Refined Input, Degraded Output: The Counterintuitive World of Compiler Behavior

THEODOROS THEODORIDIS, ETH Zurich, Switzerland
ZHENDONG SU, ETH Zurich, Switzerland

To optimize a program, a compiler needs precise information about it. Significant effort is dedicated to improving the ability of compilers to analyze programs, with the expectation that more information results in better optimization. But this assumption does not always hold: due to unexpected interactions between compiler components and phase ordering issues, sometimes more information leads to worse optimization. This can lead to wasted research and engineering effort whenever compilers cannot efficiently leverage additional information. In this work, we systematically examine the extent to which additional information can be detrimental to compilers. We consider two types of information: dead code, *i.e.*, whether a program location is unreachable, and value ranges, *i.e.*, the possible values a variable can take at a specific program location. Given a seed program, we refine it with additional information and check whether this degrades the output. Based on this approach, we develop a fully automated and effective testing method for identifying such issues, and through an extensive evaluation and analysis, we quantify their existence and prevalence in widely used compilers. In particular, we have reported 59 cases in GCC and LLVM, of which 55 have been confirmed or fixed so far, highlighting the practical relevance and value of our findings. This work's fresh perspective opens up a new direction in understanding and improving compilers.

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

Additional Key Words and Phrases: Missed Compiler Optimizations, Automated Compiler Testing

ACM Reference Format:
Theodoros Theodoridis and Zhendong Su. 2024. Refined Input, Degraded Output: The Counterintuitive World of Compiler Behavior. *Proc. ACM Program. Lang.* 8, PLDI, Article 174 (June 2024), 21 pages. https://doi.org/10.1145/3656404

1 INTRODUCTION

Compilers are complex systems with hundreds of "moving parts". An optimizing compiler must simultaneously understand the semantics of an input program using many different analyses [9, 14, 15, 21], and transform it into an equivalent program that is more efficient to execute. This process requires hundreds or even thousands of interdependent transformation steps [1, 2, 25]. Naturally, a compiler may fail to properly optimize a program, *e.g.*, due to phase ordering issues [33], unexpected interactions between its components [7], or missing analyses and optimizations [3, 6].

However, as users of a compiler, we expect it to behave consistently. If, for example, it can optimize a code snippet well in one program, we expect similar results if the same snippet occurs in another program. Similarly, a newer compiler version should optimize a given program at least as well as an older version of the same compiler. Furthermore, we expect that a compiler does better given additional program information, similarly to Listing 1 where GCC generates better code by leveraging the given hint.

Authors' addresses: Theodoros Theodoridis, ETH Zurich, Zurich, Switzerland, theodoros.theodoridis@inf.ethz.ch; Zhendong Su, ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.

ACM 2475-1421/2024/6-ART174
https://doi.org/10.1145/3656404

```
int foo(int x,
        int y) {
 if (x == y)
    return 0;
 if (x > y)
    return 1;
 else
    return -1;
}
int bar(int x,
        int y) {
 return foo(x, y);
}
```

(a) Original code

```
bar:
xorl     %eax, %eax
cmpl     %esi, %edi
je       .L6   #x == y
setg     %al   #x > y
movzbl   %al, %eax
leal  -1(%rax,%rax),%eax
.L6:
ret
```

(b) Generated ASM

```
int bar(int x, int y){
 // We know that
 // x > = y and
 // make it obvious
 // to the compiler
 if (!(x >= y))
   __builtin_unreachable();
 return foo(x, y);
}
```

(c) Refined code

```
bar:
xorl     %eax, %eax
cmpl     %esi, %edi
setne    %al #x < y
ret
```

(d) Refined ASM

Listing 1. Consistent behavior example: GCC 13 -O3 generates simpler code given the additional information. In the original assembly code (Listing 1b) two conditional instructions are used to determine the return value `je .L6` and `setg .L7`: (1) `je .L6` implements the `x == y` check by jumping to the return statement if the previous comparison result (`cmpl %esi, %edi`) is "equal", the return value is 0 (which set by initially `xor`-ing `%eax` with itself), (2) `setg .L7` implements the `x > y` check by setting the return value to 1 if the previous comparison result is "greater". In the refined assembly (Listing 1d) there is only one conditional: `setne %al` which sets the return value to 1 if the previous comparison result is "not equal".

Compilers, however, often defy our expectations—their behavior can be inconsistent. For example, a loop's source code "form" can drastically affect the generated code [13, 20] and, counterintuitively, auto-vectorizers sometimes generate better code given less information [27]. Even seemingly insignificant changes such as swapping the order of independent statements can lead to unexpected differences in the generated code. Moreover, compilers are frequently unable to use additional information hints provided by programmers [10], and improving the strength of analyses has sometimes little impact on optimizations [22].

The inconsistent and unpredictable behavior of compilers can be a major obstacle in compiler research and development. Compiler developers are aware of these issues and rely on bug reports or continuous benchmarking to identify them, but the scope of these efforts is limited. Previous work has focused on automatically identifying missed optimizations [3, 18, 29, 31]. None, however, has systematically studied this inconsistent behavior of compilers, *i.e.*, the phenomenon where *more information* about a program's semantics causes a compiler to generate *worse* code. Techniques and tools are necessary for finding such *optimization inconsistencies*, both to help with understanding the unexpected interactions between compiler components, but also to identify these issues and fix the missed optimizations, analysis weaknesses, and unexpected interactions that cause them.

