Finding Bugs in MLIR Compiler Infrastructure via Lowering Space Exploration

Jingjing Liang*‡
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
jjliang@sei.ecnu.edu.cn

Shan Huang*
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
shan.huang@stu.ecnu.edu.cn

Ting Su†
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
Shanghai, China
tsu@sei.ecnu.edu.cn

***Abstract*—MLIR is a widely adopted compiler infrastructure that supports multi-level IRs and reusable components. Ensuring its correctness is critical, as bugs can propagate to downstream systems. MLIR provides a lowering mechanism that transforms high-level programs into low-level representations through configurable sequences of passes, and allows multiple valid lowering paths for a given program. This gives rise to a lowering equivalence property: all valid lowering paths for the same MLIR program should produce semantically equivalent results. In this paper, we leverage this property and propose lowering space exploration, to effectively test the MLIR infrastructure. Our approach dynamically constructs diverse lowering paths in an adaptive, stepwise manner using a feedback-based scheduling mechanism. It finds bugs by comparing the execution results across these paths. Any inconsistencies indicate potential bugs in the MLIR infrastructure. To the best of our knowledge, this is the first work to test MLIR from the perspective of exploring its compilation space. We implement our approach in a tool named LOBE and evaluate it on latest MLIR versions. LOBE discovers 38 previously unknown bugs, including 8 miscompilations and 30 crash bugs, with 25 confirmed/fixed.**

I. Introduction

MLIR (Multi-Level Intermediate Representation) is a modern compiler infrastructure that lowers the cost of building domain-specific compilers by providing reusable components and support for multi-level IRs [1]. Its flexibility has led to rapid adoption across academia and industry [2]–[4], with over 30 downstream projects [5] spanning hardware design [6], parallel runtimes [7], and deep learning compilers [8], [9]. Given its widespread use, ensuring the correctness of MLIR is critical. The bugs within the MLIR infrastructure could have significant consequences, potentially leading to failures or incorrect behavior in the downstream compilers and systems built upon it [10], [11].

Program Lowering in MLIR. MLIR exhibits a hierarchical structure by organizing IR into *dialects*, and it can *lower* programs by converting high-level dialects into lower-level ones. Each dialect defines a set of operations, attributes, and types for a specific domain. For example, the *arith* dialect provides high-level numerical operations that reflect source-level semantics [12], whereas the *llvm* dialect defines low-level constructs resembling LLVM IR [13]. MLIR supports *conversion* mechanisms to transform IR from one dialect into a semantically equivalent form in another. For instance, the *"-convert-arith-to-llvm"* pass rewrites arithmetic operations into *llvm*. Prior to each conversion, dialect-specific or general optimization passes may be applied. We refer to the process of transforming programs composed of multiple dialects into ones using only a lowest-level dialect[1] as *lowering*. The sequence of passes applied during this process is referred to as a *lowering path*. As shown in Figure 1, an MLIR program consisting of three dialects, *i.e.*, *arith*, *complex*, and *index*, can be lowered to *llvm* through a lowering path composed of three conversion passes, *i.e.*, {*-convert-arith-to-llvm*, *-convert-complex-to-llvm*, *-convert-index-to-llvm*}.

Lowering Space of MLIR Programs. Unlike traditional compilers that define fixed compilation pipelines, such as Clang with `-O1`, MLIR provides a configurable lowering process. An MLIR program could be lowered in many different ways by varying the order of optimization and conversion passes. This implies a *lowering equivalence property*: lowering an MLIR program through different paths should produce semantically equivalent programs. This property should always hold. Because a correct compiler needs to ensure that the semantics of an input program should always be preserved throughout the compilation process. When an MLIR program is lowered to the program containing only the lowest-level dialects via different valid lowering paths, the semantics of the output programs should always be equivalent (*e.g.*, yielding the same outputs). We define the set of all possible lowering paths for a given MLIR program as the *lowering space*.

Figure 1 illustrates the concept of lowering space using an MLIR program with three dialects: *arith*, *complex*, and *index*. Each node represents the current dialects present in the program, and each edge corresponds to the application

*Jingjing Liang and Shan Huang are co-first authors of this work.

‡Jingjing Liang is also affiliated with State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China.

†Ting Su is the corresponding author.

[1]For example, *llvm* is the commonly used lowest-level dialect. In this paper, all dialects of MLIR programs are converted to *llvm*.

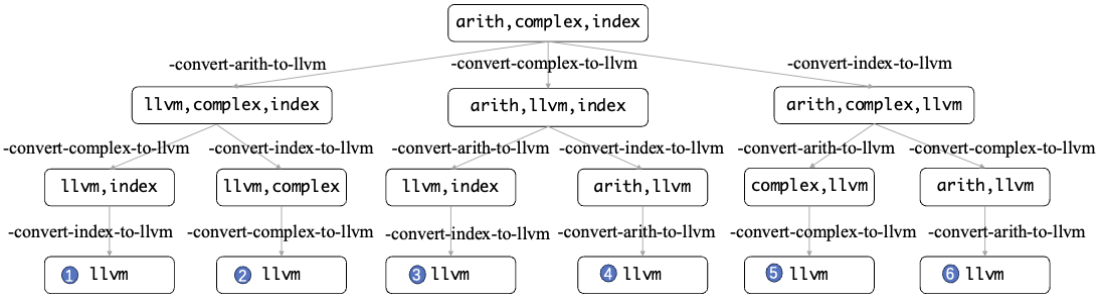


Fig. 1: An illustration of the lowering space (built with only dialect conversions) for an MLIR program consisting of three dialects. Optimizations are omitted for simplification.

of a dialect conversion pass. The lowering space of the MLIR program contains six lowering paths and all lowered program (numbered 1-6) from different lowering path should be semantically equivalent. Note that, in real-world lowering process, a arbitrary number of optimizations could be applied before applying the dialect conversion. We omit the optimizations in this example for simplicity.

Lowering Space Exploration. In this paper, we introduce *lowering space exploration*, a novel approach to automatically finding bugs in MLIR compiler infrastructure. The key idea is to construct every possible lowering paths for a given MLIR program and cross-check the execution results of their corresponding lowered programs. According to the *lowering equivalence property*, any inconsistencies, such as execution output divergences or compilation crashes, across different lowering paths within the same lowering space indicate potential bugs in the MLIR infrastructure.

To realize lowering space exploration, we face one technical challenge: *how to efficiently explore the lowering space.* In practice, the lowering space grows rapidly with more complex programs that contain diverse dialects and operations that require multiple steps to be lowered to the lowest-level program. The interaction between conversion and optimization passes leads to a factorial explosion in possibilities. Assuming an MLIR program requires k conversion passes and up to n optimization passes can be inserted between them, the number of possible lowering paths is roughly $k! \cdot ((n+1)!)^{k-1}$, where $k!$ accounts for all permutations of the conversion passes, and $((n+1)!)^{k-1}$ captures the number of ways to insert up to n optimization passes in the $k-1$ available gaps. Moreover, not all combinations lead to valid lowering paths. Exhaustively enumerating all lowering paths is unnecessary and infeasible.

To address the challenge of the vast lowering space, we investigate how to construct valid lowering paths and propose a *stepwise lowering path construction* approach. We begin by collecting a set of *atomic lowering rules*, which statically define the mapping between operations and their corresponding applicable passes. Guided by these rules, we then construct the lowering path step by step. At each step, we scan the program to identify the current set of dialects and operations, then select and apply a pass that is applicable to one or more of these operations based on the rules. This step is repeated until all dialects are lowered to the lowest-level dialect (*i.e.*, *llvm*). The sequence of passes applied throughout this iterative process constitutes a lowering path. The dynamic stepwise construction ensures that each generated path is valid, as only passes applicable to the operations present in the program are selected at each step.

