PolyJuice: Detecting Mis-compilation Bugs in Tensor Compilers with Equality Saturation Based Rewriting

CHIJIN ZHOU, BNRist, Tsinghua University, China
BINGZHOU QIAN, National University of Defense Technology, China
GWIHWAN GO, BNRist, Tsinghua University, China
QUAN ZHANG, BNRist, Tsinghua University, China
SHANSHAN LI, National University of Defense Technology, China
YU JIANG*, BNRist, Tsinghua University, China

Tensor compilers are essential for deploying deep learning applications across various hardware platforms. While powerful, they are inherently complex and present significant challenges in ensuring correctness. This paper introduces PolyJuice, an automatic detection tool for identifying mis-compilation bugs in tensor compilers. Its basic idea is to construct semantically-equivalent computation graphs to validate the correctness of tensor compilers. The main challenge is to construct equivalent graphs capable of efficiently exploring the diverse optimization logic during compilation. We approach it from two dimensions. First, we propose arithmetic and structural equivalent rewrite rules to modify the dataflow of a tensor program. Second, we design an efficient equality saturation based rewriting framework to identify the most simplified and the most complex equivalent computation graphs for an input graph. After that, the outcome computation graphs have different dataflow and will likely experience different optimization processes during compilation. We applied it to five well-tested industrial tensor compilers, namely PyTorch Inductor, OnnxRuntime, TVM, TensorRT, and XLA, as well as two well-maintained academic tensor compilers, EinNet and Hidet. In total, PolyJuice detected 84 non-crash mis-compilation bugs, out of which 49 were confirmed with 20 fixed.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Tensor Compiler Testing, Equality Saturation, Fuzzing, ML System

ACM Reference Format:
Chijin Zhou, Bingzhou Qian, Gwihwan Go, Quan Zhang, Shanshan Li, and Yu Jiang. 2024. PolyJuice: Detecting Mis-compilation Bugs in Tensor Compilers with Equality Saturation Based Rewriting. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 317 (October 2024), 27 pages. https://doi.org/10.1145/3689757

1 Introduction

Over the past decade, the ever-increasing demand for efficient and scalable execution of deep learning (DL) models has driven a surge of interest in tensor program compilation in both academic research and industry. The core process is to transform high-level tensor programs into optimized low-level code that can be efficiently executed on a specific hardware architecture. Modern tensor compilers, such as TensorRT [33], XLA [48], and TVM [5], carry out their compilation

*Yu Jiang is the corresponding author.

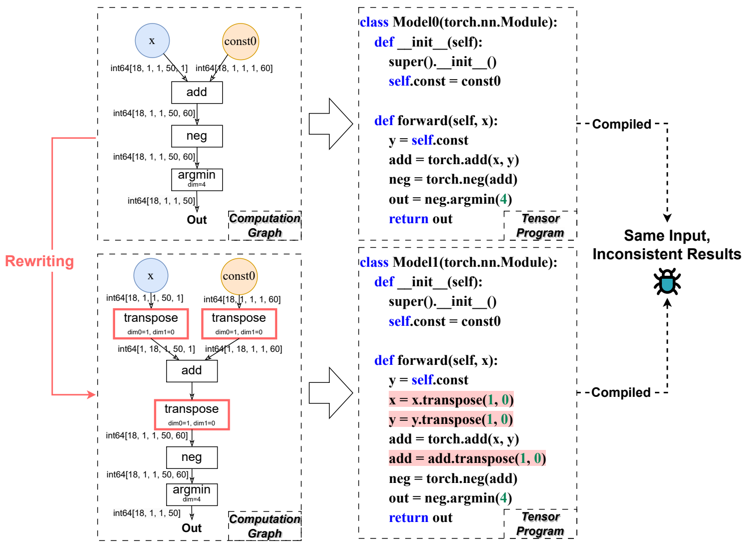
Authors' Contact Information: Chijin Zhou, BNRist, Tsinghua University, Beijing, China; Bingzhou Qian, National University of Defense Technology, Changsha, China; Gwihwan Go, BNRist, Tsinghua University, Beijing, China; Quan Zhang, BNRist, Tsinghua University, Beijing, China; Shanshan Li, National University of Defense Technology, Changsha, China; Yu Jiang, BNRist, Tsinghua University, Beijing, China.

ACM 2475-1421/2024/10-ART317
https://doi.org/10.1145/3689757

tasks as a series of tensor program transformations. By implementing transformations such as algebraic simplification and operator fusion, these compilers can optimize memory usage, decrease computational overhead, and enhance hardware utilization [24].

Despite the proliferation of optimization techniques in tensor program compilation, their correctness is not always guaranteed. These techniques require multiphase intricate transformations on the intermediate representation (IR) of the compiled tensor program, making the process error-prone. However, evaluations of most tensor compiler research [5, 53, 68, 70] tend to focus more on optimization efficiency, often neglecting the aspect of correctness. Consequently, potential bugs may remain concealed within their implementations. For example, versions of TVM from February to April 2023 incorrectly simplifies "$y - (x\%2)$" to "$y + (x\%2) - 1$" because of a minor implementation bug [50]. Such bugs caused by incorrect transformation implementation during compilation are referred to as *mis-compilation bugs.*

Different from execution failures, which often present clear symptoms such as program crashes, mis-compilation bugs tend to emerge silently, making detection particularly challenging. Most current tensor compiler fuzzers [9, 11, 27–29, 59] focus on identifying execution failures, with a few [27, 28, 30] employing differential testing to uncover mis-compilation bugs. Differential testing achieves this by comparing the execution results of target compilers with those of a reference executor. However, differential testing for tensor compilers produces a non-negligible number of false positives due to variations in numerical accuracy across different execution environments. For instance, Xiao et al. [61] reported that differential testing yields thousands of inconsistencies when comparing the execution results between TensorFlow and Glow [39]. Such discrepancies pose challenges in distinguishing actual bugs from benign differences in numerical precision. This situation underscores the urgent need for a new approach to complement differential testing.



```
class Model0(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.const = const0

  def forward(self, x):
    y = self.const
    add = torch.add(x, y)
    neg = torch.neg(add)
    out = neg.argmin(4)
    return out
```

```
class Model1(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.const = const0

  def forward(self, x):
    y = self.const
    x = x.transpose(1, 0)
    y = y.transpose(1, 0)
    add = torch.add(x, y)
    add = add.transpose(1, 0)
    neg = torch.neg(add)
    out = neg.argmin(4)
    return out
```

Fig. 1. An equivalent pair of tensor programs generated by PolyJuice. They produce significantly inconsistent execution results after compiled by TVM, triggering a mis-compilation bug.

This paper introduces PolyJuice, a fuzzer for identifying mis-compilation bugs in tensor compilers, aiming to improve the robustness of exisiting tensor compilers. Inspired by the idea of equivalence modulo inputs (EMI) [21], this paper addresses the limitations of differential testing by detecting inconsistent results between *multiple tensor programs that are semantically equivalent but syntactically different* on the same execution environment. Fig. 1 provides an illustrative example of how PolyJuice uncovers mis-compilation bugs. The two models depicted in the figure are clearly equivalent, with only minor changes in the shapes of some intermediate tensors. These changes are supposed not to influence the outputs. If these two models, when passed to the same compiler in the same execution environment, produce different results for the same inputs, it indicates the presence of a mis-compilation bug in the compiler. This approach eliminates the need for a separate reference executor, thereby avoiding the false positives often caused by cross-platform numerical accuracy variations, enhancing the reliability of bug detection.

Constructing equivalent tensor programs that can efficiently detect mis-compilation bugs is challenging. (1) **Equivalence**. Traditional compiler testing techniques [21, 22, 49] often construct equivalent programs by inserting dead code, e.g., `if(false){...}`, into a given program. However, this approach cannot be directly applicable to tensor programs. Tensor compilers primarily analyze the flow of tensors, and as such, ignore control-flow-wise dead code by design. An intuitive thought is to modify the tensors' flow of a tensor program, but it is non-trivial to maintain programs' equivalence after the modification. (2) **Test Efficiency**. The testing throughput for tensor compilers is much lower than for other applications (roughly 4 testcases per second [27] v.s. 10,000 testcases per second [72]). Therefore, each pair of equivalent tensor programs needs to be carefully constructed to maximize the potential for detecting mis-compilation bugs. However, even though the equivalence is maintained, tensor programs with naive modifications tend to experience the same optimization transformations, which is unlikely to expose logic bugs in these transformations. Consequently, a more effective modification strategy is needed to diversify the transformations that the tensor programs undergo. We address these challenges from two dimensions:

First, we propose arithmetic and structural rewrite rules to modify the dataflow of a tensor program. Arithmetic rewriting rearranges mathematical expressions without altering the final output. This is achieved by manipulating the computation graph using arithmetic properties like commutativity and associativity. On the other hand, structural rewriting changes the sequence or structure of tensor operations while keeping the final output intact, achieved by adjusting how intermediate tensors are transposed, sliced, and concatenated. The rationale behind this is that modifying arithmetic procedures and tensor shapes within tensor programs can often trigger diverse optimization opportunities.

Second, we design an efficient rewriting framework to apply the rewrite rules on a given computation graph and identify "representative" equivalent graphs. We adopt equality saturation technique [60] to find all graphs equivalent to the original graph based on the rules. After that, we search for two types of graphs among these equivalent graphs: the one that calculates its outputs using the fewest number of operations (the most simplified graph), and the one that does so using the maximum number of operations (the most complex graph). The two extreme cases have significantly different dataflow and likely experience different optimization processes during compilation, which makes them valuable for tensor compiler testing.

We implemented PolyJuice as a practical fuzzer and applied it to five well-tested industrial tensor compilers, namely PyTorch Inductor [38], OnnxRuntime [31], TVM [5], TensorRT [33], and Tensorflow XLA [48], as well as two well-maintained academic tensor compilers, EinNet [70] and Hidet [7]. In total, PolyJuice detected 84 *non-crash* mis-compilation bugs, out of which 49 were confirmed with 20 fixed. In a 48-hour experiment, compared to the state-of-the-art fuzzers NNSmith [27] and MT-DLComp [61], PolyJuice additionally uncover 7 and 25 non-crash mis-compilation bugs on TVM and XLA, respectively. These results collectively demonstrate its effectiveness.

Overall, we make the following contributions:

- **New Approach**. We proposed a new approach to detect mis-compilation bugs in tensor compilers. This is achieved by constructing equivalent tensor programs that is capable of exploring diverse optimization logic during the compilation process.
- **Practical Fuzzer**. We designed and implemented a fuzzer named PolyJuice, which leverages equality saturation to rewrite computation graphs and extract representative graphs. To the best of our knowledge, it is the first work to apply equality saturation for testing purposes. We released the prototype at https://github.com/ChijinZ/PolyJuice-Fuzzer.
- **Real-World Bugs**. We evaluated PolyJuice on seven well-maintained tensor compilers. In total, it has detected 84 non-crash mis-compilation bugs with 49 confirmed.