To this end, this work develops a general approach for finding optimization inconsistencies in compilers. Our core idea is to (1) refine an input program by adding additional information about its semantics without affecting its runtime behavior, and (2) check whether the compiler generates worse code for the refined program. For example, in Listing 1c we explicitly "tell" the compiler that `x >= y`: GCC is consistent in this case, it generates better code for the refined version (Listing 1d). Another example of adding information would be to annotate pointers with the `restrict` keyword, making it obvious that they do not alias. In general, we expect that a compiler should be able to optimize a refined program at least as well as the original one; otherwise, we have identified an optimization inconsistency.

Note that our work is orthogonal to finding missed optimizations such as the aforementioned efforts [3, 29–31]. Detection oracles like DCE Markers [31], `optdiff` checkers [3], or identifying redundant memory operations [29] aim at reliably finding missed compiler optimizations. In contrast, our work investigates the novel issue of how additional information about a program's behavior can negatively impact a compiler, leading to inferior code generation. Furthermore, as demonstrated in our evaluation (Section 4.8), our approach can uncover such issues much more effectively than merely using a missed optimization detection oracle.

We refine programs with two kinds of information: (1) dead code information, *i.e.*, by explicitly annotating dead locations as unreachable, and (2) value ranges, *i.e.*, by making the bounds of variable values on specific program locations explicit. Deriving this information is generally nontrivial, but we focus on closed (*i.e.*, taking no input) and deterministic programs where this task is straightforward (Section 3.2); such programs are often used in automated compiler testing [4, 23, 34]. Our approach, however, is general and extendable to other kinds of information (Section 4.9).

We determine if the compiler is consistent between an original and refined program by using an *oracle*. Such an oracle, given a compiler C, an original program P, and the refined one P', determines if the compilation result on the latter is degraded. Note that the oracle is not limited to the compiler's output, but it can also consider other information such as the compiler's internal representation, diagnostics, or the compiled program's runtime behavior. Our approach can be instantiated with a number of different oracles. In this work, we use three: (1) the size of the generated code, *i.e.*, whether refining a program leads to a significant binary size increase, (2) the number of Dead Code Elimination (DCE) markers [31], *i.e.*, whether refining a program leads to less eliminated dead code, and (3) the precision of value range analysis, *i.e.*, whether refining a program leads to less precise value range results. Our approach is described in detail in Section 3.

Our empirical analysis demonstrates the usefulness and applicability of our approach in uncovering a wide range of optimization inconsistencies (Section 4). We reported 40 GCC and 19 LLVM cases, out of which 39 and 16 respectively have been confirmed/fixed. We also analyzed 89 GCC and 69 LLVM unique regressions, *i.e.*, cases where a previous compiler version was consistent. These regressions were caused by changes in 18 GCC and 16 LLVM components, including, among others, alias analysis, control flow graph transformations, constant propagation, global value number, jump threading, loop transformations, peephole optimizations, value range analysis, *etc.*. Overall, this demonstrates that our technique is highly effective in finding a wide range of optimization inconsistencies, unexpected interactions between compiler components, and missed optimizations in state-of-the-art compilers. Our key contributions are:

- Formulating the optimization inconsistency problem in compilers and an automated approach for finding them;
- An implementation of our approach which we used to find and report a wide range of issues to compiler developers; and
- A systematic and quantitative study of optimization inconsistencies in GCC and LLVM.

2 AN OPTIMIZATION INCONSISTENCY EXAMPLE

We start with an example demonstrating an optimization inconsistency (Listing 2). When compiling the original program in Listing 2a with the current development version of GCC[1] at `-Os`, the compiler is able to optimize away all control flow and simplify the code, as shown in Listing 2b. To uncover the optimization inconsistency in this example, we must refine the program with additional information; the refinement is done in a semantics preserving way, *i.e.*, the refined program must have the same behavior as the original one. In this work, we refine programs by explicitly annotating dead code as unreachable and by making the ranges of variable values at specific program locations explicit.

[1]Revision `r14-5021-g94c0b26f454`

In this particular example, the inconsistency is triggered by refining function `h`: `h` is called 13 times in `main`'s outer loop and the values of its parameter, `j`, range from 1 to 65535. We "inject" this information via a conditional call to `__builtin_unreachable()` [24], as shown in Listing 2c. This annotation informs the compiler that `j`'s values can never be outside the range $[1, 65535]$. The compiler's output on the refined program, shown in Listing 2d, is significantly longer and more complex than the original one. The compiler's behavior is unexpected: it can fully optimize the original program, but it fails with the refined one, even though the only difference between the two is the constrained range of `j`'s values. This is an example of optimization inconsistency where refining a program with additional information leads to an unexpected degradation in the generated

```c
static struct { int a; int b; } c;
static int d, e, g;
static short f, i;
static void h(unsigned short j) {
  c.a = i;
}
int main() { d = 6;
  for (; d != -7; d--) {
    h(d ^ d < (0 <= 6));
    if (d >= 12) if (e) {
        for (; f; ++f) g = c.b;
        if (g) e = 0;
    }
  }
}
```

