2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

A Tale of Two DL Cities: When Library Tests Meet Compiler

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

Yongqiang Tian
The Hong Kong University of Science and Technology
Hong Kong, China
yqtian@ust.hk

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

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

Lili Huang
College of Intelligence and Computing, Tianjin University
Tianjin, China
huangll@tju.edu.cn

Ruifeng Fu
College of Intelligence and Computing, Tianjin University
Tianjin, China
frf2000@tju.edu.cn

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

Zan Wang
College of Intelligence and Computing, Tianjin University
Tianjin, China
wangzan@tju.edu.cn

***Abstract*—Deep Learning (DL) compilers typically load a DL model and optimize it with intermediate representation. Existing DL compiler testing techniques mainly focus on model optimization stages, but rarely explore bug detection at the model loading stage. Effectively testing the model loading stage requires covering diverse usages of each DL operator from various DL libraries, which shares a common objective with DL library testing, indicating that the embedded knowledge in DL library tests is beneficial for testing the model loading stage of DL compilers. With this idea, we propose OPERA to migrate the knowledge embedded in DL library tests to test the model loading stage. OPERA constructs diverse tests from various tests for DL libraries (including the tests documented in DL libraries and those generated by recent fuzzers). In total, we considered three sources of tests in DL libraries for migration. In addition, it incorporates a diversity-based test prioritization strategy to migrate and execute those tests that are more likely to detect diverse bugs earlier. We then used eight frontends from three DL compilers (e.g., TVM, TensorRT, and OpenVINO) for evaluation. OPERA detected** 170 **previously unknown bugs in total,** 90 **of which have been confirmed/fixed by developers, demonstrating the effectiveness of such the migration-based idea. The test prioritization strategy in OPERA improves testing efficiency with migrated tests by 11.9%∼47.4% on average compared to general test prioritization strategies.**

***Index Terms*—Compiler Testing, Test Migration, Test Prioritization, Deep Learning Compiler**

I. INTRODUCTION

Deep Learning (DL) compilers (e.g., TVM [1], TensorRT [2], and OpenVINO [3]) are widely utilized to optimize the performance of DL models for deployment on various hardware devices. The compilation process of a DL model typically involves three main stages [4]: ① Loading the DL model, prepared under a specific DL library (e.g., PyTorch [5] or Keras [6]), into its equivalent high-level intermediate representation (IR); ② Performing hardware-independent optimizations on the high-level IR; ③ Lowering the high-level IR into the low-level IR and conducting hardware-specific optimizations to generate code targeting specific hardware.

Similar to traditional compilers [7]–[13], DL compilers also contain bugs, which can compromise the reliability of both the compilers themselves and the models they produce. As reported [4], each stage of DL compilers contains a significant number of bugs, underscoring the need for comprehensive testing to ensure the quality of DL compilers. However, existing DL compiler testing techniques primarily focus on the two optimization stages (② and ③) mentioned earlier, neglecting the model loading stage (①). Specifically, recent DL compiler testing techniques (such as HirGen [14], Tzer [15], and TVMFuzz [16]) mainly construct tests at either high-level or low-level IRs, bypassing the model loading stage.

NNSmith, a state-of-the-art grammar-based technique [17], is designed to construct DL models for testing various stages of DL compilers. However, it primarily focuses on stress testing for optimizations by generating complicated models. In contrast, the model loading stage involves converting each operator in a model into its equivalent high-level IR individually. Therefore, effectively testing the model loading stage requires covering diverse usages of each DL operator from various DL libraries, rather than focusing on complex dependencies among operators. Furthermore, NNSmith is limited to constructing DL models solely under the ONNX library and supports a limited number of operators, making it ineffective for testing the model loading stage.

Intuitively, manually developing test generation tools that follow the corresponding grammars may meet the test requirements for the model loading stage. However, this method can be labor-intensive and error-prone due to the extensive number of DL libraries and their supported operators. Additionally, operators often involve numerous parameters, leading to complex constraints that further aggregate the difficulty of developing such tools. This highlights the need for alternative and lightweight methods to address this challenging task.

†Junjie Chen is the corresponding author

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00025

By analyzing source code, test cases, and bugs of DL compilers, we found that: (1) Testing the model loading stage of DL compilers is related to testing DL libraries. Specifically, DL compilers typically accept DL models composed of operators supported by specific DL libraries as inputs. Both the testing of the model loading stage and DL libraries share a common objective, which is to ensure the correctness of operators under various usages. While there may not be complete overlap between the corner usages of each operator for DL compilers and DL libraries, the embedded knowledge in DL library tests could potentially be beneficial for testing the model loading stage of DL compilers. (2) A few tests in DL compilers are designed with inspiration from tests documented in the ONNX library, as indicated by the comments accompanying these tests. This suggests the feasibility of leveraging the knowledge embedded in DL library tests to enhance the testing of the model loading stage to some extent.

However, due to the separate development of communities for DL compiler testing and DL library testing, there has been no systematic study to investigate the feasibility of migrating the knowledge embedded in DL library tests for testing the model loading stage. Hence, we performed the first exploration on the potential of the migration-based idea. Specifically, we design a migration-based technique, called OPERA (OPERator Adapter), to test DL compilers (especially the model loading stage) by considering three sources of tests in DL libraries for migration, i.e., tests documented in DL libraries and tests generated by two recent fuzzers (DocTer [18] and DeepREL [19]).

In fact, the direct adoption of most DL library tests for testing the model loading stage of DL compilers is not feasible due to differences in their input formats. Specifically, while DL compiler tests rely on pre-constructed DL models, DL library tests typically involve subtasks (e.g., gradient calculation and model design) in model construction, many of which cannot be represented in the form of DL models. To address this challenge, OPERA first extracts instances of DL operators from each DL library test via code instrumentation and then packages each operator instance to a model (as a migrated test for DL compilers) with the aid of model generation templates for different DL libraries. Each operator instance represents a specific usage of the operator, encompassing an operator API and its corresponding parameter settings.

Another challenge that OPERA encounters is the significant cost consideration. This is primarily due to two factors: (1) the substantial volume of migrated tests originating from various migration sources, and (2) the frequent migration and execution of tests resulting from the frequent evolution of both DL libraries and DL compilers. To address this challenge, the component of test prioritization is designed in OPERA, which prioritizes the migration and execution of tests that are more likely to uncover a diverse range of bugs in the model loading stage. After prioritization, more bugs can be detected within any given time budgets, thereby enhancing the overall test efficiency. The test prioritization component takes into account the diversity of operator instances.

In this work, we applied OPERA to test three popular DL compilers (i.e., TVM [1], TensorRT [2], and OpenVINO [3]). To balance evaluation cost and conclusion generality, for each compiler, we chose several popular DL libraries, i.e., PyTorch, Keras, and ONNX, from its supported frontends (responsible for model loading). In total, our study covered eight frontends across three DL compilers. In total, OPERA detects 170 previously unknown bugs by migrating the knowledge embedded in DL library tests, 90 of which have been confirmed/fixed, while the state-of-the-art grammar-based DL compiler testing technique (NNSmith) detects only 18 bugs within the same time budget. The results demonstrate the effectiveness of such a migration-based idea for testing the model loading stage of DL compilers. Furthermore, the test efficiency can be largely improved with the test prioritization component in OPERA. On average across eight subjects, it improves OPERA without special test prioritization by 13.1% and improves OPERA incorporating the widely-used test prioritization strategies in general software testing by 11.9%$\sim$47.4% in terms of APFD (Average Percentage of Faults Detected) [20].

