SCUZER: A Scheduling Optimization Fuzzer for TVM

XIANGXIANG CHEN, Zhejiang University, China
XINGWEI LIN, Zhejiang University, China
JINGYI WANG*, Zhejiang University, China
JUN SUN, Singapore Management University, Singapore
JIASHUI WANG, Zhejiang University, China and Ant Group, China
WENHAI WANG, Zhejiang University, China

The concept of deep learning (DL) compiler was proposed to deploy DL models more efficiently on diverse hardware through optimization techniques. As one of the most popular DL compilers, TVM incorporates three levels (high-level, schedule, and low-level) of optimizations, which can inadvertently introduce code logic bugs and build failure bugs. Among these optimizations, scheduling optimization is the core component of DL compilers, which ensures the acceleration of models on all devices. However, the existing works only focus on the testing of high-level and low-level optimizations in TVM, fail to take the most important and challenging intermediate scheduling optimization layer into consideration.

To fill the gap, we propose a **Sc**heduling optimization oriented f**uzzer** for TVM, named SCUZER, which is specially designed to effectively detect bugs introduced by the scheduling optimization. In particular, SCUZER first proposes a set of schedule-triggering mutators to actively trigger many scheduling optimizations. Meanwhile, observing that scheduling optimization is closely coupled with program data flow and operator type, SCUZER additionally proposes a set of structure-enriching mutators to enrich the structure of data flows and operators. Based on these carefully designed mutators, SCUZER then devises a multi-objective algorithm that can adaptively select different combinations of objectives at each period to guide the selection of seeds and mutators during fuzzing. We conduct extensive experiments comparing with three state-of-the-art fuzzers that can be applied in testing scheduling optimization to evaluate the effectiveness of SCUZER. The experimental results demonstrate that SCUZER outperforms the 2nd-best state-of-the-art fuzzer by 7.4% in edge coverage and achieves 7× improvement in rule-operator coverage. SCUZER has successfully detected 17 previously unknown bugs (9 are inconsistent results and 5 are inconsistent compilations) in TVM, out of which 10 have been confirmed and 5 been fixed.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: Deep Learning Compiler, Fuzzing, Scheduling Optimization, TVM

1 INTRODUCTION

Deep learning (DL) compilers take the DL models described in different DL frameworks (e.g., TensorFlow [1] and PyTorch [2]) as input and generate optimized codes for diverse DL hardware as output. Several DL compilers have been proposed, such as TVM [3], Glow [4], nGraph [5] and TensorFlow XLA [1], from both industry and

*Corresponding author

Authors' addresses: Xiangxiang Chen, chenxiangx@zju.edu.cn, Zhejiang University, China; Xingwei Lin, xwlin.roy@gmail.com, Zhejiang University, China; Jingyi Wang, wangjyee@zju.edu.cn, Zhejiang University, China; Jun Sun, junsun@smu.edu.sg, Singapore Management University, Singapore; Jiashui Wang, quhe@antgroup.com, Zhejiang University, China and Ant Group, China; Wenhai Wang, zdzzlab@zju.edu.cn, Zhejiang University, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s).
ACM 1557-7392/2024/12-ART
https://doi.org/10.1145/3705308

Just Accepted

academia. Among these DL compilers, the most popular one is TVM. Thanks to its support for various model formats, hardware platforms, and open-source feature, TVM is now one of the biggest and most widely used DL compiler projects [6, 7].

TVM employs a layered approach with specific optimizations (high-level, schedule, and low-level optimizations) to accommodate various hardware back-ends. The existing high-level and low-level optimizations of TVM are not much different from the existing DL libraries (TensorFlow, MXNet [8] and PyTorch [1]). They all take the graph-related optimization and hardware-related optimization to accelerate the models. However, these optimizations can not meet the demand of adapting various hardware back-ends [9]. To fix this, TVM further introduces a new scheduling optimization at the intermediate layer. The newly introduced scheduling optimization is based on the decoupled schedule principle derived from Halide [10]. It uses time and space-related optimizations, such as multi-core parallelism, vectorization instruction selection, and memory access optimization, to maximize the execution efficiency and performance of DL models on various hardware platforms. However, due to the various types of optimizations involved in TVM's multi-layered structure (and the possibility of incorrect user-defined optimizations), the reliability of TVM in deploying DL models is significantly compromised. Many compiler bugs arise from such incorrect optimizations, like code logic bugs and build failure bugs [11], which seriously undermines the correctness and reliability of TVM during the optimization process, especially for the less studied scheduling optimization. Given the importance of scheduling optimization in TVM and the risks of optimization itself, it is urgent to develop a testing tool for scheduling optimization.

While several recent works have been proposed to fuzz TVM, they mainly focus on testing high-level or low-level optimizations, lacking fine-grained consideration of the scheduling optimization in the intermediate layer. For high-level optimizations, NNSmith [12] employs the Z3 solver to solve constraint relations for high-level operators and utilizes a gradient-based algorithm to adjust model inputs. In general, these high-level optimization fuzzers do not design specific mutators for scheduling optimization rules, and the mutations made at the high-level layer tend to have a coarse-grained impact on the low-level layers. For low-level optimizations, TVMFuzz [13] focuses on generating test programs based on the desired program depth and randomly selecting operators from a predefined list. Tzer [7] combines an evolutionary method to mutate the optimization list and uses edge coverage to guide the whole process. Since TVMFuzz and Tzer work in a lower intermediate representation (IR) of the generated program (i.e., the converted IR after applying scheduling optimization), they are not suitable for testing scheduling optimization either.

Due to the lack of suitable testing techniques and novel attributes of scheduling optimization, it is urgent to design a scheduling optimization-specific testing technique. However, compared to existing TVM fuzzers, testing scheduling optimization effectively presents several significant challenges: ❶ Different from the optimizations of traditional compilers, the optimization of the DL compiler mainly focuses on the alterations of tensor structure. The conditions of the scheduling optimization are intricately tied to specific tensor structures and parameters, making them difficult to trigger via simple structure-related mutation methods (our experiment in the 4.4.1 also supports this view). Therefore, the necessity arises to design customized mutators for triggering scheduling optimization more effectively. ❷ Since the APIs of the scheduling layer lack sufficient documentation in reality [2], manually writing functions related to the API would require in-depth expertise and tedious manual labor. How to obtain the construction information of the API to assist model generation is also one of the challenges. ❸ Compared to the traditional program codes such as C program, the compilation and running of DL models will consume more time. Yet, existing DL compiler testing techniques only focus on the generation and mutation of IRs, neglecting the vulnerability exploration efficiency during the fuzzing process. It is thus desirable to design an adaptive strategy to improve the efficiency of the fuzzing process.

[1]PyTorch versions higher than 2.0 have implemented compilation acceleration capabilities, here we mainly refer to PyTorch before 2.0
[2]The official documentation of TVM [14] provides the parameter types of the API, but lacks the supported input tensor data types.

Just Accepted

In this work, we design and implement Scuzer, a **sc**heduling optimization-oriented f**uzzer** for TVM. To tackle the first challenge, Scuzer proposes a set of schedule-triggering mutators to actively explore as many scheduling optimizations as possible, enabling the exploration of a vast number of optimization scenarios. Meanwhile, observing that scheduling optimization is closely coupled with data flow and operator type, Scuzer additionally proposes a set of structure-enriching mutators to enhance the diversity of both the data flow structure and operator type within the test program. To overcome the second challenge, Scuzer introduces a single-operator testing method and a tree-structure-based generation algorithm to guide the reasonable substitution and generation of tensors. Before the fuzzing process, Scuzer will initially test all the operator APIs by giving each API different input data types. While the test passes the compilation, Scuzer will collect the data type of the output and save the result into a dictionary. The saved data type information then be used by the generation algorithm to randomly generate tensor programs during the fuzzing process. As evidenced by existing fuzzing techniques, efficient mutator management [15] and objective management can improve the efficiency of fuzzing [16]. To solve the third challenge, Scuzer devises a multi-objective managing algorithm to adaptively select different combinations of objectives at each fuzzing period to guide the selection of seeds and mutators. The manager module takes different combinations of objectives as the arms of a bandit algorithm and assigns weights to both the seeds and mutators based on the selected combination of objectives.

We conducted extensive experiments to evaluate the effectiveness of Scuzer, comparing it against three state-of-the-art fuzzers: LibFuzzer [17] (a general-purpose fuzzer), TVMFuzz [13] (a low-level IR fuzzer for TVM) and NNSmith [12] (a high-level IR fuzzer for DL compilers). The results demonstrate that Scuzer achieved a 7.4% higher edge coverage (than 2nd-best) and a 7× rule-operator coverage (a new metric directly measuring scheduling optimization coverage) improvement. Additionally, Scuzer outperforms existing approaches in three other performance-related metrics, i.e., operator types, schedule combinations, and length of scheduling list. We also conducted rigorous ablation studies to validate the significance and necessity of all designed components in Scuzer. The results show that the new mutators and the multi-objective managing algorithm significantly contribute to higher coverage metrics, e.g., the multi-objective algorithm leads to up to 78% improvements in the time objectives. In summary, we make the following contributions:

- We propose a new fuzzing framework specially designed for TVM scheduling optimization with customized mutators and an adaptive multi-objectives management algorithm.
- We implemented the proposed technique as a practical fuzzer (named Scuzer) for the TVM, which is released on https://github.com/cxx122/Scuzer.
- Our extensive evaluation shows that Scuzer outperforms state-of-the-art TVM fuzzers. To date, Scuzer has detected 17 bugs in TVM, 14 of which are caused by scheduling optimization (9 are inconsistent results and 5 are inconsistent compilations). Among the found bugs, 10 have been confirmed and 5 been fixed in the latest version of TVM.

2 BACKGROUND

2.1 TVM Compiler

As depicted in Figure 1, the workflow of TVM can be divided into three layers.

In the graph IR layer, TVM converts DL models from different DL frameworks into a graph IR language (i.e., Relay in TVM). This layer enables high-level optimizations related to the graph structure, such as constant folding and common sub-expression elimination. DL libraries like TensorFlow and PyTorch before the 2.0 version use the same optimizations in this layer to accelerate DL models. Several existing works, including LEMON [18], NNSmith [12], and MT-DLComp [19], focus on testing high-level optimizations in this layer.

The graph IR then divides the model into smaller subgraphs and reduces them into operator IR, known as Tensor Expression (TE) IR in TVM. The operator IR describes the computation of a DL model in a finer-grained

Just Accepted

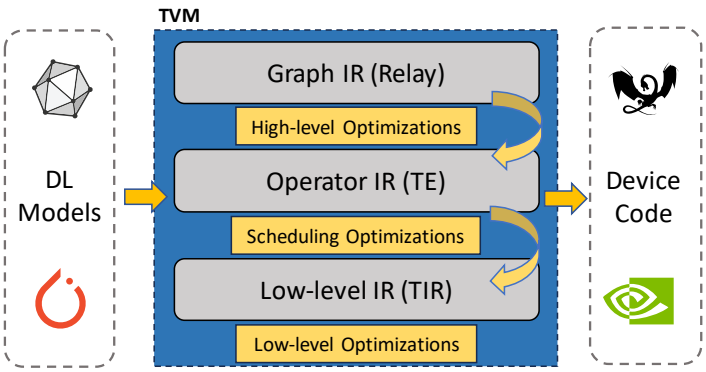


Fig. 1. The workflow of TVM.

representation than graph IR, enabling target-device-dependent optimizations to improve the computation and memory access process. The runtime program of the operator IR program is a nested loop structure. It traverses the input tensor, calculates each variable, and saves the results to the corresponding positions in the output tensor.