During the construction of a valid lowering path, the order in which dialects are lowered within an MLIR program is crucial. Each conversion pass is designed to operate on specific dialects and operations, and when multiple dialects are present, applying a pass prematurely or out of order may lead to failures. However, there is no official documentation that explicitly specifies the exact lowering order in MLIR. Therefore, we propose *feedback-based scheduling*, a dynamic strategy that adaptively selects the next operation to lower based on its priority. These priorities are continuously updated according to the success or failure of previous lowering attempts. Leveraging lowering feedback effectively steers the exploration away from repeatedly selecting operations that are not currently lowerable.

By adopting this adaptive step-by-step strategy, we are able to construct a valid lowering path. The inherent randomness in selecting which pass to apply at each step fosters diversity in the explored paths. Consequently, by iteratively applying this guided construction process, our approach achieves broader and more effective coverage of the lowering space, which in turn enhances the likelihood of finding bugs.

We implemented our approach as a tool named LOBE (**LO**wering **B**ug **E**xplorer). LOBE is a test amplification technique. In principle, it can be combined with any UB-free generator to enhance the effectiveness of generation-based MLIR testing techniques by exploring a broader range of optimization and conversion passes. On the latest MLIR versions, LOBE uncovered 38 previously unknown bugs (30 crashes, 8 miscompilations), 25 of which have already been confirmed or fixed. In comparison with the predefined fixed lowering path, LOBE detected 39 bugs when combined with two program generators, TOSASmith and Ratte [14], while testing using Ratte with a predefined fixed lowering path did not find any bugs. LOBE also achieved full dialect and operation coverage relative to Ratte with a predefined fixed lowering path and significantly outperformed it in code coverage, with increases of 81.05% in line coverage and 90.33% in branch coverage. These results highlight the effectiveness of lowering space

exploration in finding bugs and improving test coverage. Our ablation study further highlights the importance of our core components. Without atomic lowering rules, the success rate of constructing valid lowering paths plummets from 97.17% to just 0.06%. Additionally, feedback-based scheduling improves the lowering success rate by 18.17% and increases the number of successful lowering paths by 37.32%. LOBE has been made publicly available at *https://github.com/ecnusse/LOBE*.

This paper has made the following contributions:

- At the conceptual level, we propose *lowering space exploration*, a novel approach that leverages the *lowering equivalence property* to find bugs in the MLIR infrastructure. To the best of our knowledge, this is the first work to test MLIR from the perspective of exploring its compilation space.
- At the technical level, we introduce an adaptive stepwise construction of lowering paths based on a *feedback-based scheduling* mechanism, enabling efficient exploration of the lowering space and facilitating bug detection.
- At the empirical level, we implement our approach in a tool named LOBE and evaluate it on latest versions of MLIR. LOBE uncovered 38 previously unknown bugs, including 8 miscompilations and 30 crash bugs, 25 of which have been confirmed/fixed.

II. Background and Illustrative Example

This section gives some background on MLIR infrastructure (Section II-A) and illustrates our approach through a real-world bug as an example (Section II-B).

A. MLIR Compiler Infrastructure

MLIR facilitates the development of domain-specific compilers by providing a flexible and extensible framework. Specifically, MLIR utilizes *Dialects* and *Operations* to manage multi-level IRs, employs *Passes* to realize optimizations and conversions between different level IRs, and offers *Command-line Tools* to support the transformation and execution.

Dialects. A dialect in MLIR is a collection of operations, types, and attributes. Each dialect represents a specific subset of the IR for a particular domain. MLIR allows users to define custom dialects, enabling the creation of domain-specific extensions within the framework. Additionally, MLIR includes several built-in dialects, supporting a wide range of specialized operations for various domains. For example, the *tosa* dialect provides operations of tensor computations commonly found in machine learning models,

Operations. An operation in MLIR represents the fundamental computation unit within a given dialect. An operation takes a list of SSA operands as inputs and produces one or more outputs. Operations are the building blocks of MLIR programs and can range from simple arithmetic operations to complex domain-specific computations. For example, `tosa.conv2d` is an operation defined within the *tosa* dialect, which represents a 2D convolution, a fundamental computation in convolutional neural networks.

Passes. A pass in MLIR refers to a transformation applied to the IR. It takes an MLIR program as input, either converts or optimizes it, and outputs the updated semantically equivalent program. Passes are categorized into two types: (1) *Conversion Passes* transform operations from one dialect to another, usually lowering from high-level to low-level dialects. For example, the *"-tosa-to-tensor"* pass converts operations in the *tosa* dialect into their corresponding operations in the *tensor* dialect; and (2) *Optimization Passes* optimize MLIR programs as traditional compiler optimizations. In MLIR, optimization passes can be further categorized into *general* and *dialect-specific* optimization passes. For example, *"-cse"* is a general optimization pass that eliminates common sub-expressions and can be applied to all MLIR programs regardless of the used dialects. In contrast, *"-tosa-infer-shapes"* is a dialect-specific optimization pass used exclusively for programs consisting of *tosa* dialect, where it infers the output shapes of *tosa* operations based on input tensor shapes. Additionally, MLIR allows the flexible composition of these passes into compilation pipelines, enabling iterative conversion and optimization of the IR.

B. An Illustrative Example

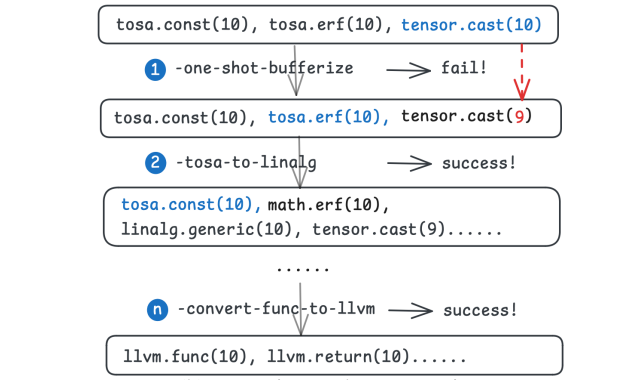
Figure 2 presents an example of the detected bug in MLIR. Figure 2a shows an MLIR program consisting of three operations in *tosa* dialect. It first declares a tensor constant (line 3), then applies the `tosa.erf` operation to compute the gaussian error function [15] of the constant (line 4), and finally prints the computed result (lines 5-6). The expected output is a 2×2 tensor filled with zeros. However, when lowering this program using a sequence of passes where *"-convert-linalg-to-loops"* is followed by the optimization pass *"-scf-for-loop-peeling="peel-front""*, the value at position (0,1) becomes a random value, indicating a bug.

Our Approach. We first construct a valid lowering path in an adaptive, step-by-step way. For the given program, we identify the dialects and operations it contains. As shown in the first box of Figure 2b, the program includes three operations, each initialized with a default priority score (*i.e.*, 10). In the first step, we randomly select `tensor.cast` and apply its corresponding pass, *"-one-shot-bufferize"*, based on the collected atomic lowering rules. This attempt fails because operations in *tosa* dialect do not yet support bufferization, so the priority of `tensor.cast` is decreased. In the second step, `tosa.erf` is selected, and its associated pass *"-tosa-to-linalg"* is successfully applied, converting it into `math.erf` and `linalg.generic`, as shown in the third box. Since the conversion succeeds, its priority remains unchanged. This adaptive process continues, dynamically updating operation priorities based on the success or failure of each transformation, until all dialects in the program are lowered to the target *llvm* dialect.

