Robustifying Debug Information Updates in LLVM via Control-Flow Conformance Analysis

SHAN HUANG, JINGJING LIANG, and TING SU*, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
QIRUN ZHANG, Georgia Institute of Technology, USA

Optimizing compilers, such as LLVM, generate *debug information* in machine code to aid debugging. This information is particularly important when debugging optimized code, as modern software is often compiled with optimization enabled. However, properly updating debug information to reflect code transformations during optimization is a complex task that often relies on manual effort. This complexity makes the process prone to errors, which can lead to incorrect or lost debug information. Finding and fixing potential debug information update errors is vital to maintaining the accuracy and reliability of the overall debugging process. To our knowledge, no existing techniques can rectify debug information update errors in LLVM. While black-box testing approaches can find such bugs, they can neither pinpoint the root causes nor suggest fixes.

To fill the gap, we propose the *first* technique to *robustify* debug information updates in LLVM. In particular, our robustification approach can find and fix incorrect debug location updates. Central to our approach is the observation that the debug locations in the original and optimized programs must satisfy a *conformance relation*. The relation ensures that LLVM optimizations do not introduce extraneous debug location information on the control-flow paths of the optimized programs. We introduce *control-flow conformance analysis*, a novel approach that determines the reference updates ensuring the conformance relation by observing the execution of LLVM optimization passes and analyzing the debug locations in the control-flow graphs of programs under optimization. The determined reference updates are then used to check developer-written updates in LLVM. When discrepancies arise, the reference updates serve as the update skeletons to guide the fixing.

We realized our approach as a tool named MetaLoc, which determines proper debug location updates for LLVM optimizations. More importantly, with MetaLoc, we have reported and patched 46 previously unknown update errors in LLVM. All the patches, along with 22 new regression tests, have been merged into the LLVM codebase, effectively improving the accuracy and reliability of debug information in all programs optimized by LLVM. Furthermore, our approach uncovered and led to corrections in two issues within LLVM's official documentation on debug information updates.

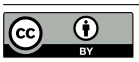
CCS Concepts: • **Software and its engineering → Compilers**; **Software testing and debugging**.

Additional Key Words and Phrases: Compiler Optimizations, Debug Information

ACM Reference Format:
Shan Huang, Jingjing Liang, Ting Su, and Qirun Zhang. 2025. Robustifying Debug Information Updates in LLVM via Control-Flow Conformance Analysis. *Proc. ACM Program. Lang.* 9, PLDI, Article 168 (June 2025), 23 pages. https://doi.org/10.1145/3729267

*Ting Su is the corresponding author.

Authors' Contact Information: Shan Huang, shan.huang@stu.ecnu.edu.cn; Jingjing Liang, jjliang@stu.ecnu.edu.cn; Ting Su, tsu@stu.ecnu.edu.cn, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China; Qirun Zhang, qrzhang@gatech.edu, Georgia Institute of Technology, Atlanta, USA.



ACM 2475-1421/2025/6-ART168
https://doi.org/10.1145/3729267

1 Introduction

Debugging is an essential activity in software development. To aid debugging, modern compilers generate *debug information* [Aggarwal and Kumar 2002] through the "-g" flag to establish a mapping between the source code and machine code. This information enables users to examine the source-level states of the program when diagnosing software failures. Debug information becomes even more important when debugging optimized code [Adl-Tabatabai and Gross 1996; Copperman 1994] because modern software is usually built with optimization enabled. Indeed, many widely used software systems (*e.g.*, Linux distributions [Gentoo Authors 2024], Mozilla Firefox[1] and the Linux software produced by GNU autotools [Calcote 2010]) depend on such debug information to diagnose failures in their optimized binaries.

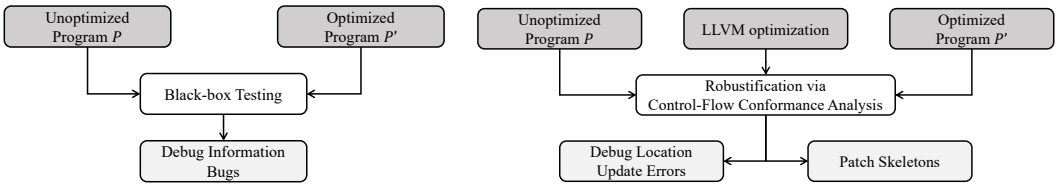
One important kind of debug information is *debug location information* (in short as *debug locations*) [LLVM Project 2024a]. Debug locations maintain the mapping between the line information of the low-level instructions in machine code and the statements in source code. Such information is critical for techniques like interactive debuggers (*e.g.*, GDB [GDB Developers 2024] and LLDB [The LLDB Team 2024]), sanitizers (*e.g.*, crash logs of AddressSanitizer [Serebryany et al. 2012]) and profile-guided optimization (*e.g.*, SamplePGO [Novillo 2014]). However, debug locations could become invalid due to various code transformations that rearrange, replace, or eliminate the instructions and statements. As a result, correctly maintaining debug location information is the key to ensuring the accuracy and reliability of debug information in optimized code.

Debug Location Update. Maintaining debug locations in the presence of compiler optimization is non-trivial. Our work focuses on addressing this problem in LLVM. A *debug location update* in LLVM involves three components: (1) the *destination instruction*, which is a single instruction requiring an updated debug location; (2) *source instructions*, whose debug locations serve as references for the update; and (3) the appropriate *update operation*, which is either `Preserve`, `Merge`, and `Drop`. These optimizations are defined in existing LLVM guidance [Kumar 2020]. When implementing LLVM optimization passes, developers must manually specify these components to ensure correct debug location updates. However, the absence of formal specifications [Copperman 1994; Hennessy 1982] for maintaining debug information makes this process error-prone. Additionally, LLVM has limited regression tests to validate these manually written updates. Consequently, incorrect debug location updates may occur, leading to bugs such as lost or misrepresented debug information. Finding and fixing these issues is crucial for ensuring reliable debugging support in LLVM.

The best-known approach to improving debug information maintenance is through testing. Prior black-box testing techniques [Assaiante et al. 2023; Di Luna et al. 2021; Li et al. 2020] can only find debug information bugs (Figure 1a) by comparing the optimized program P' with the unoptimized counterpart P. Intuitively, testing could not pinpoint location update errors in optimization implementations, let alone fix them. Indeed, many reported bugs are still open.[2] In an issue report of debug information bug for LLVM, one developer comments, "*This is sort of a known problem, and no-one has been brave enough to really address it yet, but there are some other tickets floating around about this*" [LLVM Developers 2023].

From Testing to Robustification. To fill this gap, this paper proposes a new *robustification* technique to improve the robustness of debug location updates in LLVM. In particular, robustification refers to the process of finding and fixing errors in debug location updates. Unlike traditional testing techniques that compare only the original program P and its optimized counterpart P', our robustification approach analyzes the internal behavior of LLVM optimization passes. As shown in Figure 1b, our approach inspects LLVM IR manipulations performed during the optimization of P

[1]The Windows release of Firefox is compiled with "-Z7 -O2" that enables the CodeView debug information in object files.
[2]https://github.com/llvm/llvm-project/issues?q=is%3Aissue+label%3Adebuginfo+is%3Aopen



(a) Black-box testing techniques. (b) Our robustification technique.

Fig. 1. Illustration of the difference between the black-box testing and robustification.

into P'. By leveraging internal knowledge of LLVM optimizations, our robustification approach can obtain the correct debug location updates and pinpoint errors by comparing them with these developer-written updates.

Control-flow Conformance Analysis. Central to our robustification approach is determining the "correct updates" in LLVM optimization passes, which is particularly challenging due to the lack of formal specifications. Our key observation is that debug locations in the optimized program P' and its unoptimized counterpart P should satisfy a *conformance relation*. Specifically, this relation refers to the fact that no "extra" debug location information should be introduced in any optimized path in P' beyond what exists in its original paths in P.

Ensuring the conformance relation presents unique challenges. First, we need to identify the source and destination instructions in the input program P and the optimized counterpart P', respectively. Second, we need to determine the proper debug location update operations to ensure the conformance relation between P and P', based on the identified source and destination instructions. To address both challenges, we introduce *control-flow conformance analysis*. The key enabling insight is to leverage the LLVM `Instruction` APIs used in LLVM optimization passes when optimizing the input program P. To this end, we identify four types of instruction manipulations (*i.e.*, `Create`, `Clone`, `Remove`, and `Replace`), instrument and monitor the corresponding LLVM APIs such as `Instruction::clone` and `Instruction::MoveBefore`. When optimizing P, these LLVM APIs essentially associate the original instructions in P with the optimized instructions in P', enabling us to identify the corresponding source and destination instructions. Using this information, we collect the debug location sets $\mathcal{L}_{src}$ (from P) and $\mathcal{L}_{dst}$ (from P'), based on their respective control-flow graphs. With $\mathcal{L}_{src}$ and $\mathcal{L}_{dst}$, our control-flow conformance analysis then applies a set of determination rules to determine the proper update operations that ensure the conformance relation between the optimized program P' and its unoptimized counterpart P. Finally, we compare the updates obtained by control-flow conformance analysis with the developer-written updates. Any discrepancies indicate potential errors in the debug location updates. In such cases, the obtained update serves as a patch skeleton to guide the correction process.

We have realized our approach as a tool named MetaLoc and applied it to the latest version of LLVM. MetaLoc has found 46 previously unknown debug location update errors in 18 LLVM optimization passes. More importantly, unlike existing techniques, we have successfully provided the patches to these 46 update errors based on the debug location updates determined by MetaLoc. All these patches have been accepted and landed in LLVM's codebase. Many of the fixed update errors were latent and affected a wide range of LLVM major versions (from LLVM 3 to 17). Fixing them has significantly improved the accuracy and reliability of debug locations in any program optimized by LLVM. MetaLoc has found no false positives during robustification, and our results are appreciated by LLVM developers. Additionally, informed by our control-flow conformance analysis, we have also found and patched two issues in LLVM documentation on updating debug information [Kumar 2020], which otherwise mislead compiler developers.

