# DuoReduce: Bug Isolation for Multi-layer Extensible Compilation

JIYUAN WANG, University of California, Los Angeles, USA
YUXIN QIU, University of California, Riverside, USA
BEN LIMPANUKORN, University of California, Los Angeles, USA
HONG JIN KANG, University of California, Los Angeles, USA
QIAN ZHANG, University of California, Riverside, USA
MIRYUNG KIM, University of California, Los Angeles, USA

In recent years, the MLIR framework has had explosive growth due to the need for extensible deep learning compilers for hardware accelerators. Such examples include Triton [39], CIRCT [14], and ONNX-MLIR [22]. MLIR compilers introduce significant complexities in localizing bugs or inefficiencies because of their layered optimization and transformation process with compilation passes. While existing delta debugging techniques can be used to identify a minimum subset of IR code that reproduces a given bug symptom, their naive application to MLIR is time-consuming because real-world MLIR compilers usually involve a large number of compilation passes. Compiler developers must identify a minimized set of relevant compilation passes to reduce the footprint of MLIR compiler code to be inspected for a bug fix. We propose DuoReduce, a dual-dimensional reduction approach for MLIR bug localization. DuoReduce leverages three key ideas in tandem to design an efficient MLIR delta debugger. First, DuoReduce reduces compiler passes that are irrelevant to the bug by identifying ordering dependencies among the different compilation passes. Second, DuoReduce uses MLIR-semantics-aware transformations to expedite IR code reduction. Finally, DuoReduce leverages cross-dependence between the IR code dimension and the compilation pass dimension by accounting for which IR code segments are related to which compilation passes to reduce unused passes.

Experiments with three large-scale MLIR compiler projects find that DuoReduce outperforms syntax-aware reducers such as Perses and Vulcan in terms of IR code reduction by 31.6% and 21.5% respectively. If one uses these reducers by enumerating all possible compilation passes (on average 18 passes), it could take up to 145 hours. By identifying ordering dependencies among compilation passes, DuoReduce reduces this time to 9.5 minutes. By identifying which compilation passes are unused for compiling reduced IR code, DuoReduce reduces the number of passes by 14.6%. This translates to not needing to examine 281 lines of MLIR compiler code on average to fix the bugs. DuoReduce has the potential to significantly reduce debugging effort in MLIR compilers, which serves as the foundation for the current landscape of machine learning and hardware accelerators.

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

Additional Key Words and Phrases: MLIR, Fault Localization, Multi-Layer Extensible Compilation

Authors' Contact Information: Jiyuan Wang, University of California, Los Angeles, USA, wangjiyuan@cs.ucla.edu; Yuxin Qiu, University of California, Riverside, USA, yuxin.qiu@email.ucr.edu; Ben Limpanukorn, University of California, Los Angeles, USA, ben@limpanu.com; Hong Jin Kang, University of California, Los Angeles, USA, hjkang@cs.ucla.edu; Qian Zhang, University of California, Riverside, USA, qzhang@cs.ucr.edu; Miryung Kim, University of California, Los Angeles, USA, miryung@cs.ucla.edu.

ACM 2994-970X/2025/7-ARTFSE030
https://doi.org/10.1145/3715747

**ACM Reference Format:**
Jiyuan Wang, Yuxin Qiu, Ben Limpanukorn, Hong Jin Kang, Qian Zhang, and Miryung Kim. 2025. DuoReduce: Bug Isolation for Multi-layer Extensible Compilation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE030 (July 2025), 21 pages. https://doi.org/10.1145/3715747

## 1 Introduction

LLVM [18] and the Multi-Level Intermediate Representation (MLIR) [15] framework are gaining significant traction across a wide array of compilers for machine learning and heterogeneous computing. They have revolutionized modern compiler architectures with layered and extensible compiler development, by enabling custom extension of IRs. For example, Triton [39], a cutting-edge language and compiler for machine learning developed by OpenAI, leverages the MLIR framework to translate Python kernels into Triton's IR representation. This IR is then optimized through various transformation passes before being lowered to PTX assembly, which is then fed to the CUDA compiler. The increasing adoption of MLIR compiler projects such as Triton underscores the importance of extensible compiler design, particularly in the domain of machine learning and hardware accelerators, where the underlying IRs and key optimization strategies are rapidly evolving.

The key features of MLIR are *dialects* and *compilation passes* [15, 17]. Each dialect represents a particular IR abstraction. The core MLIR project [1] alone consists of 46 dialects. Compiler developers also introduce additional *custom dialects*. A pass in MLIR transforms and optimizes IR within a dialect or from one dialect to another. The core MLIR project [1] includes 232 passes that either (1) transform the IR code from high-level dialects to low-level dialects, or (2) optimize IR code within the same dialect. For example, the `lower-host-llvm` pass translates the `host` dialect into the `llvm` dialect, while `affine-loop-fusion` pass fuses loops within the `affine` dialect. We use the term compilation path in MLIR to denote an ordered set of compilation passes that are *executed sequentially* to transform a high-level input IR to a low-level IR.

Consider the Circuit IR Compilers and Tools (CIRCT) [14] project that leverages the MLIR framework to build a compiler for heterogeneous hardware accelerators. As shown in Figure 1, CIRCT transforms high-level language code such as Python and C into Verilog. It defines 30 different hardware-related custom dialects and 206 compilation passes. Similarly, other MLIR-based compilers use a significant number of compilation passes. For instance, the GPU stencil optimization [12] consists of 27 compilation passes. This size of available compilation passes introduces significant complexity in bug isolation. Suppose that a developer attempts to reproduce the same compiler error by exhaustively testing all $2^{27}$ combinations naively by turning on and off each compilation pass. This could take more than 1000 days to complete. Take the CIRCT Python frontend, PyCDE [2] as another example. It converts high-level Python constructs to optimized Verilog code using 15 distinct compilation passes. These optimization passes include crucial tasks like resource sharing, pipelining, and clock domain management. A naive attempt would require testing $2^{15}$ combinations, taking more than 18 hours to complete.

MLIR's extensible compiler architecture enables rapid IR evolution and flexibility; however, it results in increased effort in compiler debugging for two reasons.

- *Interdependent Passes.* Compilation passes in MLIR are not isolated; they inherently depend on each other. Thus, a naive approach of toggling each pass on and off to isolate a reported bug is inefficient. Real-world MLIR compiler errors result from interactions between multiple passes. For example, the crash described in GitHub issue [3] is reproducible only if the `convert-parallel-loops-to-gpu` pass is preceded by the the `affine-loop-tile` pass out of 10 passes, and the other 8 passes are irrelevant.

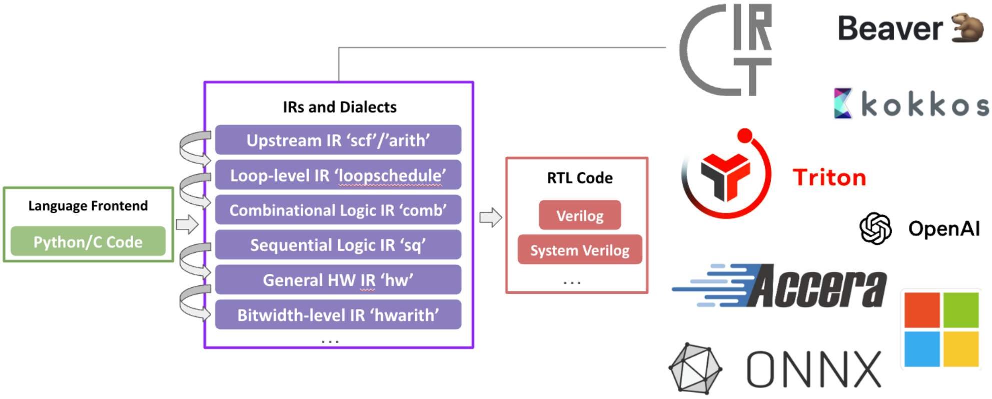


Fig. 1. CIRCT's multi-layer compilation involves 4 core dialects and 26 optimizing dialects such as hwarith and sq. An MLIR input program goes through on average 13 transformation passes and produces the resulting Verilog code. The MLIR community is growing and as of Sep. 2024, there are 30 MLIR-based compiler projects on GitHub.

- *Volume of Generated Code.* A few lines of high-level input IR can still produce a large amount of intermediate or low-level IR. Therefore, input isolation at high-level IR is inadequate, and compiler developers need pass isolation to determine which compiler optimization module is the root cause of a bug.

Existing general-purpose delta debugging (DD) tools, such as Perses [36], reduce test cases by performing syntax-aware code splitting and deletion. However, they often fail to precisely localize errors in minimized IR code due to the lack of consideration of compilation pass dependencies. For example, if a compiler developer includes an irrelevant `lower-hw-to-sv` pass, the input IR must have operations in dialect `hw`, because, by definition, this pass lowers dialect `hw` to dialect `sv`. However, if we can determine that pass `lower-hw-to-sw` is irrelevant, we can remove `hw` specific operations safely. Similarly, existing MLIR's debugging utility, `mlir-reduce` [4] considers all available compilation passes provided by a user without identifying which compilation passes are relevant, leading to poor performance.