Operator IR supports various scheduling optimizations, such as tiling, vectorization, parallelization, and fusion [10], to accelerate computation. These optimizations can rearrange loop orders for independent loops, split or merge loops to adjust the number of iterations, and modify how tensors are accessed in memory. By fine-tuning loop parameters and memory access patterns, these optimizations enhance computational speed on the target device. To date, most publications on TVM have concentrated on improvements at this layer [9, 20, 21].

After optimizing the operator IR, each subgraph is reduced to a low-level IR, known as TIR in TVM and undergoes low-level optimizations. The low-level IR can be eventually lowered to LLVM IR and benefits from LLVM's mature optimizer and code generator. Tzer [7] and TVMFuzz [13] test such low-level optimizations by generating various low-level IR programs.

In summary, the main difference between the DL compiler and the DL library is the insertion of a new intermediate operator IR and the design of scheduling optimization. Given that scheduling optimization is the critical component that sets apart a DL compiler from a DL library [10]. It is crucial to develop a specialized fuzzer specifically designed for testing and evaluating scheduling optimization.

2.2 Tensor Expression and Scheduling Optimiazation

TE describes operator computations in a purely functional language (i.e., each expression has no side effects). When viewed from the perspective of the overall TVM, graph IR (Relay) describes computation as a set of operators, and each operator can be represented as a TE program, where each TE program accepts an input tensor and produces an output tensor. As depicted in Figure 2, the TE program firstly defines an input placeholder tensor `A` to be accessed and then establishes an output compute tensor `B` with a `lambda` function to describe the computation process (the `te.sum` operator is relevant to the use of reduce operation, which aggregate the elements of a two-dimensional array into a one-dimensional array). During the later compilation process, TVM converts the tensor `B` into the form of loops and transforms the axis `i` into a `for` loop with a range `n`. The values in tensor `B` are then calculated by traversing the values in tensor `A`.

TVM employs a two-level approach: sketch and annotation to search for the best scheduling optimization. At the sketch level, TVM generates sketches by recursively applying several derivation rules. At the annotation

Just Accepted

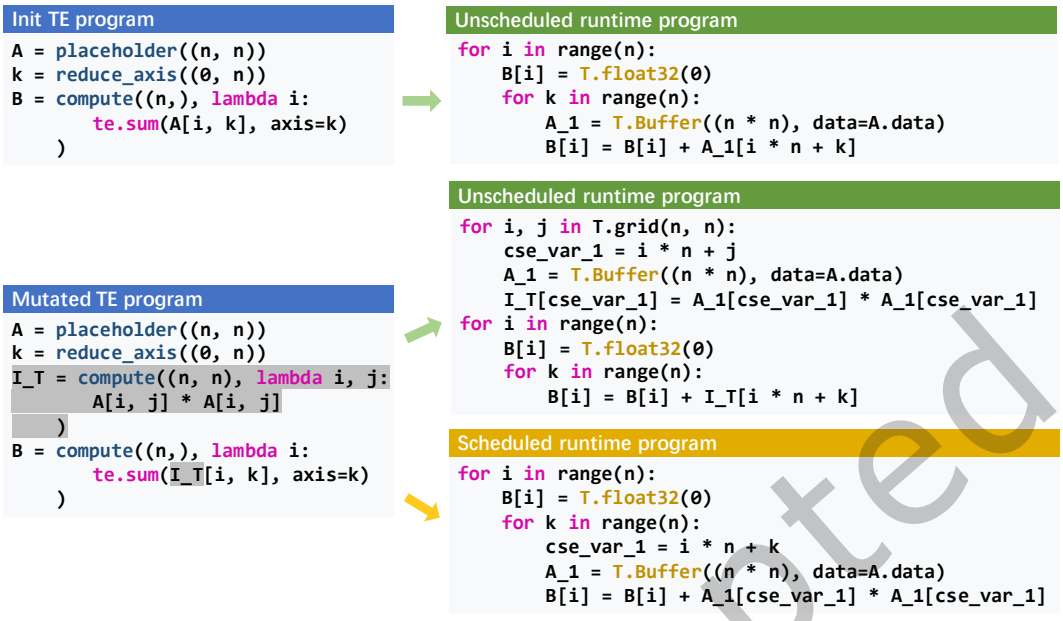


Fig. 2. An example of TE program and *Always Inline* rule application.

Table 1. Schedule-related rules shown in the TVM paper. TVM uses predicates like *IsStrictInlineable* and *HasDataReuse* to determine whether a program meets the conditions and then applies the rules.

No	Rule Name	Condition
1	Always Inline	IsStrictInlineable()
2	Multi-level Tiling	HasDataReuse()
3	Multi-level Tiling with Fusion	HasDataReuse() ∧ HasFusibleConsumer()
4	Add Cache Stage	HasDataReuse() ∧¬ HasFusibleConsumer()
5	Reduction Factorization	HasMoreReductionParallel()
...	...	...

level, TVM randomly annotates these sketches, filling in the missing information to obtain complete programs with fully specified schedules. Since the annotation is a program-independent strategy, we focus on triggering derivation rules.

We collect the derivation rules shown in the TVM [9] paper and list them in table 1. The trigger predicates of these derivation rules include *IsStrictInliable* which indicates whether a tensor node is a simple element-wise operator that can always be inlined (e.g., element-wise addition, ReLU), and *HasDataReuse* which indicates whether a tensor node is a compute-intensive operator with ample intra-operator data reuse opportunities (e.g., matmul, conv2d). These trigger predicates can be summarized as inline, data reuse, fuse, and rfactor. TVM performs static analysis on the computation definitions of tensors to get the values for these predicates. Then, these values are used to judge if the rules are met with the condition. According to our in-depth reading of the TVM source code, TVM has designed a total of five predicates. In addition to the four predicates shown in the paper, there is still one remaining predicate, which only triggers one schedule-related rule in TVM. We tried to

Just Accepted

trigger this predicate but failed to restore the triggering process due to the lack of relevant documents. To ensure the rationality of the designed mutators, here we only include the above four predicates.

The *Always Inline* rule inserts tensor with computation descriptions directly into the place where the tensor is used. It employs the *IsStrictInlineable* predicate to identify inlineable tensors in the program. Once identified, the rule performs inline optimization on these tensors to reduce memory access. The *Multi-level Tiling* rule, also known as loop blocking, partitions loops into smaller blocks to utilize the memory hierarchy effectively. Instead of processing the entire data array in one loop, data is processed in smaller blocks that fit into the faster, smaller levels of the memory hierarchy, such as caches. When data reuse is present, it means that certain data elements are accessed multiple times during the computation process. Multi-level tiling helps enhance data reuse by ensuring that frequently accessed data stays in faster levels of the memory hierarchy for longer periods. When a fusible tensor follows a data reuse tensor, the *Multi-level Tiling with Fusion* rule fuses the tiling loop with the fusible tensor, reducing intermediate variable accesses. The *Add Cache Stage* rule allocates a high-speed temporary buffer to cache reused data, speeding up memory access. The *Reduction Factorization* rule is a commonly used technique in computational graph optimization to improve performance and parallel efficiency. The core idea of this rule is to decompose a large-scale reduction operation into multiple small-scale reduction operations and run them in parallel. When the number of reduction computation loops is large, this rule decomposes the large reduction loop into multiple smaller loops.

Consider the application of the *Always Inline* rule, as illustrated in Figure 2. To trigger the *Always Inline* condition, we insert an inlineable tensor `I_T` to satisfy the *IsStrictInlineable* predicate. Before applying the schedule, the unscheduled runtime program introduces a new loop to compute the tensor `I_T`, which is subsequently used to compute the tensor `B`. Since the presence of the intermediate tensor `I_T` will result in increased memory and computational resource consumption. Here, the *Always Inline* rule is deployed to optimize the program. The rule inlines the computation of tensor `I_T` into tensor `B`, thereby enhancing the runtime performance.

According to the work aimed at optimizations testing [22–24], designing mutation operators that trigger optimization can effectively increase the probability of finding optimization-related bugs. In order to deeply test the scheduling optimization-related code during the testing process, we designed schedule-triggering mutators based on the four predicates.

Just Accepted

2.3 A Motivating Example

Figure 3 presents a bug detected by Scuzer, which causes the final result on line 3 to be inconsistent. Scuzer detected this bug by inserting a new tensor `I_T` to trigger the *Always Inline* rule. While using inequalities from the two boolean types `C<=4` and `C<4`, the *IsStrictInliable* function mistakenly assumes that `I_T` is a redundant tensor, leading to its omission. Before applying the schedule, during the first `for` loop, all values in the `I_T_1` array are assigned the value 1 (resulting from `T.exp(T.float32(0))`, which evaluates to 1). Subsequently, in the second `for` loop, the values in the `output_1` array are updated with the values from the `I_T_1` array. However, after applying the schedule, due to the exclusion of the `I_T_1` array in the scheduling optimization, the values in the `output_1` array are replaced with the unassigned values from `I_T_1` array, resulting in an inconsistency in the final results.

From this example, we can observe that the triggering of scheduling optimization bugs needs to satisfy both the derivation rule predicates and specific operation structures, which are difficult to achieve by randomly generating TE programs (like NNSmith [12] and TVMFuzz [13]). This emphasizes the importance of considering both predicate and tensor computation structures while fuzzing the scheduling optimizations.

Buggy TE program

```
k = reduce_axis((0, 1))
I_T = compute((2, 2, 10), lambda ax1, ax2, ax3:
            min(exp(const(0, dtype="float32")), axis=k)
        )
output = compute([16, 10], lambda c, d : I_T[(c <= 4), (c < 4), d])
```

Unscheduled runtime program

```
for ax1, ax2, ax3 in T.grid(2, 2, 10):
    cse_var_1 = ax1 * 20 + ax2 * 10 + ax3
    I_T_1[cse_var_1] = T.float32(3.4028234663852886e+38)
    I_T_1[cse_var_1] = T.min(I_T_1[cse_var_1], T.exp(T.float32(0)))
for c, d in T.grid(16, 10):
    output_1 = T.Buffer((160,), data=output.data)
    output_1[c * 10 + d] = I_T_1[(c <= 4) * 20 + (c < 4) * 10 + d]
```

Scheduled runtime program

```
for c_d_fused in T.parallel(160):
    output_1 = T.Buffer((160,), data=output.data)
    I_T_1 = T.Buffer((40,), data=I_T.data)
    output_1[c_d_fused] = I_T_1[(c_d_fused <= 49) * 20 + (c_d_fused < 40) *
    10 + c_d_fused % 10]
```

Fig. 3. An inconsistent bug triggered by the *Always Inline* rule.