This work makes the following major contributions:

- We introduced the idea of migrating knowledge from DL library tests to enhance the testing of the model loading stage in DL compilers.
- We designed a migration-based technique (OPERA), which integrates various migration sources (i.e., tests documented in DL libraries and those generated by recent fuzzers), along with diversity-based test prioritization.
- We conducted an extensive study to evaluate OPERA across eight frontends from three DL compilers, leading to the efficient detection of 170 previously unknown bugs.
- We released the OPERA implementation and all experimental data for replication and future use, accessible at: **https://github.com/ShenQingchao/OPERA**.

II. MOTIVATION

At the model loading stage, DL compilers take as input DL models built from various DL libraries, e.g., PyTorch, and convert them into a unified high-level IR. The model constructed by a specific DL library is a computational graph with diverse DL operators. This high-level IR, known as graph-level IR, helps hide the differences in DL models from various DL libraries, simplifying optimization execution. Each operator in the DL model is converted into semantically equivalent one or more IR expressions. For example, a `Conv2D` operator in a Keras model, along with all parameter settings (e.g., filters), is converted to `nn.conv2d` in the high-level IR of TVM.

The behavior of operators can be customized by input parameters. For example, Figure 1(a) shows the definition of the `Conv2DTranspose` operator in Keras. It includes two required parameters, `filters` and `kernel_size`, and 14 optional parameters (e.g., strides) with default values. The value selection of each parameter may affect the calculation result of a model involving the operator. To guarantee the correctness of a DL library, numerous tests containing diverse operator instances with various parameter values were constructed for DL library testing [18]. For example, to test

```
keras.layers.Conv2DTranspose(
    filters,
    kernel_size,
    strides=(1, 1),
    padding="valid",
    output_padding=None,
    data_format=None,
    dilation_rate=(1, 1),
    activation=None,
    use_bias=True,
    bias_initializer="zeros",
    kernel_regularizer=None,
    ...
)
```

(a) `Conv2DTranspose` definition in Keras

```
class Conv2dTransposeTest(TestCase):
  def _run_test(self, kwargs):
      test_utils.layer_test(
          Conv2DTranspose,
          kwargs=kwargs,
          input_shape=(2, 3, 7, 6))
...
@parameterized.named_params(
  ({ "strides": (2, 2),
     "output_padding": (1, 1) }))
def test_conv2d_transpose (self):
  kwargs = {"filters": 3,
            "kernel_size": 3, ...}
  ly = Conv2DTranspose(**kwargs)
...
```

(b) A test for `Conv2DTranspose` in the test suite of Keras

Fig. 1. A motivating example with `Conv2DTranspose`

```
def _convert_convolution(inexpr, keras_layer, etab, data_layout):
...
+ if is_deconv and keras_layer.output_padding:
+     params["output_padding"] = keras_layer.output_padding
...
```

Fig. 2. Patch for a real bug on `Conv2DTranspose` in TVM

the correctness of the `Conv2DTranspose` operator, Keras developers prepared 84 tests.

In the model loading process, DL compilers need the ability to handle various usages of these operators, including all the combinations of parameters, for correct transformation from DL operators to high-level IR. Hence, the model loading stage of DL compilers actually shares a similar test objective with the tests for these operators in DL libraries. This motivates the idea of migrating knowledge embedded in DL library tests to test the model loading stage.

Figure 2 shows a real bug [21] of `Conv2DTranspose` in TVM triggered by a test migrated from Keras testing. TVM overlooks the parameter `output_padding` when converting the operator `Conv2DTranspose`, which leads to incorrect output shape when the `out_padding` is not set to the default value (i.e., `None`). Before this bug was reported, the developer-provided tests of TVM only consisted of two operator instances to test the conversion of `Conv2DTranspose` from the DL operator to high-level IR. Moreover, in the two instances, all the optional parameters use the default values and thus are ineffective in detecting the bugs that require other, non-default parameter settings.

The tests migrated from DL library testing can help detect these cases. The test with non-default `out_padding` is absent in the tests of DL compilers, but available in the tests of Keras. In Figure 1(b), the test `Conv2dTransposeTest` from Keras, which assigns `(1, 1)` to `out_padding`, matches the bug-triggering condition for this bug, and thus can help reveal it. By migrating the Keras test to the test in the input format for TVM with this operator instance, OPERA successfully detected this previously unknown bug. After this bug was reported, it was fixed by adding the analysis on the parameter `output_padding` in the `_convert_convolution` function of TVM.

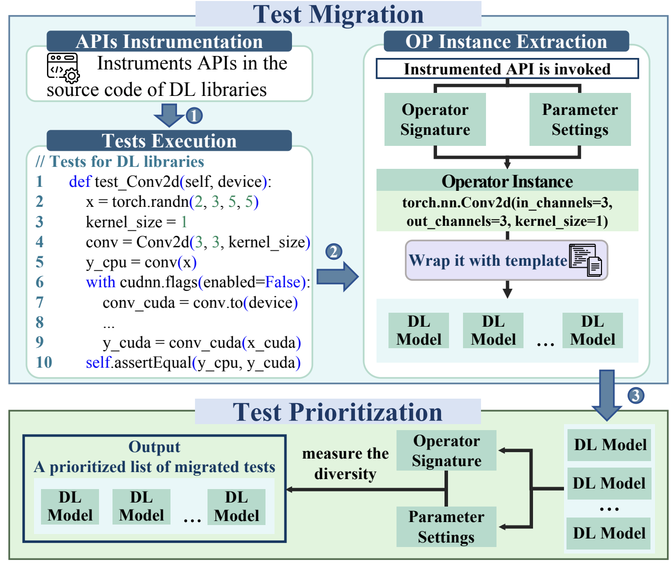

Fig. 3. Workflow of OPERA

It is non-trivial for those existing DL compiler testing techniques to detect this bug. For grammar-based techniques like NNSmith [17], developers need to manually prepare the grammar to support `Conv2DTranspose`. As operators in DL compilers usually have complicated logic and vast space of parameters, supporting this operator that can match the bug-triggering condition, requires extensive expert knowledge and costs. For mutation-based techniques [15], detecting this bug requires finding a seed test from a seed pool and applying a set of effective mutation operators, which is also non-trivial due to large search space. Therefore, in this work, we explore an alternative and lightweight method (i.e., migration-based idea) for this challenging task.

III. APPROACH

With the migration-based idea, we propose a technique, called OPERA. The workflow of OPERA is shown in Figure 3, which contains two main components: test migration and test prioritization. Specifically, OPERA first creates tests for the model loading stage by migrating knowledge embedded in DL library tests via operator instance extraction (Section III-A). Due to the large number of migrated tests and the requirement of frequent test migration and execution caused by the evolution of DL libraries and compilers, OPERA then prioritizes migrated tests based on their diversity in order to improve testing efficiency (Section III-B). Finally, OPERA incorporates two test oracles to determine whether a migrated test detects a bug in the model loading stage of a DL compiler (Section III-C).

A. Test Migration

1) Migration Sources: In OPERA, we considered both human-written tests and tool-generated tests in DL library testing as our migration sources. Human-written tests imply expert knowledge for considering various usages of each operator under the corresponding constraints. Tool-generated tests, on the other hand, can help explore corner cases. In the literature, there are many DL library testing techniques [18], [19], [22] By balancing evaluation cost and conclusion generalizability, we selected two state-of-the-art but diverse techniques (DocTer [18] and DeepREL [19]) for supporting the migration source of tool-generated tests. DocTer extracts API constraints from official documentation and then utilizes these constraints to generate tests. DeepREL infers potential API relations automatically based on API syntactic and semantic information and then synthesizes tests for invoking relational APIs. In theory, OPERA is generalizable to various DL library testing techniques and we will investigate more sources of tool-generated tests in the future.