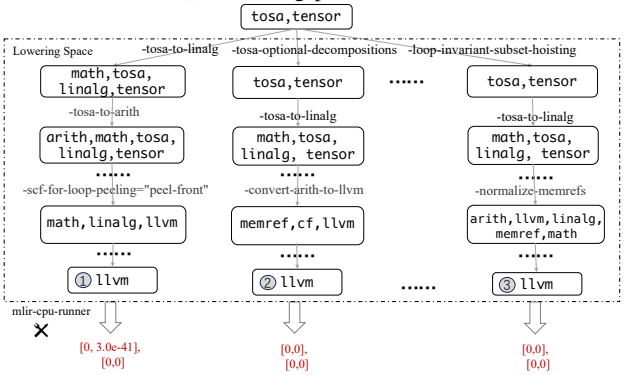
We repeat this process to generate multiple distinct lowering paths and obtain their execution results using `mlir-cpu-runner` [16]. As shown in Figure2c, the result

```
1 func.func private @printMemrefF32(tensor<*xf32>)
2 func.func @main() {
3  %0 = "tosa.const"() <{value = dense<0.0> : tensor<1x2x2xf32>}>
   : () -> tensor<1x2x2xf32>
4  %1 = tosa.erf %0 : (tensor<1x2x2xf32>) -> tensor<1x2x2xf32>
5  %cast = tensor.cast %1 : tensor<1x2x2xf32> to tensor<*xf32>
6  call @printMemrefF32(%cast) : (tensor<*xf32>) -> ()
7  return
8 }
```

(a) An MLIR program



(b) Lowering path construction



(c) Lowering space for the MLIR program

Fig. 2: An illustrative example with Bug #118790.

produced by the first path differs from all others, suggesting the presence of a faulty pass along that lowering path.

Challenge. This example illustrates both the vastness of the lowering search space and the effectiveness of atomic lowering rules and feedback-based scheduling in narrowing it. In terms of search space, lowering the input program requires k=15 conversion passes. Assuming we consider only general optimization passes, with a total of n=13 such passes available for insertion, the total number of possible lowering paths exceeds 10^{165} combinations. This highlights the impracticality of exhaustive enumeration and underscores the necessity of guided exploration strategies. *Feedback-based scheduling* improves efficiency by dynamically updating operation priorities based on previous failures. For example, `tensor.cast` should be lowered only after all *tosa* operations are handled. Lowering its priority after a failed attempt reduces the chance of premature selection in later iterations. Overall, the adaptive stepwise lowering path construction effectively reduces infeasible paths while facilitating efficient exploration of the lowering space.

III. Approach

This section first provides a formulation of lowering space exploration (Section III-A). It then introduces how this exploration is practically applied. Specifically, we describe how to construct a valid lowering path (Section III-B), followed by the overall algorithm that leverages these paths for testing (Section III-C).

A. Lowering Space Exploration

In this paper, an MLIR program is represented by the set of dialects it comprises. Formally, we define an MLIR program as $P_{D=\{d_1,\ldots,d_n\}}$, where each operation $d_i \in \mathcal{D}$, and $\mathcal{D}$ represents the set of all dialects implemented in MLIR infrastructure.

Definition III.1 (Lowering Step)**.** *Given an MLIR program P_D, a lowering step for this program is defined as $P_D \xrightarrow{O,c} P_{D'}$, where $O \subseteq \mathcal{O}$, and $c \in \mathcal{C}$. Here, $\mathcal{O}$ and $\mathcal{C}$ denote the sets of all optimization and conversion passes in MLIR infrastructure, respectively.*

The goal of a lowering step for an MLIR program is to transform one of the dialects within the program into a more low-level semantically equivalent dialect. During this process, optimization passes may also be applied to improve performance or simplify the IR. As a result, a lowering step typically consists of applying a sequence of optimization passes followed by a conversion pass, progressively transforming the program into a lower-level IR.

Definition III.2 (Lowering Path)**.** *A lowerint path γ for an MLIR program P_D is a sequence of lowering steps where the program is incrementally transformed into progressively lower-level dialects until it reaches the lowest level representation such as llvm IR. Formally, a lowering path can be represented as:*

$$\gamma : P_D \xrightarrow{O_1,c_1} P_{D'} \xrightarrow{O_2,c_2} P_{D''} \xrightarrow{O_3,c_3} \cdots \xrightarrow{O_n,c_n} P_{\{llvm\}} \tag{1}$$

where each $\xrightarrow{O_i,c_i}$ denotes a lowering step that transforms the program from one level of abstraction to a lower level.

Definition III.3 (Lowering Space)**.** *Given a program P_D, the lowering space of P_D is defined as all possible lowering paths that can perform on P_D. Formally, it is defined as:*

$$\mathcal{S}(P_D) = \{\gamma \mid Runner(P_D, \gamma) \neq \perp\} \tag{2}$$

where $Runner(P_D, \gamma)$ denotes the process of applying the lowering path γ to P_D, and then executing the resulting program. It returns the program's output if successful, or $\perp$ if the execution fails or produces no result.

Executing an MLIR program P_D with any lowering path γ from the set $\mathcal{S}(P_D)$ should yield the same output. Thus, if we find that

$$Runner(P_D, \gamma_1) \neq Runner(P_D, \gamma_2) \tag{3}$$

for $\gamma_1 \neq \gamma_2 \in \mathcal{S}(P_D)$, this indicates a potential miscompilation within the MLIR infrastructure. To ensure the correctness

and reliability of the lowering processes, it is ideal to exhaustively explore all possible lowering paths in $\mathcal{S}(P_D)$ to verify the equivalence of their outputs. This thorough examination is referred to as *Lowering Space Exploration*.

B. Lowering Path Construction

Exhaustively enumerating all lowering paths is infeasible. To address this, we construct valid lowering paths step by step. At each step, we analyze the program's current dialects and operations, and use *Atomic Lowering Rules* to identify applicable passes. To guide the process, we apply *Feedback-based Scheduling*, which dynamically updates operation priorities based on past successes or failures. Together, these components ensure correctness and improve efficiency by avoiding unproductive paths.

Atomic Lowering Rules. We define atomic lowering rules as a mapping from operations to their applicable conversion and optimization passes, since transformations in MLIR are driven by the operations within each dialect. This mapping allows us to determine which passes are applicable to each operation in a given MLIR program. The collection of these rules is a one-time effort that enables subsequent lowering path construction.

The *optimization rule*, denoted as $\mathcal{R}_{\mathcal{O}}$, is defined as a mapping from each operation $op \in \mathcal{OP}$ to the set of applicable optimization passes:

$$\mathcal{R}_{\mathcal{O}}[op] = \{o_1, o_2, \ldots, o_n\} \subseteq \mathcal{O}$$

where $\mathcal{OP}$ denotes the set of all MLIR operations, and $\mathcal{O}$ represents the set of all available optimization passes. For an operation op belonging to the dialect d, $\mathcal{R}_{\mathcal{O}}[op]$ includes both general optimization passes and specific to the dialect d.

The *conversion rule*, denoted as $\mathcal{R}_{\mathcal{C}}$, is defined as a mapping from each operation $op \in \mathcal{OP}$ to the set of applicable conversion passes:

$$\mathcal{R}_{\mathcal{C}}[op] = \{c_1, c_2, \ldots, c_m\} \subseteq \mathcal{C}$$

where $\mathcal{OP}$ denotes the set of all MLIR operations, and $\mathcal{C}$ represents the set of all available conversion passes. In most cases, each operation op has a unique corresponding conversion pass, making $\mathcal{R}_{\mathcal{C}}[op]$ a singleton set.

Feedback-based Scheduling. The lowering order of operations across different abstraction levels is subject to inherent constraints. Violating these constraints may lead to unsuccessful lowering. To improve the success rate of lowering, and guide the construction of the lowering path more efficiently, we design a feedback-based scheduling mechanism that dynamically updates priorities for operations within the program based on the outcomes of previous lowering attempts. This mechanism helps prioritize operations that are more likely to be successfully lowered, while deprioritizing those that have repeatedly failed, thereby reducing redundant attempts and accelerating progress.