Just Accepted

3 SCUZER FRAMEWORK

In this section, we present the details of Scuzer, a coverage-guided fuzzer specifically designed for TVM scheduling optimization. The overall workflow of Scuzer is illustrated in Figure 4. In each fixed-time fuzzing cycle, Scuzer assesses four objectives within the seed corpus and employs a multi-armed bandit managing algorithm to determine the optimal combination of these objectives (Section 3.3). Subsequently, Scuzer allocates the weight of each seed based on the values of the objectives in the selected combination. To activate TVM scheduling optimization while enhancing the diversity of program structure (Section 3.1), Scuzer develops four schedule-triggering mutators and five structure-enriching mutators. Using a weight matrix guided by the designed mutators, Scuzer establishes the relationship between the objectives and the mutators, assigning weights to each mutator under different objective combinations. Following the designed mutators, Scuzer utilizes the results of single-operator testing to generate a mutant conforming to the operator type constraints based on the selected seed and mutator (Section 3.2). If the mutant increases coverage, it is incorporated into the seed corpus. Next, we introduce the details of each component.

3.1 Mutators

In terms of mutator design, we consider two typical goals in fuzzing [25]: exploration and exploitation. That is, we need to 1) generate programs to test DL compilers as sufficiently as possible by exploring unused optimization code and 2) exploit different usage ways of optimization code. For a thorough examination of scheduling optimization-related code, it is essential to design mutators with both goals in mind. Therefore, we systematically design our mutators from two perspectives, i.e., schedule-triggering mutators and structure-enriching mutators.

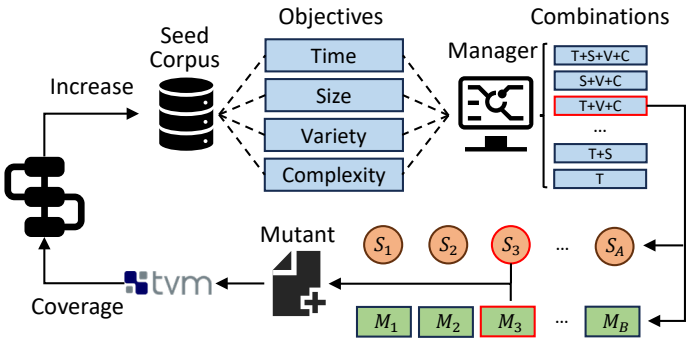


Fig. 4. Workflow of Scuzer. T+S+V+C represents the combination of objectives time, size, variety and complexity. Scuzer guides the selection of seeds and mutation operators based on the current optimal combination.

As mentioned in 2.2, TVM adopts various rules to generate sketches. To trigger more optimization code, the predicates of these rules need to be met. Existing mutators focusing on changing the operator types and parameters make it difficult to trigger these rules effectively. In order to extensively trigger the predicates of the rules and improve the efficiency of exploration, we first manually analyze the schedule-related rules, e.g., always inline and multi-level tiling, and then derive the mutation strategies aiming for extensively triggering the predicate of the rules. These strategies are defined as the schedule-triggering mutators.

We further realize that mutating data flow and operator type in TE programs can potentially advance the exploitation effectiveness of scheduling optimization fuzzing due to the following reasons. Firstly, the structure of the TE program is a directed acyclic graph driven by data flow, where operators serve as nodes and tensors as edges for the transmission of data [6]. Modifying the TE program inherently involves altering the data flow structure. Secondly, data dependency analysis is a crucial component in scheduling optimization analysis. A significant amount of optimization-analysis-related work [27–30] revolves around the dependency existing problem within the data flow to ensure that the program remains free from errors after the deployment of scheduling optimization. Moreover, the mutation of operator type is one of the most popular methods applied in the existing DL compiler testing work [7, 13]. Accordingly in Scuzer, we propose a set of structure-enriching mutators including one operator mutator to diversify operator types and three data flow-related mutators to strengthen the structure of the TE program. We also designed deletion mutators to reduce inefficient parts of the program.

To better understand our mutators, we offer illustrative examples for each mutator in Figure 5, Figure 7, and Figure 7.

3.1.1 Schedule-triggering Mutators. We designed four mutators corresponding to the schedule-related rules in the existing TVM. i.e., inline, data reuse, fuse, and rfactor. The above-mentioned four mutators can trigger most of the rules, as listed in Table 1.

Data reuse mutator. A data reuse tensor is characterized by involving reduction-related operators like sum, min, and max, which use the same data multiple times in a computation. As illustrated in Figure 5, based on the init TE program, during the mutation process, the mutator first searches for a non-data reuse tensor `B` from the tensor list. Then, it converts one of the dimensions of tensor `B` (e.g., axis `j`) into a reduction axis `k`, inserts the statement of reduce axis `k` and the reduction operator `sum` into the computation, transforms tensor `B` into a data reuse tensor. In the converted running program shown in Figure 2, the data elements `B[i]` of tensor `B` are calculated multiple times to obtain the final sum result.

Just Accepted

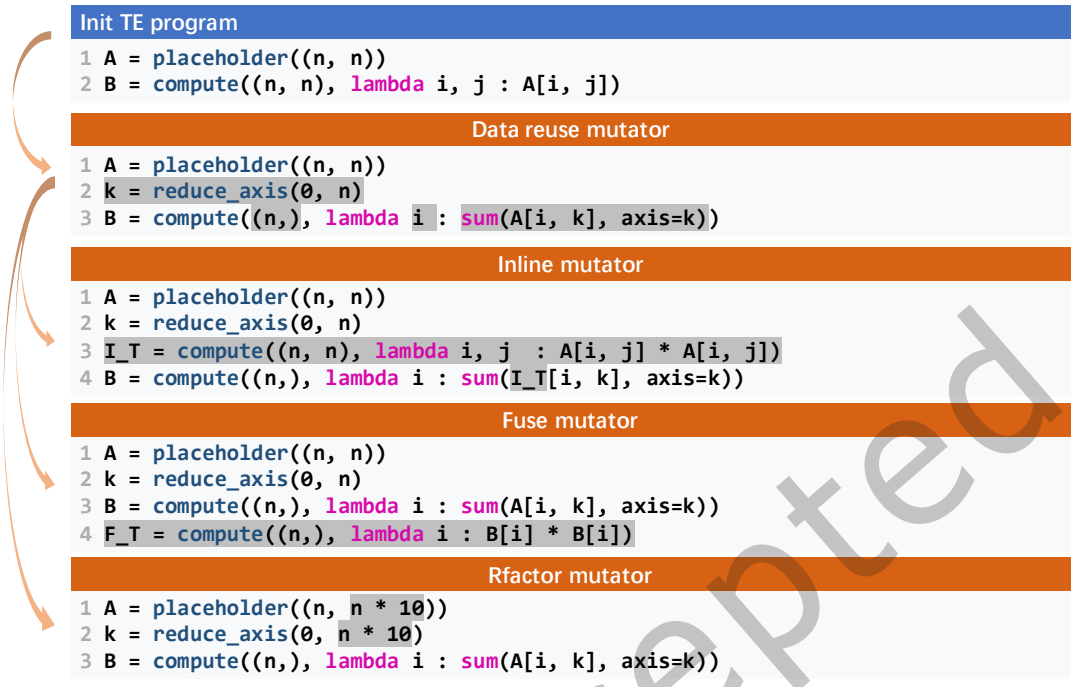


Fig. 5. Examples of schedule-triggering mutators.

Inline mutator. The inline mutator strategy is to insert a new inlineable tensor before the selected tensor. As illustrated in Figure 5, to mutate the TE program after data reuse mutation, the mutator firstly traverses the entire tensor list to select a compute tensor B. Next, the mutator will generate a new tensor I_T (the generation process is detailed in section 3.2), and then insert it in front of tensor B. Finally, the mutator replaces the producer tensor A of tensor B with the new inline tensor I_T.

Fuse mutator. The associated mutator strategy of *Multi-level Tiling with Fusion* rule is appending a new fusible tensor after the data reuse tensor. In Figure 5, to mutate the TE program after data reuse mutation, we initially search for a data reuse tensor B from the tensor list. Next, we append a new fusible tensor F_T after the tensor B. The fusible tensor F_T is configured to utilize B as its producer tensor.

Rfactor mutator. The predicate of *Reduction Factorization* is fulfilled when the tensor's reduce axis is several times larger than the other non-reduce axis. Considering this predicate, we define the mutator strategy as increasing the upper bound of the reduce axis and the corresponding dimension in the producer tensor. To illustrate, Figure 5 demonstrates that we multiply the upper bound of the reduce axis k by ten and apply the same modification to the corresponding dimension in the producer tensor A accessed by k.

3.1.2 Structure-enriching Mutators. To increase the diversity of TE programs under different rules, we further design structure-enriching mutators to modify the program's operator type and data flow, thereby enhancing the degree of exploration within the optimization-related code space.

Operator mutator. To test different operator types in the context of scheduling optimization, we have designed this mutator to alter the operator. As depicted in Figure 6, a multiplication operator is replaced by an addition operator. Since different operators have different application structures and different types of input and output. To address the type mismatch problem during operator substituting, we conducted individual tests for each

Just Accepted

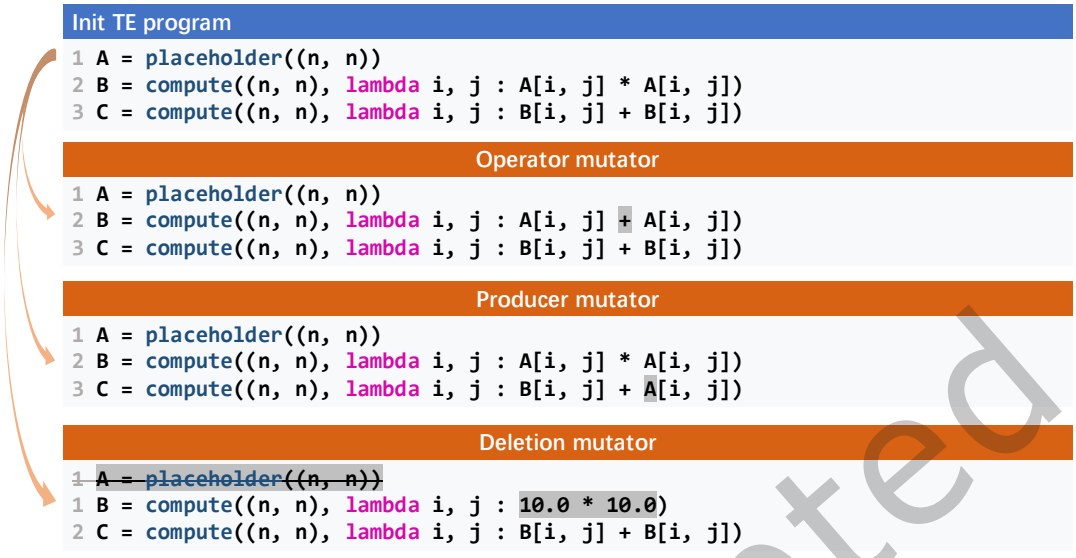


Fig. 6. Examples of structure-enriching mutators. Include operator, producer, and deletion mutators that modify one single program.