Contributions. This paper has made the following contributions:

- At the conceptual level, we propose the first technique to robustify debug location updates in LLVM, which determines reference updates for finding and fixing debug location update errors in LLVM optimization implementations.
- At the technical level, we introduce *control-flow conformance analysis*, a novel approach to determining correct debug location updates that ensure the conformance relation by monitoring four kinds of instruction manipulations when running LLVM optimization implementations and analyzing the debug locations based on the control-flow graphs of the programs under optimization.
- At the empirical level, we implement our approach as a tool named MetaLoc. The evaluation result shows the high precision of the updates determined by MetaLoc (detailed in Section 5.2). Moreover, with MetaLoc, we have found and fixed 46 previously unknown update errors in the latest version of LLVM (detailed in Section 5.3). All patches, along with 22 new regression tests, have been accepted and merged into LLVM's main trunk. Additionally, we have uncovered and patched two issues in LLVM's official documentation on updating debug information.

The paper is structured as follows. Section 2 introduces the maintenance of the debug location in LLVM and motivates our work through a concrete example. Then, we give details of the robustification approach in Section 3. Section 4 and Section 5 describe our implementation and experimental results, respectively. Finally, Section 6 surveys related work, and Section 7 concludes.

2 Background and Motivating Example

This section gives some background on LLVM's debug location maintenance (Section 2.1) and illustrates our approach with a real debug location update error in LLVM (Section 2.2).

2.1 Debug Location and Its Updates in LLVM

Debug Location. In optimizing compilers, debug location establishes the mapping between the source code and the compiled (or optimized) code. Specifically, in LLVM, debug location is maintained as metadata attached to LLVM IR instructions [LLVM Project 2024a]. It is represented in the form of a triplet (l, c, z), where l, c, and z denote the line number, the column number, and the scope (*e.g.*, the true branch of an if statement) of the corresponding source code, respectively. In this way, debug location maps the LLVM IR instruction to the corresponding source code. The code below shows an example LLVM IR instruction `%sub` (line 1), whose debug location is specified by the metadata tag "`!dbg !30`". This tag refers to the debug location (line 2), which denotes that `%sub` corresponds to line 6, column 17, and scope "!20" in the source code.

```
1 %sub = sub %a, %mul, !dbg !30
2 !30 = !DILocation(line: 6, column: 17, scope: !20)
```

Debug Location Updates. LLVM optimization passes perform various code transformations that rearrange, replace, or eliminate IR instructions. To ensure the accuracy of debug location mapping, LLVM developers need to manually update debug locations in LLVM optimization passes [Kumar 2020]. Specifically, let P and P' be the unoptimized program and the optimized counterpart, respectively, *w.r.t.* an optimization O. A *debug location update* in O consists of three elements: (1) the *update destination*, *i.e.*, the destination instruction (in P') whose debug location needs to be updated, (2) the *update sources*, *i.e.*, the source instruction(s) (in P) whose debug locations are used for the update, and (3) the *update operation*: `Preserve`, `Merge`, or `Drop` [Kumar 2020]. We formally define the semantics of these three update operations as follows.

- `Preserve`. Let s' and s be the destination and source instructions in P' and P, respectively. `Preserve` assigns the debug location of s to that of s'.
- `Merge`. Let s' and $s_1, ..., s_n$ ($n \geq 1$) be the destination and source instructions in P' and P, respectively. `Merge` assigns the merged debug location of $s_1, \ldots, s_n$ to that of s'.[3]
- `Drop`. Let s' be the destination instruction in P'. `Drop` removes the debug location of s'.

Debug Location Update Errors. Both incorrectly specified destination and source instructions and incorrect update operations can lead to *debug location update errors*. For example, retaining debug locations that should be discarded leads to incorrect (or misleading) debug locations, and discarding debug locations that should be retained leads to *lost* debug locations. The LLVM documentation [Kumar 2020] explains that "*[The* `Preserve` *operation] preserves the ability to set breakpoints on source locations corresponding to the instructions they touch. Debugging, crash logs, and SamplePGO accuracy would be severely impacted if that ability were lost,*" and "*The purpose of* `Drop` *is to prevent erratic or misleading single-stepping behavior in situations in which an instruction has no clear, unambiguous relationship to a source location.*"

2.2 A Real Debug Location Update Error in LLVM

Consider the C program in Figure 2a, the function `recursion` takes the parameter `a` and invokes recursive calls at lines 5 and 7. We can see that when function `recursion` is called by passing the argument value `2` (line 14, Figure 2a), the recursive call at line 7 will be executed. Note that the recursion call at line 5 is not reachable. However, when this C program is optimized by LLVM optimization pass TailRecursionElimination, the debug location becomes incorrect. For example, if we use LLDB to single-step the optimized program, LLDB will stop at line 5 (Figure 2b), which is misleading. Such incorrect debug location information could severely impact debugging, crash logs, and profiling-based optimization.

Root cause. The preceding incorrect debug location bug is caused by a real debug location update error (fixed by us with PR 95742) in the optimization pass TailRecursionElimination. Figure 2e shows the buggy code snippet of TailRecursionElimination which contains the debug location update error. Figure 2c and Figure 2d show the LLVM IR instructions of the function `recursion` before and after the optimization, respectively. The two blocks `if.then2` and `if.else` in Figure 2c and Figure 2d correspond to the true and false branches of the C program (Figure 2a), respectively. We can see that the debug locations in Figure 2c (before the optimization) are correct, which *accurately* map the instructions to the source code (Figure 2a). For example, line 6 in Figure 2c (highlighted) maps to line 5 in Figure 2a. In Figure 2e, this optimization clones the IR instruction `%add` (denoted by variable `AccRecInstr` at line 3) in the block `if.then2` (Figure 2c) to create a new instruction `%accumulator.ret.tr` (recorded by variable `AccRecInstrNew`, lines 3-6). This new instruction `%accumulator.ret.tr` is later inserted before the return instruction `ret i32 %accumulator.ret.tr` (recorded by variable `RI`, line 7) in the block `if.else` (Figure 2d). Note that the instructions `%add` and `%accumulator.ret.tr` are respectively highlighted in Figure 2c and Figure 2d. However, the optimization pass erroneously preserves the debug location of `%add` to `%accumulator.ret.tr`. As a result, when the new instruction `%accumulator.ret.tr` is executed, its debug location misleadingly informs the user that the execution reaches line 5 (*i.e.*, a dead branch).

Our Approach. We instrument the code of TailRecursionElimination by monitoring four kinds of instruction manipulations like `clone` (line 3 in Figure 2e). In this way, it can identify that the destination and source instructions are denoted by the variables `AccRetInstrNew` and `AccRecInstr`,

[3]Let $(l_1, c_1, z_1), (l_2, c_2, z_2), \ldots, (l_n, c_n, z_n)$ be debug locations and (l, c, z) be the merged debug location. In LLVM's implementation, $l = l_1$ if $l_1 = l_2 = \ldots = l_n$ otherwise $l = 0$. The same principle applies to c as well. The merged scope z is the intersection of $z_1, z_2, \ldots, z_n$.

```c
int recursion(int a) {
  if (a == 2) {
    int sub = a - 1;
    if (a & 1) { // Dead branch
      return recursion(sub) + 1;
    } else {     // Live branch
      return recursion(sub) * 2;
    }
  }
  return 0;
}

int main() {
  recursion(2);
}
```

(a) The C program triggering the optimization pass TailRecursionElimination.

```
(lldb) s
Process 28102 stopped
* Thread #1, name = 'a.out',
  stop reason = step in
  frame #0: a.out ` recursion(a=2)
  at main.c:5:38
   2  if (a == 2) {
   3    int sub = a - 1;
   4  if (a & 1) {
-> 5    return recursion(sub) + 1;
   6  } else {
   7    return recursion(sub) * 2;
   8  }
(lldb) v a
(int) a = 2
```

(b) When debugging, LLDB stops at line 5, the dead branch in the C program.

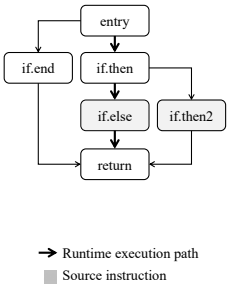
```
define dso_local i32 @recursion(i32 noundef %a) !dbg !10 {
...

if.then2:
  %call = call i32 @recursion(i32 noundef %sub1), !dbg !27
  %add = add nsw i32 %call, 1, !dbg !29
  br label %return, !dbg !30

if.else:
  %call3 = call i32 @recursion(i32 noundef %sub1), !dbg !31
  %mul = mul nsw i32 %call3, 2, !dbg !33
  br label %return, !dbg !34

...
}

!27 = !DILocation(line: 5, column: 20, scope: ..)
!29 = !DILocation(line: 5, column: 35, scope: ..)
!30 = !DILocation(line: 5, column: 13, scope: ..)
!31 = !DILocation(line: 7, column: 20, scope: ..)
!33 = !DILocation(line: 7, column: 35, scope: ..)
!34 = !DILocation(line: 7, column: 13, scope: ..)
```



(c) The LLVM IR snippet of the unoptimized function `recursion` and its CFG.

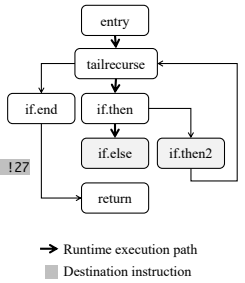
```
define dso_local i32 @recursion(i32 noundef %a) !dbg !10 {
...

if.then2:
  %add = add nsw i32 %accumulator.tr, 1, !dbg !27
  br label %tailrecurse, !dbg !29

if.else:
  %call3 = call i32 @recursion(i32 noundef %sub1), !dbg !30
  %mul = mul nsw i32 %call3, 2, !dbg !32
  %accumulator.ret.tr = add nsw i32 %accumulator.tr, %mul, !dbg !27
  ret i32 %accumulator.ret.tr, !dbg !33

...
}

!27 = !DILocation(line: 5, column: 35, scope: ..)
!29 = !DILocation(line: 5, column: 20, scope: ..)
!30 = !DILocation(line: 7, column: 20, scope: ..)
!32 = !DILocation(line: 7, column: 35, scope: ..)
!33 = !DILocation(line: 11, column: 1, scope: ..)
```



(d) The LLVM IR snippet of the optimized function `recursion` and its CFG.

```
1    void TailRecursionEliminator::cleanupAndFinalize() {
2      ...
3      Instruction *AccRecInstrNew = AccRecInstr->clone();
4      AccRecInstrNew->setName("accumulator.ret.tr");
5      AccRecInstrNew->setOperand(AccRecInstr->getOperand(0) == AccPN,
6                                 RI->getOperand(0));
7      AccRecInstrNew->insertBefore(RI);
8      // Update Skeleton: Line 781 Drop(AccResInstrNew)
9  +   AccRecInstrNew->dropLocation();  // Patch code
10     RI->setOperand(0, AccRecInstrNew);
11     ...
12   }
```

(e) The debug location update error in optimization pass TailRecursionElimination in LLVM. The error leads to incorrect debug locations in optimized programs. The comment in green shows the generated update skeleton by MetaLoc and the code in gray is the concretized patch to the error.