Initially, all operations are assigned equal priority scores, reflecting the absence of prior knowledge about which operations should be lowered first. During the iterative lowering process, the operation with the highest priority is selected. If multiple operations share the highest score, one is randomly chosen among them. When a conversion attempt fails for the selected operation, a penalty is applied to reduce its priority score, decreasing the likelihood of it being selected again in subsequent iterations. Both the initial priority and the penalty factor are configurable, allowing users to fine-tune the scheduling strategy to balance between exploration diversity and conversion stability.

Lowering Path Construction. Based on these two core components, we introduce how to adaptively construct a lowering path in a stepwise way. The procedure proceeds iteratively through alternating optimization and conversion phases. In each phase, it scans the current program, selects and applies the appropriate passes, and subsequently updates the program, the lowering path, and (in the case of conversion) the priority map. This process continues until all operations have been successfully lowered into the target dialect. By dynamically adapting to the dialects and operations present in each intermediate program during the lowering process, this approach efficiently guides the construction of valid lowering paths.

Algorithm 1 outlines the construction procedure, which takes as input an MLIR program P_D, conversion rules $\mathcal{R}_C$, optimization rules $\mathcal{R}_O$, and a priority map $\mathcal{P}$ assigning a lowering priority to each operation, and outputs a lowering path γ, which represents a sequence of optimizations and conversions applied to the input MLIR program, along with the updated operation priority map $\mathcal{P}$, which is retained to guide the construction of subsequent lowering paths. It works as follows:

- *Initialization* (lines 1–3): The algorithm begins by initializing key variables. $curStep$ tracks the number of lowering steps performed on the input program. The applied optimization or conversion passes are recorded in the lowering path γ. The dialect set D along with the operation set OP, which contains operations pending lowering, are extracted using the SCAN function.
- *Optimization Phase* (lines 5–9): In each iteration, the algorithm first enters the optimization phase, which aims to explore effective optimizations for the current program. Specifically, it randomly selects a subset of applicable optimization passes for the current program and applies them. Afterward, both the sets of pending operations and dialects, as well as the lowering path γ are updated accordingly.
- *Conversion Phase* (lines 10–18): Following optimization, the algorithm proceeds to the conversion phase, which focuses on lowering the program using a conversion pass. Specifically, it selects the operation with the highest priority for conversion (lines 10-12). If multiple operations share the same highest priority, one of them is randomly selected. If the conversion fails, the priority score of the operation is reduced to discourage its selection in future steps (lines 13-14). Notice that if the priority scores of all operations eventually drop to zero, the scheduling strategy effectively degenerates into a purely random selection process, Upon success, the sets of unlowered

Algorithm 1: Lowering Path Construction

Input: An MLIR program P_D, conversion rules $\mathcal{R}_C$, optimization rules $\mathcal{R}_O$, operation priority map $\mathcal{P}$
Output: A lowering path γ, updated operation priority map $\mathcal{P}$

1 $curStep \leftarrow 0$
2 $\gamma \leftarrow [\,]$
3 $D, OP \leftarrow \textsc{Scan}(P_D)$ *// dialects and operations manifested within the program*
4 **while** $(\exists\ d \in D$ *s.t.* $d \neq$ *llvm*$)\ \wedge\ (curStep < maxStep)$ **do**
  // — Optimization Phase —
5 $O_{AllOpts} \leftarrow \bigcup_{op \in \mathcal{OP}} \mathcal{R}_{\mathcal{O}}[op]$
6 $O_{SelectedOpts} \leftarrow \textsc{RandomSelect}(O_{AllOpts})$
7 $P_{OP'} \leftarrow \textsc{ApplyOptimization}(O_{SelectedOpts}, P_D)$
8 $\gamma \leftarrow \gamma \cup O_{SelectedOpts}$
9 $D, OP \leftarrow \textsc{Scan}(P_{OP'})$
  // — Conversion Phase —
10 $op \leftarrow \textsc{SelectHighestPriority}(OP)$
11 $c \leftarrow \mathcal{R}_C[op]$
12 $P_{OP''} \leftarrow \textsc{ApplyConversion}(c, P_{OP'})$
13 **if** *conversion fails* **then**
14 $\mathcal{P}[d] \leftarrow \max(0,\ \mathcal{P}[d] - penalty)$
15 **else**
16 $D, OP \leftarrow \textsc{Scan}(P_{OP''})$
17 $\gamma \leftarrow \gamma \cup \{c\}$
18 $curStep \leftarrow curStep + 1$ *// Increment step counter*
19 **return** γ, $\mathcal{P}$

operations and dialects, as well as the lowering path γ are updated (lines 16-17). Finally, the step counter $curStep$ is incremented (line 18). The process continues iteratively until all dialects are successfully lowered to *llvm* or a preset maximal step is reached (line 4).

Notice that if a crash occurs during the application of an optimization or conversion pass, the lowering process is immediately terminated, and the crash is reported along with the associated pass and program.

C. Overall Algorithm

Repeatedly invoking single path construction introduced in Algorithm 1 can yield a diverse set of distinct lowering paths, as this process involves randomized decisions (*e.g.*, in optimization pass selection and operation scheduling). This repetition enables a effective exploration of the lowering space,

Algorithm 2 describes our overall algorithm. Given an MLIR program P_D, conversion rules $\mathcal{R}_C$, and optimization rules $\mathcal{R}_O$, the algorithm aims to finding potential miscompilations via exploring the lowering space, *i.e.*, multiple diverse lowering paths. Specically, it begins by initializing result set $\mathcal{R}$ and operation priority map $\mathcal{P}$. Notably, $\mathcal{P}$ is initialized with uniform priority scores for all operations in P_D during the first lowering path construction. In all subsequent constructions, the updated priority map from the previous iteration is reused. This adaptive reuse enables the algorithm to learn from past failures and successes. The core of the algorithm is to repeatedly invoke the Algorithm 2 to construct various lowering paths, execute each resulting program, and compare their outputs until the given time budget is exhausted (lines 3–6). If discrepancies are observed, it indicates a potential miscompilation, which is then reported as a bug (lines 7–8).

Algorithm 2: Overall Algorithm

Input: An MLIR program P_D, conversion rules $\mathcal{R}_C$, optimization rules $\mathcal{R}_O$
Output: potential miscompilation report

1 $\mathcal{R} \leftarrow \{\ \}$ *// Initialize result map*
2 $\mathcal{P} \leftarrow \{op \mapsto p \mid op \in OP\ \}$ *// Initialize operation priority map with uniform priority p*
3 **while** *time budget not exhausted* **do**
4 $\gamma, \mathcal{P} \leftarrow \textsc{LoweringPathConstruction}(P_D, \mathcal{R}_C, \mathcal{R}_O, \mathcal{P})$
5 $r \leftarrow \textsc{Runner}(P_D, \gamma)$
6 $\mathcal{R} \leftarrow \mathcal{R} \cup \{(r, \gamma)\}$
7 **if** $\exists\ (r_i, \gamma_i),\ (r_j, \gamma_j) \in \mathcal{R}$ *s.t.* $r_i \neq r_j$ **then**
8 Report a bug with program P_D, along with the distinct results and their corresponding lowering path $\{(r_i, \gamma_i), (r_j, \gamma_j)\}$

IV. Implementation

We implement our approach as a tool named LOBE in Python, comprising approximately 4900 lines of code. LOBE leverages `mlir-opt` [17] and `mlir-cpu-runner` [16] to apply MLIR passes (*i.e.*, ApplyOptimization and ApplyConversion functions in Algorithm 1) and execute the lowered MLIR programs (*i.e.*, Runner function in Algorithm 2), respectively. We give some important implementation details of LOBE below.