2) Operator Instance Extraction: Although there are massive tests from the three sources, However, most DL library tests cannot be directly adopted to test DL compilers due to differences in their input format. Specifically, DL compilers take DL models as inputs, while the tests for DL libraries are often in the format of Python code, most of which lack a complete model structure, as shown in Figure 1(b). Hence, extracting DL models from DL library tests for DL compiler testing is non-trivial.

To achieve the goal of creating tests for the model loading stage of DL compilers by migrating knowledge embedded in DL library tests, OPERA uses *operator instances* to bridge the migration gap. An operator instance refers to a specific usage of an operator with a specific setting of its parameters (an example is shown in the Figure 3). They can be extracted from the tests for DL libraries and converted to DL models composed of a single layer for testing DL compilers. We call such DL models *single-operator models*. Note that multiple operator instances can be extracted from one DL library test, leading to obtaining a set of single-operator models (that is, migrated tests) for DL compiler testing.

Note that the primary functionality of the model loading stage lies in converting each operator in a DL model individually into an equivalent IR, known as single-operator equivalence conversion. As a result, OPERA naturally creates single-operator models for testing. Also, single-operator models facilitate follow-up bug de-duplication and localization.

Specifically, OPERA instruments APIs in the source code of DL libraries for operator instance extraction. When an instrumented API is invoked, the operator signature and its corresponding parameter values can be recorded, which collectively form an operator instance. This operator instance is then wrapped using a template as a DL model for testing the model loading stage. Due to different DL libraries having different model construction methods, we design a model generation template for each DL library to facilitate wrapping the corresponding operator instances. For instance, the template for PyTorch is shown in Figure 4, which takes an operator instance as input (Line 5) and encapsulates it in a model structure (Lines 2-5). An instance of the PyTorch model is then created

```
1 model_template =
2     f"{params_list_declare_str}"
3     + f"class Model(torch.nn.Module):\n"
4     + f"    def forward(self, *args):\n"
5     + f"        return {op_signature}(args[0], {params_settings_str})\n"
6     + f"model = Model().eval()\n"
7     + f"input_data=para_0\n"
8     + f"trace = torch.jit.trace(model, input_data)\n"
```

Fig. 4. Template for generating DL models under PyTorch

and set to evaluation mode (Line 6). Finally, with the aid of input data, the model is serialized into deployable code (i.e., TorchScript), which can be used as the test of DL compilers. We show the used template for each DL library at our project homepage due to the space limit.

B. Test Prioritization

As explained in Section I, the practicality of such a migration-based idea may still be hindered by significant cost consideration. We thus incorporate test prioritization into OPERA to detect more bugs within a given testing time budget, which facilitates investigating the efficiency improvement of this migration-based idea. According to the characteristics of our scenario, we design a diversity-based test prioritization strategy in OPERA. A migrated test is a single-operator model converted from an operator instance, and thus its core semantics lie in (1) the signature of the operator and (2) the setting of each parameter in the operator. Hence, OPERA prioritizes the set of migrated tests according to the diversity among them in terms of the two-dimensional information.

1) Diversity of Operator Signatures: The tests with different operator signatures mean that they have diverse semantics. However, how to determine the order of the tests with different operator signatures is still a challenge. In OPERA, we address it according to the following intuitions: (1) the number of tests with an operator signature is large in DL library testing, indicating that this operator receives more attention potentially due to its complicated implementation logic, more corner cases in it, etc; (2) the number of tests with an operator signature is small in the test suite equipped by the DL compiler (i.e., the test suite specific to the model loading stage), indicating that DL compiler developers still pay little attention to testing the transformation of this operator. Hence, OPERA assigns an operator OP_i a higher priority score if it occurs in the set of migrated tests for a DL library more frequently (denoted the occurrence times as $Num\_DLL\_OP_i$) but occurs in the test suite equipped by the DL compiler more rarely (denoted the occurrence times as $Num\_DLC\_OP_i$). With the intuitions, the priority score of OP_i is calculated by the ratio of $Num\_DLL\_OP_i$ over $Num\_DLC\_OP_i$.

2) Diversity of Parameter Settings: With the diversity of operator signatures, the tests with the same operator signature have the same priority, and thus how to further determine their priority is another challenge. OPERA addresses it by measuring the diversity of parameter settings for each operator. Inspired by the theory of equivalence class partitioning [23], OPERA partitions the value space of each parameter into a set of subspaces, each of which clusters the values with potentially similar testing capabilities, by performing different considerations on different types of parameters.

If the value space of a parameter is *a limited set of concrete values*, Opera treats each unique value as a unique subspace as each of them may represent a unique configuration of using this operator. If the value space of a parameter is *a range of Integers*, Opera pays more attention to some special values in DL [24]–[26], i.e., -1 (e.g., it can be used to represent the automatically-calculated dimension size in tensor shape), 0 (e.g., it can mean that no padding operation is performed when assigned to the parameter `padding`), and 1 (e.g., it can be a boundary value for the parameter `scale` in `keras.layers.Rescaling`). Hence, Opera partitions them into five sets of subspaces, i.e., $(-\infty, -2], [-1], [0], [1]$, and $[2, \infty)$. Similarly, Opera partitions the value space of *a range of Floating number* for a parameter into three sets of subspaces, i.e., $(-\infty, 0), [0]$, and $(0, \infty)$.

If a parameter is a *tensor*, which is a compound type with some typical attributes (i.e., tensor type and shape), it first partitions each tensor-type value as a unique subspace as the value space of tensor type is a limited set of concrete values. Then, it partitions the value space of tensor shape (which belongs to the List type). The type of value in the List is Integer, and thus we use the Integer space partition method for it. The size of the List refers to the dimension of the tensor and Opera considers some special values [27]. Specifically, the dimension of the tensor with batch size ranging from zero to five can be used to represent none, scalar, vector, matrix, color image (e.g., [batch, channel, height, width]), and video (e.g., [batch, channel, depth, height, width]), respectively. Therefore, the value space of the dimension is divided into seven sets of subspaces, i.e., $[0], [1], [2], [3], [4], [5]$, and $[6, \infty)$. Finally, Opera intersects the subspace partitioned by each attribute and thus forms the final set of subspaces for tensors.

For an operator signature, Opera measures the diversity score of the parameter setting for an operator instance with those of already-prioritized operator instances based on partitioned value subspaces. Specifically, for the operator instance, Opera measures the percentage of parameters or pair-wise parameter combinations, whose values cover new subspaces or pair-wise subspaces over the set of already-prioritized operator instances, inspired by combinatorial testing [28].

3) Overall Prioritization: With the two-dimensional diversity, Opera prioritizes tests based on the product of the diversity score of the operator signature and that of the parameter setting, which is called operator-instance diversity. This calculation method may not be optimal in our scenario, and we will explore more methods in the future. After adding the test with the maximum operator-instance diversity to the prioritized result, Opera updates the parameter-setting diversity of the remaining tests with the same operator signature for subsequent iterations. The prioritization process in Opera is based on the Heapsort algorithm [29] due to its high efficiency ($O(nlogn)$ time complexity and $O(1)$ space complexity).