Fig. 2. A real debug location update error in the LLVM optimization pass TailRecursionElimination.

respectively. Later, our approach runs the optimization pass against the input C program. In this way, it can identify that `AccRetInstrNew` records the destination instruction `%accumulator.ret.tr` in the block `if.else` in Figure 2d, and `AccRecInstr` records the source instruction `%add` in the block `if.then2` in Figure 2c. Via performing the control-flow conformance analysis on the source and destination instructions based on the control-flow graphs of P and P' (*i.e.*, comparing $\mathcal{L}_{src}$ and $\mathcal{L}_{dst}$, our approach determines that `Drop` is the appropriate update operation to ensure the conformance relation. Thus, the determined debug location update is performing `Drop` on `AccRetInstrNew`. By comparing `Drop` to `Preserve`, our approach thus finds this update error and generates the update skeleton at line 8 in Figure 2e. Line 9 shows the accepted fixing patch to drop the debug location of `AccRetInstrNew`.

Limitations of Existing Techniques. Prior work uses black-box testing techniques [Di Luna et al. 2021] to find debug location bugs. It (1) generates random (closed) programs like the C program in

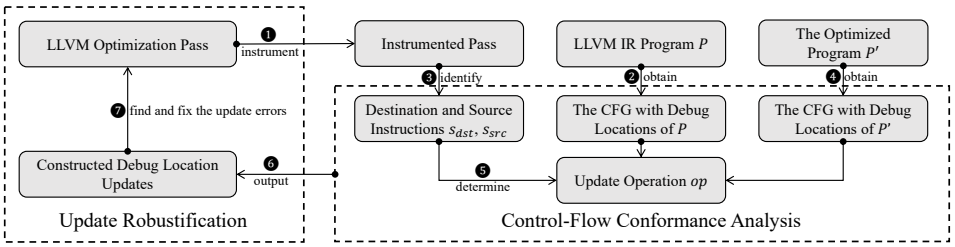


Fig. 3. Overview of our approach

Table 1. Four types of instruction manipulations capturing source and destination instructions. "DI" and "SI" denote destination and source instructions, respectively. "—" means no corresponding instruction identified.

Manipulation	Notation	Semantics	DI	SI
`Create`	$\langle$`Create`, $s\rangle$	Create a new instruction s	s	—
`Clone`	$\langle$`Clone`, $s_{new}, s_{old}\rangle$	Create a new instruction s_{new} by cloning an existing instruction s_{old}	s_{new}	s_{old}
`Move`	$\langle$`Move`, $s_{to}, s_{from}\rangle$	Move an instruction s from the position p_{from} to the position p_{to}	s_{to}	s_{from}
`Replace`	$\langle$`Replace`, $s_{to}, s_{from}\rangle$	Replace all uses of s_{from} with s_{to}	—	s_{from}

Figure 2a, (2) uses off-the-shelf debuggers like LLDB to collect the debug locations of the executed paths of the unoptimized program P and the optimized counterpart P' respectively, and (3) checks whether there exist some inconsistencies like Figure 2d (*e.g.*, line 5 that appears to be executed on P' while this line never appears to be executed on P). However, testing techniques may miss the debug location bug if the executed path cannot expose the inconsistency and cannot find lost debug location bugs. More importantly, they cannot pinpoint the update errors in the optimization or suggest fixing patches.

3 Approach

This section presents our robustification approach implemented as MetaLoc. Figure 3 illustrates the workflow of MetaLoc, which takes an optimization pass in LLVM as input. Our robustification approach determines the debug location updates via the proposed control-flow conformance analysis. Specifically, MetaLoc instruments the given optimization pass O (step ❶) and runs the instrumented pass against some unoptimized program P to generate the optimized counterpart P'. In this process, MetaLoc (1) obtains the control-flow graph of P and P' (steps ❷ and ❹), (2) identifies the source and destination instructions via monitoring specified instruction manipulations (step ❸). Next, MetaLoc determines the proper debug location update operations via control-flow conformance analysis and outputs the final debug location updates (steps ❺-❻). Finally, with the determined updates, we robustify developer-written updates in the optimization pass (step ❼).

In the following, Section 3.1 presents how MetaLoc identifies the destination and source instructions. Section 3.2 explains how MetaLoc determines updates by determining the update operations via control-flow conformance analysis. Section 3.3 presents the overall algorithm of our approach.

3.1 Identifying Update Destinations and Sources

To identify the destination and source instructions, our approach monitors the LLVM IR instruction manipulations performed by the optimization pass. In LLVM, class `Instruction` abstracts all the IR instructions. From `Instruction`'s APIs that manipulate instructions, we can distill six kinds of manipulations: `Create`, `Clone`, `Move`, `Insert`, `Delete`, `Replace`. Among the six kinds of manipulations, four of them (*i.e.*, `Create`, `Clone`, `Move`, `Replace`) can capture the destination and source instructions, while `Insert` and `Delete` cannot. Thus, the four kinds of manipulations listed in Table 1 are monitored. We give details about how they capture source and destination instructions below.

Algorithm 1: Identifying the Destination and Source Instructions

Input: The monitored instruction manipulations *IM*
Output: The pairs of destination and source instructions

```
1  DI ← ∅    // Initialize an empty set of destination instructions
2  SIM ← {} // Initialize an empty mapping from the destination instructions to source instructions
3  foreach instruction manipulation im in IM do
4      if im is ⟨Create, s⟩ then DI = DI ∪ {s}
5      else if im is ⟨Clone, s_new, s_old⟩ then
6          DI = DI ∪ {s_new}
7          SIM[s_new] = {s_old}
8      else if im is ⟨Move, s_to, s_from⟩ then
9          DI = DI ∪ {s_to}
10         SIM[s_to] = {s_from}
11 foreach instruction manipulation im in IM do
12     if im is ⟨Replace, s_to, s_from⟩ and s_to ∈ UD then
13         if SIM has an entry with key s_to then SIM[s_to] = SIM[s_to] ∪ {s_from}
14         else SIM[s_to] = {s_from}
15 return All entries in SIM
```

- `Create`: The `Create` creates a new instruction s, denoted as $\langle$`Create`, $s\rangle$. In this case, s's debug location is undefined and needs to be updated. Thus, s is identified as a destination instruction.
- `Clone`: The `Clone` creates a new instruction s_{new} by cloning an existing instruction s_{old}, denoted as $\langle$`Clone`, $s_{new}, s_{old}\rangle$. In this case, the new instruction s_{new} shares the same debug location with the cloned instruction s_{old}. Thus, s_{new} is identified as a destination instruction, and s_{old} is identified as a source instruction.
- `Move`: The `Move` moves an instruction s from the position p_{from} to the position p_{to}, denoted as $\langle$`Move`, $s_{to}, s_{from}\rangle$. In this case, the instruction s_{to} (now at the position p_{to}) is identified as a destination instruction, while the instruction s_{from} (originally at the position p_{from}), which provides the debug locations, is identified as a source instruction.
- `Replace`: The `Replace` finds all uses of the instruction s_{from} and replaces them with s_{to}, denoted as $\langle$`Replace`, s_{to}, $s_{from}\rangle$. In this case, if s_{to} is identified by `Create`, `Clone` or `Move` as the destination instruction, the replacing instruction s_{from} will be identified as the source instruction of s_{to}.

Algorithm 1 shows how MetaLoc identifies the destination and source instructions. It takes an optimization O as input and outputs the *update pairs*. An update pair $\langle s_{dst}, S_{src}\rangle$ includes the destination and the set of its corresponding source instructions identified from O. Algorithm 1 starts by initializing an empty set *DI* that collects destination instructions and an empty map *SIM* that records the pairs of the destination instructions and their corresponding source instructions (lines 1-2). All four instruction manipulations are collected in the set *IM* by scanning specific LLVM APIs (line 3). Section 4 gives the details of these specific LLVM APIs. In the first loop (lines 3-11), MetaLoc identifies the destination instructions via `Create`, `Clone`, and `Move` (lines 5, 7, 10), and the corresponding source instructions via `Clone` and `Move` (line 8 and line 11). In the second loop (lines 12-17), MetaLoc finds the source instructions for the destination instruction via `Replace` based on the results from the first loop.

Example 3.1. Consider the motivating example in Section 2.2. From the monitored manipulation ⟨`Clone`, `%accumulator.ret.tr`, `%add`⟩, `%accumulator.ret.tr` is identified as the destination instruction, *i.e.*, *DI* = {`%accumulator.ret.tr`}, and `%add` is the corresponding source instruction, *i.e.*,

SIM[`%accumulator.ret.tr`] = {`%add`}. Because there is no other monitored manipulations that manipulates `%accumulator.ret.tr`, no new source instructions are added to *SIM*[`%accumulator.ret.tr`]. Thus, the pair is ⟨`%accumulator.ret.tr`, {`%add`}⟩.

3.2 Control-Flow Conformance Analysis

After identifying the source and destination instructions, we need to determine the proper update operation (*i.e.*, `Preserve`, `Merge`, or `Drop`) for those instructions. Unfortunately, the existing LLVM documentation provides only informal guidelines on how to update debug locations correctly [Kumar 2020]. The key insight behind our approach is to leverage the control-flow structures of the original program P and its optimized counterpart P' to determine the proper debug update operations for P and P'. This is based on the observation that most LLVM optimization passes do not introduce extra debug locations into the optimized program, as extra locations could provide misleading information during debugging. Intuitively, the debug locations along a path in the optimized program P' should be a subset of those along the corresponding path in P. We formally state it as a *conformance relation*.