MLIR Program Generation. LOBE is not tied to a particular program generator. Rather, it can be combined with any generator that produces executable, UB-free MLIR programs. In our experiments, we develop TOSASmith, a custom generator that generates programs using only operations from the *tosa* dialect. Unlike MLIRSmith [10], which primarily aims at triggering crashes without guaranteeing UB-freedom, TOSASmith adopts reconditioning techniques [18], *e.g.*, limiting the bit-width of the operands of a multiplication to avoid overflow, thereby ensuring UB-freedom. Specifically, TOSASmith generates programs via a randomized composition strategy. It first initializes a variable pool with some seed variables. Then, in each iteration, it randomly selects an operation and chooses type-compatible operands from the variable pool. The result of the newly generated operation is added back into the pool. This process is repeated until the desired number of operations is reached. In our experiment, this ranges from 20 to 60 operations per program. Restricting programs to the *tosa* dialect ensures deterministic semantics and guarantees that all operations are mutually compatible. Moreover, *tosa* is a high-level dialect that offers a sufficiently large space for LOBE to explore. TOSASmith currently supports 55 of the 64 *tosa* operations, excluding those (*e.g.*, matrix multiplication) for which UB cannot be fully mitigated.

Atomic Lowering Rules Collection. We collect both conversion and optimization rules for each operation by analyzing the official MLIR documentation [19] and source code. The documentation specifies passes applicable to each dialect, which we refine to the operation level. *For optimization rules,* all operations within the same dialect are assumed to share the same common optimization rules. That is, they are associated with the same set of applicable optimization passes,

including both general and dialect-specific ones, resulting in 11 rules, one per dialect. *For conversion rules,* we retain only passes targeting *llvm*, excluding others (*e.g.*, *"-convert-arith-to-spirv"*). If a dialect maps to a single target, all its operations share one rule. For dialects with multiple targets (*e.g.*, *"-tosa-to-arith"*, *"-tosa-to-linalg"*), we analyze pass implementations using tree-sitter [20] to extract `matchAndRewrite` patterns and establish operation-to-pass mappings. For instance, `tosa.const` and `tosa.apply_scale` are handled by *"-tosa-to-arith"*. In total, we collect 91 conversion rules.

Lowering Path Construction. To select applicable optimization or conversion rules for MLIR programs in Algorithm 1, the used dialects and operations in MLIR programs should be identified (*i.e.*, SCAN function in line 8 and line 16). To this end, we implement a standalone command-line tool leveraging MLIR infrastructure that traverses the MLIR programs and records the used dialects and operations.

MLIR Program and Lowering Path Reduction. To help developers localize bugs, we reduce both the MLIR programs and the buggy lowering paths before reporting. For MLIR programs, we currently perform manual reduction by removing irrelevant parts of the program. For buggy lowering paths, we apply an iterative strategy: starting from the full faulty lowering path, we remove one optimization pass at a time. If this pass appears multiple times in the path, all of its occurrences are removed together. If removing a pass preserves the bug, the pass is deemed irrelevant; otherwise, it is considered related. After exploring all passes, we obtain a minimal set of suspicious passes.

V. EVALUATION

To evaluate LOBE, we address the following research questions:

- **RQ1:** How effective is LOBE in detecting previously unknown bugs in the MLIR infrastructure?
- **RQ2:** How effective is LOBE compared to state-of-the-art technique in testing the MLIR infrastructure?
- **RQ3:** To what extent do the two key components contribute to the successful construction of lowering paths?

A. Experimental Setup

To answer RQ1, we applied LOBE to fuzz recent MLIR revisions, covering commits from 0a44b24 to ba63150. For RQ2 and RQ3, we fixed the MLIR version to 449e2f5, the latest at the start of our experiments. By default, we set the maximum number of lowering attempts per path (*maxStep*) to 30, allocate a 30-minute time budget for each input program, and initialize each operation's priority to 10, with a penalty of 1 applied for failed conversions. We evaluate the appropriateness of the *maxStep* setting in RQ3. The remaining parameters are empirically chosen to balance exploration depth and runtime efficiency. To reduce randomness, each 12-hour experiment was repeated five times. All experiments are executed on a server equipped with an Intel(R) Xeon(R) Gold 6354 CPU @ 3.00GHz, and 500GB of memory, running Ubuntu 20.04.

Compared Work. For RQ2, we compare LOBE with testing based on *the predefined fixed lowering path*. Specifically, we use Ratte [14] as the baseline, as it is the only state-of-the-art MLIR testing technique capable of detecting both miscompilations and crashes. Ratte generates UB-free programs with semantic for each variable, *i.e.*, the expected output values, executes the programs using a predefined fixed lowering path, and detects bugs by comparing the actual execution results against the expected output. Currently, Ratte supports three MLIR dialects: *arith*, *linalg* and *tensor*. We compare LOBE against Ratte using generators: $LOBE_{TOSASmith}$ with our *tosa* generator, and $LOBE_{Ratte}$ which reuses Ratte's generator. While both $LOBE_{Ratte}$ and Ratte share the same generator, they differ in how the lowering path is constructed: Ratte relies on the predefined fixed lowering paths, identical to the one used in their original work, whereas $LOBE_{Ratte}$ employs our lowering space exploration approach to construct multiple diverse lowering paths.

For RQ3, we first evaluate the impact of different *maxStep* settings on the construction of lowering paths. We configure *maxStep* to $\{20, 30, 40, 50\}$ and measure both the number of successful lowering paths and the lowering success rate. Next, to evaluate the contribution of atomic lowering rules and feedback-based scheduling to the successful construction of lowering paths, we constructed two ablation variants of LOBE, each omitting one of the components. Without atomic lowering rules, LOBE randomly selects a pass from the full set of optimization or conversion passes during the respective phases. Without feedback-based scheduling, the conversion phase randomly selects an operation without considering operation priorities.

Evaluation Metrics. We evaluate the effectiveness of LOBE using the following four metrics:

- *Number of Detected Bugs:* For RQ1, we count confirmed and fixed bugs based on developer feedback and patch merges. For RQ2, we compare the number of bugs within a fixed time budget, deduplicating crash bugs by assertion and stack trace, and miscompilations by the faulty pass.
- *Dialect and Operation Coverage:* We record all dialects and operations involved during the lowering process. In both the optimization and conversion phases, we use the SCAN function to extract the set of dialects and operations present in the current program.
- *Code Coverage:* We report line and branch coverage of the MLIR codebase using `llvm-cov` [21], indicating how thoroughly the lowering space exercises internal MLIR logic.
- *Lowering Success Rate:* Used in RQ3 to assess the effectiveness of two core components. It measures the ratio of programs successfully lowered to *llvm* within *maxStep*:

$$lowering\_success\_rate = \frac{\#Success}{\#Success + \#Fail} \quad (4)$$

where #*Success* is the number of programs successfully lowered and executed, and #*Fail* is the number that failed

to lower within *maxStep* .

B. RQ1: Previously Unknown Bugs Detected

Table I summarizes the previously unknown bugs detected by LOBE, including the Bug Id, the pass where the bug occurred, the symptom, and the bug status. In total, LOBE detected 38 previously unknown bugs, including 30 crash bugs and 8 miscompilations. Among them, 6 bugs have been confirmed, and 19 have been fixed. These results demonstrate LOBE's effectiveness in uncovering both miscompilation and crash bugs. Analyzing the passes where the bug occurred, we find that 31 bugs were triggered in dialect-specific optimization passes, 3 bugs in general optimization passes, 3 bugs in conversion passes, and 1 bug in bufferization pass. This highlights the capability of our approach to find bugs across all kinds of passes.