C. Test Oracles

A recent study [4] revealed that most of DL compiler bugs manifested as either compiler crashes or inference inconsistencies between the original DL models and the corresponding compiled models. We thus implemented the **two test oracles** for testing the model loading stage with migrated tests.

Crash has been widely used in compiler testing [17], [30], which refers to an unexpected termination of the compilation process. To avoid crashing a DL compiler due to invalid tests, Opera uses the DL library from which the tests are migrated to check their validity in advance. If the DL library crashes when executing a test, this test is considered invalid and thus discarded before testing the DL compiler. In addition, the crashes that produce error messages like “unsupported type” and “unsupported operator” are disregarded as the unsupported features are often not treated as bugs.

Inference inconsistency means that the inference results of a test (a DL model) and its corresponding compiled model via the DL compiler under test are inconsistent [14]. Same as the existing work [14], [17], we measured the inference difference between them based on Chebyshev distance [31] and used 1e-3 as the threshold to determine whether the inference results are inconsistent. As the DL compiler aims to achieve equivalent transformation for any DL model, if the obtained inference results from them are inconsistent, it indicates that a DL compiler bug is found.

IV. EVALUATION SETUP

Our evaluation aims to study two research questions:

- **RQ1**: To what extent can Opera effectively detect bugs at the model loading stage of DL compilers?
- **RQ2**: To what extent can the test prioritization component enable Opera to detect bugs earlier?

A. Subjects

Following recent work [14], [15], [17], [32], we performed our study on three widely-studied DL compilers, including TVM [1], TensorRT [2], and OpenVINO [3]. We chose the latest versions of them (i.e., TVM v0.13, TensorRT v8.6, and OpenVINO v2023.1.0), which is helpful to answer these RQs more sufficiently by detecting previously unknown bugs. A DL compiler usually contains multiple frontends for model loading, each of which converts DL models under a specific DL library into high-level IRs. To allow for in-depth analysis, our evaluation focuses on the frontends of popular DL libraries, including PyTorch frontend, Keras frontend, and ONNX frontend. Each of them handles the models under libraries that are widely used in both research and industry [24], [26], [33], i.e., PyTorch, Keras, and ONNX libraries, respectively. In particular, we use Keras instead of Tensorflow as Keras is a high-level and easy-to-use interface of TensorFlow, and many DL models constructed by TensorFlow are saved in the Keras format for platform compatibility and portability at the deployment stage. Further, as TensorRT does not support the Keras frontend, our evaluation covers eight frontends from three DL compilers in total.

B. Baselines

We assessed the effectiveness of the migration-based idea with OPERA by comparing it with **NNSmith** [17] and **COMET** [24]. Both produce DL models with multiple operators for testing, which facilitates comparison analysis with single-operator models obtained by OPERA.

NNSmith is the state-of-the-art DL compiler testing technique, which is a grammar-based technique and can test the model loading stage as well. Specifically, it randomly generates DL models from scratch by supporting 75 operators of the ONNX library, which indicates that we can just study NNSmith on the ONNX frontend. Even though most of DL library tests cannot be directly adopted to test the model loading stage, we have to use OPERA to support the migration. There are also some DL library testing techniques that can directly generate DL models to satisfy the input format of DL compilers. We also studied such a state-of-the-art technique, i.e., COMET, which designs a set of mutation operators and a coverage-based search-based algorithm to generate diverse models for testing DL libraries.

In RQ2, we investigated the efficiency improvement of such the migration-based idea by evaluating the effectiveness of the test prioritization strategy in OPERA. Here, we considered some test prioritization strategies commonly used in general software testing for comparisons.

- **Random**. The migrated tests are randomly ordered, serving as the baseline without special test prioritization.
- **FAST** [34], which treats each test as a string, and adopts the data mining algorithms (i.e., minhashing and locality-sensitive hashing algorithms [35]) to accelerate the process of finding diverse tests by converting each string to a k-shingle (the set of its substrings of length k).
- **Total**-coverage-based prioritization, which prioritizes migrated tests based on the number of program elements (in the frontend under test) covered by each test. Here, we used statements as the representative program elements following existing work [36]–[39].
- **Additional**-coverage-based prioritization, which prioritizes migrated tests based on the number of covered statements that are not covered by the existing prioritized ones.

FAST is the state-of-the-art black-box strategy, while coverage-based prioritization is the most widely studied white-box strategy. The prioritization strategy in OPERA and the random strategy are black-box. For coverage-based strategies, Coverage.py [40] is used to collect statement coverage in frontends. For FAST, we re-used its released implementation [34].

C. Metrics

We counted **the number of detected bugs** as the metric of evaluating the test effectiveness. During the testing process, it is possible that some of the test failures are triggered by the same root cause. Hence, it is important to de-duplicate them and count the number of unique bugs.

In DL compiler, each operator in the model loading stage comprises a conversion function that is responsible for converting it into the equivalent high-level IR. As the migrated test is a single-operator model, it is convenient to determine the conversion function responsible to the operator in a failure-triggering test. Therefore, based on the identified conversion function for each test failure, we de-duplicated the test failures to obtain unique bugs. Here, we did not use the operator in each failure-triggering test for de-duplication, as different operators may be handled by the same conversion function in the DL compilers. For example, `AveragePooling2D` and `MaxPooling2D` are two different operators in the Keras library, but they are converted by the same function (i.e., `_convert_pooling`) in TVM. As the tests generated by NNSmith and COMET are DL models with multiple operators, we manually de-duplicated their test failures following the original papers [17], [24].

All bugs are detected on the latest versions of DL compilers, and thus we created a bug report for each unique bug and then submitted it to project maintainers. We counted the number of confirmed or fixed bugs by developers. Based on the feedback from developers, all of our submitted bugs that have been confirmed are unique. This indicates the accuracy of our de-duplication method.

Besides, OPERA includes a test prioritization component to improve the testing efficiency, and thus it is also important to investigate the testing efficiency of each technique. Here, we used two metrics to evaluate the effectiveness of each prioritization strategy. First, we measured **the time spent on detecting each bug**. The shorter the time is, the more effective the strategy is. Second, We adopted the widely-used metric of evaluating test prioritization, i.e., **APFD** (Average Percentage of Faults Detected) [20], to compare various test prioritization strategies following many existing studies [38], [41]–[43]. The calculation of APFD is shown in Formula 1:

$$APFD = 1 - \frac{\sum_{i=1}^{m}(p_i)}{n \cdot m} + \frac{1}{2n} \tag{1}$$

where m is the total number of detected bugs, n is the total number of tests, p_i is the rank of the first test in the prioritized result that detects the i^{th} bugs. A larger APFD value indicates a more effective strategy.

D. Implementations

We collected the test suite equipped by PyTorch v1.7, Keras v2.3, and ONNX 1.8 as the migration source of human-written tests for the corresponding frontends of the three DL compilers, respectively. We collected 32,378, 20,992, and 1,014 tests from the three test suites, respectively. We used the implementations of DocTer and DeepREL released by their works [18], [19]. As neither of them supports test generation for the ONNX library, we exclude them when using OPERA to test the ONNX frontend in DL compilers. For the PyTorch and Keras frontends, we used DocTer and DeepREL to generate the same number of tests as the corresponding human-written tests respectively, which can help compare the three migration sources fairly. All experiments were conducted on an Ubuntu 18.04 server with Intel Xeon CPU, NVIDIA GTX1080Ti GPU, and 128G RAM.

TABLE I
NUMBER OF BUGS DETECTED BY OPERA. "–" MEANS NOT APPLICABLE SINCE TENSORRT DOES NOT SUPPORT KERAS MODELS.