Definition 3.2 (Conformance Relation). Consider a control-flow path γ' in the optimized program P' and its corresponding path γ in the original program P. Let $L_{\gamma'}$ and L_{γ} denote the sets of debug locations attached to the instructions in γ' and γ, respectively. We say that γ' and γ form a conformance relation if and only if $L_{\gamma'} \subseteq L_{\gamma}$.

Note that Definition 3.2 considers the case in which a control-flow path γ' in P' is optimized from a single original path γ in P. In practice, however, compiler optimizations may merge or split control-flow paths. The conformance relation naturally generalizes to split paths, since each optimized path still corresponds to a single original path. For merged paths, the conformance relation can be extended by generalizing L_{γ} to account for the debug locations associated with instructions from all contributing original paths. Our work does not directly check conformance across all paths in the programs. Instead, we focus on the source s_{src} and destination s_{dst} instructions identified in Algorithm 1. For simplicity and clarity, for a given update pair $\langle s_{dst}, \{s_{src}, \ldots\}\rangle$, we use *dst* and *src* to denote the sets of destination $\{s_{dst}\}$ and source instructions $\{s_{src}, \ldots\}$, respectively. To determine update operations that enforce the conformance relation, we collect *debug location sets* for the destination and source instructions.

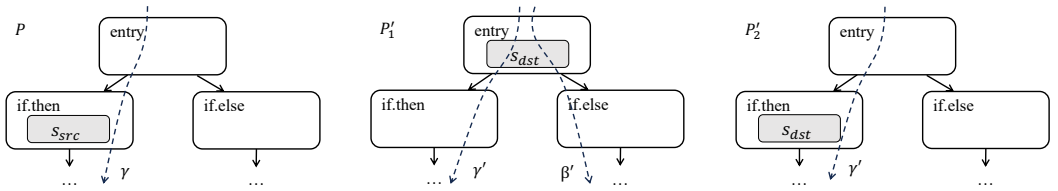
Definition 3.3 (Debug Location Sets). Consider a set of instructions $S = \{s_1, ..., s_n\}$. Let Γ_S be the set of control-flow paths that pass through any instruction in S. We define the debug location set *w.r.t.* S as

$$\mathcal{L}_S = \bigcup_{\gamma \in \Gamma_S} L_{\gamma}.$$

Therefore, based on the source and destination instructions $src = \{s_{src}, \ldots\}$ and $dst = \{s_{dst}\}$, we collect the corresponding debug location sets $\mathcal{L}_{src}$ and $\mathcal{L}_{dst}$, respectively. With these notations, we now proceed to formally introduce our control-flow conformance analysis.

Definition 3.4 (Control-Flow Conformance Analysis). Given two debug location sets, $\mathcal{L}_{src}$ from P and $\mathcal{L}_{dst}$ from P', control-flow conformance analysis determines the debug update operation for s_{dst} as follows:

- `Drop`, if $\mathcal{L}_{dst} \not\subseteq \mathcal{L}_{src}$;
- `Preserve`, if $\mathcal{L}_{dst} \subseteq \mathcal{L}_{src}$ and $|src| = 1$; and
- `Merge`, if $\mathcal{L}_{dst} \subseteq \mathcal{L}_{src}$ and $|src| > 1$.



(a) Unoptimized program P.

(b) Optimized P'_1 with an instruction movement.

(c) Optimized P'_2 with an instruction replacement.

Fig. 4. Illustration of the rationale behind control-flow conformance analysis. The dashed lines denote control-flow paths.

From Definition 3.4, we can see that, based on the control-flow graphs of P and P', the control-flow conformance analysis computes the proper debug update operation for the source and destination instructions s_{dst} and s_{dst}. The rationale behind the control-flow conformance analysis is to determine whether the destination instruction s_{dst} appears in control-flow paths where the source instructions are absent. Ideally, we can obtain the correspondence between the control-flow paths in P' and P, *i.e.*, mapping a path β' in P' to the path β in P. If there is a path $\beta' \in \Gamma_{dst}$ but the corresponding path $\beta \notin \Gamma_{src}$, we should perform a `Drop`, because preserving the debug location to s_{dst} would lead to an extra debug location on the path β' (*i.e.*, $L_{\beta'} \not\subseteq L_{\beta}$), violating the conformance relation. Consider two optimization scenarios without control-flow changes *w.r.t.* an update pair $\langle s_{dst}, \{s_{src}\}\rangle$ in Figure 4.

- Figure 4a depicts the source instruction s_{src} in the unoptimized program P. Let γ be the control-flow path passing through s_{src}. If the optimization moves s_{src} from block `if.then` to `entry`, the destination instruction s_{dst} after moving is shown in the optimized program P'_1 in Figure 4b. In this case, $\Gamma_{src} = \{\gamma\}$ while $\Gamma_{dst} = \{\gamma', \beta'\}$. If we preserve the debug location of s_{src} to s_{dst}, the path β' will have an extra debug location (*i.e.*, the debug location of s_{src}) that did not exist before the optimization. Thus, the update operation should be `Drop`.
- If the optimization replaces s_{src} with a new instruction s_{dst} inside the same basic block `if.then`, s_{dst} is the destination instruction in the optimized program P'_2 shown in Figure 4c. In this case, $\Gamma_{dst} = \{\gamma'\}$ and $\Gamma_{src} = \{\gamma\}$ and preserving the debug location of s_{src} to s_{dst} would not lead to extra debug location on any control-flow path. Thus, the correct update operation should be `Preserve`.

In practice, it is difficult to establish the precise correspondence between the control-flow paths in the optimized and unoptimized programs. Thus, our analysis *over-approximates* the ideal situation, where the precise correspondence between control-flow paths in P and P' is obtainable, by abstracting these paths (*i.e.*, Γ_{dst} and Γ_{src}) based on the debug locations that can be collected from them (*i.e.*, $\mathcal{L}_{dst}$ and $\mathcal{L}_{src}$). More specifically, we use the debug locations on the control-flow paths in P' to indicate the corresponding original control-flow paths in P.

Algorithm 2 gives the details of control-flow conformance analysis. As shown in the algorithm, the control-flow conformance analysis first collects the debug location sets $\mathcal{L}_{src}$ and $\mathcal{L}_{dst}$ (lines 1-4 and the function `CollectDebugLocsFromCFG`). Next, it determines the proper debug location update operation by comparing the two debug location sets $\mathcal{L}_{dst}$ and $\mathcal{L}_{src}$ and output it with the given update pair (lines 5-11). Note that when we collect paths from a given control-flow graph, each path is allowed to traverse the loops in the graph only once to avoid path explosion. Although the paths traverse a loop only once, the debug locations of each branch in the loop can be covered by at least one path.

Example 3.5. Consider the example in Section 2.2. Through Algorithm 1, we have identified the update pair $\langle$`%accumulator.ret.tr`, {`%add`}$\rangle$. In this step, the set $\mathcal{L}_{src}$ and $\mathcal{L}_{dst}$ is first collected. For the source instruction `%add`, the only path passing it on the control-flow graph of P is `entry` $\rightarrow$

Algorithm 2: Control-Flow Conformance Analysis

Input: An update pair $\langle s_{dst}, S_{src}\rangle$, the P's and P''s control control-flow graphs G_P and $G_{P'}$
Output: The determined debug location update

```
1 L_src ← ∅
2 foreach s_src ∈ S_src do L_src ← L_src ∪ CollectDebugLocsFromCFG(s_src, G_P)
3 L_dst ← CollectDLInfoFromCFG(s_dst, G_P')
4 if L_dst ⊈ L_src then return ⟨s_dst, S_src, Drop⟩
5 else
6     if |S_src| = 1 then return ⟨s_dst, S_src, Preserve⟩
7     else return ⟨s_dst, S_src, Merge⟩
```

`if.then` $\longrightarrow$ `if.then2` $\longrightarrow$ `return`, resulting in $\mathcal{L}_{src} = \{2, 3, 4, 5, 11\}$. For the destination instruction `%accumultor.ret.tr`, the only path passing it on the control-flow graph of P' is `entry` $\longrightarrow$ `tailrecurse` $\longrightarrow$ `if.then` $\longrightarrow$ `if.else`, resulting in $\mathcal{L}_{dst} = \{2, 3, 4, 5, 7, 11\}$. Therefore, $\mathcal{L}_{dst} \nsubseteq \mathcal{L}_{src}$ and the control-flow conformance analysis on the update pair determines the update operation `Drop`.

We prove the correctness of control-flow conformance analysis by showing that it guarantees the conformation relation. Our proof focuses on the conformance relation defined in Definition 3.2, but it can be generalized to conformance relations that involve split and merged control-flow paths.

Theorem 3.6. *The update operation obtained by control-flow conformance analysis (Definition 3.4) guarantees that the conformance relation (Definition 3.2) holds* w.r.t. *the update pair* $\langle s_{dst}, \{s_{src}, ...\}\rangle$.

Proof. Our control-flow conformance analysis outputs one of the update operations `Preserve`, `Merge`, and `Drop`. It suffices to prove that in each case, we have $L_{\beta'} \subseteq L_\beta$ for the affected path $\beta' \in \Gamma_{dst}$ in P' and its corresponding original path β in P. Consider the `Drop` update operation, *i.e.*, $\mathcal{L}_{dst} \nsubseteq \mathcal{L}_{src}$. If only the debug locations in $\mathcal{L}_{dst} \cap \mathcal{L}_{src}$ can be collected from β', the conformance relation $L_{\beta'} \subseteq L_\beta$ inherently satisfies (*i.e.*, no extra debug location is introduced) and dropping the debug location would not break the relation (only making $L_{\beta'}$ smaller). If a debug location $l \in \mathcal{L}_{dst} \setminus \mathcal{L}_{src}$ can be collected from β', the original path β does not traverse any of the source instructions (*i.e.*, $l \notin \mathcal{L}_{src}$). However, after optimization, s_{dst} appears in β'. Thus, performing `Drop` on the destination instruction ensures $L_{\beta'} \subseteq L_\beta$; otherwise, the extra debug location information from the source instructions would be introduced to β', resulting in $L_{\beta'} \nsubseteq L_\beta$. In contrast, if the update operation is `Preserve` or `Merge`, *i.e.*, $\mathcal{L}_{dst} \subseteq \mathcal{L}_{src}$, for each path $\beta' \in \Gamma_{dst}$, the relation $L_{\beta'} \subseteq L_\beta$ inherently holds and the original path β already includes the debug locations of the source instructions. Thus, preserving the debug locations of the source instructions would not lead to incorrect debug location in β', but would improve the completeness of the debug location. □

The control-flow conformance analysis, for a destination instruction, collects all the control-flow paths passing through it, including those that may not be executed at runtime. Thus, the analysis accounts for all possible execution paths, which are affected, of a program. However, the overall robustification approach (*i.e.*, determining the proper updates and reporting the incorrect developer-written updates) is neither sound nor complete.