(a) Original code

```
main:
   movl   $-7, d(%rip)
   xorl   %eax, %eax
   movl   %eax, c(%rip)
   xorl   %eax, %eax
   ret
```

(b) Original ASM

```c
// The rest of the code is unchanged
static void h(unsigned short j) {
 c.a = i;
 if (!((j >= 1) && (j <= 65535)))
     __builtin_unreachable();
}
```

(c) Refined code

```
main:                      |  .L4:                      |  .L32:                     |  .L12:
  movl  e(%rip), %ecx      |    testb %dil, %dil        |    testl %esi, %esi        |    testb %r9b, %r9b
  movl  $6, %eax           |    je    .L5               |    je    .L6               |    je    .L20
  xorl  %edi, %edi         |    movl  %ecx, e(%rip)     |    movb  $1, %dil          |    movw  %dx, f(%rip)
  xorl  %r8d, %r8d         |  .L5:                      |    xorl  %ecx, %ecx        |    movl  %esi, g(%rip)
  movl  c+4(%rip),%r10d    |    xorl  %ecx, %ecx        |  .L6:                      |    jmp   .L20
  movl  g(%rip), %esi      |    movw  %dx, f(%rip)      |    decl  %eax              |  .L10:
  xorl  %r9d, %r9d         |    movl  %ecx, c(%rip)     |    movb  $1, %r8b          |    testb %dil, %dil
  movl  $6, d(%rip)        |    movl  %esi, g(%rip)     |    jmp   .L1               |    je    .L12
  movw  f(%rip), %dx       |  .L3:                      |  .L31:                     |    movl  %ecx, e(%rip)
.L2:                       |    cmpl  $11, %eax         |    testb %r8b, %r8b        |    jmp   .L12
  cmpl  $-7, %eax          |    jle   .L6               |    je    .L10              |  .L20:
  je    .L31               |    testl %ecx, %ecx        |    movl  $-7, d(%rip)      |    xorl  %eax, %eax
  xorl  %r11d, %r11d       |    je    .L6               |    testb %dil, %dil        |    ret
  testl %eax, %eax         |  .L7:                      |    je    .L11              |
  setle %r11b              |    testw %dx, %dx          |    movl  %ecx, e(%rip)     |
  cmpw  %ax, %r11w         |    je    .L32              |  .L11:                     |
  jne   .L3                |    incl  %edx              |    xorl  %eax, %eax        |
  testb %r8b, %r8b         |    movl  %r10d, %esi       |    movl  %eax, c(%rip)     |
  je    .L4                |    movb  $1, %r9b          |
  movl  %eax, d(%rip)      |    jmp   .L7               |
```

(d) Refined ASM

Listing 2. Optimization inconsistency example: the current upstream version of GCC -Os optimizes away all control flow in the original program. Refining the program with additional information, leads to an unexpected behavior: the compiler fails to optimize the refined program and generates significantly more complex code. Adapted from bug report: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=112545

code. Note that existing (differential) testing based approaches [3, 29, 31] cannot detect this issue, as it only manifests in the refined program and not the original one. Given the standardization of the ability to “inject” information similarly to Listing 2 [11], optimization inconsistencies will likely become an even more prevalent issue in the future.

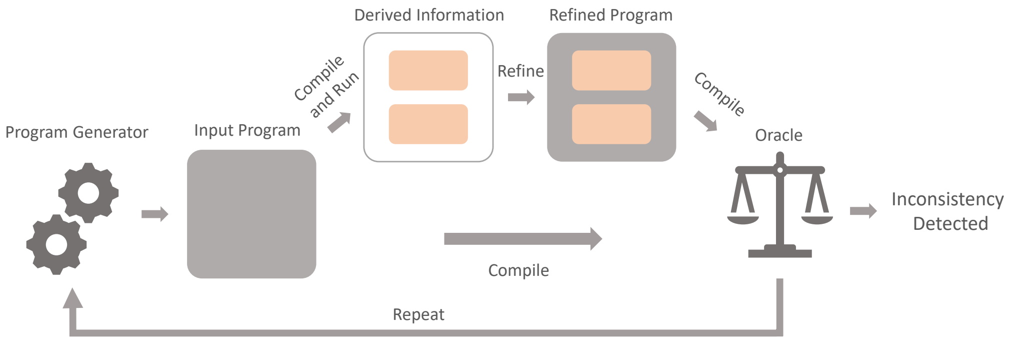


Fig. 1. Overview of our approach.

3 DETECTING OPTIMIZATION INCONSISTENCIES

This section introduces our approach for detecting optimization inconsistencies in compilers and describes our end-to-end automated testing implementation.

3.1 High-level Overview and Background

Our technique (Figure 1) detects cases where a compiler can optimize an input program, but fails with a refined version, which should be easier to optimize. Our core idea is the following: given an input program P, we can run it and observe parts of its behavior, *e.g.*, what the bounds of variable values are on specific program locations; we call this information I. If we “inject” I into P, we get a refined version, P'. Our expectation is that a compiler C should be able to optimize P' at least as well as P. If not, then we have identified an optimization inconsistency.