2 Background

2.1 Tensor Program Compilation

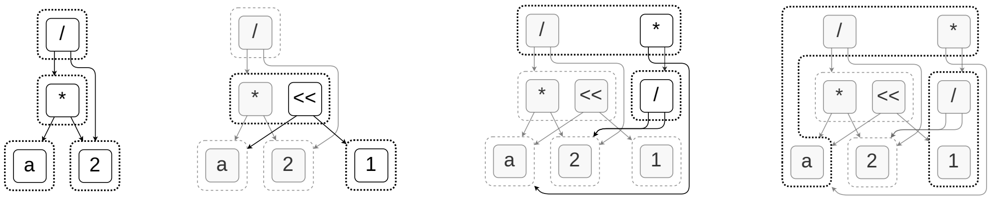
Tensor program compilation involves a series of transformations to transform high-level tensor programs into optimized low-level code that can be efficiently executed on a variety of hardware architectures. Typical optimizations in tensor program compilation mainly include algebraic simplification [14, 18, 51, 53, 63, 70], which reduces the complexity of the computation by exploiting mathematical properties and identities; operator fusion [7, 15, 16, 48, 66, 67], where multiple operators are combined into a single operator to reduce computational overhead and improve cache usage; and loop tiling [5, 40, 44, 64, 68, 69, 71, 76], which partitions computations into smaller "tiles" to exploit memory hierarchy and parallelism.

These optimization strategies, while powerful, are inherently complex and present significant challenges in ensuring bug-free implementations. For instance, when dealing with algebraic simplification, optimization developers must consider not only the semantics of the relevant operators but also the intricacies of high-dimensional data structures and the precision of data types. Developers generally implement many sanity checks to apply an algebraic simplification rule. However, if these are not thoroughly considered, errors can easily slip through, leading to potential mistakes. For example versions of TVM between February to April 2023 incorrectly simplifies "$y - (x\%2)$" to "$y + (x\%2) - 1$" during the optimization of `floormod(x,2)` [50], potentially corrupting the correctness of downstream applications.

2.2 Equality Saturation

Equality saturation [47] is a promising technique that leverage equality graph (e-graph) to implement efficient term rewrite engines. Given a set of rewrite rules, Starting from a term t, an equality saturation engine builds an initial e-graph E representing t, and then repeatedly applies the rules to expand E into a large set of equivalent terms. Fig. 2 (borrowed from the egg paper [60]) presents an e-graph for the term $(a * 2)/2$ and the ones after applying several rewrite rules. Each node along with its subtree represents a term. All terms represented by nodes in an e-class are equivalent. For example, "$*$" and "$\ll$" in Fig. 2b are in the same e-class, and therefore their represented terms $a * 2$ and $a \ll 1$ are equivalent. By maintaining the e-graph, the engine can efficiently store all equivalent terms for a given term, and thus avoid redundant computation.

The rewriting process terminates when either *saturation* is reached, meaning the rewrites added no new information to the e-graph, or until a specified timeout. After that, the engine can extract the best represented term according to a user-provided cost function. Therefore, it is often used to solve optimization problems. Many research efforts have been made to improve the efficiency of both e-graph construction and optimal term extraction [19, 35, 60]. As being more and more efficient, equality saturation has been successfully applied in the research field for program analysis [36, 65], program optimizations in compilers [63], and hardware designs [52].



(a) The e-graph of $(a * 2)/2$.

(b) After applying rewrite rule $x * 2 \longrightarrow x \ll 1$.

(c) After applying rewrite rule $(x * y)/z \longrightarrow x * (y/z)$.

(d) After applying the rules $x/x \longrightarrow 1$ and $1 * x \longrightarrow x$

Fig. 2. An example of an e-graph. Dashed boxes show e-classes, and edges connect e-nodes to their e-class children. Applying rewrites to an e-graph either adds new e-nodes and edges or merges e-classes.

3 Motivation

Test Oracle. Tensor program compilation can be seen as a sequence of program transformations for optimization, and thus the compiled program can be denoted as

$$\mathcal{P}_{compiled} = t_n(...t_1(\mathcal{P}))$$

where $\mathcal{P}$ is an input tensor program and $t_1, t_2, \ldots, t_n$ are transformations. These transformations are designed to preserve the semantic equivalence of the program while enhancing its computational efficiency. The goal of our approach is to construct an equivalent tensor program, denoted as $\mathcal{P}'$, which is distinct in terms of its dataflow from the original tensor program $\mathcal{P}$. Due to dataflow differences, this constructed program will trigger a different sequence of optimization transformations, denoted as $t_m^{'}(...t_1^{'}(\mathcal{P}'))$. This divergence in the optimization procedure serves as the basis for our testing methodology. Ideally, the execution results of $t_n(...t_1(\mathcal{P}))$ and $t_m^{'}(...t_1^{'}(\mathcal{P}'))$ are supposed to be equal for any inputs. Otherwise, we can infer that at least one of the transformations t_i is implemented incorrectly, namely a *mis-compilation bug*. We formally define our idea to detect mis-compilation bugs as follows:

$$\exists i \in I, exec(t_n(...t_1(\mathcal{P})), i) \neq exec(t_m^{'}(...t_1^{'}(\mathcal{P}')), i)$$

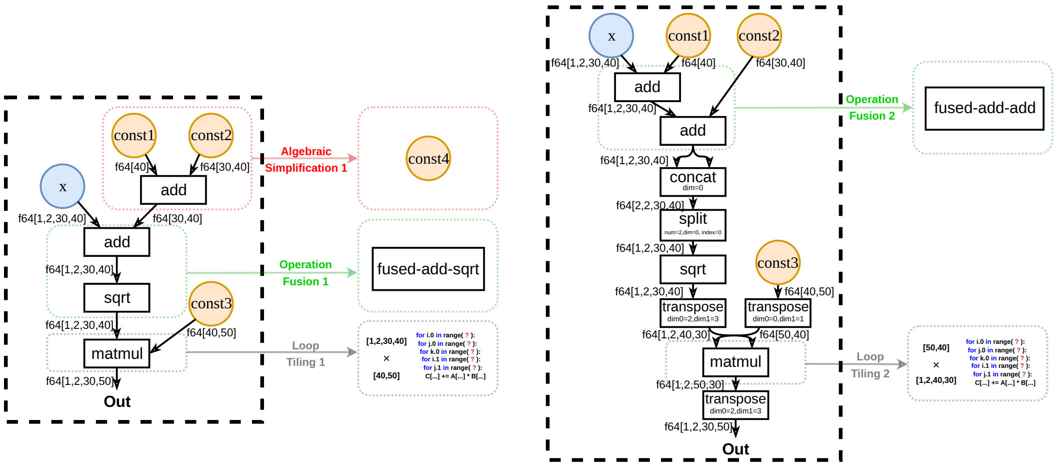
where $\mathcal{P}$ and $\mathcal{P}'$ are equivalent tensor programs, I is the input space of $\mathcal{P}$, i is a concrete input tensor, and $exec(p, i)$ is the execution result of program p for input i.

Challenge. The key challenge is to construct the equivalent tensor programs to maximize the potential for detecting mis-compilation bugs. The testing throughput for tensor compilers is significantly lower than for other applications. According to the previous study, the testing throughput for ONNXRuntime is roughly 3.5 testcases per second [27], while the throughput for traditional applications generally higher than 10,000 testcases per second [72]. Therefore, each testcase in our scenario needs to be carefully constructed to ensure the test efficiency. However, tensor programs with naive modifications tend to experience the same optimization transformations, which is unlikely to expose logic bugs in these transformations. Suppose we construct two equivalent tensor programs $\mathcal{P}$ and $\mathcal{P}'$ whose experiencing optimization transformations $t_n(...t_1(\mathcal{P}))$ and $t_m^{'}(...t_1^{'}(\mathcal{P}'))$ respectively. Ideally, we hope that their undergoing transformations are as different as possible, i.e.,

$$\min_{\mathcal{P}' \in EqSet(\mathcal{P})} |\{t_1, t_2, \ldots, t_n\} \cap \{t_1^{'}, t_2^{'}, \ldots, t_m^{'}\}| \tag{I}$$

where $EqSet(\mathcal{P})$ is a set of all programs equivalent to $\mathcal{P}$. However, the exact compilation paths cannot be predetermined before the actual execution. Therefore, during testcase construction, an effective strategy is needed to identify equivalent tensor programs that may undergo different transformations when they are compiled.

Motivating Example. Our observations and analyzes indicate that *twisting arithmetic procedures and tensor shapes within tensor programs can often lead to the activation of diverse optimization opportunities.* Fig. 3 provides an example of how two equivalent computation graphs can trigger different optimization opportunities during compilation.



```
for i.0 in range( ? ):
 for j.0 in range( ? ):
  for k.0 in range( ? ):
   for i.1 in range( ? ):
    for j.1 in range( ? ):
     C[...] += A[...] * B[...]
```

```
for i.0 in range( ? ):
 for j.0 in range( ? ):
  for k.0 in range( ? ):
   for i.1 in range( ? ):
    for j.1 in range( ? ):
     C[...] += A[...] * B[...]
```

Fig. 3. An illustrating example of two equivalent computation graphs undergo different optimization processes during compilation. Blue circles represent input tensors; orange circles denote constant tensors, black arrow denotes the dataflow between operators. Optimization opportunities for employing algebraic simplification, operation fusion, and loop tiling are highlighted in red, green, and gray respectively.

The two graphs present in the figure are equivalent because the left one is transformed to the right one through a series of three semantically-equivalent graph rewriting. First, the original graph computes $x + (const1 + const2)$. This is arithmetically transformed in the right graph to $(x + const1) + const2$ via an arithmetic rewriting rule that preserves semantic equivalence due to the commutative law of addition. The second rewriting involves the manipulation of the output tensor's shape from the last addition operation. This is achieved using the *concat* operator, with the original shape later restored via the *split* operator, i.e., $w = split(concat(w, w), 2)[0]$. The third rewriting pertains to the inputs of the *matmul* operator. The shapes of these inputs are transposed before being passed to the *matmul* operator. This rewriting rule upholds semantic equivalence, as $matmul(z, y) = transpose(matmul(transpose(y), transpose(z)))$.

Although equivalent, the two graphs are likely to undergo different optimization processes. For simplicity, we only consider three straightforward optimizations: algebraic simplification, operator fusion, and loop tiling. First, the original graph allows for constant folding due to the addition of two constants, $const1 + const2$. This algebraic simplification may be inhibited in the transformed graph, as it requires the computation of $x + const1$ before adding to $const2$. In addition, the *add* and *sqrt* operators in the original graph are likely to be fused together because both of them are injective operators [5], which are supposed to be fused. However, the *concat* and *split* operators in the transformed graph prevent this operation fusion. Furthermore, the *matmul* operator in the original graph accepts two tensor inputs whose shapes are $[1, 2, 30, 40]$ and $[40, 50]$, while the *matmul* operator in the transformed graph accepts $[50, 40]$ and $[1, 2, 40, 30]$ as input shapes. This potentially enables different loop tiling strategies during the code generation phase of tensor compilation. Conversely, the transformed graph may enable additional optimization strategies, such as fusing the two *add* operators into one.

Insight: Our key insight is to construct tensor programs that, while equivalent, exhibit a broad diversity in arithmetic procedures and intermediate tensor shapes. This diversity in the structure of computation graphs potentially enables diverse optimization opportunities, thereby enhancing the effectiveness of each testcase. Therefore, we shift our focus from research goal I to the following problem:

$$\max_{\mathcal{P}' \in EqSet(\mathcal{P})} |OpNum(\mathcal{P}') - OpNum(\mathcal{P})| \tag{II}$$

where $\mathcal{P}$ is the given tensor program and $OpNum(p)$ is the number of arithmetic and structural operations in the computation graph of tensor program p. We will give a quantitative analysis of this insight in Section 6.3.