3.3 Robustifying Debug Location Updates

This section presents the overall algorithm of our approach for robustifying debug location updates in LLVM. Specifically, we first determine debug location updates for a given optimization with the proposed control-flow conformance analysis and then check the developer-written updates with the determined updates as the reference. Any inconsistencies indicate potential debug location

Algorithm 3: Robustifying Debug Location Updates

Input: A compiler optimization O and the input program P
Output: Debug location update skeletons $\mathcal{U}_s$

1 $O' \leftarrow$ InstrumentOptimization(O)
2 $G_P \leftarrow$ ObtainControlFlowGraph(P)
3 $P', IM \leftarrow$ RunOptimization(O', P)
4 $G_{P'} \leftarrow$ ObtainControlFlowGraph(P')
5 $\mathcal{U} \leftarrow$ ConstructDebugLocUpdates(IM, G_P, G'_P)
6 $\mathcal{U}_s \leftarrow \emptyset$ // Initialize an empty set of update skeletons
7 **foreach** $U = \langle s_{dst}, S_{src}, op \rangle \in \mathcal{U}$ **do**
8 $op_{exist} \leftarrow$ IdentifyExistingUpdateOperations(s_{dst})
9 **if** $op \neq op_{exist}$ **then** $\mathcal{U}_s \leftarrow \mathcal{U}_s \cup$ ConstructSkeleton(U)
10 **return** $\mathcal{U}_s$

update errors, and update skeletons are generated according to the determined updates to assist in fixing the errors.

Algorithm 3 explains this process. It takes as input an optimization O and an unoptimized program P, and outputs the update skeletons for any found potential update errors. In line 1, it instruments the given optimization O for monitoring the four kinds of instruction manipulations (step ❶). Before running the instrumented optimization, we obtain the control-flow graph of the program P (step ❷, line 2). Next, we run the instrumented optimization O' with P, obtaining the optimized program P' and the monitored instruction manipulations. After optimization, we obtain the control-flow graph of P' (step ❸, line 4). At the core of the algorithm, debug location updates are determined via Algorithms 1 and 2 (steps ❹-❻, line 5). For each determined debug location update $\langle s_{dst}, S_{src}, op \rangle$, it compares the determined operation op with the developer-written update operation op_{exist} in the optimization O *w.r.t.* the destination instruction s_{dst} (lines 7-9). If op and op_{exist} are different, a potential update error is found and an update skeleton (illustrated in Section 4) for fixing is generated (line 10). To identify existing update operations in O, we instrument and monitor the update operations of the developer-written updates (*i.e.*, `Preserve`, `Merge`, `Drop`) in O. Note that in some cases, the destination instructions may not have explicit update operations. In such cases, we infer the existing update operations based on how the destination instructions are created. For example, if s_{dst} is created by $\langle \texttt{Create}, s \rangle$, the update operation is `Drop` (because the debug location is undefined); if s_{dst} is created by $\langle \texttt{Clone}, s_{new}, s_{old} \rangle$ or $\langle \texttt{Move}, s_{to}, s_{from} \rangle$, the update operation is `Preserve` (because the debug location is retained from s_{old} or s_{from}).

Example 3.7. Consider again the motivating example in Section 2.2. After instrumenting the pass TailRecursionElimination, we run the instrumented pass with the IR program P in Figure 2c to obtain the optimized program P' in Figure 2d and the monitored instruction operations. After obtaining the control-flow graphs of P and P', we determine the debug location update using Algorithm 2. Examples 3.1 and 3.5 further describe the determination process. The determined debug location update is $\langle$`%accumulator.ret.tr`, {`%add`}, `Drop`$\rangle$. However, the existing update operation for the destination instruction `%accumulator.ret.tr` is `Preserve`. Because of the inconsistency, an update error is reported and an update skeleton for that error is generated.

4 Implementation

We implemented our approach as a tool named MetaLoc. MetaLoc consists of two components: (1) a lightweight built-in library for LLVM, which realizes our approach, and (2) a standalone pass

Table 2. Hooks injected into the LLVM optimizations and the target LLVM APIs.

Hook	Functionality	Target LLVM APIs

`OnCreate`
`OnMove`
`OnClone`
`OnUseReplace`
	Monitor instruction manipulations of the optimization pass and record the destination and source instructions	`BinaryOperation::Create*`
		`Instruction::Move*`
		`Instruction::clone`
		`Value::replaceAllUsesWith`

`OnPreserve`
`OnMerge`
`OnDrop`
	Monitor the debug location updates implemented by developers in optimization passes	`Instruction::setDebugLoc`
		`Instruction::applyMergedLocation`
		`Instruction::dropLocation`

`OnStart`	Initialize the analysis and collect debug locations from the input unoptimized programs	Before the first statement in the optimization pass entry `*Pass::run`
`OnFinish`	Collect debug locations from the optimized programs, check potential update errors, and generate update skeletons	Before each return statement in the optimization pass entry `*Pass::run`

Table 3. Update skeletons generated by MetaLoc.

Skeleton Type	Update Skeleton	Actual LLVM Code Add by Developers
Preserving	`Line <number> Preserve(DstInst, SrcInst)`	`DstInst->setDebugLoc(SrcInst->getDebugLoc());`
Merging	`Line <number> Merge(DstInst, SrcInst1, ..., SrcInstN)`	`DstInst->applyMergedLocation(SrcInst1, SrcInst2);`
Dropping	`Line <number> Drop(DstInst)`	`DstInst->dropLocation();`

instrumentation tool for analyzing LLVM optimizations. We give some important implementation details of our approach below.

Instrumenting LLVM Optimizations. The instrumentation tool injects some hooks into LLVM optimizations to obtain the runtime information and integrate our analysis. Table 2 lists these hooks (grouped by their effects).

- *Monitoring Instruction Manipulations*. The first group of hooks (the first row in Table 2) monitors instruction manipulations (*i.e.*, `Create`, `Clone`, `Move`, `Replace`) for identifying the destination and source instructions (step ❹). In LLVM, IR instructions of the subclasses of `Instruction` have their own `Create` APIs, while other manipulations use the APIs provided by `Instruction`. Specifically, all `Create` APIs from `Instruction` and its 36 subclasses (*e.g.*, `BinaryOperator`) are monitored by `OnCreate`; three `Move` APIs, one `Clone` API, and one `Replace` API are provided by `Instruction`, and them are all monitored by the corresponding hooks listed in the table. The second group of hooks (the second row in Table 2) monitors the existing update operations (*i.e.*, `Preserve`, `Merge`, and `Drop`) implemented in the optimizations for finding potential update errors (Section 3.3). The three target APIs listed in the table are the standard APIs for updating debug locations.
- *Integrating our Analysis*. The hook `OnStart` initializes our analysis and collects debug locations from the unoptimized program P (step ❷) at the beginning of the optimization. The hook `OnFinish` collects debug locations from the optimized P' (step ❸) at the end of the optimization, executes the control flow conformance analysis, and generates update skeletons if enabled.

Collecting Debug Location. In steps ❷ and ❸, we use depth-first search to collect debug locations from the control flow graphs of P and P'. Since our approach is implemented as a built-in library, we can collect debug locations using LLVM classes like `BasicBlock` and `Instruction`.

Update Skeleton. MetaLoc generates update skeletons for identified update errors to assist in their fixing. Table 3 lists the three types of update skeletons. The column "Update Skeleton" shows the forms of update skeletons generated by MetaLoc. Note that these update skeletons include concrete variables from the LLVM optimization implementation that represent the destination and source instructions, along with the specific LLVM APIs used for the update operation. The update skeleton also suggests a reference source code line in the LLVM optimization implementation where the determined debug location update should be added. Informed by these update skeletons, LLVM developers could manually instantiate the skeletons into ready-to-use patches to fix debug location update errors. The last column of Table 3 gives some specific examples.

5 Evaluation

We evaluate MetaLoc by addressing the following two research questions:

- **RQ1**: How accurate are the debug location updates determined by MetaLoc?
- **RQ2**: How effective is MetaLoc in finding and fixing previously unknown debug location update errors in LLVM?

To answer **RQ1**, we assemble a dataset of historical update error fixes confirmed by LLVM developers as the ground truth, and investigate how many of the debug location updates determined by MetaLoc are identical to the ground truth. To answer **RQ2**, we apply MetaLoc to the latest version of LLVM at the time of our work (as of November 2024) to investigate how many previously unknown debug location update errors can be found and fixed with the help of the update skeletons generated by MetaLoc. We summarize the main results of our experiments below.

- In the dataset of 31 debug location updates implemented and confirmed by LLVM developers, MetaLoc successfully determined 30 updates which are identical to the ground truth.
- With the help of MetaLoc, we have found and fixed 46 previously unknown debug location update errors in LLVM. We filed 21 pull requests with the patches for fixing these 46 errors. All patches have been accepted and merged into LLVM. In addition, 22 new tests have been added to the regression test suite for LLVM optimizations.
- Many of the reported debug location update errors are latent and affect many LLVM major versions. These update errors were introduced into LLVM from 2008 to 2023, and eight errors have been latent since before the release of LLVM 3 (Dec 1, 2011).
- We have uncovered and fixed two issues in the LLVM's official documentation [Kumar 2020] for guiding compiler developers to update debug information. One issue gives an incorrect debug info update example (which should use `Merge` instead of `Drop`), and the other issue gives inaccurate and incomplete guidelines for using `Merge`.

5.1 Experimental Setup

Figure 3 shows the workflow of MetaLoc that takes an optimization pass and an (unoptimized) input program as input. This section introduces the experimental settings of RQ1 and RQ2 in terms of the *optimization passes*, *input programs* and *the procedure of running optimization pass with the input programs*. All experiments were conducted on a machine running 64-bit Ubuntu 20.04 LTS and equipped with an AMD 3995WX 64-core CPU and 128G RAM.