Frontend	Status	TVM	TensorRT	OpenVINO	Total
PyTorch	Fixed	9	0	7	16
	Confirmed	0	6	10	16
	Awaiting	21	25	7	53
Keras	Fixed	20	–	6	26
	Confirmed	5	–	2	7
	Awaiting	10	–	3	13
ONNX	Fixed	2	4	2	8
	Confirmed	5	5	7	17
	Awaiting	7	3	4	14
Total		79	43	48	170

E. Process

For each studied frontend, we obtained a set of migrated sets from the three migration sources with the aid of OPERA. We then tested each frontend with these migrated tests and recorded whether a test triggered a failure or not and the time spent on each test. For fair comparison, we applied NNSmith and COMET to generate DL models for testing each frontend for the same time budget as that used by OPERA (including the time spent on test generation, migration, prioritization, and execution by OPERA), respectively.

To answer RQ2, for each frontend, we constructed four variants of OPERA by replacing its diversity-based test prioritization strategy with each of the four compared strategies, respectively. By applying each variant to test each frontend, we recorded the test result and testing time of each test, and calculated the APFD value. To reduce the influence of randomness and the running environment, we repeated our experiments for five times and calculated average results.

V. RESULTS AND ANALYSIS

A. RQ1: Effectiveness

1) Bug Detection: Table I shows the number of bugs detected by tests migrated from the testing of PyTorch, Keras, ONNX libraries with the aid of OPERA, respectively. In total, 170 previously unknown bugs are detected, including 79, 43, and 48 on TVM, TensorRT, and OpenVINO, respectively. 90 bugs of them have been confirmed or fixed by developers, and the remaining bugs are being investigated by developers. The ratio of confirmed or fixed bugs is high for most of the subjects, except the PyTorch frontend for TensorRT, since its developers are inactive.

After investigations by developers, apart from three optimization bugs, the remaining 87 bugs that have been confirmed or fixed are frontend bugs. This is aligned with the goal of enhancing the testing of the model loading stage via test migration. Specifically, each migrated test by OPERA is a single-operator model, which can trigger various logic in the model loading stage, but the bugs in the optimization stages often involve more complicated models [14].

2) Root Causes of Detected Bugs: According to the feedback from developers on the 90 confirmed or fixed bugs and their patches, these bugs are caused by diverse root causes,

```
def threshold(self, inputs, input_types):
    data = inputs[0]
-   return _op.nn.relu(data)
+   threshold_f = float(inputs[1])
+   threshold_ = _op.full_like(inputs[0], fill_value=_expr.const(threshold_f))
+   value_f = float(inputs[2])
+   value = _op.full_like(inputs[0], fill_value=_expr.const(value_f))
+   return _op.where(_op.greater(data, threshold_), data, value)
```

Fig. 5. Patch for an Incorrect Code Logic bug

```
def _convert_cropping(inexpr, keras_layer, etab, data_layout):
...
-   begin = [0, 0, crop_t, crop_l]
-   end = [int32_max, int32_max, in_h - crop_b, in_w - crop_r]
+   if data_layout == "NHWC":
+       begin = [0, crop_t, crop_l, 0]
+       end = [int32_max, in_h - crop_b, in_w - crop_r, int32_max]
+   else:
+       begin = [0, 0, crop_t, crop_l]
+       end = [int32_max, int32_max, in_h - crop_b, in_w - crop_r]
    return _op.strided_slice(inexpr, begin, end)
```

Fig. 6. Patch for a Tensor Shape bug

which cover all root cause categories summarized on historical bugs in the model loading stage [4]. Among the 90 bugs, 28 bugs are caused by Tensor Shape Problem, 18 bugs are due to Type Problem, 17 bugs are Incorrect Code Logic, 13 bugs are due to Incorrect Exception Handling, 5 bugs are due to Incompatibility, 4 bugs are due to Incorrect Assignment, 3 bugs are Incorrect Numerical Computation, 1 bug is due to Concurrency, and 1 bug is due to Typo. These root causes are classified based on the existing study on DL compiler bug [4].

Next, we present two bugs detected by the migrated tests. Figure 5 depicts an Incorrect Code Logic bug in the PyTorch frontend of TVM [44]. In this bug, the `Threshold` operator from PyTorch was converted into the high-level IR of `_op.nn.relu`, which differs in its computation logic. Any non-zero value assigned to the parameter `threshold` or `value` of the `Threshold` operator will lead to wrong inference results. Indeed, a migrated test containing the operator instance `torch.nn.Threshold(threshold=2, value=1)`, helps trigger this bug during compilation. A patch [45] was committed to fix it by correcting the conversion logic for the `Threshold` operator as shown in Figure 5.

Figure 6 shows another bug [46] caused by Tensor Shape Problem in the Keras frontend. When converting the `Cropping2D` operator, TVM always considers the data layout to be `NCHW` (e.g., channel first), but `NHWC` (e.g., channel last) is also a common data layout. When TVM loads the model containing the `Cropping2D` operator and sets the parameter *data_format* to `channels_last`, which means the data layout is in the `NHWC` format, this bug can be triggered and lead to wrong inference results. This bug has been fixed using different calculation logic for different layouts.

3) Comparison with NNSmith and COMET: During the same testing time, the migrated tests by OPERA detect 79, 43, 48 bugs, while NNSmith detects 11, 2, 5 bugs and COMET detects 6, 0, 4 bugs in TVM, TensorRT, OpenVINO,

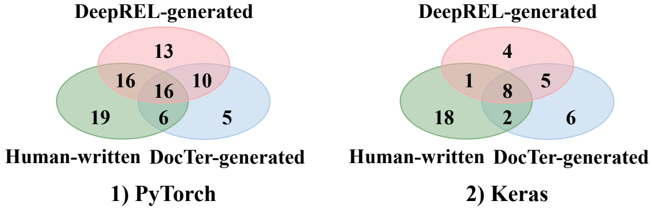

Fig. 7. Bug detection comparison among different sources

respectively. 15 of 18 bugs detected by NNSmith and 7 of 10 bugs detected by COMET are unique, which were not detected by OPERA. All these unique bugs are in the optimization stages. Besides, all frontend bugs detected by NNSmish and COMET are also detected by OPERA. OPERA detected 164 unique bugs that were not detected by NNSmith and COMET, 156 of which are frontend bugs. The results demonstrate the superiority of OPERA in testing the model loading stage and the complementarity between OPERA and the existing DL compiler testing techniques (NNSmith and COMET). The major reason for the superiority of migrated tests by OPERA over NNSmith and COMET in testing the model loading stage is that the latter two support only 75 and 72 operators while the former covers 477 operators in a lightweight manner. This implies a correlation between operator coverage and bug detection, *i.e.*, higher operator coverage in testing is likely to detect more bugs in the model loading stage.

4) Contribution of Different Migration Sources and Different Test Oracles: In this work, OPERA considers three migration sources (i.e., tests documented in DL libraries, and the tests generated by two recent fuzzers) and designs two test oracles (i.e., crash and inference inconsistency). Here, we analyzed the contribution of each migration source as well as each test oracle. Figure 7 shows the bug detection results for each migration source in OPERA, which do not include the results on the ONNX frontend as DocTer and DeepREL do not support the testing of the ONNX library. From Figure 7, all three migration sources contribute to detecting a certain number of unique bugs in PyTorch and Keras frontends, showing the complementarity among them in testing the model loading stage. Among three sources, the migration source of human-written tests detects the most bugs in both PyTorch and Keras frontends. Specifically, the migrated tests from human-written tests cover 169 PyTorch operators and 131 Keras operators, while the migrated tests from DocTer cover 65 PyTorch operators and 53 Keras operators, and the migrated tests from DeepREL cover 59 PyTorch operators and 26 Keras operators.