Just Accepted

operator and collected the corresponding input and output type information (further described in section 3.2). During the mutation process, we match operators with the same input and output types to form a candidate list and then randomly select an operator for replacement.

Producer mutator. Data dependency analysis is crucial in program parallelization [31]. To investigate the effectiveness of TVM scheduling optimization in handling dependencies when applying parallel-related optimizations, we have devised the producer mutator, which modifies the producer contained in a tensor. Specifically, in Figure 6, we select tensor `C` and analyze the producer tensors associated with it (i.e., tensor `B`). Subsequently, we identify other tensors that can serve as replacements (i.e., tensor `A`, the replacement tensor must satisfy the condition of having a larger value and dimension of the shape than the replaced tensor). Finally, we replaced the producer tensor `B`.

Deletion mutator. We introduce a deletion mutator for program size reduction to address the increased program size caused by the append or modify operations of the above mutators. In this mutator, we analyze the producer tensor of each tensor and form them into a list. By examining how the program utilizes each tensor, we identify and filter out the producer tensor called by only one other tensor, making it a candidate for deletion. Subsequently, we remove the selected producer tensor and use a random constant of the corresponding type to replace it. For instance, as shown in Figure 6, the placeholder tensor `A` is only called by tensor `B`, which qualifies it for deletion. As a result, we delete tensor `A` and its declaration, and replace it with a constant value of 10.0.

Series combine mutator. In light of the most common serial connection structure observed in the data flow, we have developed the series combine mutator. As illustrated in Figure 7, we randomly select two TE programs and performed static analysis to obtain their output and input tensors. Subsequently, we insert a connection tensor `C_T` between the two TE programs, utilizing the former program's output tensor `B_1` as the producer. Furthermore, we replace the input tensor `A_2` of the latter program with the inserted tensor `C_T`. The tensor `C_T` inherits the shape of tensor `B_2` and employs the `Select` operator to impose a limit on the axis, effectively

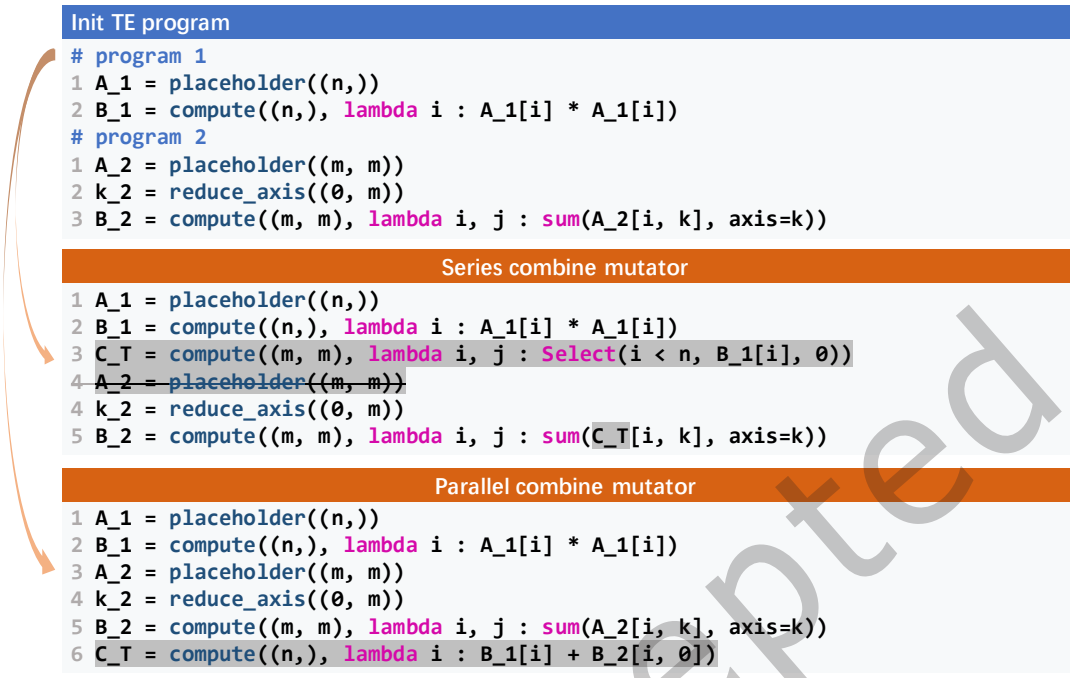


Fig. 7. Examples of structure-enriching mutators. Includes series and parallel combine mutators that take two programs as input.

Just Accepted

preventing out-of-bound access. When the axis lies within the output tensor shape of the previous program, the tensor `C_T` value is directly assigned to the tensor `B_1` value. Otherwise, it is set to 0.

Parallel combine mutator. As a compiler with parallel optimization as a key component [9], optimizing parallel structures in the data flow is also a focal point of our testing. Like the Series combine mutator, the parallel combine mutator conducts static analysis on the two selected TE programs to obtain their output tensors. To prevent out-of-bounds accesses, the join tensor `C_T` adopts the smallest shape value between tensor `B_1` and tensor `B_2` as its shape. It combines these two tensors using operators that can be adapted to all types of tensors (e.g., addition and subtraction).

3.2 Generator

To support the mutators mentioned above, an efficient tensor generator is required. However, when generating tensors, the type inconsistencies of different operators may result in runtime errors. Additionally, uncontrolled randomness in program generation can produce invalid program structures, impairing the efficacy of testing. In this subsection, we present an approach to tensor generation based on single-operator testing and randomized generation of computation tree.

As shown in Figure 8, during the single-operator testing process, we generate various types of input tensors to thoroughly test the target operator and obtain the corresponding output tensor type. The types tested include `int`, `uint`, `float`, and `bool`, encompassing different precision types like `uint32` and `uint64`. For single-input structure operators, such as the `isnan` operator, we provide n types of input tensors (where n is the number of types). For the multi-input structure operator, we supply n^d combinations of input tensors (where d is the

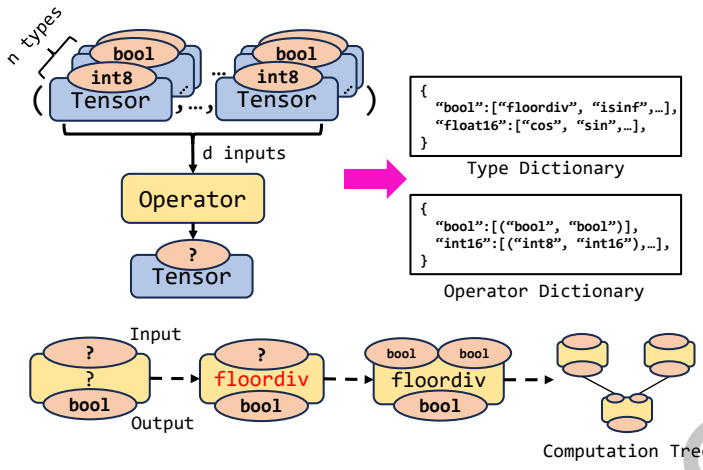


Fig. 8. Workflow of single-operator testing process and computation tree generation.

number of input tensors required by the operator). Since the number of types and input tensors is limited (up to 3, e.g., the operator `if_then_else`), the testing overhead remains relatively acceptable. We then save all the test results in the type dictionary and operator dictionary, where the type dictionary stores the operators according to the type of their output tensor, the operator dictionary saves all the output tensor types and the corresponding input tensor types for each operator.

We generate the computation of the tensor in the form of a tree. Each node in the tree contains information about the required output type, the operator, and a list of input subtrees. As demonstrated in Algorithm 1, we first generate an initial tree `initTree` by randomly selecting an operator from the type dictionary according to the required output type `tOutType` (line 18-20). Within the `ExpandTree` function, we generate a candidate list of input types `inTypesList` by searching operator dictionary `op` with the output type `outType`. We then use `TInTypeMostSelect` to select the input types that contain the most occurrences of the required input type `tInType` (line 6-7). Based on the selected `inTypes`, We iterate over each of `inTypes` and find eligible operator `subOp` to each input type `inType`. When the `maxDepth` is lower than 0, we select the operator that contains the most input type of tensor `tInType`. After packaging these operators into new subtrees, we recursively expand these subtrees and append them to the `children` list (line 8-16). The recursive process stops when the `outType` of the termination node matches `tInType` or when the `maxDepth` reaches the set limitation `LIMIT` (a negative number).

3.3 Manager

Different from binary fuzzers, testing DL model-related programs requires significantly more computational resources compared to traditional C program testing. In practice, testing a single TE program can take about one second or more, leading to poor performance on testing speed. Existing techniques [12, 13, 26] all generate test samples randomly, resulting in a certain randomness in the test time for each sample, which negatively impacts testing efficiency.

Besides minimizing the time of testing, it is crucial to consider other objectives. Since solely focusing on programs with shorter test times may negatively impact other important metrics, such as the number of optimizations explored and code branches exploited. Thus, scheduling the fuzzer involves balancing multiple objectives to ensure comprehensive and efficient testing. To fill the gap, inspired by Mobfuzz [16], we designed a multi-objective

Just Accepted

Algorithm 1: Computation Tree Generation Algorithm.

```
Input: tOutType, tInType, typeDict, maxDepth
Output: tree
Function ExpandTree(tree, maxDepth, tInType):
    (outType, op, _) ← tree;
    if maxDepth < 0 then
        if maxDepth < LIMIT or outType == tInType then
            return (outType, None, [])
    inTypesList ← op[outType];
    inTypes ← TInTypeMostSelect(inTypesList);
    children ← [];
    for inType in inTypes do
        subOpList ← typeDict[inType];
        if maxDepth < 0 then
            subOp ← TInTypeMostSelect(subOpList);
        else
            subOp ← randomly select from subOpList;
        subTree ← (type, subOp, subOpIntypes);
        children ← children ∪ ExpandTree(subTree, maxDepth − 1, tInType);
    return (outType, op, children)
opList ← typeDict[tOutType];
op ← randomly select from opList;
initTree ← (tOutType, op, []);
tree ← ExpandTree(initTree, maxDepth, tInType);
return tree
```

Just Accepted

managing algorithm for SCUZER. The multi-objective managing algorithm divides the testing process into multiple stages. During each stage, it adaptively selects the objective combination that contains the most appropriate objectives and allocates more energy to the seeds and mutators under the chosen objective combination. The consideration of the objective combination can help balance exploration and exploitation while improving the speed, thereby maximizing the testing efficiency.

3.3.1 Objective Design. We established four key objectives that need to be paid attention to during the fuzzing process. In particular, we consider the following four complementary objectives:

- **Time:** The running time of TE programs significantly influences testing efficiency. The TIME objective prioritizes seeds with shorter running times, determined by the shape value of the final tensor in the TE program.
- **Size:** The SIZE objective is calculated based on the number of nodes in the seed. Smaller seeds are preferable as they are easier to analyze after triggering the test oracle.
- **Variety:** The VARIETY objective is calculated based on the number of operator types within the seed. Seeds with a wider variety of operator types can potentially trigger more code branches in the corresponding scheduling optimization.