Optimization passes. We perform our experiments on all the 80 optimization passes under the LLVM's directory `llvm/lib/Transforms/Scalar` [LLVM Project 2024b]. They primarily focus on intra-procedural optimizations, performing various code transformations in which the instructions are rearranged, moved, or eliminated. Thus, debug location updates are necessary and crucial in these optimization passes.

Input programs. For our experiments, we used LLVM's regression test suite, consisting of LLVM IR programs, as the input for testing LLVM optimizations. Specifically, for each optimization pass in `llvm/lib/Transforms/Scalar`, there is a subdirectory under `llvm/test/Transforms/` containing the corresponding regression tests. We ran each optimization pass with all its corresponding regression tests. In the latest version of LLVM (87d36c5), there are 9590 regression tests in `llvm/test/Transforms/`. In contrast to randomly generated input programs, these regression tests are more diverse and are better at covering a wide range of code transformations in LLVM. As a result, they are more likely to involve instruction manipulations, making them suitable for determining debug location updates.

Running an optimization pass with input programs. We used the commands provided by the regression tests, *e.g.*, "`opt -passes=<optimization> -S`", to run an optimization pass with the input

programs. In this command, "opt" is the LLVM mid-end optimizer driver, "-passes" specifies the optimization pass, and "-S" denotes the assembly format. We used LLVM's debugify pass [Vedant, Kumar 2017] to assign synthetic debug locations for each instruction in the input program. For example, the command "opt -S -passes=callsite-splitting -debugify Ipad.ll" assigns debug locations and runs the CallSiteSplitting optimization pass with the input program "Ipad.ll."

Setup for RQ1. To evaluate the accuracy of the debug location updates determined by MetaLoc, we assembled a dataset of debug location updates implemented and verified by the LLVM developers as the ground truth. Specifically, our dataset contains the fixes of historical update errors in LLVM reviewed and verified by LLVM developers. To this end, we searched all the 484,935 commits of LLVM before Jan 1, 2024, to find such fixes. First, we filtered the commits on the optimization passes under llvm/Transforms/Scalar whose code revisions involve the LLVM APIs of debug location update operations (*i.e.*, setDebugLoc, applyMergedLocation, and DropLocation). After this step, we obtained 151 relevant commits. Second, we manually excluded those commits that are not real fixes of debug update errors. For example, in some cases, the LLVM debug update APIs are involved because the code containing those LLVM APIs was refactored or added for new code transformations. After this step, we obtained 55 fixes for historical update errors. Finally, we inspected the debug location updates in these collected fixes and retained those within the scope of our approach, *i.e.*, at least one destination and one source instruction could be identified via the four kinds of instruction manipulations. We thereby obtained 31 verified debug location updates as the ground-truth dataset.

Table 4 presents the detailed information of the 31 debug location updates in this dataset. The column "Commit ID" gives the LLVM commits and the fixed lines. The column "Fixing Date" denotes the dates on which the update errors were fixed. The column "Optimization Pass" denotes the optimization passes that contain the errors. The column "Operations" denotes the operations of the debug updates. The column "Bug Type" denotes the types of the bugs caused by the historical errors. We can see that these update errors are representative: (1) spanning a long duration from 2011 to 2023 and implemented in 16 different LLVM optimization passes, (2) involving all three update operations, *i.e.*, Preserve, Merge, Drop, and leading to the two different bug types.

In our experiments, we applied MetaLoc on the provided commit version of LLVM to determine the debug location update. The determined update is counted as *accurate* only when its destination and source instructions and the update operation are all *identical* to the ground truth.

Setup for RQ2. We applied MetaLoc to find potential debug location update errors in the latest revision of LLVM (87d36c5) at the time of our work. Among the total 1401 optimizing functions in the 80 optimization passes, 386 functions that directly or indirectly use the monitored APIs are instrumented by MetaLoc in 54 passes. When a new update error was found, we filed an issue report to LLVM and created a pull request to fix it. Specifically, each issue report includes (1) the code snippet of the optimization where the update error was introduced, (2) the input (unoptimized) program P exposing the error, and (3) its optimized counterpart P'. A pull request includes a patch and a new regression test. The patch is concretized from the update skeleton generated by MetaLoc. We manually transform the update skeleton into the concretized patch code (Table 3). The new regression test is constructed based on the input program P. P is added with the checks for validating the debug locations in the optimized counterpart P'.

5.2 RQ1: Accuracy of the Constructed Debug Location Updates

Table 4 gives the results of RQ1. In column "Identical?", "✓" denotes that the debug update determined by MetaLoc is identical to the corresponding update in the ground-truth dataset, and "×" denotes that the debug update determined by MetaLoc is incorrect. We can see that MetaLoc has successfully determined identical debug updates on 30 out of 31 cases. This shows that MetaLoc has high accuracy in determining correct debug location updates for different update operations.

Table 4. Constructed debug location updates for previous LLVM releases. Compared to the ground truth, MetaLoc has successfully generated 30 correct update skeletons. Only one update skeleton is incorrect.

	Commit ID	Fixing Date	Optimization Pass	Update Operation	Bug Type	Identical?
1	80d1d3a-L220	Apr 29, 2011	Reassociate	`Preserve`	Lost	✓
2	80d1d3a-L510	Apr 29, 2011	Reassociate	`Preserve`	Lost	✓
3	80d1d3a-L536	Apr 29, 2011	Reassociate	`Preserve`	Lost	✓
4	80d1d3a-L1059	Apr 29, 2011	Reassociate	`Preserve`	Lost	✓
5	33d87d9-L577	Apr 29, 2011	TailRecursionElimination	`Preserve`	Lost	✓
6	c1f7c1d-L330	Apr 30, 2011	LoopRotation	`Preserve`	Lost	✓
7	ffb798c-L2121	May 5, 2011	GVN	`Preserve`	Lost	✓
8	306f8db-L948	May 5, 2011	JumpThreading	`Preserve`	Lost	✓
9	306f8db-L1381	May 5, 2011	JumpThreading	`Preserve`	Lost	✓
10	5127c5d-L99	Jun 3, 2011	SimplifyCFGPass	`Preserve`	Lost	✓
11	89645df-L1699	Jun 11, 2015	GVN	`Preserve`	Lost	✓
12	89645df-L1767	Jun 11, 2015	GVN	`Preserve`	Lost	✓
13	5b7e21a-L315	Sep 12, 2018	CallSiteSplitting	`Preserve`	Lost	✓
14	984f1dc-L1171	Jul 19, 2017	GVN	`Preserve`	Lost	✓
15	5b7e21a-L399	Sep 12, 2018	CallSiteSplitting	`Preserve`	Lost	✓
16	35f504c-L553	Nov 18, 2018	CorrelatedValuePropagation	`Preserve`	Lost	✓
17	35f504c-L575	Nov 18, 2018	CorrelatedValuePropagation	`Preserve`	Lost	✓
18	35f504c-L598	Nov 18, 2018	CorrelatedValuePropagation	`Preserve`	Lost	✓
19	b60aea4-L1147	Mar 11, 2019	JumpThreading	`Preserve`	Lost	✓
20	b60aea4-L1251	Mar 11, 2019	JumpThreading	`Preserve`	Lost	✓
21	cc7803e-L2350	Apr 26, 2021	LoopStrengthReduce	`Preserve`	Lost	✓
22	ebd0249-L858	Sep 1, 2022	CorrelatedValuePropagation	`Preserve`	Lost	✓
23	368681f-L1153	Mar 31, 2022	GVNHoist	`Drop`	Incorrect	✓
24	52545e6-L778	Sep 29, 2022	InferAddressSpaces	`Preserve`	Lost	✓
25	84a71d5-L252	Aug 30, 2022	MergedLoadStoreMotion	`Merge`	Incorrect	×
26	84a71d5-L259	Aug 30, 2022	MergedLoadStoreMotion	`Merge`	Incorrect	✓
27	256f8b0-L844	Oct 28, 2022	StructurizeCFG	`Preserve`	Lost	✓
28	256f8b0-L956	Oct 28, 2022	StructurizeCFG	`Preserve`	Lost	✓
29	256f8b0-L977	Oct 28, 2022	StructurizeCFG	`Preserve`	Lost	✓
30	06a9c67-L943	Dec 22, 2023	CorrelatedValuePropagation	`Preserve`	Lost	✓
31	c0a986a-L1716	Jun 16, 2023	LICM	`Drop`	Incorrect	✓

Table 5. The summary of the robusitification results by MetaLoc.

Overall Results			Fixes by Update Operations			Bug Types	
#Found Update Errors	#Issued PRs	#Accepted PRs	`#Preserve`	`#Merge`	`#Drop`	#Lost	#Incorrect
46	21	21	37	2	7	37	9

Only one determined debug update is incorrect. In this case, MetaLoc determines the update operation as `Drop`, which, however, should be `Merge`. Our approach relies on the four kinds of instruction manipulations that indicate the relations between the source and destination instructions, under-approximating the overall debug location update problem. But in this case, for the destination instruction, there are source instructions that cannot be identified by the four kinds of instruction manipulations, which is out of the scope of our approach. Therefore, MetaLoc determined the incorrect update operation.

5.3 RQ2: Finding and Fixing Previously Unknown Update Errors

Table 5 summarizes the overall results. In total, 46 update errors have been found and fixed via 21 pull requests to LLVM (the column "Overall Results"). All pull requests have been accepted by LLVM developers and merged into LLVM. Specifically, among these 46 update errors, 37 errors were fixed by the `Preserve` operation, two errors were fixed by the `Merge` operation, and seven errors were fixed by the `Drop` operation (the column "Fixed by Update Operations"). Additionally, all 37 update errors fixed by `Preserve` lead to lost debug locations, while the 9 update errors fixed

Table 6. Our patches to the found debug location update errors by pull requests.