We refine programs with two kinds of additional information: (1) dead code information, *i.e.*, which parts of the code are unreachable, and (2) value ranges, *i.e.*, what the bounds of variable values are on specific program locations. We can efficiently derive these for closed and deterministic programs which are typically used for compiler testing [34] (Section 3.2). To refine a program we “inject” the derived information using `__builtin_unreachable` (Section 3.3). To detect inconsistencies, we need an *oracle* that compares a compiler’s output on the original, P, and refined, P', programs. In this work we test three different oracles in our end-to-end implementation (Section 3.4).

Program Generation. We use a standard C program generator, Csmith [34], to produce candidate test programs. Csmith generates closed and deterministic programs, which are suitable for our approach. However, our technique is not limited to Csmith, alternative generators such as YarpGen [22] can also be used.

Injecting Additional Information. Compilers offer multiple generic ways for providing additional information about the compiled program; we use the `__builtin_unreachable()` extension [24]. This extension indicates that a given location is unreachable, and the compiler can use this information to better optimize the input program. We can inject any piece of information that can be written as an expression using this builtin: `if (!(EXPR)) __builtin_unreachable()`. For example, `if (!(a == 0)) __builtin_unreachable()` tells the compiler that `a` is always zero at this program location. Other alternatives include:

- LLVM's `__builtin_assume(EXPR)` builtin function, *e.g.*, `__builtin_assume(a == 0)` tells the compiler that the expression `a` is always zero. This is similar to using: `if (!(EXPR)) __builtin_unreachable()`.
- GCC 13 also implements a similar extension: `__attribute__((assume(EXPR)))`.
- C++23's `[[assume(EXPR)]];` [11].

We wanted a solution that is available in older versions of GCC and LLVM for our evaluation, thus we chose `__builtin_unreachable()`.

Detection Oracles. To detect optimization inconsistencies, we must compare the compiler outputs on the original and refined programs. To this end, we need an *oracle* that determines whether or not the compiler is consistent. Our approach can be instantiated with any oracle that can compare the compiler's behavior outputs on the two inputs. For example, an oracle based on optdiff checkers [3] can detect inconsistencies directly in the generated assembly outputs. Another possibility is to target the precision of static analyses and whether they deteriorate on the refined programs [30]. One can also choose to compare the runtime behavior of the original and refined programs, *e.g.*, via CIDetector [29] to determine if the refined program contains more redundant (useless) operations. In this work, we utilize three oracles: significant binary size increase, DCE markers [31], and Value Range Analysis precision degradation; the three oracles are described in Section 3.4.

End-to-end Testing. Our implementation also uses the following tools:

- C-Reduce [23] in combination with sanitizers [26], to reduce test cases before reporting them to compiler developers.
- `git bisect` [8], to bisect regressions and identify the changes in compilers that introduced the detected inconsistent behavior.

3.2 Extracting Additional Program Information

We extract additional program information by running the input programs and observing their behavior. Since we test closed and deterministic programs, a single execution is enough to determine all possible behaviors. We track two kinds of information: (1) dead code information, *i.e.*, for a given program branch (*e.g.*, an if-statement or a loop) we identify if it is never executed, and (2) the value ranges of variables at specific program locations, *i.e.*, for a program location p and a variable v, we determine v's lower and upper bounds: $v \in [lb, ub]$ at p. Listing 3a shows an instrumented version of Listing 2a with branch and value range tracking. The two macros `TRACK_BRANCH` and `TRACK_VALUES` record the program's behavior; the former tracks whether branches are executed and the latter tracks the runtime values of variables. The tracking data is printed at the end of the program's execution, as shown in Listing 3b.

Extracting Dead Code Information. We track all branches during program execution, the dead branches are the ones that are never reached. For example, in Listing 3a all branches are instrumented with `TRACK_BRANCH(ID)`. After executing the program, we can determine that only `Branch0` is alive, as shown in Listing 3b; the remaining branches are dead.

Extracting Value Ranges. We extract value ranges by tracking the lower and upper runtime variable values throughout a program's execution. For example, in Listing 3a, we track the values of `j` in function `h` with `TRACK_VALUES(7, j, i)`; the tracked value range is $[1, 65535]$ (Listing 3b).

Given a variable `v`, we insert annotations before each statement that uses `v` to track its value. Barring a few exceptions (*e.g.*, variables that might be uninitialized at particular program locations), we use the following procedure: for each program statement S, and the set of local variables V used by S, track each variable in V immediately before S. Note that this is done recursively, *e.g.*, in `if(C) STMT(v);` we would track `v` both before `STMT(v)` and before `if(C)`. We use unique identifiers for

```
static void h(unsigned short j) {
 TRACK_VALUES(7, j, i);
 c.a = i; }
int main() { d = 6;
 TRACK_VALUES(0, d, e, f, g, c.b);
 for (; d != -7; d--) { TRACK_BRANCH(0);
  TRACK_VALUES(1, d);
  h(d ^ d < (0 <= 6));
  TRACK_VALUES(2, d);
  if (d >= 12) { TRACK_BRANCH(1);
   TRACK_VALUES(3, e);
   if (e) { TRACK_BRANCH(2);
    TRACK_VALUES(4, f, c.b);
    for (; f; ++f) { TRACK_BRANCH(3);
     TRACK_VALUES(5, c.b);
     g = c.b; }
    TRACK_VALUES(6, g);
    if (g) { TRACK_BRANCH(4); e = 0;}}}}}
```