Algorithm 2: The Multi-Objective Managing Algorithm.

Input: Objectives O, Initial seeds S, Mutators M, Mutator weight matrix W

$k \leftarrow 0$;
while *cur_time - start_time < LIMIT_TIME* **do**
  if *cur_time - prev_time >= 5 min* **then**
    $C_M \leftarrow None$;
    $k \leftarrow k + 1$;
    for o_i *in* O **do**
      $\overline{v}_{o_i}^k \leftarrow \sum_{r=1}^{k} v_{o_i}^r / k$;
      $R_{o_i} \leftarrow \frac{v_{o_i}^k}{\overline{v}_{o_i}^k} - \lambda \frac{v_{o_0}^k}{\overline{v}_{o_0}^k}$;
    for l *in range*$(1, 16)$ **do**
      $tFlag \leftarrow (l/8) \mod 2$;
      $sFlag \leftarrow (l/4) \mod 2$;
      $vFlag \leftarrow (l/2) \mod 2$;
      $cFlag \leftarrow l \mod 2$;
      $n_l \leftarrow tFlag + sFlag + vFlag + cFlag$;
      $R_{C_l} \leftarrow (tFlag \times o_0 + sFlag \times o_1 + vFlag \times o_2 + cFlag \times o_3)/n_l + n_l$;
      if $n_{C_l} == 0$ **then**
        $C_M \leftarrow C_l$;
    if C_M *is None* **then**
      $C_M \leftarrow arg\_max(R_{C_l} + \sqrt{2\ln(k)/n_{C_l}})$;
    $prev\_time \leftarrow cur\_time$;
  for s *in* S **do**
    $W_s \leftarrow \sum_{o_i \in C_M} v_{o_i}(s)/v_{o_i}/n_l$;
  for m *in* M **do**
    $W_m \leftarrow \sum_{o_i \in C_M} W[m][o_i]$;
  $s_a \leftarrow$ randomly select a seed using W_s list;
  $m_b \leftarrow$ randomly selct a mutator using W_m list;
  $s' \leftarrow m_b(s_a)$;
  if $new\_cov(exec(s')) == True$ **then**
    $S \leftarrow S + s'$;

- **Complexity:** The COMPLEXITY objective is calculated based on the number of scheduling optimizations triggered by the seed. Greater complexity results in more optimization code triggered, implying a larger exploitation space for the seed.

3.3.2 Objective Combination Selection. In each fuzzing iteration, SCUZER will first select the optimal combination of objectives to focus on based on the reward of each possible objective combination. We do not simply use all the objectives since different objectives emphasized at different stages of testing can enhance the efficiency of fuzzing. For example, in the initial stage of testing, when we aim to traverse different seeds quickly, we can prioritize the TIME objective and select seeds with shorter testing times. When further improvements in the TIME objective

Just Accepted

are not feasible, we allocate more energy to objectives such as variety and complexity, thereby increasing the depth of testing.

As illustrated in Algorithm 2, during the fuzzing process, Scuzer periodically selects the optimal objective combination C_M after a fixed time interval (line 3-20). Considering that a single test for the program typically lasts from one to ten seconds, we set the time interval to 5 minutes to ensure the inclusion of a sufficient number of new seeds before progressing to the next round. Each objective in a combination has two states: presence or absence. With N objectives, the total number of combinations is $2^N - 1$. In each managing period (combination selection round), we calculate the reward of each objective R_{o_i} by dividing the average value of objective $v^k_{o_i}$ in the current round by the average value of objective $\overline{v}^k_{oi}$ over all rounds (line 6-8). When $v^k_{o_i}$ exceeds $\overline{v}^k_{oi}$, the reward is large, encouraging the selection of objectives that increase rapidly. The parameter λ serves as a penalty factor, reducing the reward for an objective that would increase the testing time (where time is the 0th objective). We set the value of λ as 0.1 (refer to Mobfuzz). The reward of each combination R_{C_l} is obtained by calculating the average reward of objectives in the combination (line 9-15). Here, n_l is the number of objectives inside the combination. We further append n_l to reward combinations with more objectives.

Scuzer utilizes the commonly used UCB-1 [32] formula to find the optimal objective combination (line 18-19). In this equation, n_{C_l} represents the number of times this objective combination has been selected in the previous $k-1$ rounds. To avoid $n_{C_l} = 0$, in the earliest 15 rounds, we select each combination as the current optimal combination (line 16-17).

During each fuzzing loop, depending on the chosen optimal combination of objectives, Scuzer assigns weights to seeds and mutators to select those beneficial for the current objective combination (line 21-24). First, Scuzer divides the objective values of the seed $v_{o_i}(s)$ by the average objective value of all seeds v_{o_i} to obtain the weight of each selected objective. These weights are then summed and divided by the number of objectives inside the combination n_l to get the final weight of the seed (line 21-22). For mutating under the current objective combination, some mutators may be unfavorable. To handle this, weights are assigned to the mutators using a 9×4 weight matrix W (corresponding to our 9 mutators and 4 objectives). The value of $W[m][o_i]$ is defined as 1 or 0, representing the positive and negative influence of mutator m on the objective o_i. By querying the weight matrix W, the values of selected objectives under the corresponding mutator are summed together to obtain the weight of the mutator (line 23-24).

Scuzer then utilizes a weighted randomization approach based on the assigned weights for seeds and mutators to choose the target seed s_a and mutator m_b (line 25-26). After the execution of the mutated seed s', if the coverage increases, Scuzer appends it to the seed corpus S.

4 EVALUATION

We extensively evaluate Scuzer by answering the following research questions:

- RQ1: How effective and efficient is Scuzer compared to state-of-the-art techniques?
- RQ2: How useful is each component of Scuzer?
- RQ3: How capable is Scuzer in bug detection?

4.1 Experimental Setup

4.1.1 Baseline Selection. Since Scuzer is the first fuzzing technique specific to scheduling optimization, we did not have direct baselines. In fact, some high-level source programs can be transformed into TE programs and thus can also be adapted to fuzz scheduling optimization:

NNSmith [12] is a generator and fuzzy infrastructure primarily utilized for the automated validation of DL frameworks and compilers. Since NNSmith operates at the graph layer as a generative testing technique. We convert its generated programs to the operator layer for comparison.

Just Accepted

LibFuzzer [17] is a leading bit-level general-purpose binary fuzzer. To ensure a fair comparison with Scuzer, we applied LibFuzzer using an identical seed corpus as Scuzer for TVM.

TVMFuzz [13] is the first specialized fuzzer for TVM. The build process of TVMFuzz includes some TE IR-related interfaces, making it an appropriate baseline.

4.1.2 Test Oracle. We address the test oracle problem for finding bugs using the following criteria:

Result Inconsistency. Scuzer compiles each generated TE program twice: first without the schedule and then with the schedule. By comparing the outputs of both versions using the same input, Scuzer identifies inconsistency errors if the absolute or relative error exceeds a tolerated threshold. The variations in runtime structures can cause slight differences in computation results due to factors like numerical precision and rounding errors. The tolerance threshold is used here to help mitigate the influence of different runtime structures between scheduled and unscheduled TE programs. In Scuzer, the tolerance threshold for relative error is set to $1e-5$.

Compilation Inconsistency. Inconsistencies during compilation may indicate optimization issues that could lead to unexpected behavior or crashes at runtime. Ensuring consistent compilation helps detect these issues early, enhancing software reliability. Due to the impact of schedules on program structure, the compilation of scheduled and unscheduled TE programs may differ. A scheduled TE program might introduce a structure that fails to compile, while the unscheduled version compiles successfully. When the compilation result of a scheduled TE program differs from that of an unscheduled one, an inconsistency occurs. During fuzzing, Scuzer examines the consistency in the compilation of TE programs before and after applying the schedule and records any inconsistencies.

Program Crash. Scuzer detects crashes by analyzing the exit code of the child process. If the exit code of the child process is abnormal after running the program, it indicates an exception or crash.

4.1.3 Experimental Environment. All TE programs are saved in JSON format, and the design of mutators is also based on JSON. The single-operator testing process contains 57 operators and 12 tensor types, the type and operator dictionaries are saved in YAML format.

To construct the seed corpus, we identify TE program-related functions from the unit test of TVM and extract about 500 seeds. By executing the ONNX files of typical DL models and splitting them into multiple TE programs, we obtain approximately 600 seeds. Due to the time-intensive nature of a single test of the DL compiler and the constraint number of tens of thousands of test rounds in 24 hours, having many inefficient seeds hinders the selection of high-quality ones. To address this, we use corpus distillation, reducing seeds from thousands to around 100 by eliminating those not contributing to coverage improvement.

Scuzer records code coverage referencing the technique presented in [7]. We also modified the optimization-related source code in TVM to record the number and types of scheduling optimization triggered during the compilation process. This information is used to compute the newly defined rule-operator coverage, which will be later introduced.

The experiments were conducted on a server with an Intel(R) Xeon(R) 6226R CPU with 64 cores and 384 GB of RAM. The experiment was deployed on docker, and the operating system was Ubuntu 20.04. Since NNSmith calls sub-processes for compilation, we allocate 4 cores to each experiment to ensure unrestricted testing processes. The version of TVM under test is 0.10.dev, compiled by clang9. To mitigate the impact of randomness, we ran each experiment 5 times and calculated the average results.

4.2 Evaluation Metrics

Code Coverage. Recent works in the field of DL libraries/compiler [7, 12, 33] have all considered code coverage as one of their key metrics. Following these state-of-the-art approaches, we adopt edge coverage as one of our primary metrics.

Just Accepted

Rule-operator Coverage. As introduced in Section 3.1, the main goal of Scuzer is to enhance the exploration and exploitation of scheduling optimization-related code. To evaluate the technique's ability from this perspective, we devise a new metric called rule-operator coverage.

Consider the inconsistent bug introduced in Section 2.3 as an example. The trigger process of this scheduling optimization-related bug includes the exploration of the *Always Inline* rule and the exploitation of the predicate analysis code. Where the predicate analysis code fails to properly handle the boolean expressions `C<=4` and `C<4`. While edge coverage can indicate how thoroughly predicate-related conditions have been exploited, it does not account for special situations that developers may overlook and fail to implement in the code. To better compare the exploitation capabilities of different techniques, we introduce the concept of rule-operator coverage. This metric considers the connection between scheduling rules and operator types, aiming to calculate how many operators have been tested under each corresponding rule. It can provide a rough estimate of the degree of exploitation in scheduling optimization.