We analyzed the miscompilation cases and identified two representative examples: one consistently produces incorrect results, while the other yields randomized values. Both stem from lowering an MLIR programs consisting of *tosa*, where faulty passes introduce incorrect IR. These issues only arise after long, specific lowering paths (18 and 25 passes), and disappear if the pass order is changed. This demonstrates that such bugs are unlikely to be exposed by fixed lowering paths, highlighting the effectiveness of lowering space exploration.

```
1 %24 = memref.reinterpret_cast %20 to offset: [0], sizes: [2],
      strides: [1] : memref<i32> to memref<2xi32>
2 linalg.generic ins(%19 : memref<2xi32>) outs(%24 : memref<2xi32>) {
3      ^bb0(%in: i32, %out: i32):
4     linalg.yield %9 : i32
5 }
```

(a) Correct IR snippet before applying *"-linalg-specialize-generic-ops"*

```
1 %24 = memref.reinterpret_cast %20 to offset: [0], sizes: [2],
       strides: [1] : memref<i32> to memref<2xi32>
2 linalg.copy ins(%19 : memref<2xi32>) outs(%24 : memref<2xi32>)
```

(b) Buggy IR snippet after applying *"-linalg-specialize-generic-ops"*

Fig. 3: Bug #130002

```
1 memref.dealloc %alloc_17 : memref<1x2x1xi1>
2 memref.dealloc %alloc_16 : memref<1x2x1xi32>
3 %2 = memref.load %alloc_15[%arg0, %arg0, %arg0] : memref<1x2x1xi1>
  ......
4 %3 = affine.load %alloc_16[0, %arg1, 0] : memref<1x2x1xi32>
```

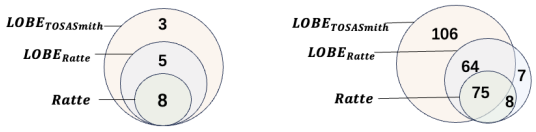
Fig. 4: Bug #130257: Buggy IR snippet after applying *"-affine-data-copy-generate"*

Figure 3 shows a miscompilation case where the *"-linalg-specialize-generic-ops"* pass incorrectly rewrites a `linalg.generic` into a `linalg.copy`, causing a semantic mismatch. Originally, `%19` is used to compute a result stored in `%24` (Figure 3a), but after transformation, `%19` is directly copied to `%24` (Figure 3b). As a result, `%24` holds an incorrect value, *i.e.*, `%19`, leading to a deterministic miscompilation.

Figure 4 illustrates a miscompilation case resulting in a randomized incorrect value. After applying the *"-affine-data-copy-generate"* pass, the compiler produces an incorrect IR, as shown in the generated snippet. Specifically, the varible `%alloc_16` is deallocated (line 2) before it is subsequently accessed by an `affine.load` operation (line 4). As a result, the loaded value `%3` may contain undefined data. This introduces use-after-deallocation, leading to random behavior across multiple executions.

> **RQ1:** In total, LOBE uncovered 38 previously unknown bugs, comprising 8 miscompilations and 30 crash bugs. Of these, 25 have been confirmed or fixed.

C. RQ2: Comparison with Ratte



(a) Dialects (b) Operations

Fig. 5: Number of covered dialects and operations

Table II shows the number of bugs detected by LOBE and *Ratte*. Notably, *Ratte* fails to detect any bugs. In contrast, LOBE, using two generators, detects a total of 39 bugs with 18 of them commonly detected by both generators. Moreover, 10 bugs detected by $LOBE_{Ratte}$ are not captured by $LOBE_{TOSASmith}$. This is because TOSA programs generated by TOSASmith exercise only a subset of the operations from the three dialects (*arith*, *linalg*, and *tensor*) that are covered by programs generated by Ratte. Thus, $LOBE_{TOSASmith}$ is able to expose only a subset of the bugs detected by $LOBE_{Ratte}$. These results underscore the effectiveness of LOBE in enhancing existing testing by systematically exploring diverse lowering paths. To further understand the underlying reasons for this improvement, we analyze the dialect/operation and code coverage achieved by each technique.

Figure 5 shows the dialect and operation coverage for $LOBE_{TOSASmith}$, $LOBE_{Ratte}$, and *Ratte*. At the dialect level (Figure 5a), $LOBE_{TOSASmith}$ covers all dialects reached by the others and identifies three unique ones. This demonstrates that, with a high-level dialect generator and our lowering space exploration approach, it is possible to achieve broad dialect coverage. At the operation level (Figure 5b), $LOBE_{Ratte}$ fully subsumes *Ratte*, indicating that lowering space exploration is more effective than relying on a fixed lowering path, when using the same generator. While $LOBE_{TOSASmith}$ covers most operations reached by the others (about 80% of $LOBE_{Ratte}$ and 90% of *Ratte*), it misses a few low-level operations. This suggests that not all low-level operations are necessarily invoked during the lowering process from high-level dialects. Therefore, combining multiple generators or increasing the diversity of initial programs can be beneficial for improving operation-level coverage.

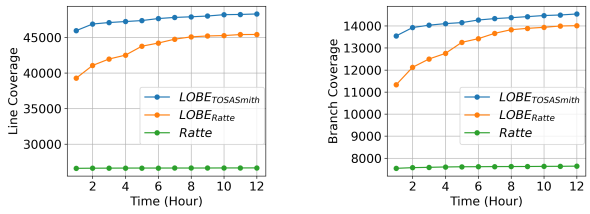
TABLE I: Summary of previously unknown bugs detected by LOBE

Bug Id	Pass	Symptom	Status	Bug Id	Pass	Symptom	Status
118268	Conversion	Crash	Fixed	121020	Dialect Specific(affine)	Crash	Fixed
118450	General	Crash	Fixed	121091	Dialect Specific(memref)	Crash	Fixed
118784	Dialect Specific(math)	Miscompilation	Reported	121321	Dialect Specific(affine)	Crash	Confirmed
118790	Dialect Specific(scf)	Miscompilation	Confirmed	121328	Dialect Specific(linalg)	Crash	Reported
119308	Dialect Specific(memref)	Crash	Reported	122076	Dialect Specific(arith)	Crash	Reported
119351	Dialect Specific(affine)	Miscompilation	Fixed	122090	Dialect Specific(memref)	Crash	Fixed
119353	Dialect Specific(func)	Crash	Confirmed	122094	Dialect Specific(linalg)	Crash	Confirmed
119364	General	Crash	Fixed	122208	Dialect Specific(affine)	Crash	Fixed
119367	Dialect Specific(memref)	Crash	Reported	122210	Dialect Specific(affine)	Crash	Fixed
119378	Dialect Specific(scf)	Crash	Fixed	122211	Dialect Specific(affine)	Crash	Reported
119525	Dialect Specific(affine)	Crash	Fixed	122213	Dialect Specific(affine)	Crash	Reported
119658	Dialect Specific(math)	Crash	Reported	122226	Dialect Specific(linalg)	Crash	Reported
119683	Dialect Specific(memref)	Miscompilation	Reported	122227	Dialect Specific(affine)	Crash	Fixed
119825	Dialect Specific(scf)	Miscompilation	Reported	122231	Dialect Specific(affine)	Crash	Fixed
120001	Conversion	Crash	Fixed	122247	Dialect Specific(linalg)	Crash	Reported
120186	Conversion	Crash	Fixed	122258	Dialect Specific(linalg)	Crash	Confirmed
120189	Dialect Specific(affine)	Crash	Fixed	130002	Dialect Specific(linalg)	Miscompilation	Fixed
120193	General	Crash	Fixed	130257	Dialect Specific(affine)	Miscompilation	Fixed
120535	Bufferization	Crash	Confirmed	140259	Dialect Specific(scf)	Miscompilation	Reported

TABLE II: Number of detected bugs

Techniques	Crash	Miscompilation	Total
$LOBE_{TOSASmith}$	25 $^{(17)}$	4$^{(1)}$	29$^{(18)}$
$LOBE_{Ratte}$	26$^{(17)}$	2$^{(1)}$	28$^{(18)}$
$Ratte$	0	0	0

Superscripts indicate the number of common bugs between $LOBE_{TOSASmith}$ and $LOBE_{Ratte}$.