(a) Instrumented program

```
Values0: d[6,6],
         e[0,0],
         f[0,0],
         g[0,0],
         c.b[0,0]

Branch0: Alive

Values1: d[-6,6]

Values2: d[-6,6]

Values7: j[1,65535],
         i[0,0]
```

(b) Tracked Information

Listing 3. Program information tracking example. Listing 3a is an instrumented version of Listing 2a.

each combination of variable (sets) and program location; the collected variable value ranges are always associated with the corresponding location.

Implementation. Calls to TRACK_BRANCH and TRACK_VALUES expand to function calls that record the program's behavior:

- Every invocation of TRACK_BRANCH(ID) records the branch ID as alive.
- Every invocation of TRACK_VALUES(ID, v0, v1, . . .) updates the value ranges of the tracked variables in the program location identified by ID; for each tracked variable v and its value val: if val < lb then lb = val and if val > ub then ub = val. The bounds are undefined initially and are set to the first observed value.

3.3 Refining Programs with Additional Information

Injecting Dead Code Information. Given that a program's branch is dead, we annotate it with a call to __builtin_unreachable(). For example, we know that Branch1 is dead in Listing 3a:

```
for (; d != -7; d--) { //Branch0
  h(d ^ d < (0 <= 6));
  if (d >= 12) { /*Branch1*/ __builtin_unreachable();
    // the rest of the program is unmodified
```

This does not change the program's behavior as this branch is never executed. We do the same for all types of branches (loops, switch statements, *etc.*).

Injecting Value Range Information. Given a known value range, we inform the compiler about it with a conditional: if (!(LB <= x && x <= UB)) __builtin_unreachable(); tells the compiler that `x` $\in$ [`LB`, `UB`] in this program location. In the case of Listing 3a, we know that `j` $\in [1, 65535]$ (`Values7` in Listing 3b); we can express this information by injecting a conditional unreachable annotation as shown in Listing 2c. Note that a call to `__builtin_unreachable` guarded by a value range constraint is not necessarily just exercising a compiler's ability to eliminate dead code. For example, in Listing 5b the unreachable annotation bounds the values of the induction variable `g` (by constraining variable `h`); however, this information is not relevant to the dead code of this example (and its elimination).

3.4 Detecting Optimization Inconsistencies

To detect optimization inconsistencies, we must compare the compiler outputs on the original and refined programs. In our example of Listing 2a and the refined version of Listing 2c, using binary size as an oracle reveals the issue: the binary size of the refined program is significantly (around 3×) larger than the binary size of the original program. Our approach is general and can be instantiated with a number of different oracles; in this work we implement and test three different ones.

Size Oracle. The size oracle compares the binary sizes of the compiled original and refined programs. If there is a size difference above a threshold (we used 200 bytes), we consider this an optimization inconsistency. We found and reported the example of Listing 2a using this oracle. Note that this oracle is suitable when optimizing for binary size (*e.g.*, using `-Os`), as a drastic size increase is not necessarily an optimization inconsistency indication when optimizing for performance (*e.g.*, using `-O3`). Examples of cases found with this oracle are shown in Listing 2 and Listing 6a.

DCE Oracle. The dead code elimination (DCE) oracle uses DCE markers [31] to identify inconsistencies. These markers identify pieces of dead code a compiler has eliminated. DCE markers can be implemented as function calls, *e.g.*, given an input program with `if (Condition) { DCEMarker(); . . . }`, if the compiler's output does not contain the call to the marker (*i.e.*, `callq DCEMarker`), then the compiler has eliminated the dead code inside the if-statement. A compiler's ability to remove dead code depends on many interactions between its analyses and transformations. Thus, a missed DCE opportunity may indicate another missed optimization or an unexpected interaction between compiler components.

We use the DCE oracle in the following way: a compiler C eliminates the set of DCE markers EM in a program P, and the set EM' in its refined version P', if $m \in EM$ and $m \notin EM'$, then we have identified an optimization inconsistency: C can eliminate m in P but not P', even though P' is a more constrained version of P and should be easier to optimize. Examples of cases found with this oracle are shown in Listing 4 and Listing 5.

VR Oracle. The value range (VR) oracle targets a compiler's ability to infer variable value ranges. We do not directly extract the compiler's inferred ranges, but we estimate them using a compiler agnostic method inspired by DCE markers, which we call *value range markers* (VR Markers):

(1) We use our instrumentation as described in Section 3.2 to identify the value ranges of variables at specific program locations.
(2) We add markers of the form `if (!(LB <= x && x <= UB)) VRMarker();` at these locations.
(3) We detect inconsistencies in the same way as with the DCE oracle, but using VR markers instead of DCE markers.

The compiler can eliminate a VR marker only if it can infer that the corresponding variable values are in the [`LB`, `UB`] range. An example case found with this oracle is shown in Listing 6b.