Alternatively, a developer may use LLVM's crash message [18] to get hints on the culprit compilation passes; however, it always points to the last pass before a crash rather than the actual set of problematic passes that produce culprit IR code. In other words, it identifies too few compilation passes and fails to include all relevant passes that cause the crash. For example, on the previous GitHub issue [3], LLVM's crash message identifies the `convert-parallel-loops-to-gpu` pass as the culprit, overlooking the actual buggy pass `affine-loop-tile` with specific tile size parameters. Furthermore, relying on LLVM crash messages for debugging is inadequate for non-crash compiler bugs that instead result in incorrect code generation. For example, the Verilog code generated in Figure 2c contains an error due to the assignment of `regvar` occuring with two operations in lines 5-6, rather than the correct single operation in line 7. The extra operations at lines 5-6 need one more time frame to complete, which leads to incorrect time-signal generation in the EDA simulation, as stated in the GitHub bug issue [5]. Since there is no explicit compiler crash message, no hints are provided to the developer.

Driven by the complexity of debugging multi-layer extensible compilation, we introduce a new technique called DuoReduce, which has three key components:

```
hw.module @Test(in %x :
    i8, in %clock:
    i1) {

  %regvar = sv.reg :
      !hw.inout<i8>
  sv.assign %regvar, %x
      : i8

  %regwithinit = sv.reg
      init %x :
      !hw.inout<i8>
}
```

(a) **Original MLIR code**

```
module {
  hw.module @Test(in %x : i8,
      in %clock : i1) {
    %regvar = sv.reg
        {hw.verilogName =
        "regvar"} :
        !hw.inout<i8>
    sv.assign %regvar, %x : i8
    %regwithinit = sv.reg
        init %x
        {hw.verilogName =
        "regwithinit"} :
        !hw.inout<i8>
    hw.output}}
```

(b) **MLIR code after pass “lower-hw-to-sv”**

```
module Test(  // o.mlir:1:1
  input [7:0] x, //
      o.mlir:1:20
  input      clock //
      o.mlir:1:32
);
  reg [7:0] regvar; //
      o.mlir:3:13
  assign regvar = x; //
      o.mlir:4:3
//correct: reg [7:0] regvar
    = x;
  reg  [7:0] regwithinit = x;
endmodule
```

(c) **Wrong verilog code after pass “export-verilog”**

Fig. 2. The CIRCT bug # 6317 from GitHub [5] does not include which compilation passes were used to detect this bug. If a user want to compile program (a) with pass `export-verilog` only to reproduce the bug, the compiler will crash with message *“Error: Unsupported operation found in design”* This shows the need of identifying a correct ordered set of compilation passes to reproduce the same bug. The original MLIR code (a) in the hw dialect must go through a first compilation pass `lower-hw-to-sv` to produce code (b) and go through a second compilation pass `export-verilog` to produce the Verilog code (c). There are 205 compilation passes available in CIRCT; thus, it is extremely challenging to isolate 2 out of 205 passes.

- **Dependency-Aware Compilation Path Reduction:** To efficiently reduce the search space of compilation passes, DuoReduce reasons about dependencies among compilation passes and computes a valid compilation path for a given IR input.
- **MLIR Transformations:** To reduce the buggy IR code effectively, DuoReduce applies two MLIR-specific transformations: *return operand rollback* and *constant replacement* based on the insight that MLIR compilation crashes often result from operator usage rather than operand values. By replacing operands with constants or returning intermediate operands instead of the final ones, DuoReduce can decompose the input IR into finer-grained units, suitable for reduction.
- **IR-Compilation Pass Dual-Dimensional Reduction:** By exploiting cross-dependencies between IR code and compilation passes, DuoReduce removes unnecessary passes. In our experiment, each pass removal results in not needing to inspect 281 lines of compiler code to inspect on average.

We conduct a systematic evaluation of DuoReduce on all reproducible Github bug issues for three open-source MLIR projects in heterogeneous hardware and machine learning domains: MLIR [1], CIRCT [14], and ONNX-MLIR [22]. We compare DuoReduce's performance against three baseline approaches—(1) Perses [36], a syntax-aware delta debugger, (2) MLIR-reduce [4], a state-of-the-practice MLIR reducer, and (3) Vulcan [42], a delta debugger with general code transformations. We also conduct ablation studies to quantify the benefit of each of the three components above. We use two assessment metrics: (a) the IR code reduction rate, and (b) the compilation pass reduction rate and how it translates to reduction in the amount of MLIR compiler code to be inspected, and (c) the overall time saving for debugging.

In terms of IR code reduction, DuoReduce outperforms Perses and Vulcan by 32% and 22% respectively. In terms of reducing compilation passes, DuoReduce achieves 14% pass reduction than DuoReduce without dual-dimensional reduction. This translates to not needing to inspect 281 lines of MLIR compiler code per bug. Suppose that each IR program goes through 18 compilation passes on average and each compilation trial takes 2 seconds to finish. Naively enumerating all pass combinations to find buggy passes needs $2^{18}$ compilation attempts, taking 145 hours. By reasoning about compilation pass dependencies, DuoReduce reduces the IR debugging time from 145 hours to 572.35 seconds. In summary, this paper makes the following contributions:

(1) We develop DuoReduce that effectively manages the complexities of multi-layer extensible compilation with a large number of compilation passes, enabling precise fault localization.
(2) Compared to prior work that does not reduce the dimension of compilation passes, DuoReduce performs IR code reduction and compilation pass reduction in tandem, achieving 17%, 7%, and 12% higher reduction on compilation passes for MLIR, CIRCT, and ONNX-mlir, which translates to not needing to inspect 7389, 1131, and 201 lines of compiler code respectively.
(3) Overall, DuoReduce achieves an estimated speedup of 901×. For MLIR compilers with more than 20 passes, DuoReduce can deliver even greater speedup, 1931× on average.
(4) DuoReduce outperforms existing reducers in terms of IR code dimension with MLIR-specific transformations. DuoReduce achieves 17%, 15%, and 11% higher IR code reduction on average on MLIR, CIRCT, and ONNX-mlir.

In the current AI era, where companies such as AWS, Microsoft, and Google invest significant resources for in-house AI processor and hardware compiler development, DuoReduce has the significant potential to expedite the overall development time of extensible optimizing compilers. The remainder of this paper is organized as follows. Section 2 introduces MLIR and a motivating example. Section 3 presents the design of DuoReduce. Section 4 provides the design of our experiments and their results. Section 5 introduces the related work. We draw the conclusions of our work in Section 6, and make artifacts available in Section 7.

## 2 Background

### 2.1 Multi-Level Intermediate Representation

MLIR [15] aims to offer a unified infrastructure that can represent IR at multiple levels of abstraction. Its core idea is to define extensible IRs that can be customized with domain-specific *dialects* [15] using *compilation passes* specific to particular domains, such as machine learning, high-performance computing, or embedded systems. All 30 MLIR projects listed on MLIR's homepage [1] define a large number of compilation passes; Triton [39] from NVIDIA has 258 passes, CIRCT has 205 passes, and ONNX-MLIR has 34 passes. As such, MLIR enables code optimization and translation from high-level domain-specific abstractions to lower-level architecture-specific machine code through multiple compilation passes. We use the term *compilation path* $P$ to refer to a sequence of *compilation passes* $P = [P_i, 1 \leq i \leq n]$ For each compilation path $P$, we define two relations.

- **Execution Order Relation**: we define a binary relation $\leq$ to indicate the *execution order*, such that $P_i \leq P_j$ means $P_i$ is executed before $P_j$ in the compilation path $P$.
- **Dependence Relation**: A compilation pass $P_j$ depends on another pass $P_i \rightharpoondown P_j$ if executing $P_j$ without executing $P_i$ leads to a different compilation outcome in terms of reproducing the same bug symptom. It is important to note executing $P_j$ after $P_i$ (i.e., $P_i \leq P_j$) does not directly imply $P_j$ depends on $P_i$ (i.e., $P_i \rightharpoondown P_j$). However, $(P_i \rightharpoondown P_j) \Rightarrow (P_i \leq P_j)$.

DuoReduce finds the shortest sublist $P'$ that can preserve the same bug as the original compilation path $P$. For each pass $P_i \in P'$, each $P_k$ that $P_i$ depends on (i.e., $P_k \rightharpoondown P_i$) must be included in $P'$. For example, in the GitHub issue 82382 [3], the original post contains $P$ with 10 passes. DuoReduce finds an alternative shortest path $P'$ with 2 passes: `affine-loop-tile` ⇁ `convert-parallel-loops-to-gpu`.