(a) Line coverage (b) Branch coverage

Fig. 6: Coverage information over a 12-hour budge

Figure 6 shows the evolution of line and branch coverage. As observed, both line and branch coverage exhibit similar trends across the techniques. $LOBE_{TOSASmith}$ consistently achieves the highest coverage throughout the experiment. We use Mann Whitney U test [22] to assess the statistical significance of coverage results. Specifically, for line coverage, $LOBE_{TOSASmith}$ achieves a final coverage of 48,294 lines, while $LOBE_{Ratte}$ reaches 45,417 lines, representing 81.05% and 70.27% improvements over $Ratte$, respectively. These improvements are statistically significant (both $p = 0.008 < 0.05$). For branch coverage, $LOBE_{TOSASmith}$ covers 14,539 branches, while $LOBE_{Ratte}$ covers 14,007, representing 90.33% and 83.36% improvements over $Ratte$, respectively. Both improvements are statistically significant ($p = 0.016$ and $p = 0.008$, respectively). These results indicate that lowering space exploration is more effective than using a fixed lowering path, as it enables broader coverage by dynamically adapting to the semantics of the input program. This expanded coverage potentially increases the opportunity to uncover bugs.

Comparing the results of $LOBE_{Ratte}$ and $Ratte$, we observe that with the same generator, lowering space exploration used in $LOBE_{Ratte}$ achieves significantly broader coverage than the fixed lowering path used in $Ratte$. In particular, $Ratte$ shows minimal improvement over time, suggesting that merely increasing the diversity of input programs, without varying the lowering path, provides limited testing capability. In contrast, exploring multiple lowering paths by reordering conversions and injecting additional optimizations enables more diverse transformations and leads to more effective testing.

> **RQ2:** Within a five repated 12-hour budget, LOBE with two generators detects **39** bugs, whereas Ratte with a predefined fixed lowering path detects none. Additionally, LOBE improves line coverage by 81.05% and branch coverage by 90.33%.

D. RQ3: Ablation Study

Figure 7 shows the impact of different *maxStep* settings on the number of successful lowering paths (bar chart) and the overall lowering success rate (line chart). We observe that setting *maxStep* to 30 achieves a good balance between these two metrics. While increasing *maxStep* generally improves the success rate, it also reduces the total number of successful lowering paths due to increased time spent on failing paths. Based on this insight, we set *maxStep* to 30 as the default configuration in our experiments.

Table III shows the effectiveness of feedback-based scheduling and atomic lowering rules in lowering path construction. We report the number of successful (#Success) lowering paths and the success rate (Rate). As shown in the table, enabling feedback-based scheduling improves the overall lowering success rate by 18.17% and increases the number of successful lowering paths by 37.32%. In contrast, enabling atomic lower-

ing rules leads to a dramatic improvement, boosting the lowering success rate by 161,850% and the number of successful lowering paths by 218,930%. Without atomic lowering rules, it is nearly impossible to construct valid lowering paths.

Notably, the benefit of feedback-based scheduling is more pronounced in $LOBE_{TOSASmith}$, as *tosa* is a higher-level dialect than those in $LOBE_{Ratte}$ (*i.e.*, *arith*, *linalg*, *tensor*). Upon inspecting the lowering logs, we found that lowering from *tosa* to *llvm* requires more steps on average (18 vs. 6/11/10), leading to a larger and more sensitive exploration space. In such cases, the feedback-based scheduling effectively guides the search by dynamically adjusting operation priorities, avoiding inefficient exploration and improving efficiency.

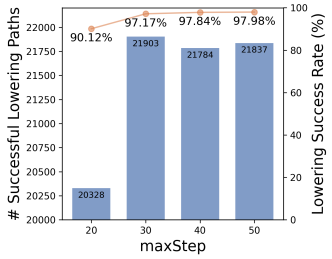


Fig. 7: Number of successful lowering paths and lowering success rate under different *maxStep* settings.

TABLE III: Results of the number of successful lowering paths and lowering success rates for variants

	$LOBE_{Ratte}$		$LOBE_{TOSASmith}$		**Total**	
	#Success	Rate	#Success	Rate	#Success	Rate
All	12822	97.49%	9081	96.72%	21903	97.17%
w/o feedback	7323	69.75%	8627	96.95%	15950	82.23%
w/o rules	10	0.17%	0	0%	10	0.06%

RQ3: Without atomic lowering rules, the success rate plummets from 97.17% to just 0.06%. Feedback-based scheduling improves the lowering success rate by 18.17% and the number of successful lowering paths by 37.32%.

VI. Discussion

Floating-point Roundoff Error. Our approach is theoretically free of false positives, as the *lowering equivalence property* should always hold. However, in practice, certain optimizations in MLIR, such as *"-arith-emulate-unsupported-floats"*, may introduce floating-point roundoff errors, potentially leading to false positives. To address this, we follow prior work [23] and report a miscompilation only when the output deviation exceeds a threshold (set to 1 in our experiments). We also manually inspect such cases to ensure correctness.

Parameter Configuration. Our approach involves several parameters, including *maxStep*, time budget, and the initial priority of operation and penalty values. For *maxStep*, we have empirically validated that setting it to 30 offers a good balance between lowering success rate and the number of successful lowering path (as shown in Figure 7). The other parameters are set empirically in our experiments. Regarding the 30-minute time budget per input program, we observe that it enables the exploration of an average of 600 lowering paths, which already constitutes a substantial search effort in practice. The initial priority (10) and penalty (1) guide the scheduling based on compilation feedback. In the worst case, the process falls back to random selection, which can still successfully construct paths by exploring the lowering space.

Difficulties in fixing Miscompilations. Compared to crash bugs, miscompilations are significantly more difficult to diagnose and fix. Identifying the faulty pass requires analyzing a long sequence of transformations to locate where the semantic deviation is introduced. In our reported cases, most miscompilation bugs involve more than 15 passes, making it difficult to trace the root cause. Without a clear indication of which pass introduced the error, developers are often unable to determine who is responsible for the issue. Moreover, many of the miscompilations we discovered are related to complex memory allocation and bufferization logic. As illustrated in Figure 4, resolving such issues often requires non-trivial modifications to the transformation logic, making them considerably more challenging and time-consuming to address.

VII. Related Work

MLIR Testing. In recent years, significant progress has been made in MLIR testing, with most efforts focusing on test program generation. These approaches generally fall into two categories: generating UB-free programs to detect both miscompilations and crashes [14], [24], and producing diverse programs to uncover crash bugs [10], [11], [25]. *For the first category,* Ratte [14] adopts an iterative process that alternates between program generation and semantic evaluation to construct well-defined programs. The final generated program includes the interpreted values of variables, which are then compared against the values obtained after applying a predefined, fixed lowering path followed by execution. DESIL [24] defines a set of UB-elimination rules to sanitize existing programs, and then predefines a lowering path based on the operations present in the program. Different optimization passes are inserted along this path to compare execution results and check for inconsistency. *For the second category,* MLIRSmith [10] is a grammar-based generator that randomly produces syntactically valid MLIR programs. Building on this, MLIRod [11] introduces operation-dependency coverage to guide the mutation of existing programs, aiming to uncover more bugs. To further reduce manual effort, SynthFuzz [25] automatically derives mutation strategies from existing programs and leverages them to generate new programs.

Our approach detects both miscompilations and crashes, but differs significantly from existing techniques. Rather than focusing on generating UB-free programs and executing them along predefined fixed lowering paths, LOBE leverages the multi-level architecture of the MLIR framework to dynamically construct multiple lowering paths. This enables broader and deeper exploration of the lowering space, leading to more comprehensive testing of the MLIR infrastructure.