Through further analysis, we found that among the 170 bugs detected by OPERA, 101 are detected by the test oracle of crash (including 59 confirmed/fixed bugs) while 69 are detected by the test oracle of inference inconsistency (including 31 confirmed/fixed bugs). The results demonstrate that the two test oracles are complementary for detecting DL compiler bugs with OPERA.

5) False Positives: There are only 9 false positives produced by the migrated tests in total. All of them are caused by the test oracle of inference inconsistency between the DL compiler and the corresponding DL library. Specifically, two of them are the bugs in the Keras library rather than the DL compiler, which have been fixed in the latest version of Keras. In this work, we assume that the bug occurs at the DL compiler when there is an inference inconsistency between a DL library and a DL compiler, thus leading to the two false positives.

Three false positives are due to the undefined behaviors in the `Mod`, `RoiAlign`, `Trilu` operators. For example, *Mod* takes the dividend tensor and the divisor tensor as inputs and produces the remainder of them. If the dividend is zero, the result will be platform-dependent. That is, this false positive is caused by the undefined behavior at division by zero, which is also meaningful as the OpenVINO developer commented: "It is a good catch. We will count on this issue in case we face undefined behavior later."

The remaining 4 false positives are due to randomness in operators (e.g., `Bernoulli` and `RandomUniformLike`). For example, `Bernoulli` takes as input a tensor containing probabilities and draws the binary random number from a Bernoulli distribution. `RandomUniformLike` generates a tensor with random values drawn from a uniform distribution. The false positives caused by randomness may be filtered out by checking whether these inference inconsistencies also exist between different versions of the DL library.

B. RQ2: Efficiency

We explored the efficiency improvement of our migration-based idea by investigating whether the test prioritization component in OPERA can help detect more bugs with a given testing time budget. Figure 8 shows the number of detected bugs by OPERA and its variant without special prioritization (OPERA_{random}) with the testing process proceeding on each subject. As OPERA has extra time spent on test prioritization but OPERA_{random} does not, we included its prioritization time into the testing time of OPERA for fair comparison. Note that the total time cost across different subjects is inconsistent due to the varying number of migrated tests from different DL libraries (presented in Section IV-D) and the differing compilation time across DL compilers.

From Figure 8, OPERA always detects more bugs than OPERA_{random} regardless of the given testing time budget. In particular, OPERA spends 16.21, 29.31, 0.32, 37.33, 1.00, 7.95, 29.81, 0.42 hours on detecting all the bugs found in the experiment presented in Section V-A on each subject (in the order shown in Figure 8), while OPERA_{random} spends 53.01, 58.56, 0.54, 125.46, 1.40, 50.76, 57.22, 0.56 hours respectively. On average, the application of test prioritization in OPERA leads to a more than 55.88% reduction in time spent on bug detection, confirming the contribution of the test prioritization component of OPERA in efficiency improvement.

We also compared the test prioritization strategy designed in OPERA and several existing test prioritization strategies in general software testing , based on all the migrated tests and all

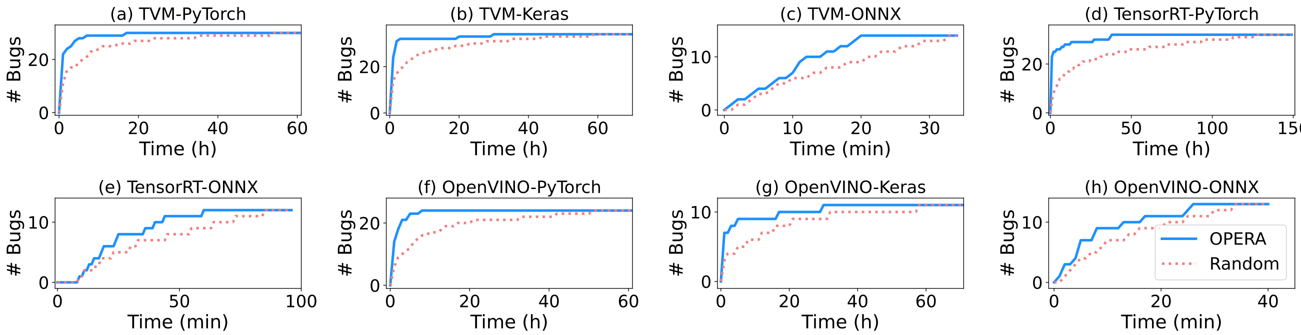

Fig. 8. Trend of bug detection effectiveness with the testing process proceeding

TABLE II
COMPARISON AMONG DIFFERENT PRIORITIZATION STRATEGIES IN TERMS OF APFD

Compiler	Frontend	OPERA	Random	FAST	Total	Additional
TVM	PyTorch	**0.984**	0.913	0.930	0.815	0.778
	Keras	**0.976**	0.905	0.906	0.691	0.868
	ONNX	**0.727**	0.577	0.487	0.316	0.575
TensorRT	PyTorch	**0.982**	0.867	0.930	0.706	0.854
	Keras	–	–	–	–	–
	ONNX	**0.768**	0.645	0.628	0.663	0.334
OpenVINO	PyTorch	**0.983**	0.877	0.927	0.608	0.670
	Keras	**0.946**	0.850	0.919	0.750	0.740
	ONNX	**0.816**	0.716	0.689	0.323	0.416
Average		**0.898**	0.794	0.802	0.609	0.654

the detected bugs in the experiment presented in Section V-A. Table II shows the comparison results among OPERA and its variants with four existing test prioritization strategies (introduced in Section IV-B) in terms of APFD. We found that OPERA performs the best among all the test prioritization strategies on all eight subjects. On average across all eight subjects, the APFD value of OPERA is 0.898 with the improvement of 13.1%, 11.9%, 47.4%, 37.2% over OPERA with random order, FAST, total-coverage-based test prioritization, and additional-coverage-based test prioritization, respectively. The results demonstrate the effectiveness of the test prioritization strategy designed in OPERA. This also indicates that designing a test prioritization strategy specific to this migration scenario is more effective than general strategies regardless of white-box or black-box strategies.

The test prioritization strategy in OPERA contains two aspects: diversity of operator signature and diversity of parameter settings. We also performed an *ablation study* to measure the contribution of each aspect by constructing two variants of OPERA: OPERA_{op} (only using diversity of operator signature to prioritize tests) and OPERA_{para} (only using diversity of parameter settings to prioritize tests). On average across all eight subjects, the APFD values of OPERA_{op} and OPERA_{para} are 0.613 and 0.832, while the APFD value of OPERA is 0.898, demonstrating the contribution of each aspect.

VI. DISCUSSION

Generalizability. We evaluated OPERA by migrating testing knowledge from three DL libraries to test eight frontends of three DL compilers. The consistent conclusions demonstrate the generalizability of it. Hence, it is promising to leverage OPERA for testing more frontends of more DL compilers in the future. Besides, the rich set of migrated tests can be directly used to test the other software taking DL models as inputs (e.g., model converters like MMdnn [47] that converts a DL model under one DL library into the equivalent model under another DL library) without any adaptation. The richness and diversity of these migrated tests may be also helpful for them.