CIRCT [14] has 196 passes to compile high-level Python or C code to custom hardware expressed in low-level RTL. Figure 2 shows an example compilation process based on two passes. The registers `%regvar` (line 3 in Figure 2a) and `%regwithinit` (line 6) are assigned and initialized the value `%x`.

```
1   func.func @main() {
2     func.func @matmul(%arg0, %arg1, %arg2)
3       ...
4     call @matmul(%2, %0, %1):(memref<128x128xf32>,
          memref<128x128xf32>, memref<128x128xf32>)
          -> ()
5     scf.for %arg0 = %c0 to %c128 step %c1 {
6       scf.for %arg1 = %c0 to %c128 step %c1 {
7         %3 = memref.load %2[%arg0, %arg1] :
              memref<128x128xf32>
8         %4 = arith.cmpf oeq, %3, %cst_0 : f32 -> i1
9           cf.assert %4, "Matmul does not produce the
                right output"}}
10  ...
11  return}}
```

(a) A buggy IR code reported in the MLIR GitHub issue 56914 [6]: compute matrix-multiplication and verify.

```
1     func.func @main() {
2   - ~~func.func matmul(%arg0, %arg1, %arg2)~~
3       ...
4   - ~~call matmul(%2, %0, %1):(memref<128x128xf32>,
          memref<128x128xf32>, memref<128x128xf32>) ->
          ()~~
5   - ~~scf.for %arg0 = %c0 to %c128 step %c1 {~~
6   - ~~  scf.for %arg1 = %c0 to %c128 step %c1 {~~
7   - ~~%3 = memref.load %2[%arg0, %arg1] :
          memref<128x128xf32>~~
8   - ~~%4 = arith.cmpf oeq, %3, %cst_0 : f32 -> i1~~
9   + %4 = arith.constant 1: i1
10      cf.assert %4, "Matmul does not produce the
            right output"}
11      }
12  ...
13  return}}
```

(b) After applying constant replacement at lines 8-9, DuoReduce removes lines 2-7.

```
1  Stack dump:
2  0.      Program arguments: mlir-cpu-runner -e main -entry-point-result=void
3   #0 0x000055d56f369ee0 PrintStackTraceSignalHandler(void*)
4   #1 0x000055d56f367904 SignalHandler(int)
5   ...
```

(c) LLVM's crash message shows only the last pass `mlir-cpu-runner`, which is not the root cause of the crash.

Fig. 3. MLIR bug #56914 [6]: the original input IR code in Fig 3a and 9 compilation passes to reproduce the crash in Fig 3c. If she compiles the input IR with all available 9 passes, 4480 lines of compiler code should be inspected. In Fig 3b, DuoReduce successfully removed the `@matmul` with "constant replacement" transformation in Section 3.2. DuoReduce removes 6 irrelevant compilation passes out of 9, leaving only 1245 lines of compiler code to inspect.

The original MLIR code is first converted to dialect `sv` in Figure 2b with pass `lower-hw-to-sv` and then lowered to System Verilog code with pass `export-verilog` in Figure 2c: $P_1$=`lower-hw-to-sv` and $P_2$=`export-verilog`, where $P_2 \rightharpoondown P_1$. When a bug occurs in a compilation path with $n$ passes, developers may naively enumerate all $2^n$ *combinations* of passes. Such a naive attempt can easily take up to 100 hours when $n$ is over 18. DuoReduce addresses this very problem by accounting for dependencies among compilation passes and by applying IR code reduction in tandem, reducing such time to less than 10 minutes.

## 2.2 Motivating Example

Figure 3a illustrates a real-world MLIR GitHub issue [6], which resulted in a segmentation fault. The result of matrix multiplication `@matmul` on line 3 is stored in `%2` and checked on line 8. The original compilation path has 9 passes: `convert-linalg-to-loops`, `affine-loop-unroll`, `convert-scf-to-cf`, `convert-arith-to-llvm`, `convert-linalg-to-llvm`, `convert-memref-to-llvm`, `convert-func-to-llvm`, `reconcile-unrealizedcasts`, and `mlir-cpu-runner`.

The crash message's stack dump in Figure 3c indicates that the last used pass is `mlir-cpu-runner`. However, the message does not display the preceding faulty pass `convert-func-to-llvm` responsible for the issue.

For Figure 3a, no further IR reduction is possible with existing delta debuggers such as Perses or Vulcan. However, this minimized IR presents ambiguity. *Is the crash caused by flawed implementation of* `@matmul` *called at line 4? Is the crash due to incorrect transformation of the nested* `for` *loop at lines 5-6?*

DuoReduce performs dual-dimensional fault localization based on two insights:

(1) MLIR compiler bugs are often caused by *operation invocations* rather than by the values of operands. For example, in Figure 3, using `cf.assert` at line 9 with the `convert-func-to-llvm` pass causes the bug, and the value of operand `%4` is irrelevant.
(2) IR code reduction can eliminate unnecessary passes. For example, a loop optimization pass `affine-loop-unroll` becomes unnecessary when `for` loops are removed.

After performing syntax-aware IR reduction, DuoReduce applies *constant replacement* and *return operand rollback* transformations, introduced in Section 3.2. For example, replacing `%4` on line 8 with the constant `0` retains the bug, allowing us to remove related IR using `%3` on line 7 and `@matmul` in line 4. By further removing IR constructs in the nested `for` loop (lines 5-6), DuoReduce eliminates the associated `affine-loop-unroll` pass.

Figure 3b shows the minimized code, reproducing the same error reported in Figure 3a. This example shows that it is possible to exclude the `@matmul` function from the culprit IR code and remove 6 out of 9 passes. It suggests that the underlying error is caused by the 3 remaining passes: `convert-arith-to-llvm`, `convert-func-to-llvm`, or `mlir-cpu-runner`.

## 3 Approach

DuoReduce takes the original IR code $C$, the compilation path $P$, and a separate oracle $O$ to check if the error message remains identical to the original. It outputs the minimized IR code $C\_ans$ and the reduced path $P'$ to reproduce the bug.

At the heart of DuoReduce is a three-fold approach.

(1) **Compilation Path Reduction**: It reduces unnecessary compilation passes by identifying dependencies among the involved passes, as described in Section 3.1.
(2) **IR Code Reduction**: In the IR code dimension, it performs syntax-aware reduction in conjunction with MLIR-specific transformation, as described in Section 3.2.
(3) **Dual-Dimensional Reduction**: As the IR code is reduced, DuoReduce's dual-dimensional reduction approach narrows down necessary compilation passes by analyzing the relation between IR code and passes, as described in Section 3.3.

Algorithm 1 describes DuoReduce's process. In lines 1-7, DuoReduce removes as many unnecessary compilation passes as possible (Section 3.1). Once a minimized compilation path is established, DuoReduce applies IR code reduction on line 8 until no further reduction is possible. Next, DuoReduce applies MLIR-specific transformations on line 11 and applies IR code reduction again on the transformed code on line 14 (Section 3.2). DuoReduce then performs dual-dimensional reduction on the compilation path on line 18 (Section 3.3).

### 3.1 Dependency-Aware Compilation Path Reduction

MLIR compilers have dependencies among compilation passes, as shown in Figure 2. Naively, one may examine all possible pass combinations—$2^n$ combinations where $n$ is the number of passes. However, this approach may lead to a large number of invalid compilation attempts, due to dependencies among passes. In fact, our evaluation showed that exhaustively examining all $2^9$ combinations resulted in 89% invalid compilation attempts for a GitHub issue involving 9 passes [3]. Such exhaustive attempts are inefficient. Therefore, DuoReduce identifies and considers dependencies among passes to avoid invalid compilation attempts.

DuoReduce performs this (1) pass reduction phase before the (2) IR code reduction phase and the (3) joint reduction in both dimensions. Our evaluation shows that, without the initial path reduction (1), it takes 16.7% more time to reach the same IR code reduction rate.

**Algorithm 1** DuoReduce takes as input the original IR program $C$, the original compilation path $P$, and an oracle $O$. DuoReduce first gets the reduced compilation path $P'$ by identifying the dependency of each pass. Then DuoReduce applies delta debugging with MLIR transformations on the IR code $C$. In the end, it applies dual-dimensional reduction on the reduced path $P'$ to get $P\_ans$.

```
Require:
  • $C \leftarrow$ Original IR Program
  • $P \leftarrow$ Original Compilation Path
  • $O \leftarrow$ Oracle that checks whether the bug is preserved
Ensure:
  • $D_i \leftarrow$ Shortest path ending with $P_i$ that can reproduce the bugs with the original $C$
  • $P' \leftarrow$ Shortest path that can reproduce the bugs with the original IR program $C$
  • $C\_ans \leftarrow$ Reduced IR Program
  • $P\_ans \leftarrow$ Shortest compilation path that can reproduce the bugs with $C\_ans$
 1: for $n = N$ downto 1 do
 2:     $D_n$ = DEPEND_GEN($P[1:n]$) // return the shortest path ending with $P_n$.
 3:     if Len($P'$) > Len($D_n$) then
 4:         $P' = D_n$ //Update $P'$ with the shorter path if possible
 5:     end if
 6: end for
 7: $C\_ans \leftarrow$ DD_IR($P', C, O$)
 8: $Flag \leftarrow$ True
 9: while $Flag$ do //Flag is used to check whether the reduction still happens
10:     $C\_cand \leftarrow$ IR_Trans($C\_ans, O$)
11:     $Flag \leftarrow$ False
12:     if Len(DD_IR($P', C\_cand, O$)) < $C\_ans$ then
13:         $C\_ans \leftarrow$ DD_IR($P', C\_cand, O$)
14:         $Flag \leftarrow$ True
15:     end if
16: end while
17: $P\_ans$ = DUO_REDUCTION($P', C\_ans, O$)//Applies dual-dim reduction to further reduce $P'$
18: return $C\_ans, P\_ans$
```

