Hong Kong PhD Fellowship Scheme, MSRA Collaborative Research Grant.

REFERENCES

[1] February 2021. Glow. https://ai.facebook.com/tools/glow/.
[2] February 2021. Keras. https://keras.io/.
[3] February 2021. nGraph. https://www.intel.com/content/www/us/en/artificial-intelligence/ngraph.html.
[4] February 2021. PyTorch. https://pytorch.org/.
[5] February 2021. TensorFlow. https://www.tensorflow.org/.
[6] February 2021. TVM. https://tvm.apache.org/.
[7] Sven Amann, Sarah Nadi, Hoan A Nguyen, Tien N Nguyen, and Mira Mezini. 2016. MUBench: A benchmark for API-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 464–467.
[8] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.
[9] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. 2021. Automated Query Reformulation for Efficient Search based on Query Logs From Stack Overflow. In *Proceedings of 43rd IEEE/ACM International Conference on Software Engineering*. IEEE, 1273–1285.
[10] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*. 2722–2730.
[11] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *Proceedings of 39th IEEE/ACM International Conference on Software Engineering*. 700–711.
[12] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test case prioritization for compilers: A text-vector based approach. In *2016 IEEE International Conference on Software Testing, Verification and Validation*. 266–277.
[13] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234.
[14] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*. 180–190.
[15] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced Compiler Bug Isolation via Memoized Search. In *Proceedings of 35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89.
[16] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.
[17] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-guided configuration diversification for compiler test-program generation. In *Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering*. 305–316.
[18] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2021. Coverage Prediction for Accelerating Compiler Testing. *IEEE Transactions on Software Engineering* 47, 2 (2021), 261–278.
[19] Junjie Chen, Zhuo Wu, Zan Wang, Hanmo You, Lingming Zhang, and Ming Yan. 2020. Practical Accuracy Estimation for Efficient Deep Neural Network Testing. *ACM Transactions on Software Engineering and Methodology* 29, 4 (2020), 30:1–30:35.
[20] Xiang Chen, Chunyang Chen, Dun Zhang, and Zhenchang Xing. 2019. Sethesaurus: Wordnet in software engineering. *IEEE Transactions on Software Engineering* (2019).
[21] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. 197–208.
[22] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).
[23] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 95–105.
[24] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. 2018. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058* (2018).
[25] Prasun Dewan and Rajesh Hegde. 2007. Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proceedings of the 10th European Conference on Computer-Supported Cooperative Work*. Springer, 159–178.
[26] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 509–519.
[27] Alastair F Donaldson, Hugues Evrard, and Paul Thomson. 2020. Putting Randomized Compiler Testing into Production (Experience Report). In *Proceedings*

of 34th European Conference on Object-Oriented Programming. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[28] W Keith Edwards. 1997. Flexible conflict detection and management in collaborative applications. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*. 139–148.

[29] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering*. 385–396.

[30] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial networks. *arXiv preprint arXiv:1406.2661* (2014).

[31] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 78–88.

[32] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and practices of differential testing. In *Proceedings of 41st IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 71–80.

[33] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 810–822.

[34] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 23:1–23:10.

[35] Foyzul Hassan and Xiaoyin Wang. 2018. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of 40th IEEE/ACM International Conference on Software Engineering*. IEEE, 1078–1089.

[36] Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. 2020. Intel Nervana Neural Network Processor-T (NNP-T) Fused Floating Point Many-Term Dot Product. In *Proceedings of IEEE 27th Symposium on Computer Arithmetic*. IEEE, 133–136.

[37] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering*. 1110–1121.

[38] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.

[39] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An Empirical Study on Bugs Inside TensorFlow. In *Proceedings of International Conference on Database Systems for Advanced Applications*. Springer, 604–620.

[40] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[41] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[42] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. 2016. Policy compression for aircraft collision avoidance systems. In *Proceedings of 2016 IEEE/AIAA 35th Digital Avionics Systems Conference*. 1–10.

[43] Adrian Kingsley-Hughes. 2017. Inside Apple's new A11 Bionic processor. *ZDNet, September* (2017).

[44] 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*. 216–226.

[45] 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*. 386–399.

[46] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[47] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The Deep Learning Compiler: A Comprehensive Survey. arXiv:2002.03794 [cs.DC]

[48] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. 25–33.

[49] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 65–76.

[50] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 43–54.

[51] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 617–628.

[52] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. 329–339.

[53] Amin Nikanjam, Mehdi Morovati, Foutse Khomh, and Houssem Ben Braiek. 2021. Faults in Deep Reinforcement Learning Programs: A Taxonomy and A Detection Approach. *arXiv preprint arXiv:2101.00135* (2021).

[54] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*. 1027–1038.

[55] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[56] 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]

[57] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.

[58] Forrest Shull, Sally Godfrey, Andre Bechtel, Raimund L Feldmann, Myrna Regardie, and Carolyn Seaman. 2008. Making Use of a Decade of Widely Varying Historical Data: SARP Project. (2008).

[59] 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*. 849–863.

[60] 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*. 294–305.

[61] Yi Sun, Yuheng Chen, Xiaogang Wang, and Xiaoou Tang. 2014. Deep learning face representation by joint identification-verification. In *Proceedings of Advances in neural information processing systems*. 1988–1996.

[62] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705.

[63] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *Proceedings of 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.

[64] Susana M Vieira, Uzay Kaymak, and João MC Sousa. 2010. Cohen's kappa coefficient as a performance measure for feature selection. In *Proceedings of International Conference on Fuzzy Systems*. IEEE, 1–8.

[65] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug characteristics in blockchain systems: a large-scale empirical study. In *Proceedings of 2017 IEEE/ACM 14th International Conference on Mining Software Repositories*. IEEE, 413–424.

[66] 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*. 788–799.

[67] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing Test Inputs for Deep Neural Networks via Mutation Analysis. In *Proceedings of 43rd IEEE/ACM International Conference on Software Engineering*. 397–409.

[68] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2020. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *arXiv preprint arXiv:2009.06520* (2020).

[69] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing Numerical Bugs in Deep Learning via Gradient Back-propagation. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. to appear.

[70] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised Log-based Anomaly Detection via Probabilistic Label Estimation. In *Proceedings of 43rd IEEE/ACM International Conference on Software Engineering*. 1448–1460.

[71] 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*. 283–294.

[72] Jerrold H Zar. 2005. Spearman rank correlation. *Encyclopedia of biostatistics* 7 (2005).

[73] Tim Zerrell and Jeremy Bruestle. 2019. Stripe: Tensor compilation via the nested polyhedral model. *arXiv preprint arXiv:1903.06498* (2019).

[74] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN*

Conference on Programming Language Design and Implementation. 347–361.

[75] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. In *Proceedings of 40th IEEE/ACM International Conference on Software Engineering*. IEEE, 886–896.

[76] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.

[77] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 826–837.

[78] 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.