Improving regression testing. The migrated tests by OPERA can help enrich regression test suites of DL compilers due to the diversity of bugs they detected. There are 39 migrated tests by OPERA that have been integrated into the official test suites of the DL compilers by developers, which have been used for regression testing in Continue Integration (CI). Moreover, the test prioritization strategy in OPERA has been demonstrated effective, which can be also used for optimizing the execution of regression tests in DL compilers.

Coverage-based testing of the model loading stage. Operator coverage plays a critical role in testing the model loading stage. This suggests that if some automatic test generation techniques are designed, they can take operator coverage as guidance. Similarly, if we incorporate more migration sources into OPERA according to the conclusions of the complementary effectiveness among different migration sources, operator coverage can be used as the acceptance criterion.

Stage-specific testing. Although some DL model generation techniques (e.g., NNSmith and COMET) were proposed. Their effectiveness is quite limited in testing the model loading stage. Similarly, the test prioritization strategies widely-studied in general testing cannot effectively improve the test efficiency for our migrated tests. In contrast, the design of OPERA (including both test migration and prioritization components) considers the unique characteristics of the model loading stage, achieving promising effectiveness and efficiency. This highlights the importance of stage-specific testing, which can be generalized to improve the testing of other stages.

Comparing with direct model generation. Although designing a technique to generate single-operator models directly based on existing test generation fuzzers (e.g., DocTer and DeepREL) guided by the two diversity metrics (presented in Section III-B) can be efficient, it has little influence on the overall DL compiler testing. This is because the most time-consuming step for OPERA is test execution (including model compilation) rather than test generation, which accounts for over 95% of the total time. Additionally, separating the test prioritization step in OPERA helps optimize the execution of tests from multiple sources (including human-written tests and the tests generated by different fuzzers), indicating a global optimization strategy. In contrast, the test prioritization conducted by a fuzzer at each iteration is a local optimization strategy and it can not migrate tests from human-written tests in DL libraries, which actually detected the most DL compiler bugs (presented in Section V-A4), for testing DL compilers. Hence, we proposed a migration-based testing technique (i.e., OPERA) rather than designing a model generator directly, considering the generalizability and effectiveness.

Avoiding false positives. Undefined behaviors and randomness are two main reasons leading to false positives during the testing process with migrated tests by OPERA. The method of avoiding false positives caused by randomness has been discussed in Section V-A5. However, there is still no method that can automatically detect undefined behaviors in operators, leaving the elimination of false positives caused by undefined behaviors as an open challenge. Borrowing the knowledge in detecting traditional undefined behaviors [48], [49] may help relieve this problem, which can be regarded as our future work.

Community appreciation. Besides confirming and fixing our detected bugs and integrating some of our migrated tests into their official test suites, the TVM community also appreciated our contribution in the TVM forum many times, e.g., "Detecting and fixing frontend bugs is very important work. You make a great contribution". In particular, the TVM community has invited the first author of this work to join them as a reviewer for TVM, because of the contribution of "continuously improving frontend".

Threats to Validity. The threats to validity mainly lie in the subjects and metrics. To ensure generalizability of OPERA, we considered eight frontends from the three popular DL compilers (i.e., TVM, TensorRT, and OpenVINO) for evaluation following the existing studies [14], [15]. There are also some metrics for evaluation test prioritization, such as APFDc [20] and RAUC-k [50], [51]. In our work, we used the most widely-used APFD metric and showed the trend of the number of detected bugs with the testing process proceeding. Moreover, we also used the RAUC-k metric but put the results at our project homepage due to the space limit and consistent conclusions.

Besides, our work may suffer from the human subject threat as we reported the detected bugs to developers for confirmation and fixing. To reduce this threat, we de-duplicated all test failures (see Section IV-C) and only reported the unique bugs. All the responses from developers are positive and we received appreciation and confirmation from DL compiler communities as well, which reduces this threat.

VII. RELATED WORK

A. DL Compiler Testing

Recently, several techniques have been proposed for testing DL compilers. According to the format of generated tests, they can be divided into IR-based test generation and model-based test generation. The former directly skips the model-loading stage and targets the testing of compiler optimizations, including HirGen [14], Tzer [15], and TVMFuzz [16]. MT-DLComp [32] and NNSmith [17] are of another category, *i.e.*, model-based test generation, which can cover all stages of DL compilers. MT-DLComp proposes semantics-preserving mutation to generate equivalent DL models to support metamorphic testing. NNSmith, the state-of-the-art technique, constructs DL models from scratch based on the corresponding grammar. Both of them mainly focus on generating valid and diverse models to comprehensively trigger bugs in optimization stages. As they can only generate DL models represented under the ONNX library and support a limited number of operators, they are ineffective in testing the model loading stage.

Different from them, the goal of OPERA is to enhance the testing of the model loading stage in DL compilers. Its core idea is to migrate test inputs from DL library testing, which can obtain DL models represented under various DL libraries in a lightweight way.

B. DL Library Testing

Many techniques have been proposed to test DL libraries. According to the test format during the generation of tests, they can be mainly divided into graph-level [22], [24], [33], [52] and API-level [18], [19], [53]–[55] test generation. In the first category, CRADLE [56] makes the first attempt to test DL libraries with differential testing. Subsequently, LEMON [22], Audee [33], EAGLE [57], and COMET [24] are proposed to generate DL models using a set of mutation rules. In the API-level test generation, Predoo [58] takes the first step to test DL libraries at the operator level. It mutates the original tests to maximize output precision errors. TitanFuzz [59] utilizes LLMs to generate and mutate tests for testing DL libraries.

Unlike them, we proposed the idea of test migration from DL library testing to enhance the testing of DL compilers. The tests generated by these DL library testing techniques can be the migration sources of OPERA. Indeed, OPERA has integrated the tests generated by DocTer and DeepREL as migration sources. In the future, we can incorporate more techniques to enrich the migration sources of OPERA. That is, our methodology is orthogonal to DL library testing techniques.

C. Test Migration

Several test migration techniques [60]–[65] have been proposed for various software. For example, Sebastian et al. [63] proposed a framework to extract differential unit tests from system tests for regression testing. Zhong et al. [64] designed

LERE to extract tests from bug reports of one traditional compiler to detect bugs in another compiler.

Different from them, OPERA migrates knowledge from DL library testing to enhance the testing of DL compilers. Our new scenario brings unique challenges for test migration, making the existing techniques inapplicable. The significant challenge is to handle the fundamental difference between DL library testing and DL compiler testing. As most tests for DL libraries do not include complete DL models but are just Python code, OPERA instruments to extract DL operators and wraps them as single-operator models with templates to fill the gap. Besides, a novel test prioritization strategy specific to our new scenario is designed for improving test efficiency, which has been demonstrated more effective than the existing test prioritization strategies for general software testing in our study.

VIII. CONCLUSION

In this work, we propose OPERA, a migration-based technique, to test the model loading stage in a lightweight manner. OPERA uses tests documented in DL libraries, and the tests generated by two recent fuzzers as migration sources. Then, OPERA extracts the operator instances from DL library tests and wraps them based on templates into DL models as migrated tests. To improve the testing efficiency, OPERA includes a diversity-based test prioritization strategy. By applying OPERA to eight frontends of three popular DL compilers (i.e., TVM, TensorRT, and OpenVINO), OPERA detected 170 previously unknown bugs (including 90 confirmed bugs). The diversity-based test prioritization strategy in OPERA achieves the average improvement of 11.9%$\sim$47.4% compared to general test prioritization in terms of APFD.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers for their thoughtful and constructive comments on this work. This work was supported by the National Natural Science Foundation of China (Grant Nos. 62322208, 12411530122, 62232001), CCF Young Elite Scientists Sponsorship Program (by CAST), and Hong Kong Research Grant Council/General Research Fund (Grant No. 16205722).