Differential Testing for Compiler. Our approach can be categorized into differential testing [26], which provides a strong oracle for detecting bugs. The core idea is to compile and execute the same input program across different settings, such as different compiler implementations [23], [27]–[29], optimization levels [30], [31], or hardware backends [32], and check for behavioral inconsistencies. For example, Csmith [33] randomly generates C programs and detects miscompilations by comparing their execution results across GCC and LLVM. HirGen [32] generates diverse computational graphs to test whether the compiled results produced by the TVM framework yield consistent execution outputs across different hardware platforms, such as CPU and GPU. Debug^2 [31] detects debug location bugs by comparing the debugging behavior of programs compiled with and without optimizations.

Using different compiler implementations as an oracle is not applicable to MLIR, as there are no equivalent alternative implementations for MLIR. LOBE proposes a novel oracle based on the consistency of execution results across different lowering paths—a capability uniquely enabled by MLIR's multi-level IR architecture. In this context, traditional optimization levels can be seen as a subset of the lowering space explored by our approach, while hardware backends are orthogonal and can be integrated as an extra testing dimension.

Both JoNM [34] and our approach explore the compilation space to uncover bugs. However, JoNM focuses on validating JIT compilers by enumerating all possible compilation paths at the function level, choosing between compilation and interpretation for each function. In contrast, our approach constructs diverse lowering paths by systematically exploring different passes and their application orders in the compilation pipeline. Since exhaustively enumerating all possible lowering sequences in MLIR is infeasible due to the vastness of the space, our contribution lies in dynamically constructing lowering paths based on the operations present in the current program, and adaptively selecting the next operation to lower based on compilation feedback. This technique efficiently explores the lowering space, enabling scalable and systematic testing of the MLIR infrastructure.

VIII. Conclusion

In this paper, we propose lowering space exploration, a novel approach to automatically finding bugs in MLIR infrastructure. Our approach dynamically constructs lowering paths based on compilation feedback. By comparing the execution results across these different paths, we can effectively identify inconsistencies that signal bugs.

We implemented our approach in a tool called LOBE. Our evaluation demonstrates the effectiveness of LOBE: it discovered 38 previously unknown bugs (including 8 miscompilations and 30 crash bugs), with 25 already confirmed or fixed. Comparison with the state-of-the-art shows that lowering space exploration significantly improves bug detection and coverage. LOBE detected 39 bugs, while Ratte found none. It achieved full dialect and operation coverage relative to Ratte and further increased line coverage by 81.05% and branch coverage by 90.33%. Our ablation results show both feedback-based scheduling and atomic lowering rules are essential to lowering path construction.

Acknowledgment

We thank the anonymous ASE reviewers for their valuable feedback. This work was supported in part by National Key Research and Development Program (Grant 2022YFB3104002) and Shanghai Trusted Industry Internet Software Collaborative Innovation Center. This work was also supported by Open Research Project of the State Key Lab for Novel Software Technology, Nanjing University (Grant KFKT2025B06).

References

[1] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[2] M. P. Lücke, O. Zinenko, W. S. Moses, M. Steuwer, and A. Cohen, "The MLIR transform dialect: Your compiler is more powerful than you think," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, 2025, pp. 241–254.

[3] Y. Wang, C. Tirelli, L. Orlandic, J. Sapriza, R. R. Álvarez, G. Ansaloni, L. Pozzi, and D. Atienza, "An mlir-based compilation framework for cgra application deployment," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2025, pp. 33–50.

[4] J. Zhuang, S. Xiang, H. Chen, N. Zhang, Z. Yang, T. Mao, Z. Zhang, and P. Zhou, "ARIES: An agile mlir-based compilation flow for reconfigurable devices with ai engines," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2025, pp. 92–102.

[5] MLIR Project. (2025) Users of MLIR. [Online]. Available: https://mlir.llvm.org/users/

[6] (2025) CIRCT. [Online]. Available: https://github.com/llvm/circt

[7] (2025) Nod Distributed Runtime: Asynchronous fine-grained op-level parallel runtime. [Online]. Available: https://www.amd.com/en.html

[8] (2025) iree. [Online]. Available: https://github.com/iree-org/iree

[9] (2025) onnx-mlir. [Online]. Available: https://github.com/onnx/onnx-mlir

[10] H. Wang, J. Chen, C. Xie, S. Liu, Z. Wang, Q. Shen, and Y. Zhao, "MLIRSmith: Random program generation for fuzzing mlir compiler infrastructure," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1555–1566.

[11] C. Suo, J. Chen, S. Liu, J. Jiang, Y. Zhao, and J. Wang, "Fuzzing MLIR compiler infrastructure via operation dependency analysis," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1287–1299.

[12] (2025). [Online]. Available: https://mlir.llvm.org/docs/Dialects/ArithOps/

[13] (2025). [Online]. Available: https://mlir.llvm.org/docs/Dialects/LLVM/

[14] P. Yu, N. Wu, and A. F. Donaldson, "Ratte: Fuzzing for miscompilations in multi-level compilers using composable semantics," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 966–981.

[15] (2025) Error Function. [Online]. Available: https://en.wikipedia.org/wiki/Error_function

[16] (2025) MLIR — Running and Testing a Lowering. [Online]. Available: https://www.jeremykun.com/2023/08/10/mlir-running-and-testing-a-lowering/

[17] (2025) Tutorial of mlir-opt. [Online]. Available: https://mlir.llvm.org/docs/Tutorials/MlirOpt/

[18] B. Lecoeur, H. Mohsin, and A. F. Donaldson, "Program reconditioning: Avoiding undefined behaviour when finding and reducing compiler bugs," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1801–1825, 2023.

[19] (2025) Passes in MLIR. [Online]. Available: https://mlir.llvm.org/docs/Passes/#-one-shot-bufferize

[20] (2025) Introduction of tree-sitter. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[21] (2025) llvm-cov - emit coverage information. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-cov.html

[22] P. E. McKnight and J. Najab, "Mann-Whitney U Test," in *The Corsini Encyclopedia of Psychology*. John Wiley & Sons, 2010, pp. 1–1.

[23] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "NNSmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023, pp. 530–543. [Online]. Available: https://doi.org/10.1145/3575693.3575707

[24] C. Suo, J. Wang, Y. Wang, J. Jiang, Q. Shen, and J. Chen, "DESIL: Detecting silent bugs in mlir compiler infrastructure," *arXiv preprint arXiv:2504.01379*, 2025. [Online]. Available: https://arxiv.org/abs/2504.01379

[25] B. Limpanukorn, J. Wang, H. J. Kang, E. Z. Zhou, and M. Kim, "Fuzzing MLIR compilers with custom mutation synthesis," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2025.

[26] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[27] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with yarpgen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 196:1–196:25, 2020. [Online]. Available: https://doi.org/10.1145/3428264

[28] G. Ye, T. Hu, Z. Tang, Z. Fan, S. H. Tan, B. Zhang, W. Qian, and Z. Wang, "A generative and mutational approach for synthesizing bug-exposing test cases to guide compiler fuzzing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1127–1139. [Online]. Available: https://doi.org/10.1145/3611643.3616332

[29] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "Automated conformance testing for javascript engines via deep compiler fuzzing," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 435–450. [Online]. Available: https://doi.org/10.1145/3453483.3454054

[30] Y. Li, S. Ding, Q. Zhang, and D. Italiano, "Debug information validation for optimized code," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 1052–1065. [Online]. Available: https://doi.org/10.1145/3385412.3386020

[31] G. A. Di Luna, D. Italiano, L. Massarelli, S. Österlund, C. Giuffrida, and L. Querzoni, "Who's debugging the debuggers? exposing debug information bugs in optimized binaries," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 1034–1045.

[32] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 248–260.

[33] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[34] C. Li, Y. Jiang, C. Xu, and Z. Su, "Validating JIT compilers via compilation space exploration," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 66–79.