*Problem Definition.* Given a compilation path $P = [P_i, 1 \leq i \leq n]$, DuoReduce identifies the minimum sublist of $P$ as $P' = [P_{i_1}, .., P_{i_m}]$, where $1 \leq i_1 < \ldots < i_m \leq n$, preserving the same bug, meaning (1) $P'$ is a sublist of $P$; (2) the number of passes included in $P'$ is minimal; and (3) for each pass $P_i \in P'$, the passes that $P_i$ depends on (i.e., $P_k \rightharpoonup P_i$) must also be included in $P'$.

$$P' = \underset{P' \subseteq P}{\operatorname{argmin}} |\{P_i \in P' | (P_k \rightharpoonup P_i) \Rightarrow (P_k \in P')\}| \quad (1)$$

*Identifying Pass Dependencies.* Given the original compilation path $[P_1, P_2, \ldots, P_N]$, DuoReduce identifies the dependent passes for each pass in a *backward order* in lines 1-3 of Algorithm 1, in the decreasing order of $P_N$ to $P_1$. We denote the dependencies of $P_n$ as $D_n$, which is initialized to an empty set. DuoReduce checks $P_n$'s preceding passes (*i.e.,* $P_1, ..., P_{n-1}$) and adds the pass to $D_n$, if it is necessary for preserving the bug with a compilation path ending with $P_n$. As shown in line 3 of Algorithm 2, DuoReduce first constructs a set of compilation paths $\{C_1, ..., C_n\}$, where each $C_i$ is the path executing from $P_1$ up to $P_{i-1}$ (i.e., $C_i = [P_1, \ldots, P_{i-1}]$) and is later used to determine whether $P_n$ depends on $P_i$. For example, $C_3 = [P_1, P_2]$. Next, DuoReduce tests whether $P_n$ depends on $P_i$, $i < n$, by constructing a modified path $C_i'$. $C_i'$ concatenates $C_i$ and current $D_n$ and adds $P_n$ at

**Algorithm 2** `DEPEND_GEN`: Dependency-Aware Reduction Algorithm

```
Require:
  • $Code$ ← IR Program
  • $P$ ← A Compilation Path ends with $P_n$, where $P_i$ is each compilation pass, and $P_i$ is execute
    sequentially after $P_{i-1}$ on this path P.
  • $O$ ← Oracle that checks whether the bug is preserved
Ensure:
  • $D_n$ ← A sequence of all the compilation passes that $P_n$ depends on.
  • $C_{shortest}$ ← The shortest compilation path ends with $P_n$ that can satisfy the oracle $O$ with
    $Code$.
1: $D_n$ = []
2: for $i = n - 1$ to 1 do
3:     $C_i = [P_1, \ldots, P_{i-1}]$
4:     $C'_i$ = Concat($C_i$, $D_n$).add($P_n$)
5:     if $O(C'_i, Code)$==false then
6:         $D_n.add(P_i)$
7:     end if
8: end for
9: return $C_{shortest} = D_n.add(P_n)$
```

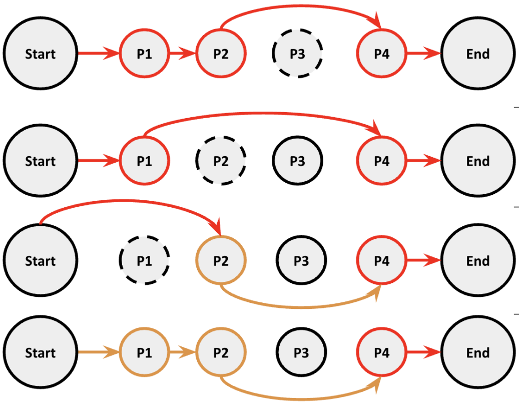
the end, as shown in line 4 of Algorithm 2. In this way, $C'_i$ reserves all passes that $P_n$ depend on, except for $P_i$. If $C'_i$ fails to retain the same behavior, it indicates that $P_n$ depends on $P_i$, and $P_i$ is added to $D_n$, as shown in lines 5-7 of Algorithm 2. Otherwise, we can safely remove the $P_i$.

Figure 4 shows a running example. We start with finding the dependency $D_4$ for $P_4$, where $D_4$ is initially an empty list []. For that, DuoReduce incrementally constructs this list by testing the removal of preceding passes P3 to P1. First, DuoReduce checks if $P_4$ depends on $P_3$ by removing it from the compilation passes. Thus, DuoReduce executes $[P_1, P_2, P_4]$. $[P_1, P_2, P_4]$ is constructed by concatenating $C_3 = [P_1, P_2]$ and $D_4 = []$, and adding the target pass $P_4$ together. Because it retains the same bug, $P_4$ does not depend on $P_3$ and $P_3$ can be safely removed. Next, DuoReduce checks if $P_4$ depends on $P_2$ by executing $[P1, P4]$, which is constructed by concatenating $C_2 = [P_1]$ and $D_4 = []$, and adding the target pass $P_4$. However, it does not reproduce the same bug; thus, DuoReduce adds $P_2$ into $D_4$. DuoReduce then checks the dependency against $P_1$ by removing $P_1$ and executing $[P_2, P_4]$, which is constructed by concatenating $C_1 = []$ and $D_4 = [P_2]$, and adding $P_4$. If the same bug is not reproduced, DuoReduce adds $P_1$ to $D_4$. Finally, DuoReduce identifies the passes that $P_4$ depends on as $D_4 = [P_1, P_2]$, and constructs the shortest path that ends with $P_4$, $[P_1, P_2, P_4]$. This algorithm has the time complexity of $O(n^2)$, where $n$ is the total number of compilation passes.

### 3.2 Transformation-Based Code Reduction

Prior work found that applying program transformations can improve delta debugging [23, 33, 34, 42]. In MLIR, we observe that compilation crashes are often caused by *operation invocations* rather than the values of operands. We thus hypothesize that altering MLIR operands can offer new opportunities for further IR code reduction. We design two operand-specific transformations—*constant replacement* and *return operand rollback*. If altering an operand successfully retains the bug behavior, we can eliminate all dependent operands.

***1. Constant Replacement.*** The *Constant Replacement* transformation substitutes operands in the IR code with constants of the same type. In Figure 5a, a GitHub issue [7] causes a compiler crash occurs due to the invocations of `vector.broadcast` for the operand `%111` at line 9 and `vector.fma`

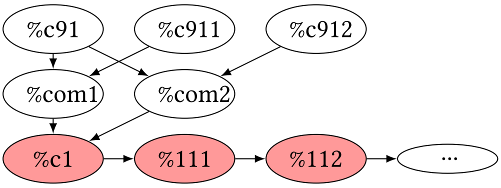


| Path to try | Our finding |
|---|---|
| Try to skip P3;<br>Run P1→P2→P4 | P3 can be discard |
| Try to skip P2;<br>Run P1→P4 | P2 cannot be discard |
| Try to skip P1;<br>Run P2→P4 | P1 and P2 cannot be discard |
| Find shortest path ends with P4: P1→P2→P4 | |

Fig. 4. Red edges indicate the current passes being tested, and the golden vertices highlight dependent passes. DuoReduce finds the shortest path $P$ from **Start** to **P4**. P4=`export-verilog` does not depend on P3=`-inline` but depends on P1=`lower-hw-to-sv` and P2=`lower-calyx-to-hw`.

```
1   func.func @func1(%arg0: tensor<5x5xf16>, %arg1:
          tensor<?x12xi16>, %arg2: vector<5x5xi1>) ->
          i1 {
2     %c91 = arith.constant 91 : index
3     %c911 = arith.constant 911 : index
4     %c912 = arith.constant 912 : index
5     %com1 = index.divs %c911, %c91
6     %com2 = index.divs %c912, %c91
7     %c1 = index.divs %com1, %com2
8    //Replace with "%c1 = arith.constant 42 : index"
           still trigger the crash
9     %111 = vector.broadcast %c1 : index to
            vector<21x12x12xindex>
10    %112 = vector.fma %111, %111, %111 :
            vector<21x12x12xf32>
11  //Insert "return %112 : vector<21x12x12xf32>" +
          replace the function return type to
          "vector" still trigger the crash
12    vector.print %112 : vector<21x12x12xf32>
13    %ans = arith.constant true
14    return %ans : i1}
```

(a) This IR code crashes LLVM due to the invocation of `vector.broadcast` at line 9 and `vector.fma` at line 10, with pass `vector-unrolling`.



(b) Dataflow graph for (a): DuoReduce applies constant replacement for %c1 and return operand rollback for %112. It then removes the operands that %c1 depends on such as %com1, and all the operands after %112 such as %ans.

```
func.func @func1(...) -> vector<21x12x12xf32> {
  %c1 = arith.constant 42 : index
  %111 = vector.broadcast %c1 : index to
         vector<21x12x12xindex>
  %112 = vector.fma %111, %111, %111 :
         vector<21x12x12xf32>
  return %112 : vector<21x12x12xf32>}
```

(c) Constant replacement on %c1 and return operand rollback on %112 enabled removal of lines 2-7 and lines 12-13 in (a).