Note that the DCE oracle identifies differences in how the compiler handles dead (unreachable) parts of the input program. In contrast, the VR oracle relies on VR markers that are placed in alive (reachable) parts of the program (we cannot measure value ranges of variables in non-executed locations). Also note that if a compiler simply ignores an unreachable directive, none of the oracles would detect an inconsistency, as the compiler's output on the refined program should be the same as the original (*e.g.*, a marker is either eliminated or not eliminated in both).

3.5 End-to-end Practical Realization

Our tool for automatically detecting optimization inconsistencies works in the following steps:

(1) Generate a program P with Csmith [34].
(2) Instrument the program with tracking code and run it to identify its dead branches and precise value ranges. Refine P with this information into P'. If a marker based oracle is used, markers are also inserted in P and P'.
(3) Use the oracle together with compiler C on P and P' to identify an optimization inconsistency. If nothing is found go to step 1.
(4) Given an older version of the compiler, C_{old}, check if this is a regression, *i.e.*, if C_{old} given the additional information does not exhibit the identified inconsistency. If it is a regression, bisect the compiler history to identify the offending commit/change.
(5) Use C-Reduce [34] to reduce the P' to a minimal example that still exhibits the inconsistency (and regression).

Reduction. The last step of our end-to-end procedure is to reduce the program to a minimal one. This minimal program, must still exhibit the inconsistency, *i.e.*, the oracle should detect a difference if dead code or value range information is injected. The reduction is done directly on the refined program P' as follows:

(1) Preprocessing step: we remove from P' unreachable annotations, and markers if using a marker oracle, that are unrelated to the identified issue. We convert the remaining annotations into macros that are enabled and disabled via command line flags, *e.g.*, we would replace a `if (!(LB <= x && x <= UB)) __builtin_unreachable();` with `INJECTION_MACRO`. When compiling P we can use `-DINJECTION_MACRO=""` to disable the information injection and `-DINJECTION_MACRO="if (!(LB <= x && x <= UB)) __builtin_unreachable();"` to enable it.
(2) During each reduction step, we accept or reject a reduced program Q. Note that we obtain the refined version Q' by enabling the unreachable annotation via a command line flag. We accept Q if:
    (a) The macro corresponding to the annotation is still present (if not we cannot obtain Q').
    (b) The oracle still identifies the inconsistency between Q and Q', *i.e.*, there is a significant size increase or a previously eliminated VR/DCE marker becomes missed (depending on which oracle is used).
    (c) If the unreachable annotation uses a value range, we also need to update the lower and upper bounds as they may have changed in Q'. We do this by running the program and printing the corresponding values. We insert the tracking code by defining the macro corresponding to the `INJECTION_MACRO` accordingly. We do the same for VR markers.
    (d) Q passes various correctness tests (*e.g.*, compiler sanitizers).
    (e) If the issue is a regression, Q' does not exhibit the inconsistency with the older compiler.

4 EMPIRICAL ANALYSIS

We evaluate the effectiveness and practical utility of our approach. We first examine the prevalence of optimization inconsistencies in GCC (13.1.1) and LLVM (16.0.4) on a corpus of 10,000 Csmith programs. We then discuss the variation across different optimization levels and compilers and investigate how this prevalence evolves across compiler versions, and we explore the diversity of these issues. Finally, we present examples of the cases that we have reported to compiler developers.

4.1 Research Questions and Result Highlights

We aim to answer the following research questions on optimization inconsistencies:

- **RQ1**: How prevalent are they? (Section 4.3)
- **RQ2**: How do they vary across optimization levels and compilers? (Section 4.4)
- **RQ3**: How long-latent are they? (Section 4.5)
- **RQ4**: Are they caused by changes in a diverse set of compiler components? (Section 4.6)
- **RQ5**: How useful is our approach in practice for compiler development? (Section 4.7)
- **RQ6**: Is refining programs effective at uncovering optimization inconsistencies? (Section 4.8)

Result Summary. Both compilers are affected by optimization inconsistencies: we detect them in 17.00% and 8.90% of the tested programs with GCC and LLVM, respectively. Most of the detected cases manifest in a single compiler, only 21.54% of the affected programs are common. GCC's optimization levels are more diversely affected than LLVM's: 43.71% of the programs with inconsistencies for LLVM affect all of its optimization levels, but only 11.94% for GCC. Several inconsistencies are long-latent, *e.g.*, out of the 1,700 cases detected with GCC 13.1.1, 900 can be traced back to GCC 9.5.0. Similarly, out of the 890 detected with LLVM 16.0.4, 647 can be traced back to LLVM 12.0.1. We also analyzed 89 GCC regressions and 69 LLVM ones: using the commits that introduced them, we find that they are caused by changes in 18 and 16 different compiler components (*e.g.*, alias analysis, loop transformations, peephole optimizations, *etc.*). Finally, we have reported 59 cases to compiler developers, out of which 55 have been confirmed or fixed.