Let $COV(R, \Omega)$ be the rule-operator coverage for the given sequence of rules R (for each TE program, TVM will search a sequence of rules and apply them in order) and operator type set Ω. The coverage examples can be expressed as pairs (r_{ij}, ω_k), where r_{ij} represents a rule pattern in the sequence R (we use the rule pattern here since the application of some rules depends on the prior application of other rules, which makes the application of rules like a pattern) and ω_k represents an operator type in the set Ω.

```
# program 1
R:[Always Inline, Multi-level Tiling, Add Cache Stage]
Ω:[min, *, -]
# coverage(6)
r12,ω1: (Always Inline → Multi-level Tiling, min)
r12,ω2: (Always Inline → Multi-level Tiling, *)
r12,ω3: (Always Inline → Multi-level Tiling, -)
r23,ω1: (Multi-level Tiling → Add Cache Stage, min)
r23,ω2: (Multi-level Tiling → Add Cache Stage, *)
r23,ω3: (Multi-level Tiling → Add Cache Stage, -)

# program 2
R:[Multi-level Tiling, Add Cache Stage]
Ω:[*, +]
# coverage(1)
~~r23,ω2 : (Multi-level Tiling → Add Cache Stage, *)~~
r23,ω4 : (Multi-level Tiling → Add Cache Stage, +)
```

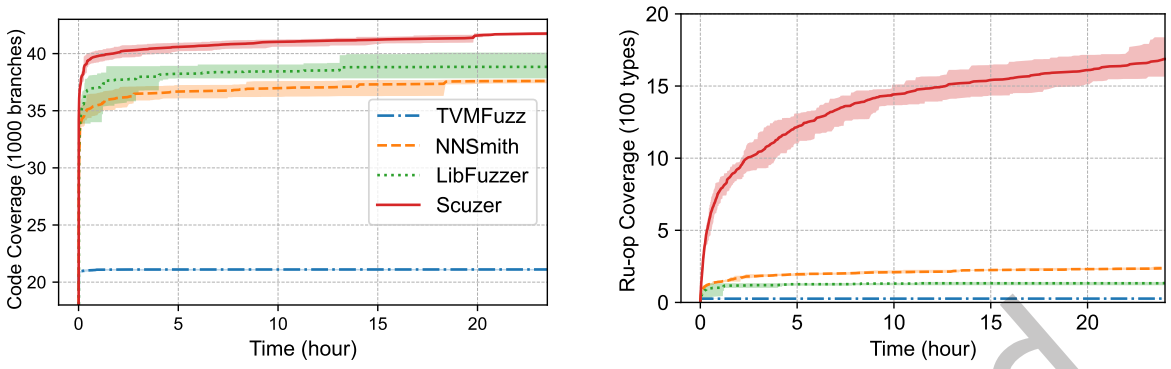
Fig. 9. A specific example of rule-operator coverage. The rule-operator coverage of the two programs is calculated in order.

More specifically, as shown in Figure 9. The first program triggers the application of three rules: *Always Inline*, followed by *Multi-level Tiling*, and finally *Add Cache Stage*. This results in two rule application patterns: *Multi-level Tiling* → *Multi-level Tiling* and *Multi-level Tiling* → *Add Cache Stage*. We estimate the variety of the program by counting the number of included operator types and define the connection between each operator and the rule application pattern as a coverage metric, which represents the degree of exploitation. The first program can increase coverage by six, while the second program only increases coverage by one since the remaining coverage has already been achieved by the first program.

Number of Detected Bugs. In line with prior work on testing or fuzzing the DL compiler [7, 12, 26], we also report the number of bugs detected by Scuzer and compare it to the baselines.

4.3 RQ1: Performance Comparison with Baselines

Just Accepted



(a) Code coverage

(b) Rule-operator coverage

Fig. 10. Coverage comparison with baselines.

4.3.1 Code Coverage and Rule-operator Coverage. To mitigate the potential impact of varying library imports among different techniques (e.g., NNSmith importing Graph IR's library files while LibFuzzer does not), We collect seed files from the four compared works that contribute to the code coverage increase during the fuzzing process and then run these seeds again in the same testing program to get final coverage result. As shown in Figure 10a, We set the generation time of the collected seed files as the x-axis and reevaluated their code coverage under the same environment to ensure consistent evaluation, using it as the y-axis. Note that we scaled down the vertical axis (1000 branches for each value in code coverage and 100 types for rule-operator coverage) to make the plot more compact.

In code coverage, Scuzer is able to beat other compared techniques at the very beginning and eventually achieves 7.4% higher coverage than the 2nd-best baseline (i.e., LibFuzzer). TVMFuzz's code coverage is significantly lower than the other techniques as its generated programs only contain a single tensor, making it difficult to trigger any scheduling optimization. While NNSmtih has made commendable strides in testing high-level optimizations, our experiments reveal its limitation in testing scheduling optimization. Being a generation-based method in relay IR, it faces challenges in testing deeper code paths at this layer due to the absence of high-quality seeds and IR-specific mutations. Using the same seeds as Scuzer, LibFuzzer outperforms other state-of-the-art techniques in code coverage. However, since LibFuzzer is a general-purpose fuzzer, its lack of knowledge regarding the specific structure at the scheduling level and the absence of optimization-related mutations hinder its ability to acquire higher code coverage than Scuzer.

To make the coverage evaluation more precise, We performed code analysis on the files related to scheduling optimization in TVM. As shown in the Figure 11. The code region related to scheduling optimization consists of two parts: the optimization implementation part, located in the `/src/te/schedule` and `/src/tir/schedule` folders, and the optimization predicate analysis part in the `/src/arith` folder. The coverage of the optimization implementation code indicates the triggering of the optimization, corresponding to the exploration goal. The coverage of the optimization predicate analysis code indicates the degree of testing different structure samples under specific optimization, corresponding to the goal of exploitation. According to the results, Scuzer achieves a total coverage of 14,266.4 in the scheduling optimization region, which is 9.8% higher than the second-highest technique, LibFuzzer. While LibFuzzer achieves similar code coverage in exploring scheduling optimizations (implementation part), its performance in exploiting the optimization predicate analysis part is worse.

Just Accepted

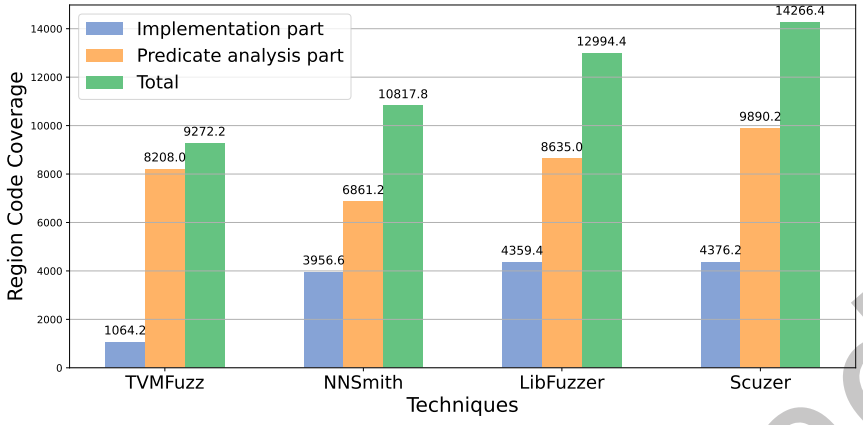
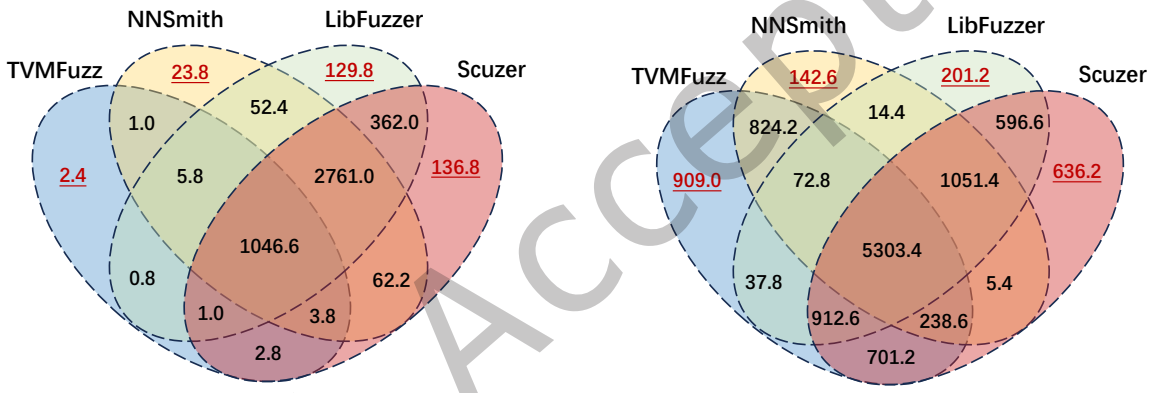


Fig. 11. Region code coverage comparison with baselines. The shown coverage result is the average of five experiments.



(a) Implementation part

(b) Predicate analysis part

Fig. 12. Venn diagram of region code coverage. The shown coverage result is the average of five experiments.

To show the unique coverage for each studied technique, Figure 12 further breaks down the coverage sets of different fuzzers through Venn diagrams. From the figure, we can observe that Scuzer covers different areas compared to the other fuzzers. Specifically, in terms of the implementation part, Scuzer can achieve 136.8 unique coverages, which is comparable to LibFuzzer. Additionally, Scuzer can attain 3× the unique coverage of LibFuzzer in the predicate analysis part. Although TVMFuzz can achieve higher unique code coverage than Scuzer in this part, it performs poorly in the implementation part, achieving almost no unique coverage.

Figure 10b presents our results on the rule-operator coverage. In terms of rule-operator coverage, NNSmith demonstrates superior performance compared to LibFuzzer. This advantage can be attributed to NNSmith's specific design for operators, whereas LibFuzzer lacks access to this structural-specific information. Although LibFuzzer achieves better code coverage, it may not comprehensively evaluate the ability of testing optimization. We propose the rule-operator coverage metric to address this limitation and enable a more comprehensive comparison. In the rule-operator comparison, we observe that Scuzer outperforms the second-ranked NNSmith

Just Accepted

by a margin of 7 times, indicating that Scuzer can test significantly more operator types on the specific rule, which exactly matches our design goal in testing scheduling optimizations.

4.3.2 Objective-related Metrics. In addition to comparing the aforementioned coverage metrics, we conducted further analysis by examining the average number of operator types, optimization patterns, and the length of the optimization sequence observed in collected seeds. Figure 13 shows the results. Regarding the operator types metric, TVMFuzz exhibits higher values compared to other approaches, owing to its generation structure based on operator selection. However, due to its limitation of generating programs with only single tensors, its capability to trigger scheduling optimizations is constrained, resulting in a zero value in the optimization patterns metric. Scuzer outperforms other existing approaches across the remaining metrics. While its performance in terms of median metrics is not higher than LibFuzzer, it surpasses LibFuzzer in terms of scope. This indicates that Scuzer is able to mutate and generate seeds with a greater variety of operator types and optimization quantities.