4 Approach

This section provides a detailed description of our proposed fuzzer PolyJuice, which aims to implement our insight (research goal II) to efficiently detect mis-compilation bugs in tensor compilers.

Our approach unfolds in two main procedures. First, we generate a set of equivalent graphs for a given computation graph by using specific arithmetic and structural rewrite rules, which alter the arithmetic procedures and tensor shapes of intermediate tensors while preserving the final results. Second, from this set of equivalent graphs, we extract two extreme cases: the one that calculate its output using the fewest number of operations (namely *the most simplified graph*), and the one that does so using the maximum number of operations (namely *the most complex graph*). Despite equivalent, these two graphs have significantly different dataflow and are likely to undergo different optimization processes during compilation. In this way, we can explore the full range of the compiler's optimization logic using a minimal and highly efficient set of testcases.

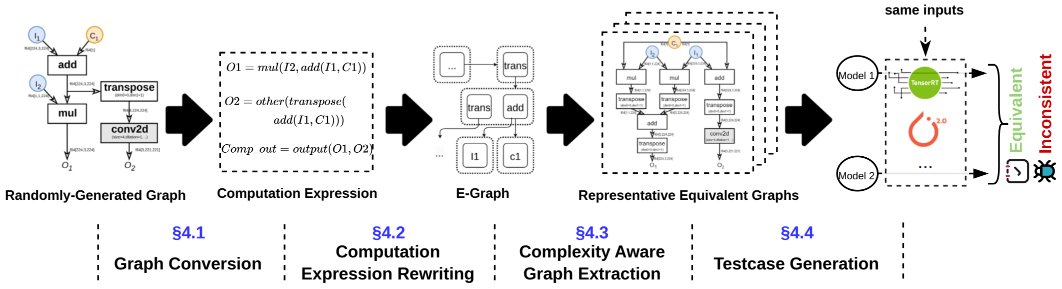


Fig. 4. Overall workflow of PolyJuice.

Fig. 4 illustrates the overall workflow of PolyJuice. First, it converts the computation flow of a given tensor computation graph into a custom internal representation, referred to as Computation Expression. This unique representation allows for the standardization of diverse computation flows into a unified format. Consequently, irrespective of the origin or structural complexity of the computation graph, our framework can seamlessly process and accept it as input. Second, it leverages an equality saturation engine to perform equivalent rewriting on this expression to generate an equality graph (e-graph). This e-graph encapsulates all expressions equivalent to the original one. Third, within the expansive realm of the e-graph, it identifies the most simplified and the most complex equivalent expressions, and converted them back to computation graphs. Finally, it leverages the computation graphs to generate testcases. We will delve deeper into each of these steps in the rest of this section.

4.1 Graph Conversion

In this step, PolyJuice converts the computation flow within a given tensor computation graph to a computation expression. Fig. 5 presents the syntax of the computation expression we defined.

$$
\begin{aligned}
\langle graph\_out \rangle &::= \textbf{\textit{graph\_out}}\ (\langle out\_expr \rangle^{+}) \\
\langle out\_expr \rangle &::= \textbf{\textit{outnode}}(\langle expr \rangle) \\
\langle expr \rangle &::= \langle def\_expr \rangle \mid \langle other\_expr \rangle \\
\langle def\_expr \rangle &::= \textbf{\textit{input}}\ (\langle id \rangle) \\
&\quad \mid\ \textbf{\textit{const}}\ (\langle id \rangle) \\
&\quad \mid\ \textbf{\textit{add}}\ (\langle expr \rangle, \langle expr \rangle) \quad \text{/* tensor1, tensor2 */} \\
&\quad \mid\ \textbf{\textit{mul}}\ (\langle expr \rangle, \langle expr \rangle) \quad \text{/* tensor1, tensor2 */} \\
&\quad \mid\ \textbf{\textit{matmul}}\ (\langle expr \rangle, \langle expr \rangle) \quad \text{/* tensor1, tensor2 */} \\
&\quad \mid\ \textbf{\textit{transpose}}\ (\langle expr \rangle, \langle int \rangle, \langle int \rangle) \quad \text{/* tensor, trans dim0, trans dim1 */} \\
&\quad \mid\ \textbf{\textit{split2}}\ (\langle expr \rangle, \langle int \rangle, \langle int \rangle) \quad \text{/* tensor, split dim, split output index */} \\
&\quad \mid\ \textbf{\textit{concat2}}\ (\langle expr \rangle, \langle expr \rangle, \langle int \rangle) \quad \text{/* tensor1, tensor2, concat dim */} \\
\langle other\_expr \rangle &::= \textbf{\textit{other}}(\langle operator\_id \rangle\ , \langle expr \rangle^{+}, \langle int \rangle) \quad \text{/* id, tensor1, ..., output index */}
\end{aligned}
$$

Fig. 5. Syntax of PolyJuice's computation expressions. $\langle graph\_out \rangle$ represents the computation flow of a tensor computation graph. $\langle def\_expr \rangle$ denotes an expression that operated by pre-defined operators. $\langle other\_expr \rangle$ denotes an expression that operated by operators other than $\langle def\_expr \rangle$. $\langle var \rangle^{+}$ means that the variable $\langle var \rangle$ can repeat one or more times. Comments highlighted in blue indicate the meanings of arguments accepted by the operators.

In the computation expressions we defined, a $\langle graph\_out \rangle$ can represent the computation flow of a given computation graph. A $\langle out\_expr \rangle$ can represent the computation flow of an output node of a computation graph. A $\langle expr \rangle$ denotes a sequence of computation operations acting on inputs, which we refer to as *computation flow*. For example, $matmul(add(x, y), transpose(z, 0, 1))$ represents a computation flow where inputs x and y are added together, and the resultant value is then matrix-multiplied with a transposed input z.

It is worth noting that our defined computation expression does not model all operators. Given that tensor compilers often provide thousands of operators for user convenience, and the supported operator sets vary across different compilers, it becomes impractical to model all operators. Consequently, our computation expressions only model the operators used in the computation expression rewriting step, i.e., $\langle def\_expr \rangle$ in the syntax, while others are kept as the $\langle other\_expr \rangle$. As per the definition of $\langle other\_expr \rangle$, it's clear that it records the inputs (i.e., $\langle expr \rangle^{+}$) of the operator and the index of the output of this expression. Therefore, it can express arbitrary operators even they are multiple inputs and multiple outputs.

We take the left-side computation graph in Fig. 4 as an example to illustrate the conversion process. For any given tensor computation graph, we disregard tensor-specific information such as shapes and data types, focusing solely on the computation flow of each output. When dealing with operators defined in the syntax of computation expressions, like `add` and `mul`, we directly convert them into corresponding $\langle def\_expr \rangle$. For `conv2d`, an operator we do not model, we convert it into a $\langle other\_expr \rangle$ with an assigned operator ID 0x99. Its output index is 0 since O_2 is the first (and only) output of `conv2d`. Thus, we can effortlessly convert the graph in Fig. 4 to the following expression:

$$
\begin{aligned}
O_1 &= outnode(mul(I_2, add(I_1, c_1))) \\
O_2 &= outnode(other(0x99, transpose(add(I_1, c_1), 0, 1), 0)) \\
out &= graph\_out(O_1, O_2)
\end{aligned}
$$

The computation expression only records the computation flow information. Tensor-specific information such as shape and data type, as well as constant values and weights of operators, are not included. Therefore, we also maintain a graph's metadata alongside the computation expression during the conversion process. The metadata keeps track of tensor-specific information to ensure that a computation expression can be accurately restored to a tensor computation graph.

Efforts for Extension. Including new defined operators in the computation expression is effortless. Basically, it requires less than 5 lines of code in our prototype. For simplicity, in Fig. 5, we only show the operators used in the computation expression rewriting step, which does not mean that our approach is limited to these operators.

4.2 Computation Expression Rewriting

In this step, PolyJuice performs equivalent rewriting on a given computation expression. We first introduce the equivalent rewrite rules used in PolyJuice, and then introduce how PolyJuice performs rewriting on computation expressions.

Rewrite Rules. To produce equivalent computation expressions, PolyJuice utilizes a collection of arithmetic and structural rewrite rules. The rewrite rules we define are listed in Table 1. The arithmetic rewrite rules are inspired by TASO [14] and primarily based on arithmetic properties such as commutativity and associativity. The structural rewrite rules are based on the invariant properties of operators `transpose`, `split2`, and `concat2`.

To ensure the equivalence of structural rewrite rules, we introduce variables $r_1, r_2, ..., r_6$ whose values are randomly generated under specific constraints. For the sake of simplicity in our discussion,

Table 1. Arithmetic and structural rewrite rules used in PolyJuice. x,y, and z are logical variables which can match any $\langle expr \rangle$ in our computation expression; c is a logical variable which can match any constant $\langle expr \rangle$; $r_1, r_2, \ldots, r_6$ are randomly generated values, each needing to meet specific constraints when a rewrite rule is applied. "dim" is a function that returns the dimension of a tensor.

Type	Rewrite Rule	Constraints
Arithmetic	$add(x,y) \rightarrow add(y,x)$	-
	$mul(x,y) \rightarrow mul(y,x)$	-
	$add(add(x,y),z) \rightarrow add(x,add(y,z))$	-
	$mul(mul(x,y),z) \rightarrow mul(x,mul(y,z))$	-
	$matmul(matmul(x,y),z) \rightarrow matmul(x,matmul(y,z))$	-
	$mul(add(x,y),z) \rightarrow add(mul(x,z),mul(y,z))$	-
	$matmul(add(x,y),z) \rightarrow add(matmul(x,z),matmul(y,z))$	-
Structural	$transpose(transpose(x,a,b),a,b) \rightarrow x$	-
	$add(x,y) \rightarrow transpose(add(transpose(x,r_1,r_2),transpose(y,r_3,r_4)),r_5,r_6)$	$r_1 = dim(x) - maxdim(x,y) + r_5$; $r_2 = dim(x) - maxdim(x,y) + r_6$; $r_3 = dim(y) - maxdim(x,y) + r_5$; $r_4 = dim(y) - maxdim(x,y) + r_6$.
	$mul(x,y) \rightarrow transpose(mul(transpose(x,r_1,r_2),transpose(y,r_3,r_4)),r_5,r_6)$	$r_1 = dim(x) - maxdim(x,y) + r_5$; $r_2 = dim(x) - maxdim(x,y) + r_6$; $r_3 = dim(y) - maxdim(x,y) + r_5$; $r_4 = dim(y) - maxdim(x,y) + r_6$.
	$matmul(x,y) \rightarrow transpose(matmul(transpose(y,r_1,r_2),transpose(x,r_3,r_4)),r_5,r_6)$	$r_1 = dim(y) - maxdim(x,y) + r_5$; $r_2 = dim(y) - maxdim(x,y) + r_6$; $r_3 = dim(x) - maxdim(x,y) + r_5$; $r_4 = dim(x) - maxdim(x,y) + r_6$.
	$concat2(x,y,c) \rightarrow transpose(concat2(transpose(x,c,r_1),transpose(y,c,r_2),r_3),c,r_4)$	$r_1 = r_2 = r_3 = r_4$
	$x \rightarrow split2(concat2(x,x,r_1),r_2,r_3)$	$0 \le r_1 = r_2 < dim(x); 0 \le r_3 \le 1.$
	$x \rightarrow concat2(split2(x,r_1,0),split2(x,r_1,1),r_1)$	$0 \le r_1 < dim(x)$
	$graph\_out(x) \rightarrow graph\_out(r_1,x,r_2)$	$r_1, r_2 = intermediateTensors()$