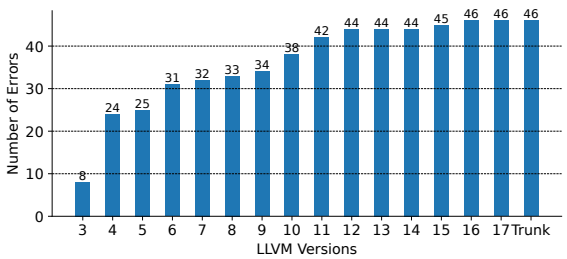
	Pull Request ID	Optimization Pass	Bug Type	#Errors
1	76118	CorrelatedValuePropagation	Lost	1
2	86236	GVNHoist	Incorrect	1
3	86269	TailRecursionElimination	Incorrect	1
4	91443	IndVarSimplify	Lost	4
5	91581	JumpThreading	Lost	3
6	91729	LICM	Lost	2
7	91839	LoopLoadElimination	Lost, Incorrect	3
8	92545	NaryReassociate	Lost	2
9	92859	GVNSink	Incorrect	1
10	95742	TailRecursionElimination	Incorrect	2
11	96045	DivRemPairs	Lost	5
12	96849	SeparateConstOffsetFromGEP	Lost	2
13	96889	JumpThreading	Lost	2
14	97038	InferAddressSpaces	Lost	1
15	97085	LoopFlatten	Lost	1
16	97145	LowerConstantIntrinsics	Lost	1
17	97384	SpeculativeExecution	Incorrect	1
18	97389	SimplifyCFGPass	Lost	1
19	97519	LoopStrengthReduce	Lost	2
20	97662	SimpleLoopUnswitch	Lost, Incorrect	6
21	98789	SimpleLoopUnswitch	Lost	4
#Total				46

by `Merge` and `Drop` lead to incorrect debug locations (the column "Bug Types"). Table 6 gives the details of these reported errors and fixes. Specifically, these fixes can be classified into the following three categories based on the expected update operations:

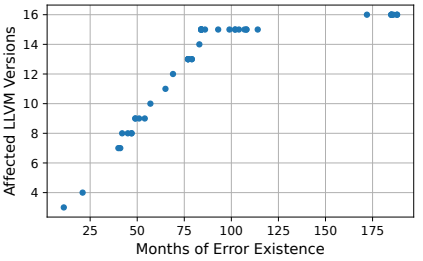
- Thirty-seven update errors were fixed with the `Preserve` operation. In these cases, the optimizations discard the debug locations that should be retained, which could lead to lost debug locations. We examined the buggy optimization implementations that contain these errors. We find that most of these errors occur when a code transformation replaces instructions in one basic block but the debug locations have not been retained.
- Two update errors were fixed with the `Merge` operation. These two errors occur when multiple source instructions are hoisted to a predecessor block or sunk to a successor block, but the debug location of only one source instruction is preserved. In fact, the debug locations of all the source instructions should be merged. Such errors lead to incorrect debug locations.
- Seven update errors were fixed with the `Drop` operation. These seven errors occur when the instructions are moved or cloned across different basic blocks. However, the debug locations of the moved or cloned instructions have not been dropped, which introduces extra debug locations into the optimized code. Such errors lead to incorrect debug locations.

From the results in Tables 4 and 6, we note that lost debug locations are more prevalent than incorrect debug locations. This observation also aligns with the statistics of the constructed dataset in Table 4. The possible reason is that compiler developers are more likely to forget to update the debug locations when moving instructions within the same basic block. Lost debug locations are more difficult to find compared to incorrect debug locations. For example, prior work using black-box testing techniques cannot find lost debug locations.

Affected LLVM Passes. We break down the debug update errors by the affected optimization passes. Table 6 gives the detected errors from 18 optimization passes in total. 14 passes are affected by lost debug locations, and these passes primarily handle code transformations that replace instructions within the same basic block. Additionally, six passes are impacted by incorrect debug locations, and these passes are responsible for instruction movement across blocks or modifications



(a) Affected LLVM versions and number of found errors.



(b) Affected LLVM versions and the duration of found errors.

Fig. 5. Statistics of the 46 debug location update errors found by MetaLoc.

to the program's control-flow structure. The SimpleLoopUnswitch pass has the highest number of errors (10) we have identified, including both lost and incorrect debug location errors. Notably, eight of these errors were introduced in 2017.

Affected LLVM Versions. We further investigated the impact of the 46 update errors found by MetaLoc. Figure 5a illustrates the impact of these update errors on various LLVM versions by listing the number of debug location update errors. We can see that these update errors affected a wide range of LLVM versions—from the earliest LLVM 3 to the most recent LLVM 17 and the trunk version, and each version in between has been affected. Figure 5b shows how long these update errors have persisted in the major versions of LLVM, and how many LLVM versions were affected. We can see that many update errors were introduced a long time ago and have been hidden for a long duration. For example, eight update errors affect LLVM 3, which was released about thirteen years ago (Dec 1, 2011).

5.4 Samples of Debug Location Update Errors Found and Fixed by MetaLoc

This section selects and discusses two previously unknown update errors found by MetaLoc and the corresponding determined updates. Because Section 2.2 covers a `Drop` case, this section focuses on the `Preserve` and `Merge` cases.

An Error Fixed with `Preserve`. Figure 6a shows a previously unknown update error found in pass LoopLoadElimination. The pass eliminates redundant load operations within loops by hoisting the load instruction and reusing the loaded values in the loops. To reuse the loaded values in a target loop, the pass inserts a new PHI instruction (recorded by variable `PHI`) into the loop (lines 5-6) to replace the uses of the hoisted load instruction (recorded by variable `Cand.Load`, line 13). The pass does not preserve the debug location of the hoisted load instruction for the newly created PHI instruction, resulting in lost debug locations.

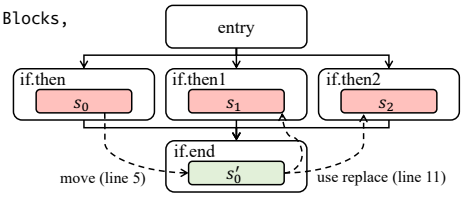
Through control-flow conformance analysis, MetaLoc determines a debug location update using `Preserve`, reports the inconsistency between the determined update and the existing update using `Drop`, and generates the update skeleton at line 14 in Figure 6a. Then, we patch the error with the code concretized from the update skeleton, as shown in line 15.

An Error Fixed with `Merge`. Figure 6b shows a previously unknown error found in pass GVNSink. This pass sinks similar instructions for redundant value computations leveraging Global Value Numbering (GVN). In line 5, the sinking process first chooses one of the similar instructions (recorded by variable `I0`) and moves it to the specific sinking position. Then, the moved instruction is used to replace all the instruction uses of the other similar instructions recorded by the variable `I` in a for loop (line 11). The sunk instruction preserves its original debug location because the movement does not remove or replace one's debug location. Figure 6b gives the control-flow graph

```
1  void propagateStoredValueToLoadUsers(const StoreToLoadForwardingCandidate &Cand,
2                                       SCEVExpander &SEE) {
3    ...
4
5    PHINode *PHI = PHINode::Create(Initial->getType(), 2, "store_forwarded"); // Creating
6    PHI->insertBefore(L->getHeader()->begin());
7    PHI->addIncoming(Initial, PH);
8
9    ...
10
11   PHI->addIncoming(StoreValue, L->getLoopLatch());
12
13   Cand.Load->replaceAllUsesWith(PHI); // Use replacing
14   // Update Skeleton: Line 481 Preserve(PHI, Cond.Load)
15 + PHI->setDebugLoc(Cond.Load->getDebugLoc()); // Patch code
16 }
```

(a) The previously unknown error found in LoopLoadElimination, which is fixed with the `Preserve` operation (line 15), *i.e.*, preserving the debug location of the instruction pointed by `Cond.Load` to that pointed by `PHI`.

```
1  void GVNSink::sinkLastInstruction(ArrayRef<BasicBlock *> Blocks,
2                                    BasicBlock *BBEnd) {
3    ...
4    // Moving
5    I0->moveBefore(&*BBEnd->getFirstInsertionPt());
6
7    ...
8
9    for (auto *I : Insts)
10     if (I != I0) {
11       I->replaceAllUsesWith(I0); // Use replacing
12       // Update Skeleton: Line 925 Merge(I0, I, ...)
13 +     I0->applyMergedLocation(I0->getDebugLoc(), I->getDebugLoc()); // Patch code
14     }
15   ...
16 }
```



(b) The previously unknown error found in GVNSink. The error is fixed with the `Merge` operation (line 13), *i.e.*, merging the debug locations of instructions pointed by `I` and `I0`.

Fig. 6. Two sample debug location update errors found and fixed by MetaLoc in the LLVM optimizations.

to illustrate the sinking process. In the graph, the instruction s_0 (recorded by `I0`) in block `if.then` is moved to block `if.end` (denoted by s'_0) and then replaces the uses of instruction s_1 (recorded by `I`) in block `if.then1` and s_2 in `if.then2`. However, preserving only one of the instructions' debug locations leads to an incorrect debug location bug. For example, when debugging, if the program execution path enters `if.end` from `if.then1`, the debugger prints the debug location of s'_0, which points to the unreachable block `if.then`, misleading the users.

Through control-flow conformance analysis, MetaLoc determines an update using `Merge`. Then, it reports the inconsistency and generates the update skeleton shown in line 12. Concretizing from the update skeleton, we patch the error with the code in line 13.

5.5 Discussions

Limitations of MetaLoc. First, our approach under-approximates the overall debug location update problem in optimizing compilers. It relies on instruction manipulations to capture source and destination instructions, which form the basis of control-flow conformance analysis. However, when optimizations rebuild the IR (*e.g.*, code transformation via `IRBuilder` APIs), our approach cannot identify the source instructions for debug location updates. Nevertheless, the four instruction manipulations (Table 1) are general in compiler optimizations and 54 LLVM optimization passes use these manipulations and the corresponding APIs. Second, due to control-flow modifications during optimization, our control-flow conformance analysis is flow-insensitive (*i.e.*, an over-approximation) by design, which may result in incorrect debug location updates using `Drop`. Third, not all LLVM APIs that align with our selected kinds of instruction manipulations are covered in the implementation.

Currently, we support APIs in the `Instruction` class and its subclasses, as they perform dedicated manipulations without side effects. However, APIs and developer-defined wrappers that align with the manipulations but have side effects are excluded. For example, the `Create` APIs in `IRBuilder` can both create instructions and assign debug locations simultaneously. Though these limitations may lead to missed or incorrect debug location updates in theory, the evaluation in Section 5.2 shows that MetaLoc achieves high accuracy, correctly determining debug location updates in 30 out of 31 cases. In addition, as shown in Section 5.3, MetaLoc did not produce false positives in our experiments, and all reported patches are correct and have been accepted into LLVM.