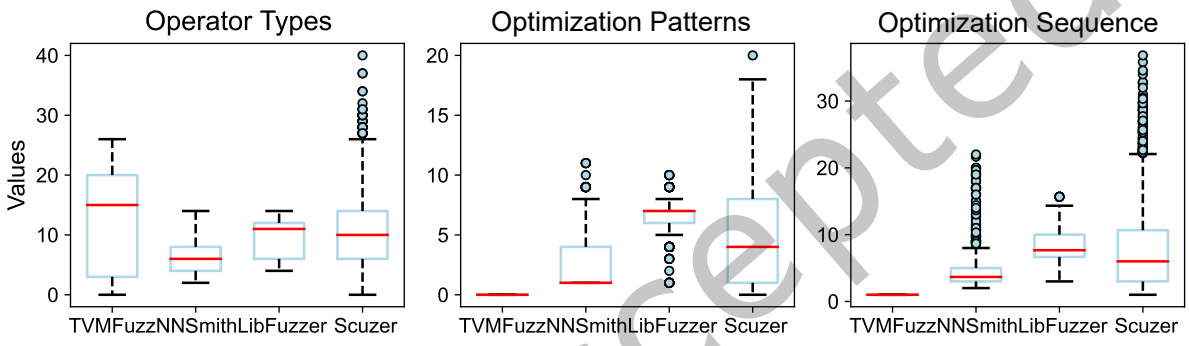


Fig. 13. Comparison with state-of-the-arts on the number of operator types, optimization patterns, and the length of the optimization sequence.

Table 2. Comparison of vulnerability funding capability. The table lists the bug results of each repeat experiment and summarizes the number of different bugs found in the five experiments in total.

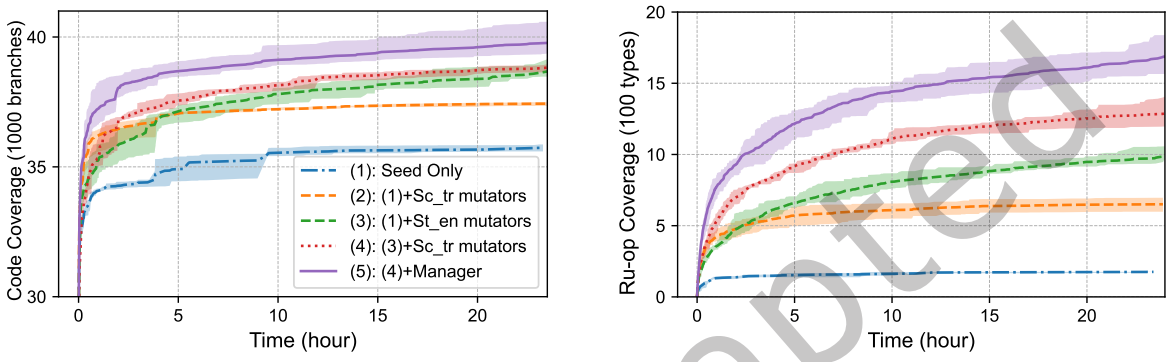
Techniques	1	2	3	4	5	Total
Scuzer	8	7	5	9	7	10
TVMFuzz	0	0	0	0	0	0
NNSmith	1	1	1	0	1	2
LibFuzzer (Scuzer seeds)	0	0	0	0	0	0

4.3.3 Vulnerability Finding Capabilities. We also conducted a comparative analysis of the vulnerability finding capabilities of these techniques. Here, we use manual analysis to find vulnerabilities. We de-duplicate bug triggers based on the reported error messages, which include the triggered test oracle and corresponding optimization types, as well as the characteristics of trigger programs such as operator types and tensor shapes. After preliminary screening of the bugs, we automatically convert the remaining triggers into executable Python files to reproduce the bugs. During the 24-hour fuzzing period of five experiments, Scuzer demonstrated its effectiveness by identifying 10 different bugs in total. TVMFuzz did not trigger any scheduling optimizations, as indicated in Figure 13, leading to no bug discoveries. NNSmith exhibited some bug-finding ability, successfully finding 2

Just Accepted

inconsistent bugs. Both of them are related to the overflow of data types including int64 and int16. LibFuzzer reported numerous issues. However, they all turned out to be false positives upon further investigation. Most of the issues cannot be reproduced when converting the testing environment from C to Python. The remaining issues are related to floating-point precision loss. For instance, when an operator multiplies a minor error by a larger number, the error in the result is amplified.

4.4 RQ2: Effectiveness of each Component



(a) Code coverage

(b) Rule-operator coverage

Fig. 14. Coverage study for different components. The Sc_tr mutators are the abbreviation of schedule-triggered mutators, and the St_en mutators are the abbreviation of structure-enriching mutators.

4.4.1 Overall Comparision. To address RQ2, we initially create a variant of SCUZER specifically designed to test the initial seed files. Subsequently, we incorporate structure-enriching mutators, schedule-triggering mutators, and manager modules into this variant. The coverage trends after adding each component are depicted by curves (1) through (5) in Figure 14a and Figure 14b. Note that we directly use the code coverage information collected during the fuzzing process of SCUZER here. This makes the final coverage curve of SCUZER in Figure 10a and Figure 14a slightly different. From curves (1) to (2) and (1) to (3), we can observe that structure-enriching mutators and schedule-triggering mutators have positive effects on both coverage metrics. Curves (3) and (4) further confirm the effectiveness of schedule-triggering mutators in addition to structure-enriching mutators. According to the improvement in rule-operator coverage, it is evident that the inclusion of schedule-triggering mutators can significantly enhance the efficiency of testing scheduling optimization. Lastly, upon comparing curves (4) and (5), we observe that the manager module improves code coverage by 2.4%, contributing approximately one-quarter of the total increased coverage for SCUZER. This demonstrates that our multi-objectives algorithm outperforms random selection, achieving deeper code paths and higher scheduling optimization testing ability.

4.4.2 Effectiveness of the Generator Module. We conduct further investigations into the effectiveness of the generator. As depicted in Algorithm 1, the generator utilizes a dictionary that stores single-operator testing results to guide the computation tree generation process. To assess its impact, we create a technique variant denoted as SCUZER$_r$, which generates a computation tree by randomly selecting operators. We then run SCUZER$_r$ alongside SCUZER for 4 hours, recording the number of valid tests (i.e., tests that passed the compile process). The results indicate that SCUZER outperforms SCUZER$_r$, generating 5419 valid tests compared to SCUZER$_r$'s 3600 tests.

Just Accepted

Table 3. Comparison of objectives values under different variants.

No	Variants	Time	Size	Variety	Complexity
1	Seed only	11725.2	144	8.47	4.86
2	1+Sc_tr mutators	24317.2	511	10.03	4.96
3	1+St_en mutators	25531.4	505.8	9.35	4.65
4	3+Sc_tr mutators	22973.4	585	9.90	4.02
5	4+Manager	45391.8	577.2	10.66	4.74

4.4.3 Effectiveness of the Manager Module. To evaluate the manager's effectiveness, we calculate the average values of four different variants under different objectives and present them in Table 3. To illustrate, the variant with manager components outperformed the other variants in the TIME objective, with 45,391 iterations, which is 78% higher than the second-best variant. And it also positively impacted the VARIETY objective, leading to a 6% increase in value. The scheduling-trigger mutator negatively impacts the average value of COMPLEXITY objectives when schedule-triggered mutators cooperate with structure-enriching mutators. However, the inclusion of the manager component effectively eliminates this adverse effect. Note that the manager component's performance in the SIZE objective is not optimal. This can be attributed to its adoption of a multi-objective algorithm during the fuzzing process, which leads to specific periods of focus primarily on the COMPLEXITY and VARIETY objectives. More structures in a program are required to fulfill these objectives, resulting in an increased demand for larger seed size.

4.5 RQ3: Bug Root Cause Analysis and Case Studies

Table 4. Summary of detected bugs.

Bug Type	**Total Number**	**Comfired**	**Fixed**
Result Inconsistency	9	7	3
Compilation Inconsistency	5	1	1
Crash	2	1	1
Infinite loop	1	1	0
Total	17	10	5

As shown in Table 4, we found 17 bugs in the *0.9.0* and *0.10.0dev* version of TVM, 9 of them are related to result inconsistency, 5 are compilation inconsistency, and 2 are due to crash. The remaining bug is because TVM uses the parameters in the TE program to set the number of 'for loops' during the function process, but ignores the case where the parameters are negative, resulting in a large number of 'for loops'. To date, 10 of these bugs have been confirmed, and 5 of them have been fixed. Next, we show two confirmed typical bugs to better illustrate the relevance of the found bugs to schedule and our mutators (the details of other found bugs are summarized on our Github).

Mis-optimization of asymmetric operations. This bug is triggered by the operator mutator. As shown in Figure 15, this bug modifies the originally defined symmetric addition in the `common_reduce` API to an asymmetric subtraction. Before applying the schedule, the value in array `B` equals `0` minus the value in array `A`. But after applying the schedule, the program allocates a new array `B_rf_1` for storing intermediate values. The value in `B_rf_1` is calculated in the first for loop. Then in the latter for loop, the value in `B_rf_1` is taken to calculate the value in the array `B`. This allocation to `B_rf_1` divides the calculation process into two independent phases,

Just Accepted

Buggy TE program

```
reduce_fun = comm_reducer(lambda x, y:x - y, lambda t:const(0, dtype=t))
A = placeholder((64, 64))
k = reduce_axis((0, 64))
B = compute((64,), lambda i:reduce_fun(A[i, k], axis=k))
```

Unscheduled runtime program

```
for i in range(64):
    B[i] = T.float32(0)
    for k in range(64):
        A_1 = T.Buffer((4096,), data=A.data)
        B[i] = B[i] - A_1[i * 64 + k]
```

Scheduled runtime program

```
for i_k_outer_fused in T.parallel(2048):
    cse_var_1 = i_k_outer_fused * 2
    B_rf_1[i_k_outer_fused] = T.float32(0)
    A_1 = T.Buffer((4096,), data=A.data)
    B_rf_1[i_k_outer_fused] = B_rf_1[i_k_outer_fused] - A_1[cse_var_1]
    B_rf_1[i_k_outer_fused] = B_rf_1[i_k_outer_fused] - A_1[cse_var_1 + 1]
for ax0 in T.parallel(64):
    B[ax0] = T.float32(0)
    for k_outer_v in range(32):
        B[ax0] = B[ax0] - B_rf_1[ax0 * 32 + k_outer_v]  <--Actual
-------------------------------------------------------------------------------------------------------------
        B[ax0] = B[ax0] + B_rf_1[ax0 * 32 + k_outer_v]  <--Expected
```

Fig. 15. A bug related to data reuse.

Just Accepted

allowing for parallel calculation or data-dependent optimization. However, it alters the computation process of the output value by converting the single subtraction into a double subtraction, resulting in an addition operation, which contradicts the previous subtraction. The correct runtime program is expected to use the addition operation, not the subtraction, to calculate the final result.

Incomplete divide-by-zero detection. The compilation inconsistency test oracle identified a bug originating from the operator mutator, which replaces the original operators with operators like `te.floormod` and `tir.Div`, leading to division by zero in the program. To clarify, before applying the schedule, the values in `tensor_1` are updated during iterations of `rv0` and `rv1`. When either `rv0` or `rv1` becomes zero, it triggers a division by zero in the subscript operation of the `placeholder_1` array (i.e., `(ax2 + 2) % rv0`). However, this TE program passed compilation with no error message. Interestingly, after the schedule is applied, the program detects this problem and reports it during compilation. It is hard to find such a bug without checking for the consistency of the compilation process since the program only reports an error message after the deployment of scheduling optimization.