Fig. 5. Code example from MLIR GitHub issue 64074 [7]. DuoReduce applies program transformation on the 1-minimal IR code and enables a further 75% reduction.

for the operand %112 at line 10. The data flow graph in Figure 5b shows that the code is already 1-minimal. DuoReduce replaces the operand %c1 on line 7 with a random constant 42 of the same type `index` on line 8. Because the bug is induced by the operation `vector.broadcast` rather than the specific value of %c1, replacing line 7 with line 8 triggers the same crash.

DuoReduce then begins another DD cycle. All preceding operands before %c1 are eliminated because %c1 is a constant. DuoReduce safely removes all operands that lead to the previous %c1

```
...
//IR Dump Before AffineLoopUnroll(affine-loop-unroll)
module {
  func.func @main() {
  ...
  %4 = arith.constant false
  cf.assert %4, "Matmul does not"}}
  ...
//IR Dump Before ArithToLLVMConversionPass(convert-arith-to-llvm)
module {
  func.func @main() {
  ...
  %4 = arith.constant false
  cf.assert %4, "Matmul does not"}}
  ...
//IR Dump Before ConvertFuncToLLVMPass(convert-func-to-llvm)
module {
  func.func @main() {
  ...
  %4 = llvm.mlir.constant(false) : i1
  cf.assert %4, "Matmul does not"}}
```

Fig. 6. After removing `@matmul` in Figure 3, the pass `affine-loop-unroll` is no longer relevant. DuoReduce removes 4 out of 7 passes, and reduces the inspection scope from 2279 to 1245 lines.

and that are not used later—operands `%c91`, `%c911`, `%c912`, `%com1`, and `%com2`. This removes lines 2 through 6, as shown in Figure 5c.

***2. Return Operand Rollback.*** The *Return Operand Rollback* transformation returns intermediate operands instead of the last operand. DuoReduce updates the return statement and the return type of the function. In Figure 5, instead of returning `%ans` at line 14, DuoReduce inserts a `return` statement after each operand to determine if the bug persists. For example, DuoReduce produces a transformation to return at `%112` and updates the return type from `i1` at line 1 to the type of operand `%112`, which is type `vector`. After returning `%112`, DuoReduce safely removes all subsequent operands because they are no longer invoked, thus removing lines 12-13.

### 3.3 IR-Path Dual-Dimensional Reduction

The existing MLIR framework provides the ability to display the impacts of each compilation pass on the IR code through the `mlir-print-ir-before-all` option. When we compile Figure 6 with this option and the compilation passes, (`affine-loop-unroll`, `convert-arith-to-llvm`, `convert-func-to-llvm`), MLIR prints the IR code before each pass. By comparing the IR code before and after each pass, DuoReduce identifies which passes are no longer necessary. In this example, the IR remains unchanged after the `affine-loop-unroll` pass but is altered following the `convert-func-to-llvm` pass. It indicates that the latter pass has an effect on the IR code, while the former does not.

For example, in Figure 3a, the `for` loops in the original IR code necessitate the `affine-loop-unroll` pass. However, after the IR code is reduced to Figure 3b which no longer contains these loops, the `loop-unrolling` pass no longer has an effect. Similarly, this approach effectively eliminates the passes `convert-linalg-to-loops`, `convert-scf-to-cf`, `convert-linalg-to-llvm`, `convert-memref-to-llvm`, and `affine-loop-unroll`, resulting in 63% pass reduction. This in turn reduces the inspection scope from 2279 lines of compiler code to 1245 lines.

### 3.4 Overall Time Complexity

DuoReduce operates in three phases: Phase 1 performs compilation path reduction, Phase 2 focuses on IR code reduction, which involves IR code transformation, and Phase 3 applies dual-dimension reduction on the compilation path.

Phase 1 has a time complexity of $O(p^2)$, where $p$ represents the number of compilation passes. Phase 2 consists of two key components: IR transformation and syntax-guided delta debugging reduction. Let $n$ denote the number of operands in the IR code. In the worst-case scenario, delta debugging reaches 1-minimal $n$ times, with each time the IR transformation enabling further reduction. For IR transformation, both constant replacement and return operand rollback traverse each operand once, leading to a time complexity of $O(n)$. Since IR transformation may be applied $n$ times in the worst case, the total cost accumulates to $O(n^2)$. For delta-debugging, since we reach 1-minimal $n$ times, it means we apply delta-debugging $n$ times. While the time cost of each delta-debugging is linear [43], the overall cost for syntax-guided delta debugging is $O(n^2)$. Thus, the total time complexity for phase 2 is $O(n^2) + O(n^2) = O(n^2)$. Phase 3 iterates over each compilation pass and verifies its results, leading to a time complexity of $O(p)$.

Summing the complexities of all three phases, the total time complexity for DuoReduce is $O(p^2) + O(n^2) + O(p) = O(n^2) + O(p^2)$, where $p$ is the number of compilation passes and $n$ is the number of operands in code.

## 4 Evaluation

DuoReduce has three main components: MLIR code transformation, dual-dimensional reduction, and dependency-aware pass reduction. We examine the following research questions to answer the effectiveness of each component:

**RQ1:** How effective is DuoReduce in terms of IR code reduction? How much reduction can be achieved by DuoReduce's MLIR code transformation?

**RQ2:** How effective is DuoReduce in terms of compilation pass reduction? How much reduction can be achieved by DuoReduce's dual-dimensional reduction?

**RQ3:** How much speedup can DuoReduce achieve with dual-dimensional reduction?

### 4.1 Experiment Design

We select three large MLIR compiler projects, listed in Table 1. The systems used in our evaluation, including MLIR [15], CIRCT [14], and ONNX-MLIR [22], are significantly-sized real-world projects (444k, 171k, and 96k LOC for MLIR, CIRCT, and ONNX-MLIR, respectively) with 295 active contributors among recent papers on MLIR testing [41]. We examined all issues related to these systems and filtered out 62 out of 134 issues which lacked MLIR code inputs or corresponding compilation passes required to reproduce the reported bugs. The GitHub issues are chosen for the latest version of each project. Out of the remaining 72 issues, we removed 41 issues that did not require further IR code reduction because the provided IR was already less than 10 lines of code. In the end, we selected 31 reproducible issues for our evaluation, which include all reproducible, non-trivial bugs from 3 widely-used MLIR projects on GitHub: onnx-mlir, CIRCT, and MLIR-core. The detailed benchmarks are provided in the replication package with the GitHub issue IDs and links.

For each IR input reported for a given compiler bug, we determined the super-set of candidate passes based on two criteria: (1) the structure of the IR program, and (2) the dialects used. Most GitHub issues provide an initial set of compilation passes that MLIR developers use to reproduce the reported bugs. These passes serve as the starting point for our delta-debugging. In cases where such path is not explicitly provided, we use the names of dialects used in the IR program and map to a corresponding set of compilation passes. For instance, if an IR program contains loops and vector operations, we assume that vector unrolling passes such as *-test-vector-unrolling-patterns=unroll-based-on-type* should be included in the scope. Similarly, if the IR program uses the `arith` dialect, lowering passes such as *-convert-arith-to-llvm* should be included. Column **Bug LoP** in Table 1 shows the number of candidate passes for the reported bugs.

Table 1. The systems used in our evaluation are significantly-sized real-world projects (444k, 171k, and 96k LOC for MLIR, CIRCT, and ONNX-MLIR, respectively). They define a large number of compilation passes: 234, 206, and 34 respectively. On the left, BugLoC stands for lines of IR code and BugLoP stands for the number of compilation passes for each bug on average.

| Project | LoC | # Compilation Pass | Commit Message | # GitHub Issues | Bug LoC | Bug LoP |
|---|---|---|---|---|---|---|
| **MLIR [15]** | 444k | 234 | 1730 | 23 | 132.3 | 17.9 |
| **CIRCT [14]** | 171k | 206 | 2051 | 6 | 32.1 | 20.2 |
| **ONNX-mlir [22]** | 96k | 34 | 2915 | 2 | 23.2 | 17 |

**Evaluation metrics.** The following metrics are typically used in previous work on DD [20, 42, 45].

- Average reduction: reduction in IR code size and the number of compilation passes. It is calculated as $\frac{|x|-|x'|}{|x|}$, where $|x|$ and $|x'|$ are the original and reduced IR code size or the length of the compilation path.
- Successful reduction: the number of cases in which buggy compilation passes and IR code segments have been successfully localized by the debugging tool in a 4-hour time limit.
- Time usage: time in seconds that each tool takes.

**Baselines.** To answer RQ1, we evaluate DuoReduce against Perses+ [8], Vulcan+ [42], `mlir-reduce+` (`circt-reduce+` for the CIRCT bug issue), and DuoReduce_NoTran (DuoReduce without MLIR code transformations).