4.2 Evaluation and Implementation Setup

Methodology. For **RQ1-4** and **RQ6**, we use the DCE oracle to identify optimization inconsistencies; note that the goal of our systematic evaluation is to study the prevalence of optimization inconsistencies, but not to compare the oracles, thus we only use one. For **RQ5**, we use the Size, DCE, and VR oracles. For each combination of test program, compiler, and optimization level, we use the procedure described in Section 3.4 to detect issues. In Section 4.3, Section 4.4, and Section 4.5, we report the number of programs that contain at least one issue, the number of additional information entries that result in inconsistencies (*e.g.*, how many branches marked as unreachable or how many injected variable value ranges), and the number of DCE markers through which we can detect them. In Section 4.6, we focus on regressions and bisect the compiler history to identify the offending commits. In Section 4.7, we show examples of reduced cases that we have reported to compiler developers. In Section 4.8, we evaluate the effectiveness of refining programs by comparing our approach with a differential testing approach that does not refine programs.

Test Corpus. We use a corpus of 10,000 C programs generated by Csmith [34] for our empirical analysis. Csmith programs are self-contained and do not require inputs, thus we can compute precise information about them (*e.g.*, dead code and variable value ranges). The median number of eliminated markers per program is 54 for GCC and 55 for LLVM, the maximum is 276 and 276, respectively. We use these markers to detect inconsistencies. The median number of injected pieces of information (dead branches and variable value ranges) per program is 2 for GCC and 2 for LLVM, the maximum is 53 and 56, respectively. Note that the generated programs are not guaranteed to contain dead code. However, the majority does, as by construction, most Csmith programs do.

Implementation. We have implemented our approach using LLVM's LibTooling and a small Python library that handles test case generation, the actual testing, reduction, and bisection by orchestrating the various related programs (the compilers under test, Csmith, C-Reduce, *etc.*).

Table 1. Detected optimization inconsistencies for GCC and LLVM on a corpus of 10,000 Csmith programs. We consider only injected value range information, only injected dead code information, and both kinds.

Opt Lvl	Value Range Info Injected		Dead Code Info Injected		Both	
	GCC	LLVM	GCC	LLVM	GCC	LLVM
# programs with detected optimization inconsistencies						
O1	412	381	243	384	589	628
O2	525	399	730	346	1,043	601
O3	472	397	755	348	1,008	599
Os	493	393	779	359	1,095	607
Union	1,168	627	971	503	1,700	890
# injected information annotations revealing inconsistencies						
O1	787	1,236	529	835	1,316	2,071
O2	1,717	1,427	1,770	776	3,487	2,203
O3	1,722	1,378	1,808	779	3,530	2,157
Os	1,443	1,320	1,813	761	3,256	2,081
Union	4,149	2,406	2,787	1,212	6,936	3,618
# DCE markers originally eliminated → missed						
O1	1,254	1,001	520	1,157	1,656	1,919
O2	1,324	1,005	1,510	1,017	2,565	1,799
O3	1,128	944	1,561	972	2,362	1,697
Os	1,236	963	1,486	1,008	2,573	1,743
Union	3,562	1,753	2,364	1,601	5,342	2,958

Experimental Setup. We used a 64-core AMD Ryzen Threadripper 3990X and Arch Linux (6.3.9 kernel) based system. We tested GCC 13.1.1 and LLVM 16.0.4 at `-O1`, `-O2`, `-O3`, and `-Os`[2] for the experiments in Section 4.3 and Section 4.4. We tested previous major versions up to GCC 9.5.0 and LLVM 12.0.1 for Section 4.5, and we used the upstream versions for Section 4.6 and Section 4.7. All our empirical analysis experiments (10,000 test programs × 2 compilers × 5 versions × 4 optimization levels) took around two weeks, including the time needed for generating the corpus, refining it, testing on all compiler versions and optimization levels, and generating the results.

4.3 Prevalence in GCC and LLVM

We measure the prevalence of optimization inconsistencies in GCC and LLVM on our test corpus. Both compilers at all optimization levels are affected (Table 1):

- We detect issues in 17.00% of the tested programs with GCC and in 8.90% with LLVM.
- The percent of injected facts causing inconsistencies is 2.33% for GCC and 1.96% for LLVM.
- The percentage of markers identifying issues is 0.94% for GCC and 0.54% for LLVM.
- Injecting value range information results in more detected issues than dead code information. The former causes 1,168 programs to have issues with GCC and the latter 971; for LLVM the numbers are 627 and 503. One reason for this difference is that there are typically many more opportunities for value range annotations (due to the many variables a program has) versus dead code ones.

[2]We used the standard optimization levels for our testing. `O0` would unlikely produce interesting results as most optimizations are disabled at `O0`. We omitted `Oz` and `Ofast` because they are not as widely used as the others.

4.4 Variance across Compilers and Optimization Levels

The detected inconsistencies vary both across compilers and optimization levels.

Across Compilers. Most issues are compiler-specific (Figure 2):

- Only 21.54% of the programs with detected inconsistencies is common to both compilers, 58.24% is exclusive to GCC, and 20.23% to LLVM.
- Only 3.68% of the injected information causing inconsistencies affects both compilers, 64.46% affects only GCC, and 31.86% only LLVM.
- Only 4.86% of the markers through which issues are detected are common, 62.63% reveal issues only for GCC, and 32.51% only for LLVM.