we shorten the term "transpose" to "trans" hereafter. Consider the structural rewrite rule $add(x, y) \rightarrow trans(add(trans(x, r_1, r_2), trans(y, r_3, r_4)), r_5, r_6)$ as an example. Suppose that we have a concrete computation expression $add(I_1, I_2)$ where I_1, I_2 denotes two tensors with shapes $[6, 5, 4, 3, 2]$ and $[1, 3, 2]$, respectively. Due to the broadcasting principle in tensor operations, the result of adding these two tensors retains the shape of $[6, 5, 4, 3, 2]$. When we apply the rewrite rule, we actually randomly transpose I_1 and I_2, and then transpose back after the addition of I_1 and I_2. However, the dimensions we transpose for these tensors are different. Suppose the expression after rewriting is $trans(add(trans(I_1, r_1, r_2), trans(I_2, r_3, r_4)), r_5, r_6)$. If r_1 and r_2 are randomly generated as values 2 and 3, then $trans(I_1, r_1, r_2)$ will have a shape of $[6, 5, 3, 4, 2]$. In this case, r_3 and r_4 should be values 0 and 1, so that $trans(I_2, r_3, r_4)$ has a shape of $[3, 1, 2]$ and can be added to $trans(I_1, r_1, r_2)$. Similarly, r_5 and r_6 should be values 2 and 3 to ensure that the rewritten expression is equivalent to the original $add(I_1, I_2)$. Therefore, the expression after rewriting is $trans(add(trans(I_1, 2, 3), trans(I_2, 0, 1)), 2, 3)$. In conclusion, the constraints presented in Table 1 are necessary to ensure the equivalence of these structural rewrite rules.

Rewriting. Given an expression and the aforementioned rewrite rules, PolyJuice aims to find the expressions that calculate its outputs using the fewest number of operations, and the ones that do so using the maximum number of operations. However, naively applying the rewrite rules directly will not find the optimal solution. This is because applying a rewrite rule for the given expression may prevent a better solution. For example, let's suppose we are trying to find the most simplified expression for $add(add(I_1, trans(I_2)), trans(I_3))$. If we mistakenly apply the $add(x, y) \rightarrow trans(add(trans(x), trans(y)))$ to the expression, the expression will transform into $add(trans(add(trans(I_1), I_2)), trans(I_3))$, which cannot be further simplified. In reality, the simplest expression should be $add(I_1, trans(I_2, I_3))$. This issue of deciding when to apply which rewrite rule is known as the phase ordering problem [20, 47, 60] in the compiler community.

To find the optimal expressions, we leverage *equality saturation* technique [60], which is commonly used for program analysis and program optimization. By leveraging this technique, we can efficiently find both the most simplified and the most complex equivalent expressions. Equality saturation operates by maintaining a data structure known as e-graph, which represents a set of terms and the equivalences between them. In our scenario, the nodes of an e-graph represent a $\langle expr \rangle$ in our computation expression syntax. Each node is associated with an equivalence class, and the e-graph ensures that two nodes are in the same equivalence class if and only if their corresponding $\langle expr \rangle$ expressions are equivalent. Each edge in the e-graph connects a $\langle expr \rangle$ to its sub-expressions. An e-graph is said to be *saturated* when no more rewrite rules can be applied to it. Once an e-graph has been saturated, the equality saturation technique ensures that the e-graph encapsulates all computation expressions that are equivalent to the original one.

Fig. 6 demonstrates how equality saturation constructs an e-graph and find the most simplified expression for an expression "$add(add(I_1, trans(I_2)), I_3)$". The left-side graph is the initial e-graph representing the original expression, and the right-side graph is the saturated e-graph. For the sake of simplicity in our discussion, in this example, we only consider three rewrite rules:

- $add(x, y) \rightarrow trans(add(trans(x), trans(y)))$
- $add(add(x, y), z) \rightarrow add(x, add(y, z))$
- $trans(trans(x)) \rightarrow x$

Additionally, we prohibit recursive rewriting to prevent an explosion of nodes in the e-graph. Initially, the e-graph only contains the original computation expression. Equality saturation then continuously applies the above three rewrite rules to the e-graph, which introduces new nodes into the graph and annotates them with the appropriate equivalence classes to maintain congruence. The right-side graph in Fig. 6 is a saturated e-graph, which encapsulates all expressions equivalent

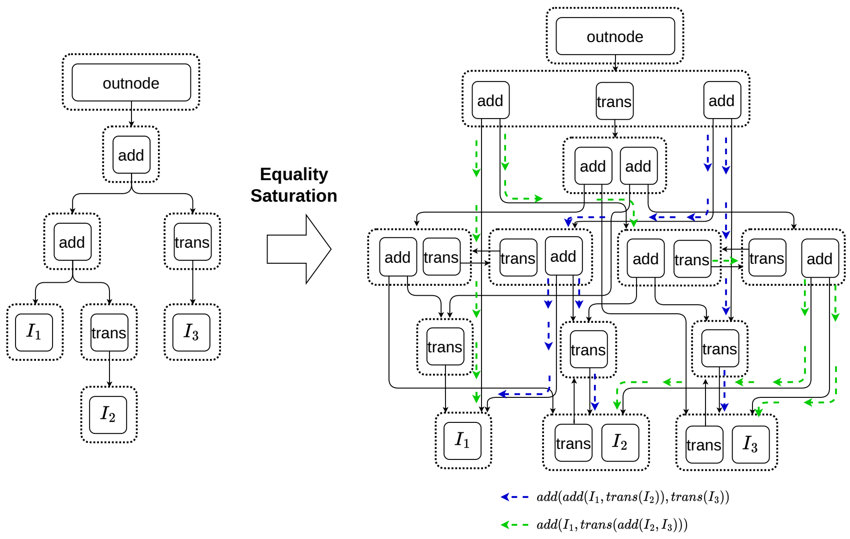

outnode
add
add
trans
I_1
trans
I_2
I_3
Equality Saturation
$add(add(I_1, trans(I_2)), trans(I_3))$
$add(I_1, trans(add(I_2, I_3)))$

Fig. 6. E-graph of $add(add(I_1, trans(I_2)), trans(I_3))$ after equality saturation. Solid boxes are nodes and present $\langle expr \rangle$. Dashed boxes represent equivalence classes, in which all the values of $\langle expr \rangle$ nodes are equal. All computation expressions equivalent to the original expressions are encapsulated in this e-graph. Blue and green arrows denote the original expression and the most simplified expression, respectively.

to the original graph. We now provide an example to explain how equivalent expressions are encapsulated in the e-graph. Let's look at the flow highlighted with green arrows: it begins from the top add node, splits into two, with the left arrow leading to the `I1` node and the right arrow leading to an equivalence class containing a `trans` node and an add node. From this class, the green arrow chooses to flow from the `trans` node to another equivalence class, which also contains a `trans` node and an add node. Then, it flows from the add node in this class, and finally flows to `I2` and `I3` nodes. This green arrow flow forms a tree structure, which represents the $add(I_1, trans(add(I_2, I_3)))$ expression. This expression is obviously equivalent to the original expression. Similarly, in this e-graph, trees rooted from nodes in the same equivalence class represent a set of equivalent computation expressions. These trees encapsulate all the possible forms of the original computation that are equivalent, providing a comprehensive view of all equivalent expressions.

4.3 Complexity Aware Graph Extraction

In this step, PolyJuice aims to extract representative computation expressions from the e-graph and then convert them into corresponding tensor computation graphs. Algorithm 1 depicts the entire process of extracting expressions and converting them into computation graphs.

First, PolyJuice performs a depth-first search for the e-graph E from $\langle graph\_out \rangle$ and remove all back edges (line 1). This ensures the output e-graph $E_{acyclic}$ becomes an acyclic graph, and thus sets an upper bound for the most complex expression. Second, PolyJuice identifies the most simplified and the most complex graphs in the e-graph (Line 2-3). This is achieved by applying a fixed-point traversal over e-classes (Line 7-13). Specifically, this traversal records the optimal cost of each equivalent class, and iteratively update the costs. Once reaching a fixed-point, the traversal finds an optimal expression on the e-graph based on the cost function. For the most simplified expression, the *costFuncForMost* is employed, where the cost metric (line 8) represents the least number of nodes constituting the sub-graphs of each e-class. Conversely, for the most complex expression,

Algorithm 1: Complexity Aware Graph Extraction

```
Input  : Original expression $O$
         Tensor metadata $M$
         Saturated e-graph $E$
Output : Two equivalent tensor computation graphs $G_1, G_2$
1  $E_{acyclic} \leftarrow removeBackEdge(E)$ // break cycles in e-graph
2  $mostSimplOut \leftarrow$ searchOptimal($E$, $cosFuncForFewest$)// search for the most simplified
3  $mostComplOut \leftarrow$ searchOptimal($E_{acyclic}$, $cosFuncForMost$) // search for the most complex
4  $G_1$ = restoreToGraph($mostSimplOut$, $M$)
5  $G_2$ = restoreToGraph($mostComplOut$, $M$)
6  return $G_1, G_2$
   // fixed-point iterations to find the cost-optimal solution
7  Function searchOptimal(E, costFunc):
8      $cost \leftarrow$ emptyMap() // store optimal value of each eclass
9      while $cost$ still updated do // until reach fixed-point
10         for $eclass$ in $E$ do // update cost of each eclass based on its children's cost
11             $cost[eclass]$ = costFunc($cost[eclass]$, $cost$, $eclass.children$)
12     end
13     return findOptimalExpr(E, cost) // find the optimal expression based on the cost
```

the *costFuncForFewest* is employed, where the cost metric represents the greatest number of nodes constituting the sub-graphs of each e-class. Based on the cost of each e-class, PolyJuice can adopt a depth-first search to extract the optimal expression in the e-graph. This step involves traversing from the root node and selecting the child node with the minimum/maximum cost from each e-class. Finally, having identified the extremes of complexity within the expressions, PolyJuice leverages the recorded metadata, including tensor shape, data type and constant/weight values, to restore these expressions back into tensor computation graphs (Lines 4-6).

Correctness. A tensor computation graph may contain multiple output nodes. This algorithm guarantees the correct extraction of both the most simplified and the most complex graphs across all output nodes. Specifically, this means that for the most simplified graph, the computational flow leading to each output node involves the fewest possible operations, while for the most complex graph, it involves the maximum number of operations. We take the extraction of the most simplified graph as an example to illustrate the correctness. After the fixed-point iteration, each e-class in the e-graph has an optimal cost value, which denotes the fewest number of nodes among all the sub-graphs rooted from this e-class. Next, Algorithm 1 line 13 finds optimal expressions based on these cost values, i.e., selecting the child node with the minimum cost from each e-class. Consequently, for a graph's expression $out = graph\_out(O_1, O_2, \ldots)$, the resulting sub-graph for each output node O_i is guaranteed to incorporate the minimal number of nodes among all sub-graphs originating rooted by the e-class of O_i. For the extraction of most complex graph, the process is similar. Therefore, the correctness of Algorithm 1 is guaranteed.