Perses is a syntax-aware delta debugger that leverages the ANTLR grammar and prunes the search space by avoiding generating syntactically invalid programs according to the grammar. We configured Perses to use the base MLIR grammar [26]. We then construct **Perses+** by adding DuoReduce's compilation path reduction, since Perses does not account for the dimension of compilation passes at all and it would be unfair to compare DuoReduce against Perses directly. In short, **Perses+** = Perses + DuoReduce's dependency-aware compilation path reduction + DuoReduce's dual-dimensional reduction.

Vulcan, an enhanced version of Perses, introduces additional code transformation rules, such as identifier and subtree replacement, to achieve further reduction. These general transformation rules are not designed for IR code transformations and are thus less effective than DuoReduce. Similar to Perses+, we configured Vulcan with the base MLIR grammar [26]. Vulcan+ adds compilation path reduction to the original Vulcan algorithm: **Vulcan+** = Vulcan + DuoReduce's dependency-aware compilation path reduction + DuoReduce's dual-dimensional reduction. In short, **Vulcan+** replace DuoReduce's MLIR transformation with Vulcan's general code transformations.

`mlir-reduce` is a utility provided by the MLIR compiler community to reduce the size of the IR code, and `circt-reduce` [9] is built on top of `mlir-reduce` by considering CIRCT dialects. These domain-specific reducers employ multiple strategies to minimize the input: directly removing operations (akin to Perses), and applying optimization rewrites [15]. However, since the optimization rewrites are designed for compilation, not reduction, they are conservative in terms of reducing IR inputs. Similarly, we construct **mlir-reduce+** = `mlir-reduce` + DuoReduce's dependency-aware compilation path reduction + DuoReduce's dual-dimensional reduction.

To answer RQ2, we compare DuoReduce against its downgraded version DuoReduce_No2Dim without dual-dimension reduction. We demonstrate why the LLVM compiler crash message alone cannot provide correct information to help developers localize the buggy pass. To answer RQ3, we evaluate DuoReduce against its downgraded version DuoReduce_NoDep, DuoReduce without identifying compilation pass dependences. DuoReduce_NoDep exhaustively tries all pass combinations, instead of considering the dependences among passes.

Table 2. Effectiveness and Efficiency for DuoReduce. DuoReduce achieves the highest IR code reduction compared to Perses+ and Vulcan+. MLIR transformations enable 31.6% additional IR size reduction. DuoReduce_NoDep cannot finish the reduction for 29 out of 31 GitHub issues in the 4-hour time limit, since it requires $2^n$ trials, where $n$ is the number of compilation passes.

| ID | Tool Name | IR Code | | Compilation Path | | Time Usage |
|---|---|---|---|---|---|---|
| | | Successes Reduction | Average Reduction | Successful Reduction | Average Reduction | |
| **1** | mlir-reduce+ | 27 | 26.1% | 27 | 84.2% | 530.65s |
| **2** | Perses+ | 31 | 47.2% | 31 | 87.2% | 414.37s |
| **3** | Vulcan+ | 31 | 51.1% | 31 | 89.3% | 1336.55s |
| **4** | DuoReduce_NoTran | 31 | 47.2% | 31 | 87.2% | 397.45s |
| **5** | DuoReduce_NoDep | 2 | / | 2 | / | ~ 145h |
| **6** | DuoReduce_No2Dim | 31 | 62.1% | 31 | 77.1% | 570.77s |
| **7** | Compiler Crash Msg | / | / | 15 | 94.4% | N/A |
| **8** | DuoReduce | **31** | **62.1%** | **31** | **91.7%** | **572.35s** |

```
1   module {
2     func.func @func2(...) {
3       %c11 = arith.constant 11 : index
4       %c12 = arith.constant 12 : index
5       %com1 = index.divs %c11, %c11
6       %com2 = index.divs %c12, %c12
7       %c2 = index.divs %com1, %com2
8       ...
9       %92 = affine.apply affine_map<(d0, d1, d2,
            d3) -> (d0 - 16)>(%46, %28, %c2, %43)
10      scf.index_switch %92
11      default {....}}
```

(a) The 1-minimal code example: the compilation crash is due to `scf.index_switch`.

```
1   module {
2     func.func @func2(%arg0: tensor<?x?x?xi1>,
          %arg1: tensor<5x5xi32>) {
3       %c2 = arith.constant 2 : index
4       ...
5       %92 = affine.apply affine_map<(d0, d1, d2,
            d3) -> (d0 - 16)>(%46, %28, %c2, %43)
6       scf.index_switch %92
7       default {....}}
```

(b) DuoReduce applies constant replacement for %c2, enabling further removal of all the operands %c2 depends on.

Fig. 7. MLIR GitHub issue 64071 [10]. DuoReduce achieves 4 more lines of code reduction with the constant replacement compared to Perses+ and Vulcan+, and only takes 617.4 seconds compared to Vulcan+ which takes 2604 seconds, resulting in 3.87× speedup.

**Experimental environment.** All experiments are performed on a machine with an AMD Ryzen 2950X 16-Core Processor with 32 GB RAM running on Ubuntu 22.04.

### 4.2 RQ1: Effectiveness of MLIR Code Transformations

As shown in Table 2, `mlir-reduce+` has the worst performance, since it applies the naive ddmin [45] approach. DuoReduce outperforms `mlir-reduce+`, Perses+/DuoReduce_NoTran, and Vulcan+ by 138%, 31.6%, and 21.5%, respectively, in terms of the IR reduction rate. Although DuoReduce shares the same compilation pass reduction with the above baselines, the improved IR code reduction leads to a better compilation pass reduction rate. As shown in the column **Compilation Path - Average Reduction**, compared to `mlir-reduce+`, Perses+/DuoReduce_NoTran, and Vulcan+, DuoReduce achieves 7.5%, 4.5%, and 2.4% higher reduction rate respectively.

Figure 7 shows an example where DuoReduce outperforms Perses+ and Vulcan+. The compilation crash is caused by the logic in `scf.index_switch` on line 10 in Figure 7a, which works like the `case` statement in C/C++. Perses+ and `mlir-reduce+` reach 1-minimal in Figure 7a; however, the code can be further reduced if we replace the operand `%c2` on line 7 with a constant, since

```
func.func @main() {
  %in_buf = memref.alloc() :
        memref<16x230x230x3xf32>
  %filter_buf = memref.alloc() :
        memref<64x7x7x3xf32>
  %out_buf = memref.alloc() :
        memref<16x112x112x64xf32>
  linalg.conv_2d_nhwc_fhwc {dilations = dense<1>
        : tensor<2xi64>, strides = dense<2> :
        tensor<2xi64>}
      ins (%in_buf, %filter_buf:
            memref<16x230x230x3xf32>,
            memref<64x7x7x3xf32>)
      outs (%out_buf: memref<16x112x112x64xf32>)
  return}
```

(a) The above code example crashes because of the loop tiling. It takes two tensors as input and computes the 2D convolution.

```
Stack dump:
0. Program arguments: mlir-opt
      --convert-linalg-to-affine-loops
      --affine-loop-tile=tile-sizes=4,28,28,..
      --affine-loop-unroll=unroll-factor=4
      --canonicalize --affine-parallelize
      --lower-affine --canonicalize
      --gpu-map-parallel-loops
      --convert-parallel-loops-to-gpu conv2d.mlir
...
#22 0x00005c544cb547a3 processParallelLoop...
...
#25 0x00005c544cb53b93 ... const
      /home/mlir/lib/Conversion/SCFToGPU/
      SCFToGPU.cpp:642:11
```

(b) The compiler crash message localizes the crash to the wrong pass: `convert-parallel-loops-to-gpu`.

Fig. 8. MLIR GitHub issue 82382 [3]. DuoReduce finds the buggy pass `affine-loop-tile` in the results while the compiler crash message doesn't, which motivates the need for DuoReduce.

the crash is due to the invocation of `scf.index_switch` instead of the value of %c2. DuoReduce replaces the operands %c2 with a random constant 2 and further reduces the IR code to Figure 7b.

Compared to Perses+, DuoReduce takes 38.1% more time to finish the entire DD process on average, as shown in the **Time Usage** column of Table 2. It is because after reaching the fixed point, DuoReduce applies IR program transformations and enables further reduction that Perses+ cannot reach. However, by using these transformations, DuoReduce achieves a 14.9% higher reduction rate and outperforms Perses+ by 31.6%.

Although Vulcan+ also applies code transformations, its general code transformations are not suitable for MLIR and thus less effective. For example, in Figure 7a, Vulcan+ cannot eliminate the logic dependency of the operand %c2, which prevents it from removing the code on lines 3-6. Vulcan+'s identifier replacement technique, which substitutes one identifier with another, fails to break the logic dependency on the operand on line 7. Similarly, its subtree replacement strategy, which shares the same logic as identifier replacement, cannot address this dependency either. Additionally, the time complexity of Vulcan+'s transformations is $O(n^2)$ [42], which is significantly higher than DuoReduce's transformation complexity of $O(n)$, as discussed in Section 3. For instance, in Figure 7, Vulcan+ takes 2604 seconds to complete the transformation, whereas DuoReduce only requires 617.4 seconds. As indicated in the **Time Usage** column of Table 2, DuoReduce consistently outperforms Vulcan+, reducing the time required from 1336.55 seconds to 572.35 seconds on average. DuoReduce also achieves an 11.0% higher reduction rate and outperforms Vulcan+ by 21.5%.