MetaLoc's Generality. Our work introduces the first LLVM-based approach to inferring debug location update principles through control-flow analysis, *i.e.*, determining the proper updates via control-flow conformance analysis. The concept of instruction manipulations is general, making our approach applicable to other CFG-based IR systems that utilize these manipulations. For instance, it could potentially be adapted for GCC, which also employs a CFG-based IR and a similar debug location maintenance mechanism. However, GCC's compilation pipeline and IR structure differ from LLVM's. We anticipate that applying our technique to GCC's implementation would require considerable effort.

Limitations of the Existing Debug Location Maintenance in LLVM. In LLVM IR, debug locations must be attached to instructions, and one instruction can only have one debug location attached. These limitations result in a tradeoff between correctness and completeness of debug locations in optimized code. While ensuring the correctness of the debug locations in optimized programs, dropping debug locations leads to debug location loss, compromising completeness. When merging instructions, preserving any debug location of the source instructions could invalidate the debug information. LLVM developers balance this tradeoff by retaining the debug locations of instructions that may cause crashes (*e.g.*, memory instructions such as store instructions) for better postmortem analysis, the retained debug locations may be misleading in debugging.

Completeness of Debug Location Information. Ideally, the debug information in the optimized programs should be correct and complete. In this work, we focus on determining correct debug location updates to ensure the debug locations in the optimized program meet the conformance relation, *i.e.*, the correctness. However, ensuring the completeness of the debug locations is inherently more challenging, due to the discussed limitations of its maintenance in LLVM. Some degree of debug location loss is inevitable after optimization. Furthermore, designing an oracle to directly detect debug location loss caused by incorrect updates is difficult, as optimizations themselves can introduce loss. Our work retains debug locations that are unlikely to lead to misleading debug information to improve completeness.

Refinement to LLVM Documentation. In addition to the patches to update errors in LLVM, we have also patched the LLVM documentation on updating debug information. Specifically, we have fixed an incorrect update example and refined an inaccurate example. These two examples are in the section "*When to merge instruction locations*" and are not consistent with our control-flow conformance. The incorrect update example suggests that developers should drop the debug location when hoisting or sinking identical instructions. However, the `Drop` operation should be only used when retaining the debug location which violates the conformance relation of debug location. The inaccurate example only suggests the `Merge` operation for hoisting or sinking identical instructions from "*both sides of a CFG diamond*". We refine this example by complementing the example description: "*Hoisting identical instructions from all successors of a conditional branch or sinking those from all paths to a post-dominating block. ... For each group of identical instructions being hoisted/sunk, the merge of all their locations should be applied to the merged instruction.*"

Instrumentation Overhead. The instrumentation overhead in our work primarily stems from the control-flow conformance analysis, which involves CFG traversals to collect debug locations

and debug location set comparisons. The overhead is negligible (running the `opt` command with the original and instrumented pass both take approximately 15 milliseconds per test case), as the instrumented pass only processes a small LLVM IR snippet per run.

6 Related Work

Debug Information Testing. Ensuring the accuracy and reliability of debug information in optimized programs is crucial to improving the usability and user experience of modern debugging utilities [Stinnett and Kell 2024]. Since the first work [Hennessy 1982] that introduced techniques to recover variable information in source code from optimized programs, how to debug optimized programs has been extensively studied [Adl-Tabatabai and Gross 1996; Copperman 1994]. Despite these efforts, incorrect or lost debug information still frustrates software developers. The seminal work [Li et al. 2020] proposes the first testing framework and introduces actionable programs for detecting incorrect debug value information by inspecting debugger-printed values. Debug^2 [Di Luna et al. 2021] generalizes the prior work to testing the correctness of both debug values (*i.e.*, variable information) and locations by four designed invariants. Instead of focusing on the correctness of debug information, conjecture-based oracles [Assaiante et al. 2023] are proposed to test the completeness of debug variable information, *i.e.*, finding lost debug variable information. The conjectures are empirically derived, which tell when a variable should be presented by the debugger. All of these prior works use black-box testing techniques and check the consistency of the debugger-interpreted debug information. Different from these works, our approach analyzes the code of the optimization implementations to determine the correct debug location updates. Therefore, our approach can pinpoint debug location update errors in the optimization implementation and also help fix these errors by generating update skeletons.

Automated Program Repair. In recent years, APR techniques have been proposed to reduce bug-fixing efforts. Some of the APR techniques target bugs in particular scenarios. ProveNFix [Song et al. 2024] utilizes temporal property to deal with temporal bugs, such as memory leaks, unchecked return values, and double-free. FootPatch [van Tonder and Le Goues 2018] relies on separation logic to fix bugs related to resource release, freeing memory, and null pointer dereferences. Other general APR techniques usually employ a variety of methodologies, such as search-based [Le Goues et al. 2011], constraint-based [Nguyen et al. 2013], template-based [Liu et al. 2019], and learning-based [Zhang et al. 2023] approaches, to fix general types of bugs. To our knowledge, none of the existing techniques can fix debug location update errors in LLVM. Existing general APR techniques are based on the generate-and-validate paradigm. They typically rely on a test suite with at least one failing test case to perform bug localization and validate the correctness of the generated patch. As a result, APR techniques primarily address existing bugs. In contrast, our approach identifies new debug information bugs by determining the correct debug location updates. These updates serve as a reference, enabling the detection and fixing of potential update errors in LLVM.

7 Conclusion

This paper has proposed the first technique for robustifying debug location updates in LLVM via a novel approach called control-flow conformance analysis. For a given LLVM optimization, the technique automatically constructs debug location updates and then uses them to pinpoint the update errors and suggest ready-for-use fixing patches. We have realized this technique as a tool named MetaLoc. With MetaLoc, we have found and fixed 46 previously unknown update errors in the latest version of LLVM. All the patches, along with 22 new regression tests, have been accepted and merged into LLVM. Informed by our approach, we have also uncovered and patched two issues in LLVM's official documentation on updating debug information. Our work provides a new perspective on improving the reliability of debug information updates in LLVM.

Acknowledgments

We thank the anonymous PLDI reviewers and the shepherd for their valuable feedback. This work was supported in part by National Key Research and Development Program (Grant 2022YFB3104002), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100. This work is also supported by the United States National Science Foundation (NSF) under grants No. 2114627 and No. 2237440. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

Artifact

The data and implementation referenced in this paper have been persistently archived [Shan et al. 2025]. The latest version of MetaLoc is also made publicly available on *https://github.com/ecnusse/MetaLoc*.

References

Ali-Reza Adl-Tabatabai and Thomas Gross. 1996. Source-level debugging of scalar optimized code. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 33–43. doi:10.1145/231379.231388

Sanjeev Kumar Aggarwal and M. Sarath Kumar. 2002. Debuggers for Programming Languages. In *The Compiler Design Handbook*. 295–382.

Cristian Assaiante, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, and Leonardo Querzoni. 2023. Where Did My Variable Go? Poking Holes in Incomplete Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 935–947. doi:10.1145/3575693.3575720

John Calcote. 2010. *Autotools: A Practioner's Guide to GNU Autoconf, Automake, and Libtool* (1st ed.). No Starch Press, USA.

Max Copperman. 1994. Debugging optimized code without being misled. *ACM Trans. Program. Lang. Syst.* 16, 3 (may 1994), 387–427. doi:10.1145/177492.177517

Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. 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 (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1034–1045. doi:10.1145/3445814.3446695

GDB Developers. 2024. *GDB: The GNU Project Debugger*. Retrieved 2024-7 from https://sourceware.org/gdb/

Gentoo Authors. 2024. *Project:Quality Assurance/Backtraces*. Retrieved 2024-8 from https://wiki.gentoo.org/wiki/Project:Quality_Assurance/Backtraces

John Hennessy. 1982. Symbolic Debugging of Optimized Code. *ACM Trans. Program. Lang. Syst.* 4, 3 (jul 1982), 323–344. doi:10.1145/357172.357173

Vedant Kumar. 2020. *How to Update Debug Info: A Guide for LLVM Pass Authors*. Retrieved 2024-2 from https://llvm.org/docs/HowToUpdateDebugInfo.html

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1052–1065. doi:10.1145/3385412.3386020

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 31–42.

LLVM Developers. 2023. *Incorrect debug info generated at -O3*. Retrieved 2024-8 from https://github.com/llvm/llvm-project/issues/68898#issuecomment-1760446717

LLVM Project. 2024a. *DILocation - LLVM Language Reference*. Retrieved 2024-7 from https://llvm.org/docs/LangRef.html#dilocation

LLVM Project. 2024b. *LLVM's Analysis and Transform Passes*. Retrieved 2024-10 from https://llvm.org/docs/Passes.html

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

Diego Novillo. 2014. SamplePGO - The Power of Profile Guided Optimizations without the Usability Burden. In *2014 LLVM Compiler Infrastructure in HPC*. 22–28. doi:10.1109/LLVM-HPC.2014.8

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

Huang Shan, Liang Jingjing, Su Ting, and Zhang Qirun. 2025. *Artifact for MetaLoc – Robustifying Debug Location Updates in LLVM via Control-Flow Conformance Analysis*. doi:10.5281/zenodo.15023927

Yahui Song, Xiang Gao, Wenhua Li, Wei-Ngan Chin, and Abhik Roychoudhury. 2024. ProveNFix: Temporal Property-Guided Program Repair. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 226–248.

J. Ryan Stinnett and Stephen Kell. 2024. Accurate Coverage Metrics for Compiler-Generated Debugging Information. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction* (Edinburgh, United Kingdom) *(CC 2024)*. Association for Computing Machinery, New York, NY, USA, 126–136. doi:10.1145/3640537.3641578

The LLDB Team. 2024. *The LLDB Debugger*. Retrieved 2024-2 from https://lldb.llvm.org

Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*. 151–162.

Vedant, Kumar. 2017. *The debugify utility pass*. Retrieved 2024-9 from https://llvm.org/docs/HowToUpdateDebugInfo.html#the-debugify-utility-pass

Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–69.

Received 2024-11-15; accepted 2025-03-06