These case studies show that by incorporating schedule-triggering mutators and structure-enriching mutators with an efficient multi-objective algorithm, Scuzer is capable of uncovering a wider range of TVM bugs that existing approaches may struggle to identify.

Buggy TE program

```
placeholder_1 = placeholder((1, 64, 52, 52, 4))
rv0 = reduce_axis((0, 2))
rv1 = reduce_axis((0, 2))
tensor = compute((1, 64, 26, 26, 4), lambda ax0, ax1, ax2, ax3, ax4:
         max(placeholder_1[ax0, ax1, floormod((ax2 + 2), rv0),
         (Div((ax3*2), rv1)), ax4], axis=[rv0, rv1])
     )
```

Unscheduled runtime program

```
for rv0, rv1 in T.grid(2, 2):
    cse_var_1 = ax1 * 2704 + ax2 * 104 + ax3 * 4 + ax4
    placeholder_1 = T.Buffer((692224,), data=placeholder.data)
    tensor_1[cse_var_1] = T.max(tensor_1[cse_var_1], placeholder_1[ax1* 10816
        + (ax2 + 2) % rv0 * 208 + T.Div(ax3 * 2, rv1) * 4 + ax4])
```

Scheduled runtime program

```
"Check failed: pb->value != 0 (0 vs. 0) : Divide by zero"
```

Fig. 16. A bug caused by inconsistency compilation.

5 THREATS TO VALIDITY

Threat to Internal Validity: Though we use single-operator testing to help record operator-related information, not all operators are suitable for this method. For example, the output type of the IF_THEN_ELSE operator actually depends on whether the conditional statement is true or false. The output type varies if the conditional statement differs. In response to this situation, the method we adopted was to conduct multiple repeated tests to improve the relevant information of the operator as much as possible.

Threat to External Validity: To optimize runtime performance and minimize overhead, TVM adopts a strategy of not strictly detecting out-of-bounds programs, which may lead to inconsistent results in the report files due to out-of-bounds access. To address this issue, we focus on the access between different tensors and designed effective access strategies during the design of mutators. These strategies have been proven to help in handling such situations, ensuring more reliable and accurate results in test oracle reports. In the process of testing TVM, we have also observed that the scheduling itself introduces a loss of precision, resulting in minor inconsistencies in the final results. If the program amplifies this error during its calculations (e.g., by multiplying by a very large number), it can lead to false positives in our program. To mitigate this, we manually filter out false positives that exhibit a small gap in the final result and with specific features that may cause the loss of precision. Another external threat comes from the generalizability of SCUZER.

6 GENERALITY AND LIMITATIONS

What distinguishes the DL compiler from the DL library is the introduction of scheduling optimization, which allows the model to be optimized for different chip structures. Previous research has not sufficiently prioritized the testing of DL compilers, neglecting the aspect of orthogonal scheduling optimization. SCUZER is the first testing tool specifically designed to address this crucial function of DL compilers. However, due to its specialized design, SCUZER inevitably faces issues related to generality and certain limitations.

The generality of SCUZER: While SCUZER is a technique specifically designed for TVM, its methodology is

Just Accepted

universal and has the potential to be applied to other DL compilers. Despite the different definitions and designs for scheduling optimization in various DL compilers, their fundamental techniques are similar. By modifying the implementation of mutators, the technique can be adapted for the scheduling optimization of different DL compilers. Furthermore, the multi-objective algorithm proposed by Scuzer is also applicable to other optimization frameworks. The proposed objectives are not dependent on a specific program structure and encompass universal indicators such as triggered optimization, program diversity, testing speed, and program size. Additionally, the rule-operator coverage introduced can serve as a reference for evaluating the exploration and exploitation capabilities of the existing tools which focus on the testing of optimization framework.

The limitations of Scuzer and future work: Scuzer also has certain limitations. It only takes TVM as the test object, and most of the code is implemented using TVM's open API, which makes it difficult to directly test other DL compilers. In this work, we only tested scheduling optimization on specific CPU hardware. While software bugs may not differ significantly across different hardware platforms, variations in testing performance on various hardware cannot be ruled out. One of our future works is to investigate the differences in testing DL compilers on different hardware. Moreover, Scuzer currently focuses solely on the testing of scheduling optimization. In the future, we plan to integrate Scuzer with testing tools for other optimization layers to create a comprehensive testing tool that covers all layers of DL compilers.

7 RELATED WORK

7.1 DL System Testing

The testing of DL libraries and compilers has garnered significant interest, with several notable works in this area. DocTer [35], FreeFuzz [36] and RelFuzz [37] use extraction and inference methods to obtain API static and dynamic information from documents and open source code to perform API testing of DL libraries. CRADLE [38] introduces differential testing to identify potential inconsistencies by directly executing existing DNN models on different DL libraries. LEMON [18] employs mutation rules for DL models in conjunction with a heuristic strategy to guide the model generation process. NNSmith [12] further enhances model and input generation effectiveness through symbolic constraint solving and gradient-driven search. In contrast to NNSmith, HirGen [26] focuses on the high-level optimization stage and introduces three operator-related coverage criteria to guide model generation. While the mentioned works effectively test DL compilers, they overlook the different compilation stages. Addressing this gap, TVMFuzz [13] uses predefined grammar rules to generate arbitrary low-level IRs automatically. More recently, Tzer [7] employs coverage feedback to perform joint mutation of both the low-level IR and optimization passes for TVM. Although Tzer has shown promising results over TVMFuzz, its low-level IR mutation may not adequately test higher-level optimizations, as demonstrated in NNSmith's experiments. Despite the significant testing efforts dedicated to high-level and low-level optimizations in DL compilers, the core scheduling optimization has yet to receive specific attention in existing works.

7.2 Compiler Testing

The field of compiler testing has witnessed significant research efforts. Csmith [39] is a technique that generates C programs containing various language constructs, such as control flow statements, structs, arrays, and C expressions, to explore C compilers. Another technique, LangFuzz [40], has been successful in uncovering bugs in the JavaScript interpreter of Mozilla Firefox. It creates a large pool of code fragments generated from a set of sample input files using a parser. To address issues related to undefined behaviors and to create more diverse test programs, mutation-based techniques [41, 42, 42] have been proposed for fuzzing compilers. These techniques involve mutating existing seed input programs to produce new test cases. Moreover, some studies have adopted the concept of Equivalence Modulo Inputs (EMI) to develop semantics-preserving mutation techniques [24, 43–45]. These techniques are particularly useful for triggering compiler optimizations like inlining and

Just Accepted

dead code elimination while ensuring that the overall semantics of the program remains intact. However, when it comes to testing the newly introduced scheduling optimization, existing compiler fuzzing techniques may encounter challenges. While these techniques can be applied to C-like low-level IR in DL compilers, they may not be fully suitable for effectively testing the newly introduced scheduling optimization.

8 CONCLUSION

In this work, we develop a scheduling optimization-oriented fuzzer for TVM, namely Scuzer, which includes four schedule-triggering mutators and five structure-enriching mutators. To maximize the overall effectiveness, Scuzer also employs a multi-objective algorithm to manage the selection of seeds and mutators. Our evaluation results demonstrate the superiority of Scuzer compared to state-of-the-art approaches. Specifically, Scuzer achieves a 7.4% increase in edge coverage and a remarkable 7× improvement in our introduced rule-operator coverage. Additionally, we validate the effectiveness of all proposed mutators with a well-designed generator and the multi-objective algorithm. Moreover, Scuzer successfully detects 17 previously unknown bugs, with 14 related to scheduling optimization, and 5 have been fixed in the latest version.

ACKNOWLEDGMENTS

We thank reviewers for their constructive feedback. This research is supported by the Key R&D Program of Zhejiang under Grant 2022C01018, the NSFC Program under Grant 62102359, the Ministry of Education, Singapore under its Academic Research Fund Tier 2 (Award ID: T2EP20222-0037), and the Cryptography Research Experimental Environment Platform Construction Project under Grant 202204.

Just Accepted

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

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

[4] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. arXiv:1805.00907 https://arxiv.org/abs/1805.00907

[5] 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. (2018). arXiv:1801.08058 https://arxiv.org/abs/1801.08058

[6] 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. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[7] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–26.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv:1512.01274 https://arxiv.org/abs/1512.01274

[9] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

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

[11] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.

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

[13] David Pankratz. 2020. TVMFuzz: Fuzzing Tensor-level Intermediate Representation in TVM. Retrieved March 18, 2024 from https://github.com/dpankratz/TVMFuzz

[14] Ehsan M. Kermani. 2024. Tensor Expression API. Retrieved March 18, 2024 from https://tvm.apache.org/docs/reference/api/python/te.html

[15] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[16] Gen Zhang, Pengfei Wang, Tai Yue, Xiangdong Kong, Shan Huang, Xu Zhou, and Kai Lu. 2022. MobFuzz: Adaptive Multi-objective Optimization in Gray-box Fuzzing. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-199/

[17] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

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

[19] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–28.

[20] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. 2022. DietCode: Automatic optimization for dynamic tensor programs. *Proceedings of Machine Learning and Systems* 4 (2022), 848–863.

[21] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the gap between auto-tuners and hardware-native performance. *Proceedings of Machine Learning and Systems* 4 (2022), 204–216.

[22] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 977–989.

[23] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709.

[24] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 56–68.

[25] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.

[26] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing Deep Learning Compilers with HirGen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 248–260. https://doi.org/10.1145/3597926.3598053

[27] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A Padua. 1993. Automatic program parallelization. *Proc. IEEE* 81, 2 (1993), 211–243.

[28] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. 1991. The I test: an improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel & Distributed Systems* 2, 03 (1991), 342–349.

[29] E Knuth Donald et al. 1999. The art of computer programming. *Sorting and searching* 3, 426-458 (1999), 4.

[30] William Pugh. 1992. A practical algorithm for exact array dependence analysis. *Commun. ACM* 35, 8 (1992), 102–114.

Just Accepted

[31] Paul M Petersen and David A Padua. 1996. Static and dynamic evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems* 7, 11 (1996), 1121–1132.
[32] Rajeev Agrawal. 1995. Sample mean based index policies by o (log n) regret for the multi-armed bandit problem. *Advances in applied probability* 27, 4 (1995), 1054–1078.
[33] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing Automatic Differentiation in Deep-Learning Libraries. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1174–1186. https://doi.org/10.1109/ICSE48619.2023.00105
[34] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308
[35] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–188.
[36] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.
[37] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. 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* (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 44–56. https://doi.org/10.1145/3540250.3549085
[38] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. https://doi.org/10.1109/ICSE.2019.00107
[39] 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.
[40] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler
[41] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. *IPSJ Transactions on System LSI Design Methodology* 7 (2014), 91–100.
[42] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.
[43] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.
[44] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76.
[45] 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.

Received 31 March 2024; revised 20 July 2024; accepted 16 October 2024

Just Accepted