> With IR code transformations, DuoReduce outperforms the state-of-the-art DD methods, Perses and Vulcan, in terms of IR code reduction by 31.6% and 21.5%, respectively.

### 4.3 RQ2: Effectiveness of Dual-Dimensional Reduction

We evaluate DuoReduce against its downgraded version DuoReduce_No2Dim and LLVM compiler crash message to show the benefit of dual-dimensional reduction.

As shown in Table 2, comparing row **6** and row **8** shows that DuoReduce achieved 14.6% higher pass reduction while only taking 1.58 seconds longer than DuoReduce_No2Dim. LLVM compiler

```
1   module {
2     func.func @omp_target() {
3       %alloca = memref.alloca() : memref<64x64xf64>
4       ...
5       %0 = omp.map_info var_ptr(%alloca :
              memref<64x64xf64>, tensor<?xi32>)
              map_clauses(to) capture(ByRef) ->
              memref<64x64xf64>
6       ...
7       omp.parallel {
8         ...
9         %2 = vector.load %arg0[%arg2, %arg3] :
            memref<64x64xf64>, vector<16xf64>
10        ...}}
11      return}}
```

(a) This code example crashes with a compilation path of 4 passes. This requires the developer to inspect 2423 lines of MLIR compiler code for debugging.

```
module {
  func.func @omp_target() {
    %alloca = memref.alloca() : memref<64x64xf64>
    %0 = omp.map_info var_ptr(%alloca :
          memref<64x64xf64>, tensor<?xi32>)
          map_clauses(to) capture(ByRef) ->
          memref<64x64xf64>
    return}}
```

(b) This reduced IR code enables DuoReduce to reduce the compilation path from 4 passes: `convert-vector-to-llvm`, `finalize-memref-to-llvm`, `convert-arith-to-llvm`, and `convert-openmp-to-llvm`, to a single pass: `convert-openmp-to-llvm`, and only requires the developer to inspect 284 lines of MLIR compiler code for debugging.

Fig. 9. MLIR GitHub issue 76579 [11]. With dual-dimensional reduction, DuoReduce removes 3 redundant passes and achieves better reduction than DuoReduce_No2Dim, which translates to not needing to inspect 2139 lines of MLIR compiler code.

crash messages cannot handle non-crash bugs such as incorrect code generated for CIRCT GitHub issue 6317 [5] in Figure 2. In total, LLVM fails to localize correct buggy passes for 52% GitHub issues.

Take Figure 8 as an example. The original code in Figure 8a transforms a 2-dimensional tensor using `linalg.conv_2d` on line 5 and stores the result in `out_buf` on line 7. LLVM reports a crash in pass `convert-parallel-loops-to-gpu`, as shown at #22 in Figure 8b. However, the crash is actually triggered by the parameter chosen in pass `affine-loop-tile`. Unlike LLVM crash report, DuoReduce localizes all four buggy passes: `convert-linalg-to-affine-loops`, `affine-loop-tile=...`, `affine-loop-unroll`, `gpu-map-parallel-loops`.

Some compilation passes are not related to the crash but cannot be removed since they are required to compile the parts of the code that are also unrelated to the crash. With dual-dimensional reduction, DuoReduce may remove both the code and passes unrelated to the crash. For example, in Figure 9a, to successfully compile the code, the vector at line 9 requires the pass `convert-vector-to-llvm` to lower the `vector` to `llvm` dialect. However, after removing line 9, which is unrelated to the crash, DuoReduce is able to safely remove the pass `convert-vector-to-llvm` while the crash remains. In summary, compared to DuoReduce_No2Dim, DuoReduce achieves an additional 75% reduction of compilation passes. This eliminates 2,139 lines of unrelated compiler code for the developer to examine while debugging the crash. On average, DuoReduce achieves a 91.7% reduction rate compared to DuoReduce_No2Dim with 77.1%.

> Compiler crash messages cannot correctly isolate culprit passes in 16 out of 31 GitHub issues. DuoReduce outperforms DuoReduce_No2Dim and achieves a higher pass reduction of 14.6%, translating to not needing to examine 281 lines of MLIR compiler code on average.

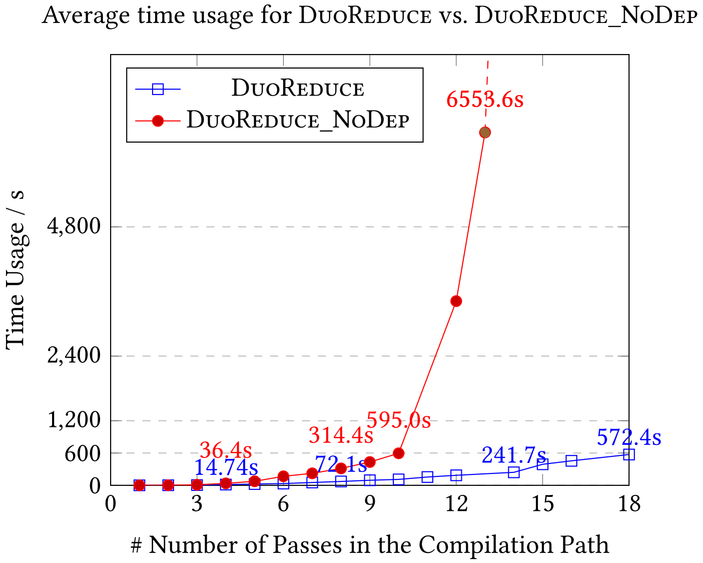


Fig. 10. Time usage for DuoReduce and DuoReduce_NoDep. The x-axis represents the number of passes, and the y-axis represents the average time usage. DuoReduce achieves higher speedup when more compilation passes are involved. For example, for 13 compilation passes, DuoReduce takes 241.7s, while DuoReduce_NoDep takes 6553.6s, resulting in 27× speedup.

### 4.4 RQ3: Effectiveness of Compilation Pass Dependence-Aware Reduction

As shown in Table 2 row **5** and row **8**, DuoReduce_NoDep cannot finish the compilation pass reduction for 29 out of 31 GitHub issues in the 4-hour time limit, while DuoReduce finishes all the tasks with an average time of 572.35 seconds. In our evaluation, DuoReduce_NoDep cannot finish the reduction tasks in the 4-hour time limit when the number of compilation passes reaches 14.

Take the compilation path from GitHub issues [3] in Figure 8 as an example. The original compilation path has 9 passes, as listed in the stack dump after the *Program arguments*. DuoReduce first figures out the dependency relation in the compilation path. For example, `affine-loop-unroll` depends on `covert-linalg-to-affine-loops` and `affine-loop-tile=...`. With DuoReduce's dependency-aware reduction, DuoReduce achieves the same reduction rate with 101 compilation trials, compared to DuoReduce_NoDep with $2^9$=512 compilation trails, resulting in a 5.1× speedup.

We compare DuoReduce against DuoReduce_NoDep to examine DuoReduce's scalability as the number of compilation passes grows. Figure 10 shows that DuoReduce's efficiency escalates with the increase in the number of compilation passes. This improvement is because more passes lead to a denser web of dependencies among the passes. With the compilation pass dependency, DuoReduce effectively reduces the number of unnecessary compilation trials. For example, for the GitHub issues with 4 passes like the example in Figure 9, DuoReduce requires 14.7 seconds on average to complete the debugging, compared to DuoReduce_NoDep, which takes 36.4 seconds, resulting in a 2.47× speedup. The efficiency gains are more pronounced with 13 passes, where DuoReduce averages 241.7 seconds, outperforming DuoReduce_NoDep's 6553.6 seconds, thereby achieving a 27× speedup. DuoReduce_NoDep cannot finish the delta debugging process within the time limit (4 hours) when the number of compilation passes reaches 14, while DuoReduce finished its process for 18 compilation passes in 572.4 seconds.

> DuoReduce finishes compilation path reduction with an average time of 572.35 seconds, while DuoReduce_NoDep requires an estimated 145 hours, achieving a 901× speedup. The result shows the effectiveness of compilation pass dependency aware reduction.

## 5 Related Work

**Delta Debugging.** Delta debugging is a seminal work for input reduction to identify the minimal failure-inducing changes between two program versions [44, 45]. Based on *ddmin*, Ghassan et al. developed Hierarchical Delta Debugging (HDD) [32]. It applies delta debugging at each level of a program's input, working from the coarsest to the finest levels. Recent work proposed several enhancements on HDD [20, 21, 24]. ProbDD [40] is a probabilistic DD algorithm that learns from testing history to select elements based on the probabilities. RCC [37] uses ZIP and SHA to compress the generated variants to speed up program reduction. Perses [36] ensures that each reduction step considers only syntactically valid variants to avoid futile effort on syntactically invalid variants, and was later extended to specific domains [38, 47]. None of these reducers can handle bug isolation across two dimensions: code and compilation passes. Existing DD is inefficient for today's extensible multi-layer compilation and can take up to 145 hours to isolate culprit compilation passes along with the minimized IR program. DuoReduce achieves 14.6% more reduction with this dual-dimension isolation.