4.4 Testcase Generation

After obtaining equivalent tensor computation graphs, PolyJuice converts them into an executable testcase that can be sent to tensor compilers, during which PolyJuice ensures that the parameters (including constant values and weights of operators like Conv2d) of the equivalent models are consistent. Most tensor compilers accept ONNX, Torch, and TensorFlow models as inputs. Therefore, PolyJuice supports the conversion of computation graphs into these three model types. The following python code is an illustrative testcase example consisting of two models.

```
class Model1(...):
  ...
class Model2(...):
  ...
while(MAX_TIMES):
  inputs = rand()
  res1 = Model1().run(inputs)
  res2 = Model2().run(inputs)
  for out in outputs:
    assert_allclose(res1[out], res2[out])
```

The two models are lowered from the generated equivalent graphs. PolyJuice feeds a set of identical inputs to the two models and check if every output is consistent. If an inconsistency is detected, then a mis-compilation bug has been found in the tensor compiler being tested.

5 Implementation

Implementation Details. We implemented PolyJuice in roughly 3,000 lines of Python code and roughly 2,000 lines of Rust. We reuse NNSmith [27], a tensor compiler test tool that can generate diverse and valid tensor computation graphs, to randomly generate inputs for PolyJuice's rewriting module. Our rewriting module is designed to support arbitrary computation graphs as inputs, allowing for seamless collaboration with NNSmith. During computation expression rewriting, we use egg [60] as the equality saturation engine. Our search for the most simplified and most complex equivalent graphs is conducted on the top of egg's EGraph data structure.

After the rewriting process, PolyJuice generates a set of equivalent tensor computation graphs, which are then converted into executable testcases. PolyJuice can lower its tensor computation graphs to three types of model formats: Torch, ONNX, and TensorFlow. For Torch model format, we leverage Torch.Fx [41] to generate python code of the model lowered by our computation graphs. For ONNX model format, we first lower the graphs to Torch model, and then export to ONNX models. For TensorFlow model format, we implement lower code for every operator in order to generate Python code of TensorFlow models.

We mitigate floating-point numerical stability issues [37] in the same way as NNSmith [27]. Specifically, since the overall magnitude of most differences caused by the issue is small, we check output equivalence by comparing the relative difference between two outputs with high error tolerance. We also exclude certain structural patterns known to exacerbate numeric discrepancies, such as Sigmoid followed by Floor.

6 Evaluation

We applied PolyJuice to real-world tensor compilers to assess its effectiveness in identifying mis-compilation bugs. Our evaluation investigates the following questions:

- **RQ1**(§6.1): Can PolyJuice detect mis-compilation bugs in real-world tensor compilers?
- **RQ2**(§6.2): How well does PolyJuice perform compared to state-of-the-art related fuzzers?
- **RQ3**(§6.3): Can our equality saturation based rewriting approach generate tensor programs that undergo different compilation paths?
- **RQ4**(§6.4): Does the rewriting process introduce significant overhead during testing?

Experiment Setup. We conducted our evaluation on a machine equipped with an AMD EPYC 7763 CPU (2.25GHz) with 128 cores and an NVIDIA GPU (V100-32G), running Ubuntu 22.04 LTS. We applied PolyJuice to five widely-used industrial tensor compilers, namely Torch Inductor [38],

Table 2. Tensor compilers tested by PolyJuice.

Tensor Compiler	Vendor/Published at	Description
Torch Inductor	Meta	A built-in compiler in PyTorch.
TensorRT	Nvidia	A runtime highly optimized for NVIDIA GPUs.
ONNXRuntime	Microsoft	A graph-optimized DNN library for ONNX models.
XLA	Google	A built-in compiler in TensorFlow.
TVM	Apache/OSDI'18	An end-to-end compiler with auto scheduling.
Hidet	ASPLOS'23	A tensor compiler improves parallelizable computations.
EinNet	OSDI'23	A derivation-based tensor program optimizer.

TensorRT [33], TVM [5], ONNXRuntime [31], and XLA [48], and two representative and well-maintained academic tensor compilers: EinNet [70] and Hidet [7]. Their brief introductions are listed in Table 2. We tested their latest stable versions during our real-world bug detection. For our comparative study, we used two related tensor compiler fuzzers as benchmarks: NNSmith [27] and MT-DLComp [61]. NNSmith, one of the most successful tensor compiler fuzzers, is known for its ability to generate diverse and valid computation graphs. MT-DLComp, on the other hand, integrates two equivalent mutation strategies for metamorphic testing.

6.1 Mis-Compilation Bug Detection

We intermittently ran PolyJuice for a month to find mis-compilation bugs in the latest stable versions of the industrial and academic tensor compilers listed above. It's important to note that while PolyJuice identified a number of crash/exception bugs, we only reported the bugs that triggered inconsistent outputs between two equivalent tensor programs constructed by PolyJuice. This is because our primary contribution is to uncover a new detection approach of silent mis-compilation bugs. The crash/exception detection capability of PolyJuice aligns with NNSmith, as we built PolyJuice on the top of NNSmith. This capability is not attributed to our proposed test oracle. The evaluation in this section only focuses on the unique contribution of PolyJuice.

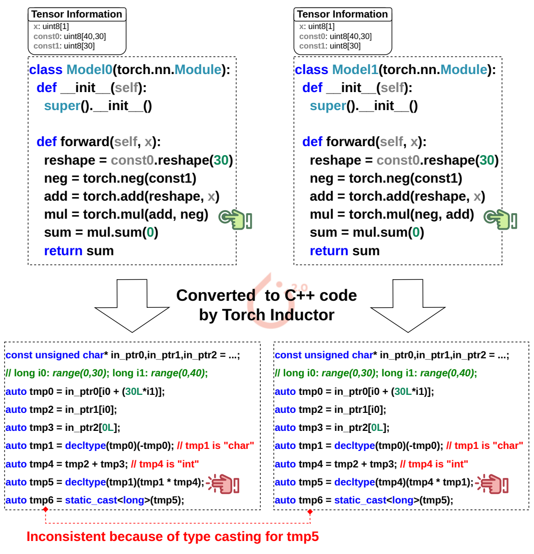
Table 3. Statistics of non-crash mis-compilation bugs reported by PolyJuice.

	Reported	Confirmed	Fixed
Torch Inductor	12	9	9
TensorRT	10	8	7
ONNXRuntime	11	0	0
TVM	7	0	0
XLA	25	13	0
Hidet	4	4	4
EinNet	15	15	0
Total	84	49	20

Bug Overview. Table 3 presents the statistics of the non-crash mis-compilation bugs reported by PolyJuice. We have filed a total of 84 bugs for the seven tensor compilers. Out of these, 49 have been confirmed, with 20 already fixed. Only PyTorch evaluated the severity of bugs, where 4 were marked with "high-priority bug" label out of the 10 bugs we reported. All reported bugs were identified by PolyJuice when it discovered pairs of equivalent tensor programs producing inconsistent outputs for identical inputs. These bugs, which do not cause crashes, are challenging to detect given that most existing tensor compiler testing tools primarily focus on crash bugs. We will provide more details on the comparative study in Section 6.2. Overall, this shows that PolyJuice can effectively detect mis-compilation bugs.

Bug Classification. The equivalent tensor programs are constructed based on arithmetic and structural rewrite rules. In total, out of all the 49 confirmed bugs, 24 are caused by arithmetic rewrite rules, and 25 are caused by structural rewrite rules. This indicates that both arithmetic and structural rules are effective for bug detection. In terms of reproducibility across different hardware, 1 of the confirmed bugs were reproducible on both GPU and CPU backends. 14 of the confirmed bugs were reproducible on the GPU backend, and 34 were reproducible on the CPU backend. The reason that we found more bugs on CPU backend is twofold. First, existing tensor compilers tend to be more robust on the GPU backend than on the CPU backend. Second, our testing environment had more CPU resources than GPU, leading us to allocate more testing tasks to the CPU backend of tensor compilers.

Feedback from Developers. The quantity of reported bugs in each compiler largely depends on the responsiveness of the developers. We usually waited for developers' feedbacks before continuing our bug-finding efforts. Therefore, a higher number of reported bugs in a compiler doesn't indicate lower code quality, but rather, it signifies the high responsiveness of the developers. In the case of ONNXRuntime and TVM, we did not receive any feedbacks after reporting bugs, so we only reported bugs found in the first 12 hours. The developers of other tensor compilers were responsive to our reports, typically confirming the reported bugs within a week. Two of our reported Inductor bugs are simplified and merged into PyTorch code base as unit tests by developers. All our reported Hidet bugs are merged into Hidet code base as unit tests by developers. We received positive feedback from developers, which not only validated our efforts but also indicated their commitment to fix these bugs. For example, Hidet developers commented "*Thanks for reporting these bugs (seems found by some awesome fuzzer) to us! ... I can reproduced all the bugs you have found. ... This issue is more serious than I thought.*" on our reported issues.


Tensor Information
Tensor Information
Converted to C++ code by Torch Inductor
Inconsistent because of type casting for tmp5

```
x: uint8[1]
const0: uint8[40,30]
const1: uint8[30]
class Model0(torch.nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, x):
    reshape = const0.reshape(30)
    neg = torch.neg(const1)
    add = torch.add(reshape, x)
    mul = torch.mul(add, neg)
    sum = mul.sum(0)
    return sum
```

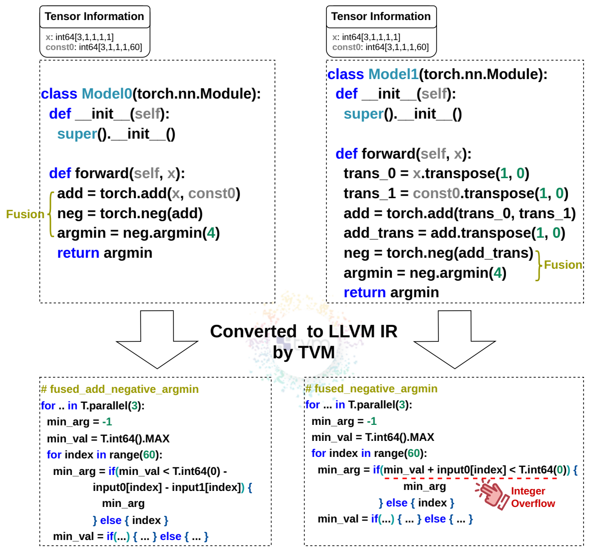
```
x: uint8[1]
const0: uint8[40,30]
const1: uint8[30]
class Model1(torch.nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, x):
    reshape = const0.reshape(30)
    neg = torch.neg(const1)
    add = torch.add(reshape, x)
    mul = torch.mul(neg, add)
    sum = mul.sum(0)
    return sum
```

```
const unsigned char* in_ptr0,in_ptr1,in_ptr2 = ...;
// long i0: range(0,30); long i1: range(0,40);
auto tmp0 = in_ptr0[i0 + (30L*i1)];
auto tmp2 = in_ptr1[i0];
auto tmp3 = in_ptr2[0L];
auto tmp1 = decltype(tmp0)(-tmp0); // tmp1 is "char"
auto tmp4 = tmp2 + tmp3; // tmp4 is "int"
auto tmp5 = decltype(tmp1)(tmp1 * tmp4);
auto tmp6 = static_cast<long>(tmp5);
```

```
const unsigned char* in_ptr0,in_ptr1,in_ptr2 = ...;
// long i0: range(0,30); long i1: range(0,40);
auto tmp0 = in_ptr0[i0 + (30L*i1)];
auto tmp2 = in_ptr1[i0];
auto tmp3 = in_ptr2[0L];
auto tmp1 = decltype(tmp0)(-tmp0); // tmp1 is "char"
auto tmp4 = tmp2 + tmp3; // tmp4 is "int"
auto tmp5 = decltype(tmp4)(tmp4 * tmp1);
auto tmp6 = static_cast<long>(tmp5);
```

Fig. 7. A mis-compilation bug in Torch Inductor. Python code on the top are the equivalent tensor programs constructed by PolyJuice. The difference between them is the input order of `torch.mul`. C++ code on the bottom is generated by Torch Inductor. The root cause is the implicit type casting for the `tmp5` variable.