REFERENCES

[1] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 579–594.

[2] "Nvidia tensorrt," Accessed: 2024, https://developer.nvidia.com/tensorrt.

[3] "Intel openvino," Accessed: 2024, https://docs.openvino.ai/2022.3/home.html.

[4] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "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*, 2021, pp. 968–980.

[5] "Pytorch," Accessed: 2024, https://pytorch.org/.

[6] "Keras," Accessed: 2024, https://keras.io/.

[7] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 294–305. [Online]. Available: https://doi.org/10.1145/2931037.2931074

[8] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in gcc and llvm," *Journal of Systems and Software*, vol. 174, p. 110884, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302740

[9] S. Chaliasos, T. Sotiropoulos, G.-P. Drosos, C. Mitropoulos, D. Mitropoulos, and D. Spinellis, "Well-typed programs can go wrong: A study of typing-related bugs in jvm compilers," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct 2021. [Online]. Available: https://doi.org/10.1145/3485500

[10] Z. Wang, D. Bu, A. Sun, S. Gou, Y. Wang, and L. Chen, "An empirical study on bugs in python interpreters," *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 716–734, 2022.

[11] J. Chen, C. Suo, J. Jiang, P. Chen, and X. Li, "Compiler test-program generation via memoized configuration search," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2035–2047.

[12] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers. in 2023 ieee/acm 45th international conference on software engineering (icse). 56–68," 2023.

[13] J. Chen, H. Ma, and L. Zhang, "Enhanced compiler bug isolation via memoized search," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 78–89.

[14] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 248–260. [Online]. Available: https://doi.org/10.1145/3597926.3598053

[15] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–26, 2022.

[16] "Tvmfuzz," Accessed: 2024, https://github.com/dpankratz/TVMFuzz.

[17] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 530–543. [Online]. Available: https://doi.org/10.1145/3575693.3575707

[18] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "Docter: documentation-guided fuzzing for testing deep learning api functions," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 176–188.

[19] Y. Deng, C. Yang, A. Wei, and L. Zhang, "Fuzzing deep-learning libraries via automated relational api inference," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 44–56.

[20] A. Mor, "Evaluate the effectiveness of test suite prioritization techniques using apfd metric," *IOSR Journal of Computer*, vol. 16, no. 4, pp. 47–51, 2014.

[21] "An incorreect conversion about conv2dtranspose in tvm," Accessed: 2024, https://github.com/apache/tvm/pull/15060.

[22] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "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*, 2020, pp. 788–799.

[23] A. Bhat and S. Quadri, "Equivalence class partitioning and boundary value analysis-a review," in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE, 2015, pp. 1557–1562.

[24] M. Li, J. Cao, Y. Tian, T. O. Li, M. Wen, and S.-C. Cheung, "Comet: Coverage-guided model generation for deep learning library testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, jul 2023. [Online]. Available: https://doi.org/10.1145/3583566

[25] D. L. Powers, *Boundary value problems*. Elsevier, 2014.

[26] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Transactions on Software Engineering and Methodology*, 2022.

[27] Y. Panagakis, J. Kossaifi, G. G. Chrysos, J. Oldfield, M. A. Nicolaou, A. Anandkumar, and S. Zafeiriou, "Tensor methods in computer vision and deep learning," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 863–890, 2021.

11

[28] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, feb 2011. [Online]. Available: https://doi.org/10.1145/1883612.1883618

[29] R. Schaffer and R. Sedgewick, "The analysis of heapsort," *Journal of Algorithms*, vol. 15, no. 1, pp. 76–100, 1993.

[30] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[31] R. Coghetto, "Chebyshev distance," *Formalized Mathematics*, vol. 24, no. 2, pp. 121–141, 2016.

[32] D. Xiao, Z. Liu, Y. Yuan, Q. Pang, and S. Wang, "Metamorphic testing of deep learning compilers," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–28, 2022.

[33] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 486–498.

[34] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 222–232.

[35] O. Jafari, P. Maurya, P. Nagarkar, K. M. Islam, and C. Crushev, "A survey on locality sensitive hashing algorithms and their applications," *arXiv preprint arXiv:2102.08942*, 2021.

[36] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[37] J. Zhou, J. Chen, and D. Hao, "Parallel test prioritization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–50, 2021.

[38] Z. Chen, J. Chen, W. Wang, J. Zhou, M. Wang, X. Chen, S. Zhou, and J. Wang, "Exploring better black-box test case prioritization via log analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–32, 2023.

[39] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 656–667.

[40] "Coverage.py," Accessed: 2024, https://coverage.readthedocs.io/.

[41] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 523–534.

[42] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *2016 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 2016, pp. 266–277.

[43] Y. Lou, J. Chen, L. Zhang, and D. Hao, "A survey on regression test-case prioritization," in *Advances in Computers*. Elsevier, 2019, vol. 113, pp. 1–46.

[44] "An incorrect code logic bug of tvm," Accessed: 2024, https://github.com/apache/tvm/issues/14805.

[45] "A patch for fixing a code logic bug of tvm," Accessed: 2024, https://github.com/apache/tvm/pull/14820.

[46] "A tensor shape problem bug of tvm," Accessed: 2024, https://github.com/apache/tvm/pull/15053.

[47] "Microsoft mmdnn," Accessed: 2024, https://github.com/microsoft/MMdnn.

[48] Z. Shen, "The impact of undefined behavior on compiler optimization," in *Proceedings of the 2021 European Symposium on Software Engineering*, 2021, pp. 45–50.

[49] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming undefined behavior in llvm," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 633–647, 2017.

[50] H. A. Güvenir and M. Kurtcephe, "Ranking instances by maximizing the area under roc curve," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 10, pp. 2356–2366, 2012.

[51] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 397–409.

[52] J. Gu, X. Luo, Y. Zhou, and X. Wang, "Muffin: Testing deep learning libraries via neural architecture fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1418–1430.

[53] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing automatic differentiation in deep-learning libraries," *arXiv preprint arXiv:2302.04351*, 2023.

[54] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 995–1007.

[55] H. J. Kang, P. Rattanukul, S. A. Haryono, T. G. Nguyen, C. Ragkhitwetsagul, C. Pasareanu, and D. Lo, "Skipfuzz: Active learning-based input selection for fuzzing deep learning libraries," *arXiv preprint arXiv:2212.04038*, 2022.

[56] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1027–1038.

[57] J. Wang, T. Lutellier, S. Qian, H. V. Pham, and L. Tan, "Eagle: creating equivalent graphs to test deep learning libraries," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 798–810.

[58] X. Zhang, N. Sun, C. Fang, J. Liu, J. Liu, D. Chai, J. Wang, and Z. Chen, "Predoo: precision testing of deep learning operators," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 400–412.

[59] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[60] X. Qin, H. Zhong, and X. Wang, "Testmig: Migrating gui test cases from ios to android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 284–295.

[61] S. Talebipour, Y. Zhao, L. Dojcilović, C. Li, and N. Medvidović, "Ui test migration across mobile platforms," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 756–767.

[62] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 54–65.

[63] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, 2008.

[64] H. Zhong, "Enriching compiler testing with real program from bug report," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[65] M. Abdi and S. Demeyer, "Test transplantation through dynamic test slicing," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 35–39.