Many delta debugging tools consider the underlying language features and apply various program transformations to decompose an input program into fine-grained units [16]. Vulcan [42] applies general code program transformations including identifier and subtree replacement. LPR [46] combines LLMs and language-generic reduction tools to refine the results of program reduction. C-Reduce [34] tackles this problem by leveraging domain-specific program transformations to reduce C/C++ programs. DDSMT [33] serve as the domain-specific reducers for SMT-LIBv2. CHISEL [19] uses reinforcement learning to select steps in *ddmin* that are more likely to satisfy the target oracle. The MLIR and CIRCT projects also develop their own debugger utilities [14, 15]. DuoReduce is the first to reason about compilation pass dependencies in tandem with IR code reduction.

J-Reduce [23], a domain-specific reducer for Java, performs code isolation by considering call dependency and field access information among connected code components. DuoReduce's dependency-aware compilation path reduction is different from JReduce's dependency-aware code reduction. In DuoReduce, we determine whether one compilation pass requires the presence of another compilation pass before proceeding with IR code isolation. In essence, DuoReduce is a dual-dimension debugging over compilation pass and code isolation, while JReduce is a single-dimension debugging over code isolation.

**Multi-Layer Compiler Testing and Debugging.** MLIR developers enable debugging with compiler users and developers with `mlir-reduce` and the Action framework [15]. Qingchao et al. [35] present a systematic study of DL compiler bugs. Nikolaos et al. [30] develop a debugging and repair tool on top of TVM for DL framework conversions. It works on DNN transformation and detects faults introduced in model parameters and the model graph. None of these leverages the information gained from the IR code to help the localization on the compilation pass dimension.

Multi-layer compiler testing, for example, testing for TVM [13] and MLIR, is a hot domain. Compiler testing involves applying program transformation and mutation to generate test cases. MLIRSmith [41] generates random MLIR programs. Unlike grammar-based fuzzers, it encodes domain-specific constraints that are difficult to encode in CFG. Ma et al. [31] propose HirGen, which generates computation graphs represented by IR for testing TVM. CLSmith [25] is a grammar-based fuzzer for OpenCL. NNSmith [27] and Neuri [28] generate computation graphs for testing deep learning compilers. Tzer [29] is a fuzzer for the TVM tensor compiler. Like DuoReduce, Tzer mutates both the TVM IR and the compilation passes. However, Tzer does not consider dependencies among passes. While the above work focuses on a test generation problem, DuoReduce targets a debugging problem instead—test input reduction.

## 6 Conclusion

Multi-Level Intermediate Representation (MLIR) is increasingly gaining prominence in compiler development in the domain of machine learning, high-performance computing, and embedded systems. By serving as a bridge between high-level languages and low-level machine code, MLIR facilitates optimizations across multiple layers of abstraction. Some MLIR compilers have over 200 compilation passes available. As of September 2024, the top 3 MLIR projects Triton, CIRCT, and tensorflow-mlir have 258, 206, and 364 compilation passes available. Due to the complexity of multi-pass compilation, debugging extensible compilers is challenging.

We present DuoReduce, a novel dual-dimensional debugging approach designed for such extensible compiler development. Its innovation centers around handling dependencies among compilation passes to streamline the debugging search space and utilizing MLIR program transformation to further decompose the IR code into fine-granular units. Demonstrated through experiments on three large MLIR projects, DuoReduce significantly outperforms all seven baselines. Compared to Perses and Vulcan, it improves the IR reduction rate by 31.6% and 21.5% respectively. While none of the baselines isolate culprit passes, DuoReduce achieves an estimated 901× speedup with dependency-aware compilation pass reduction. The experiment results prove DuoReduce's effectiveness in debugging multi-layer extensible compilers, showing significant potential to reduce the development cost of extensible optimizing compilers.

## 7 Data Availability

Per the open science policy, we make DuoReduce's artifacts, benchmark programs, and datasets available at https://doi.org/10.5281/zenodo.13751881.

## 8 Acknowledgement

We would like to thank the anonymous reviewers for their valuable feedback. We thank Yaoxuan Wu for his kind help in picturing the algorithm for compilation path reduction. The participants of this research are in part supported by NSF grants CNS 2106838, CCF 2106404, CCF 2426162, CCF 2426161, Cisco gift funding, UCR Senate Awards, Amazon Science gift funding, and Samsung contract.

## References

[1] 2024. https://mlir.llvm.org/.
[2] 2024. https://circt.llvm.org/docs/PyCDE/basics/.
[3] 2024. https://github.com/llvm/llvm-project/issues/82382.
[4] 2024. https://mlir.llvm.org/docs/Tools/mlir-reduce/.
[5] 2024. https://github.com/llvm/circt/issues/6317.
[6] 2024. https://github.com/llvm/llvm-project/issues/56914.
[7] 2024. https://github.com/llvm/llvm-project/issues/64074.
[8] 2024. https://github.com/uw-pluverse/perses.
[9] 2024. https://github.com/llvm/circt/tree/main/test/circt-reduce.
[10] 2024. https://github.com/llvm/llvm-project/issues/64071.
[11] 2024. https://github.com/llvm/llvm-project/issues/76579.
[12] Nick Brown, Maurice Jamieson, Anton Lydike, Emilien Bauer, and Tobias Grosser. 2023. Fortran performance optimisation and auto-parallelisation by leveraging MLIR-based domain specific abstractions in Flang. In Proceedings

of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. 904–913.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 578–594.

[14] CIRCT Contributors. 2023. CIRCT: Circuit IR Compilers and Tools. https://github.com/llvm/circt

[15] LLVM Contributors. 2023. MLIR Language Reference. https://mlir.llvm.org/docs/LangRef/

[16] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 1017–1032.

[17] Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. 2022. IRDL: an IR definition language for SSA compilers. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 199–212.

[18] Juliana Freire, D Koop, Emanuele Santos, Carlos Scheidegger, Cláudio Silva, and Vo Huy. 2011. The Architecture of Open Source Applications. Vol. 1. Chapter LLVM.

[19] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 380–394.

[20] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation. 31–37.

[21] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse hierarchical delta debugging. In 2017 IEEE international conference on software maintenance and evolution (ICSME). IEEE, 194–203.

[22] Tian Jin, Gheorghe-Teodor Bercea, Tung D Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, et al. 2020. Compiling onnx neural network models using mlir. arXiv preprint arXiv:2008.08272 (2020).

[23] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 556–566.

[24] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical delta debugging algorithm. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation. 16–22.

[25] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. ACM SIGPLAN Notices 50, 6 (2015), 65–76.

[26] Ben Limpanukorn, Jiyuan Wang, Hong Jin Kang, Eric Zitong Zhou, and Miryung Kim. 2024. Fuzzing MLIR by Synthesizing Custom Mutations. arXiv preprint arXiv:2404.16947 (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 Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 530–543.

[28] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. 2023. Neuri: Diversifying dnn generation via inductive rule inference. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 657–669.

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

[30] Nikolaos Louloudakis, Perry Gibson, José Cano, and Ajitha Rajan. 2023. Fault Localization for Buggy Deep Learning Framework Conversions in Image Recognition. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 1795–1799.

[31] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing deep learning compilers with hirgen. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 248–260.

[32] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In Proceedings of the 28th international conference on Software engineering. 142–151.

[33] Aina Niemetz and Armin Biere. 2013. ddSMT: a delta debugger for the SMT-LIB v2 format. In Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT. 8–9.

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

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

conference and symposium on the foundations of software engineering. 968–980.

[36] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In Proceedings of the 40th International Conference on Software Engineering. 361–371.

[37] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2023. On the Caching Schemes to Speed Up Program Reduction. ACM Transactions on Software Engineering and Methodology 33, 1 (2023), 1–30.

[38] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2023. On the Caching Schemes to Speed Up Program Reduction. ACM Transactions on Software Engineering and Methodology 33, 1 (2023), 1–30.

[39] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. 10–19.

[40] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic delta debugging. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 881–892.

[41] H. Wang, J. Chen, C. Xie, S. Liu, Z. Wang, Q. Shen, and Y. Zhao. 2023. MLIRSmith: Random Program Generation for Fuzzing MLIR Compiler Infrastructure. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE Computer Society, Los Alamitos, CA, USA, 1555–1566. https://doi.org/10.1109/ASE56229.2023.00120

[42] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the limit of 1-minimality of language-agnostic program reduction. Proceedings of the ACM on Programming Languages 7, OOPSLA1 (2023), 636–664.

[43] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? ACM SIGSOFT Software engineering notes 24, 6 (1999), 253–267.

[44] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (Charleston, South Carolina, USA) (SIGSOFT '02/FSE-10). Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/587051.587053

[45] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[46] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis.

[47] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. PPR: Pairwise Program Reduction. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 338–349.

Received 2024-09-12; accepted 2025-01-14