Bug Study 1: Incorrect Type Casting. Fig. 7 shows a mis-compilation bug detected by PolyJuice in Torch Inductor (PyTorch version: 2.1.0). The two tensor programs are constructed by the arithmetic rewrite rule $mul(x, y) = mul(y, x)$. While simple, this rewrite rule can reveal deeply hidden mis-compilation bugs. After being compiled into executable code, the two tensor programs produce significantly different outputs. Both converted C++ programs in the figure are identical except for the type casting of the `tmp5` variable: one is casting to the type of `tmp1`, and the other is casting to the type of `tmp4`. This type casting is the root cause of this bug. Both `tmp2` and `tmp3` are of `unsigned char` type, however, according to C99 Standard section 6.3.1.1 [34], adding two char-type variables triggers an integer promotion, so `tmp4` is promoted to an `int` type. Subsequently, when calculating `tmp5`, Torch Inductor explicitly converts the type of `tmp5` to the type of its first argument. Therefore, when `tmp1` is the first argument, `tmp5` is of char type; whereas when `tmp4` is the first argument, `tmp5` is of int type. This difference leads to the inconsistent outputs of the two compiled tensor programs. Developers patch this issue by explicit typecasting for the add operator of char type, making `tmp4=decltype(tmp2)(tmp2+tmp3)`. Developers also added a unit test based on our report to prevent this issue from happening again. Although this bug is buried deep, PolyJuice detects it within one-hour testing, demonstrating its effectiveness.



```
class Model0(torch.nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, x):
    add = torch.add(x, const0)
    neg = torch.neg(add)
    argmin = neg.argmin(4)
    return argmin
```

```
class Model1(torch.nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, x):
    trans_0 = x.transpose(1, 0)
    trans_1 = const0.transpose(1, 0)
    add = torch.add(trans_0, trans_1)
    add_trans = add.transpose(1, 0)
    neg = torch.neg(add_trans)
    argmin = neg.argmin(4)
    return argmin
```

```
# fused_add_negative_argmin
for .. in T.parallel(3):
  min_arg = -1
  min_val = T.int64().MAX
  for index in range(60):
    min_arg = if(min_val < T.int64(0) -
              input0[index] - input1[index]) {
                min_arg
              } else { index }
    min_val = if(...) { ... } else { ... }
```

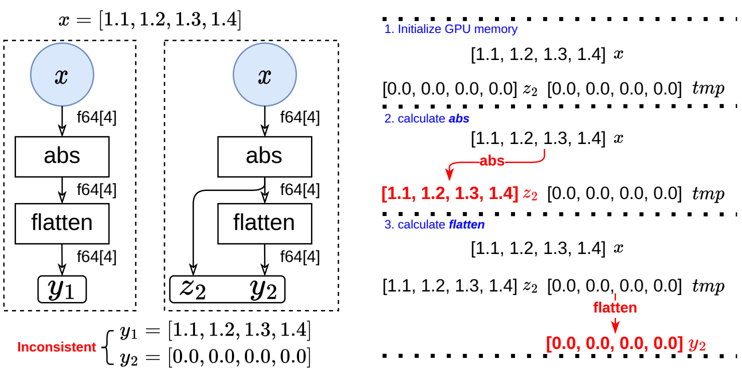
```
# fused_negative_argmin
for ... in T.parallel(3):
  min_arg = -1
  min_val = T.int64().MAX
  for index in range(60):
    min_arg = if(min_val + input0[index] < T.int64(0)) {
                min_arg
              } else { index }
    min_val = if(...) { ... } else { ... }
```

Fig. 8. A mis-compilation bug in TVM. Python code on the top are the equivalent tensor programs constructed by PolyJuice. In the negative-argmin fusion kernel, TVM creates a condition that causes integer overflow and always returns false when finding the index of minimal values. The correct condition in this fusion kernel should be `min_val < 0 - input0[index]`.

Bug Study 2: Incorrect Operator Fusion. Fig. 8 shows a mis-compilation bug detected by PolyJuice in TVM (version 0.12.0). The presented two tensor programs are constructed by the structural rewrite rule $add(x, y) \rightarrow trans(add(trans(x, r1, r2), trans(y, r3, r4)), r5, r6)$. The *argmin* operator is to find the index of the minimum value in the given tensor along a specific dimension. For the left-side tensor program, TVM fuses the `add`, `neg`, and `argmin` operations into a fusion kernel. To find the index of minimum values, the kernel first initializes variables `min_arg` and `min_val` to -1 and the maximum int64 value, respectively. It then traverses indexes and updates `min_arg` and `min_val` by finding the smaller `"-input0[index]-input1[index]"` each time. During the compilation of this tensor program, everything works as expected.

However, for the right-side tensor program, TVM only fuses the `neg` and `argmin` operators into a fusion kernel. When finding the index of minimum values, it is supposed to find the minimum `"-input[index]"`, i.e., if the `min_val < -input[index]` condition is false, it will update `min_val`. In TVM's implementation, it mistakenly converts the condition to `min_val + input[index] < 0`. Since `min_val` is initialized to the maximum int64 value, the `min_val + input[index]` causes integer overflow, and is less than 0 in most cases (depending on how LLVM handles the overflow). As a result, the `min_arg` cannot be properly updated. Although this bug is buried deep, PolyJuice detects it within half-hour testing, showing its effectiveness.

Bug Study 3: Incorrect Tensor Processing. Fig. 9 illustrates a mis-compilation bug detected by PolyJuice in Hidet (version 0.3,0). The outputs y_1 and y_2 of the two tensor programs are expected to be identical because the second tensor program only additionally outputs an intermediate tensor z_2, which should not influence the result of y_2. The right-side figure shows the internal process logic of Hidet when executing the second tensor program. During Hidet's optimization process, it allocates two memory areas (denoted as z_2 and *tmp*) for storing the output of the `abs` operation, but only updates the output to the first memory area, i.e., the one denoted by z_2. As a result, the second memory area (i.e., *tmp*) retains all zero values. However, it uses the second memory area to calculate y_2, leading to an incorrect result for y_2. The Hidet developers acknowledged the severity of this bug, commenting, "This issue is more serious than I thought." PolyJuice was able to detect this bug within ten minutes of testing.



$x = [1.1, 1.2, 1.3, 1.4]$
x
f64[4]
abs
f64[4]
flatten
f64[4]
y_1
x
f64[4]
abs
f64[4]
flatten
f64[4]
z_2
y_2
Inconsistent
$y_1 = [1.1, 1.2, 1.3, 1.4]$
$y_2 = [0.0, 0.0, 0.0, 0.0]$
1. Initialize GPU memory
[1.1, 1.2, 1.3, 1.4] x
[0.0, 0.0, 0.0, 0.0] z_2 [0.0, 0.0, 0.0, 0.0] tmp
2. calculate abs
[1.1, 1.2, 1.3, 1.4] x
abs
[1.1, 1.2, 1.3, 1.4] z_2 [0.0, 0.0, 0.0, 0.0] tmp
3. calculate flatten
[1.1, 1.2, 1.3, 1.4] x
[1.1, 1.2, 1.3, 1.4] z_2 [0.0, 0.0, 0.0, 0.0] tmp
flatten
[0.0, 0.0, 0.0, 0.0] y_2

Fig. 9. A mis-compilation bug in Hidet. During execution of the second tensor program, after calculating the abs, it allocates two memory area z_2 and *tmp*, but only stores the result to z_2. However, it uses the value in another memory area for y_2 calculation.

6.2 Comparative Study

We evaluate the effectiveness of our approach by comparing it with two representative comparators: MT-DLComp [61], which uses two mutation strategies to add dead code to computation graphs for mis-compilation bug detection; and NNSmith [27], which generates diverse and valid computation graphs and uses exception monitor and differential testing as its bug detectors. It is worth noting that PolyJuice is built on the top of NNSmith, and thus inherits NNSmith's exception monitor and differential testing capabilities. However, in this comparative study, we disable PolyJuice's differential testing capability to investigate the unique contribution of our proposed test oracle. We chose TVM (version 0.12) and XLA (version 2.15.0) as the test targets because these two compilers are the only ones supported by all three tools. We ran each test tool on each compiler for 48 hours, repeated three times. We triaged crash/exception bugs based on their runtime failure stacktrace, and triage mis-compilation bugs based on the graph structure of the testcases. We manually filtered out false positives from differential testing, which is mainly introduced by numerical accuracy variations across different platforms.

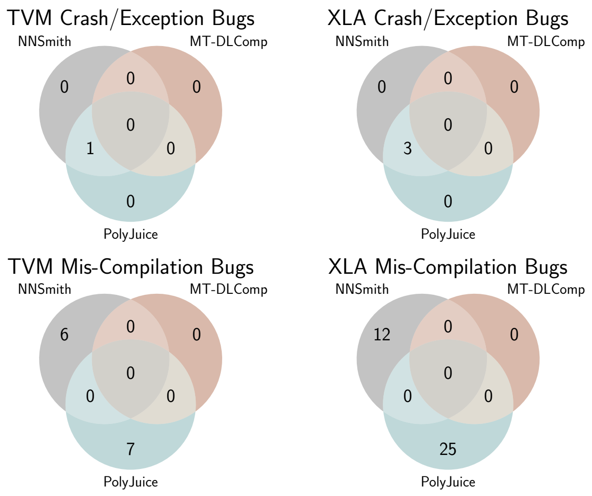


Fig. 10. The overlap of crash bugs and mis-compilation bugs reported by the three test tools in TVM and XLA after 48-hour fuzzing campaigns.

As shown in Fig. 10, our proposed test oracle can detect mis-compilation bugs that are not detected by NNSmith and MT-DLComp. In terms of crash/exception bugs, PolyJuice performs in line with NNSmith. This is because PolyJuice reuses NNSmith's code for the original computation graph generation, enabling PolyJuice to generate all graphs that NNSmith can generate. In terms of mis-compilation bugs, the area within NNSmith denotes the bugs detected by differential testing. In total, 6 bugs and 12 bugs were detected by differential testing in TVM and XLA, respectively. The area within the PolyJuice represents bugs found by our proposed test oracle. PolyJuice detected additional 7 and 25 bugs in TVM and XLA, respectively, indicating that the proposed test oracle can complement differential testing. MT-DLComp did not report any bugs in TVM and XLA, which is inline with the result in its paper's evaluation [61].

6.3 Effectiveness of Equality Saturation Based Rewriting

In our equality saturation based rewriting, we extract two representative equivalent graphs, i.e., the most simplified and the most complex graphs among all equivalent computation graphs. This rewriting strategy is based on an insight that the two extreme cases can experience different transformations during compilation. In this subsection, we validate if this insight works.

We use NNSmith (with its default settings) to generate 5,000 computation graphs. For each graph, we run the equality saturation engine and generate an e-graph, which encapsulates all graphs that equivalent to the original graph. Upon the e-graph, we compare two different extraction strategies: (1) extracting the most simplified and the most complex graphs, denoted as PolyJuice; (2) randomly extracting two graphs, denoted as PolyJuice^{naive}. Next, we run the graphs extracted by PolyJuice and PolyJuice^{naive} through TVM to observe the transformations they undergo.

We use Longgest Common Subsequence (LCS) difference and edit distance to measure the difference between the transformations of two extracted graphs. Let us consider that after two graphs are executed, the transformation sequences are $T_1 = [t_1, t_2, \ldots, t_n]$ and $T_2 = [t_1', t_2', \ldots, t_m']$.

The LCS difference, aligned with the POSIX diff utility [13], computes the LCS of T_1 and T_2, then identifies the number of elements not part of the LCS. The edit distance measures the minimum number of operations required to convert T_1 to T_2, implemented by Levenshtein Algorithm [3]. We calculate the average improvement as $(\sum_{i=1}^{N}(\frac{diff_1^i - diff_2^i}{diff_2^i}))/N$, where $diff_1^i$ is the executed transformation difference (LCS Diff or edit distance) between two graphs extracted by PolyJuice for the i-th testcase, and $diff_2^i$ is the difference between two graphs extracted by PolyJuice^{naive} for the same testcase.

Table 4. PolyJuice's improvement over PolyJuice^{naive} on the differences of transformations that extracted equivalent tensor programs experience during compilation.

LCS Difference		Edit Distance	
avg impr	stderr	avg impr	stderr
112.98%	0.09	150.06%	0.27

Table 4 presents the results of the comparison. Three out of 5,000 testcases include $diff_2^i = 0$ and thus are excluded from the calculation of average improvement to prevent division by zero. The average improvement of LCS difference and edit distance are 112.98% and 150.06%, respectively. Based on the standard error of the mean (denoted as stderr in the table), we can establish a 95% confidence interval for the average improvement: interval [95.34%, 130.62%] for the LCS difference, and interval [97.14%, 202.98%] for the edit distance. This indicates that the two extreme cases of equivalent graphs extracted by PolyJuice are likely to undergo considerably more varied transformations than two randomly extracted graphs do. Therefore, our insight is validated.

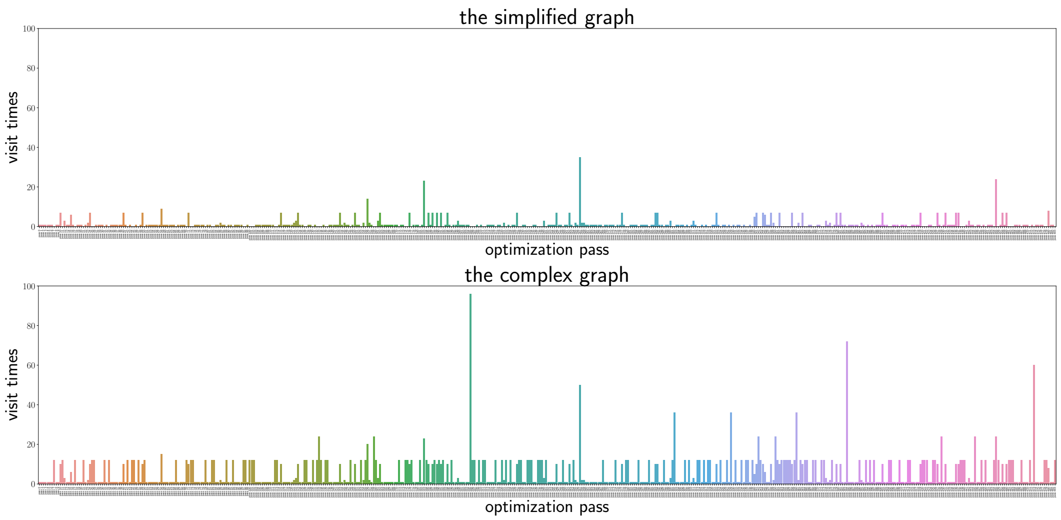


Fig. 11. An example of the visiting frequency distribution of optimization passes in TVM between two equivalent graphs extracted by PolyJuice.

In addition to transformation sequence differences, We also investigate the visiting frequency difference of transformations. Fig. 11 shows a random example of the visiting frequency distribution of optimization passes in TVM between two equivalent graphs extracted by PolyJuice. Each x-axis label represents an optimization pass, numbered starting from 0. We can see that the two graphs exhibit different visiting frequency distributions, indicating that the two graphs undergo different transformations during compilation. This further validates the effectiveness of our equality saturation based rewriting approach.

6.4 Overhead During Testing

Fuzzing throughput is a critical metric for evaluating the effectiveness of a fuzzer [25, 55–57, 72]. We conduct a 5000-iteration testing campaign on TVM and XLA to evaluate the overhead introduced by PolyJuice. Table 5 provides a detailed breakdown of the time consumption during testing. The expression rewriting and graph extraction columns represent the overhead introduced by PolyJuice. Two other steps of PolyJuice, namely graph conversion and testcase generation, consume less than 0.01 ms, and hence, their details have been omitted from the table. The original graph generation column denotes the time spent by NNSmith for generating the initial tensor computation graphs. From the table, we can observe that the majority of the time is spent in the execution of the compiler and the original graph generation. Notably, TVM expends more than 1,000 milliseconds on its compilation and code execution processes. By contrast, PolyJuice typically uses less than 2 milliseconds for expression rewriting and graph extraction tasks. As a result, PolyJuice introduces a maximum overhead of 0.11% and 1.53% when testing TVM and XLA, respectively. It is also noteworthy that the number of graph nodes does not significantly influence the time spent by PolyJuice, but it does largely impact the execution time of tensor compilers. Therefore, as the number of graph nodes increases, the overhead introduced by PolyJuice becomes negligible.

Table 5. Average time taken during PolyJuice's testing. Time spent by expression rewriting and graph extraction are the overhead caused by PolyJuice.

Compiler	# graph node	original graph generation	expression rewriting	graph extraction	compiler execution
TVM	5	26.76ms	0.33ms	1.16ms	1367.85ms
	10	41.72ms	0.33ms	1.63ms	1865.97ms
	15	53.04ms	0.35ms	1.64ms	2059.14ms
XLA	5	22.35ms	0.27ms	1.23ms	74.09ms
	10	35.92ms	0.26ms	1.23ms	97.24ms
	15	87.00ms	0.26ms	1.25ms	121.11ms

7 Discussion

Design Choices. PolyJuice is designed to be a non-intrusive test tool, capable of testing tensor compilers without requiring access to their source code. This design choice makes PolyJuice easily be applied to a wide range of tensor compilers regardless of their code availability. A potential improvement is to incorporate code instrumentation to obtain fine-grained source-code-level optimization information. This would enable our test tool to know the execution path of tensor compilers when executing a tensor program, potentially leading to a more effective graph extraction strategy. However, since some compilers are close-sourced, we choose our current design for broad applicability and ease of use. We leave the instrumentation-based approach as our future work.

False Positives. Despite our test oracle eliminating false positives caused by differential testing's cross-environment numerical accuracy issues, PolyJuice reported two types of false positives. The first type stems from triggering undefined behavior in tensor compilers. For instance, in divide-by-zero scenarios, most tensor compilers output NaN. However, after certain optimizations, TensorRT outputs a number other than NaN, which developers consider expected. We mitigate this by filtering out graphs that trigger undefined behaviors.

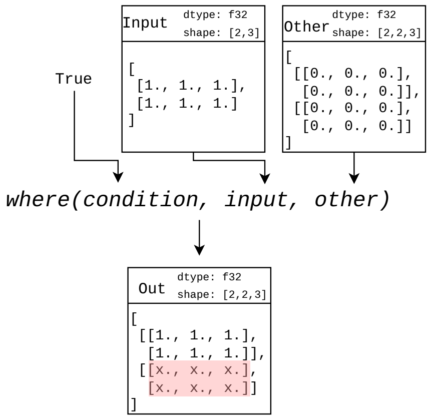


Fig. 12. A false positive reported by PolyJuice due to unstable outputs in TensorRT. In other tensor frameworks/compilers, the appended values (highlighted in red) are filled with the same values of the Input tensor. In TensorRT, these values are filled with random values from GPU memory.

The second type is related to unstable outputs when some compilers process specific operators. For example, TensorRT processes the `where(cond,x,y)` operator differently from other libraries or compilers. `where(cond,x,y)` returns a tensor of elements selected from either tensor `x` or tensor `y`, depending on the condition `cond`. As Fig. 12 shows, when handling this operator, if the shapes of `x` and `y` differ, TensorRT fills the difference with random values from GPU memory, resulting in random outputs. Developers confirmed that this behavior is expected in TensorRT. We mitigate this by filtering out these unstable operators during graph generation. These false positives constitute less than 8% of the bugs reported by PolyJuice, thus not significantly impacting its overall effectiveness. Besides, once we encounter a false positive and filter it out according to developers' feedback, this false positive will never present again. Therefore, we believe that false positives are not a major concern for PolyJuice.

Common Mis-Compilation Bugs. While bugs vary greatly across different tensor compilers, the majority of bugs we uncovered originate from operator fusion and low-level code generation. During the operator fusion phase, certain compilers may have implicit preconditions for input models to perform fusion. However, some edge cases may fail to meet these conditions, leading to bugs. During low-level code generation, the transition of data types between high-level code and low-level code is particularly susceptible to errors. Our practice corroborates the findings of previous studies [45] that type problems and incorrect code logic are the most common bugs in tensor compilers.

Test Tool Deployment. PolyJuice can be integrated into tensor compiler development in two main ways. Developers can run PolyJuice on their compilers for a specified time slot (e.g., 24 hours) to identify bugs. Alternatively, they can use PolyJuice to generate a test suite for continuous integration during development. We will release a test suite of 10,000 testcases, each in three formats: ONNX file, PyTorch code, and TensorFlow code. We believe that PolyJuice and this test suite can enhance the code quality of tensor compilers.

Future Work. PolyJuice enables several promising opportunities for future work. First, the development of more effective equivalent rewrite rules is a promising direction. PolyJuice provides a platform for researchers to effortlessly experiment with various rewrite rules. We believe that a new class of rewrite rules, beyond arithmetic and structural rewriting, could also contribute significantly to the robustness of tensor compilers. Second, future work could investigate new bug type based on PolyJuice. We believe the concept of equivalent graphs could be applicable to other types of bugs, such as performance bugs and high GPU memory usage bugs. Third, it would be also promising to extend PolyJuice to test bugs in all kinds of deep learning hardware accelerators other than CPU and GPU. Recently, many new hardware accelerators [42] have been proposed to assist deep learning model to deploy in various real-world scenarios. This would be a good backend target for mis-compilation bug findings.

8 Related Work

Reliability Study on ML Systems. Research efforts have been made to study the reliability of DL frameworks such as PyTorch, TensorFlow, as well as their compilers. Notably, studies on real-world bugs in DL frameworks provide insights into the reliability of these systems. For example, Shen et al. [45] analyze the root cause of 603 bugs arising in three popular tensor compilers and provide a series of valuable guidelines for improving the reliability of tensor compilers. Cao et al. [4] analyze 210 StackOverflow posts related to performance problems in ML systems, and point out that API misuse and buggy DL libraries often cause performance problems. Guan et al. [10] analyze 371 model optimization bugs in model training, compiling and deployment stages, and unveil the major root causes of these bugs. All these studies highlight the importance of reliability in ML systems and developers can benefit from them to implement more reliable ML systems. Our work complements these studies by implementing a test approach to detect mis-compilation bugs, facilitating the improvement of the tensor compilers' reliability.

Tensor Compiler Testing. Testing serves as an essential method for uncovering the weaknesses within software systems, including DL libraries and tensor compilers. A variety of techniques have been proposed in recent years. Most of these techniques such as Audee [11], Lemon [59], GraphFuzzer [29], Muffin [9], NNSmith [27], Tzer [28], and HirGen [30] aim to construct valid testcases through program analysis to expose bugs in target systems. As large language models (LLMs) become more popular, some research efforts, e.g., FuzzGPT [6], have begun to harness the power of LLMs to generate testcases that are not only effective but also highly nuanced, targeting deep learning libraries and tensor compilers. These techniques focus on testcase (i.e., computation graph) generation, which serve as the initial input for PolyJuice's workflow. Orthogonally, our work focuses on mis-compilation bug detection by constructing equivalent computation graphs.

On the other hand, a few test oracles are proposed for testing DL libraries and tensor compilers. EAGLE [54] uses different APIs to implement identical computation graphs and verify their execution results, targeting implementation bugs in DL library APIs. Our work adopts a totally different approach and focuses on different types of bugs. MT-DLComp [61] introduces two types of dead code in tensor programs to construct equivalent programs to detect mis-compilation bugs. The first type is to insert universal obscure conditions as dead code, and the second is to insert per-input obscure conditions. Different from it, our contribution is to provide arithmetic and structural rewriting strategies to rewrite a tensor program, and implement a framework to find representative equivalent programs to trigger diverse transformation logic in tensor compilers.

Traditional Compiler Testing. Compiler testing has been studied extensively over the past few decades. One key direction is to generate valid testcases to expose bugs in compilers. Csmith [62] tests C/C++ compilers by generating C/C++ programs that avoid undefined behaviors. CodeAlchemist [12] assembles and mutate existing code snippets to generate JavaScript programs for

JavaScript engines. On the other hand, several techniques focus on constructing test oracles to detect mis-compilation bugs. Notably, for C/C++ compiler testing, a practical technique named EMI [21] was proposed. This technique constructs equivalent C/C++ programs by inserting dead code into a given program. JIT-Picking [2] detects non-crash JIT bugs by executing the JavaScript code twice: once with the JIT compiler enabled and once without it, i.e., solely relying on the interpreter. For Java virtual machine testing, CSE [23] inserts semantics-preserving code into Java programs to efficiently explore compilation space in order to detect crashes or mis-compilation bugs in JVM. Our work differs from these techniques in that we focus on tensor compilers and our proposed test oracle is based on tensor-compiler domain-specific knowledge, i.e., arithmetic and structural invariant instead of dead code insertion.

New Oracle for Testing Other Systems. Since correctness bugs are often silent in complex systems, many research efforts are paid to combine domain knowledge to design new test oracle to detect correctness bugs in specific systems. For database engine testing, PQS [43] was introduced to generate queries that are designed to retrieve a specific data entity for verifying the integrity of the database engine; Mozi [8, 26, 58] constructs equivalent database engines through adjusting configurations to detect correctness bugs. For operating system testing, B3 [32] and Hydra [17] finds different test oracle such as formal specifications and crash consistencies to validate correctness of filesystems; BVF [46] leverages structured and sanitized Programs to verify the correctness of eBPF programs. For web browser testing, Janus [73–75] leverages rendering change statuses of two web pages to conduct differential testing to detect rendering logic bugs in web browsers. Our work serves as a new test oracle for tensor compilers, which can inspire the design of test oracles for other complex systems.

Term Rewriting Systems. Term rewriting is a time-tested approach for equation reasoning. Equality saturation [47, 60] is a technique to implement term rewriting efficiently using an e-graph. Among all the equality saturation engines, egg [60] is the most successful one, so we reuse it in our expression rewriting process. This engine is often used in the research field for program analysis [36, 65], program optimizations in compilers [63] and hardware designs [52]. In contrast to the these applications, we focus on a distinct use case: leveraging equality saturation to assess the reliability of tensor compilers.

9 Conclusion

We presented a new test approach which constructs semantically-equivalent tensor programs to detect mis-compilation bugs in tensor compilers. We realized it as PolyJuice, which generates equivalent tensor programs that have significantly different dataflow and are likely to experience different optimization processes during compilation. During our testing period with PolyJuice, it has detected 84 non-crash mis-compilation bugs on seven well-maintained tensor compilers. 49 bugs were confirmed by developers, out of which 20 were fixed. Our future work will focus on integrating more equivalent rewriting rules into PolyJuice.

Data-Availability Statement

The artifact that supports Section 6 is available on Zenodo [1]. We also release the prototype of PolyJuice at https://github.com/ChijinZ/PolyJuice-Fuzzer.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful feedback. This research is sponsored in part by the National Key Research and Development Project (No.2022YFB3104000) and NSFC Program (No.92167101, 62021002).

References

[1] 2024. PolyJuice: Detecting Mis-Compilation Bugs in Tensor Compilers with Equality Saturation Based Rewriting. Zenodo. https://doi.org/10.5281/zenodo.12671619

[2] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022.*

[3] Paul E Black. 1999. *Algorithms and theory of computation handbook, "Levenshtein distance".* CRC press. https://www.nist.gov/dads/HTML/Levenshtein.html

[4] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding performance problems in deep learning systems. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022.*

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018.*

[6] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *46th IEEE/ACM International Conference on Software Engineering, ICSE 2024.*

[7] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-Mapping Programming Paradigm for Deep Learning Tensor Programs. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023.*

[8] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin : Grammar-Free DBMS Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022.*

[9] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022.*

[10] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A Comprehensive Study of Real-World Bugs in Machine Learning Model Optimization. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023.*

[11] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated Testing for Deep Learning Frameworks. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020.*

[12] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019.*

[13] The IEEE and The Open Group. 2023. diff - compare two files. https://pubs.opengroup.org/onlinepubs/009604499/utilities/diff.html. (visited on September 1, 2023).

[14] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *27th ACM Symposium on Operating Systems Principles, SOSP 2019.*

[15] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019.*

[16] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. 2021. DeepCuts: a deep learning optimization framework for versatile GPU workloads. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021.*

[17] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *27th ACM Symposium on Operating Systems Principles, SOSP 2019.*

[18] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 77:1–77:29.

[19] Thomas Koehler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. 2024. Guided Equality Saturation. *Proc. ACM Program. Lang.* 8, POPL (2024), 1727–1758.

[20] Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. In *27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012.*

[21] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014.*

[22] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT Compilers via Compilation Space Exploration. In *29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023.*

[23] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT Compilers via Compilation Space Exploration. In *29th Symposium on Operating Systems Principles, SOSP 2023.*

[24] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2021. The Deep Learning Compiler: A Comprehensive Survey. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 708–727.

[25] Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 1–27.

[26] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation. In *46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*.

[27] 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 *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023*.

[28] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–26.

[29] Weisi Luo, Dong Chai, Xiaoyue Run, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-based Fuzz Testing for Deep Learning Inference Engines. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021*.

[30] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing Deep Learning Compilers with HirGen. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023*.

[31] microsoft. 2023. ONNX Runtime: cross-platform, high performance ML inferencing and training accelerator. https://github.com/microsoft/onnxruntime. (visited on September 1, 2023).

[32] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*.

[33] Nvidia. 2023. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. (visited on September 1, 2023).

[34] open std. 2023. ISO/IEC 9899:TC3. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf. (visited on September 1, 2023).

[35] Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. 2023. Equality Saturation Theory Exploration à la Carte. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1034–1062.

[36] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015*.

[37] Douglas M. Priest. 1992. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations.* Ph. D. Dissertation. USA. UMI Order No. GAX93-30692.

[38] PyTorch. 2023. Accelerated CPU Inference with PyTorch Inductor using torch.compile. https://pytorch.org/blog/accelerated-cpu-inference/. (visited on September 1, 2023).

[39] PyTorch. 2023. Glow - Compiler for Neural Network hardware accelerators. https://github.com/pytorch/glow. (visited on September 1, 2023).

[40] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013*.

[41] James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022*.

[42] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2020. Survey of Machine Learning Accelerators. In *2020 IEEE High Performance Extreme Computing Conference, HPEC 2020*.

[43] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*.

[44] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor Program Optimization with Probabilistic Programs. In *NeurIPS*.

[45] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*.

[46] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. 2024. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In *19th European Conference on Computer Systems, EuroSys 2024*.

[47] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*.

[48] Tensorflow. 2023. https://www.tensorflow.org/xla. https://www.tensorflow.org/xla. (visited on September 1, 2023).

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

[50] tvm. 2023. PR: Fix a bug of iter map floormod(x,2) simplify. https://github.com/apache/tvm/pull/14571. (visited on September 1, 2023).

[51] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022*.

[52] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*.

[53] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021*.

[54] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022*.

[55] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021*.

[56] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Jingzhou Fu, Zhuo Su, Qing Liao, Bin Gu, Bodong Wu, and Yu Jiang. 2024. Data Coverage for Guided Fuzzing. In *33rd USENIX Security Symposium, USENIX Security 2024*.

[57] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. 2022. Odin: on-demand instrumentation with on-the-fly recompilation. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022*.

[58] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021*.

[59] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*.

[60] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29.

[61] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1 (2022), 15:1–15:28.

[62] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*.

[63] Yichen Yang, Phitchaya Mangpo Phothilimthana, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021*.

[64] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. 2023. Cocktailer: Analyzing and Optimizing Dynamic Control Flow in Deep Learning. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023*.

[65] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI (2023), 468–492.

[66] Jie Zhao and Peng Di. 2020. Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020*.

[67] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022*.

[68] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. USENIX Association.

[69] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022.*

[70] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, and Zhihao Jia. 2023. EINNET: Optimizing Tensor Programs with Derivation-Based Transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023.*

[71] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, and Lidong Zhou. 2023. PIT: Optimization of Dynamic Sparse Deep Learning Models via Permutation Invariant Transformation. In *29th Symposium on Operating Systems Principles, SOSP 2023.*

[72] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020.*

[73] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. 2023. Towards Better Semantics Exploration for Browser Fuzzing. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 604–631.

[74] Chijin Zhou, Quan Zhang, Bingzhou Qian, and Yu Jiang. 2025. Janus: Detecting Rendering Bugs in Web Browsers via Visual Delta Consistency. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025.*

[75] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: browser API fuzzing with dynamic mod-ref analysis. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022.*

[76] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022.*

Received 2024-04-05; accepted 2024-08-18