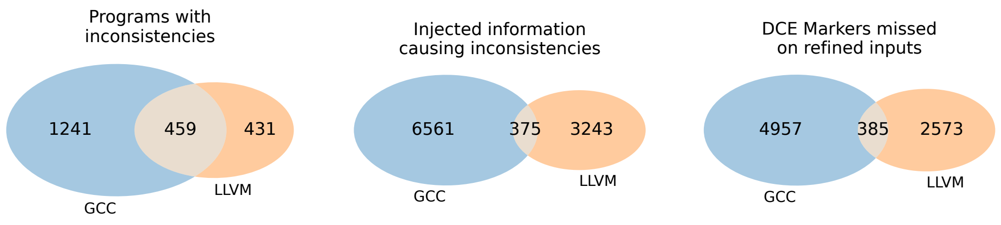
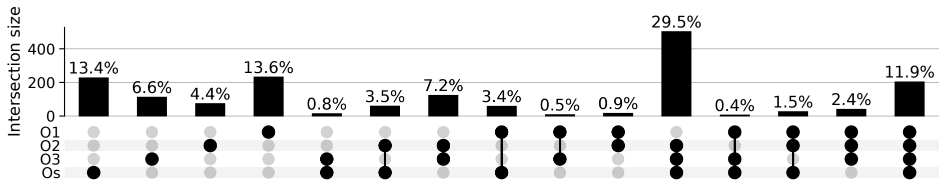
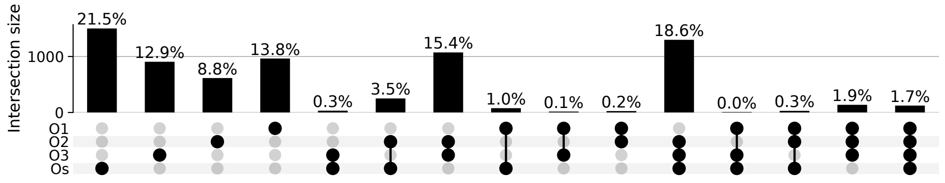


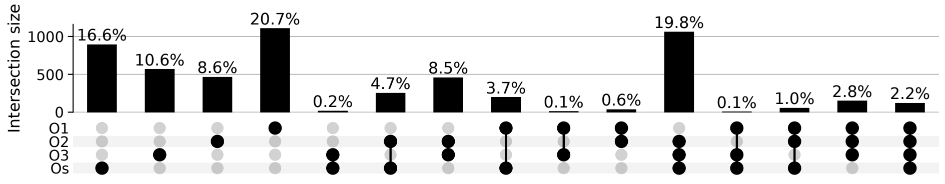
Fig. 2. Most issues are compiler-specific: 78.46% of the affected programs, 96.32% of the issue-causing injected information, and 95.14% of the DCE markers that become missed.



(a) Programs: 37.94% are affected by only a single level

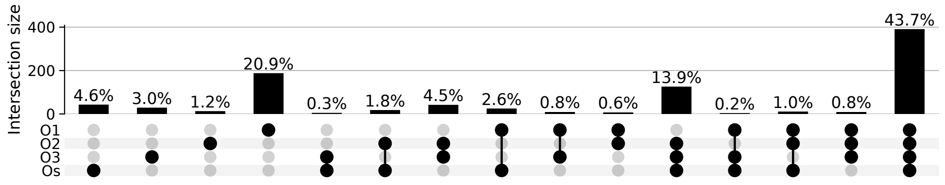


(b) Injected information: 57.01% affects only a single level.

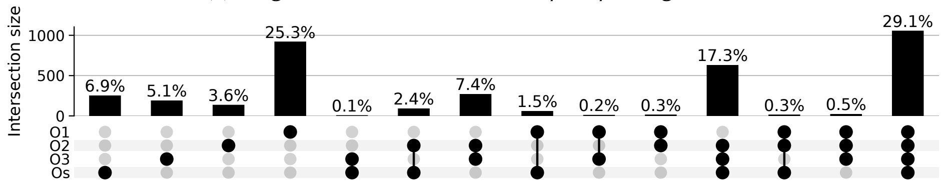


(c) Markers: 56.50% become missed only in a single level.

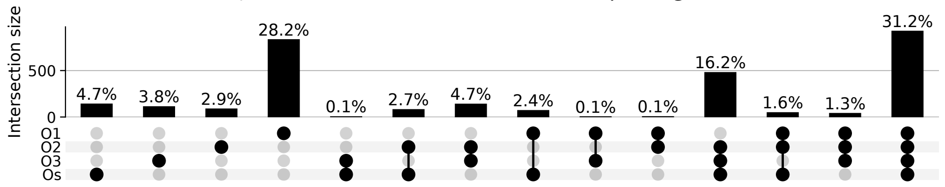
Fig. 3. GCC: Inconsistencies across optimization levels. (Empty sets are omitted.)



(a) Programs: 29.78% are affected by only a single level



(b) Injected information: 40.99% affects only a single level.



(c) Markers: 39.59% become missed only in a single level.

Fig. 4. LLVM:Inconsistencies across optimization levels. (Empty sets are omitted.)

Across Optimization Levels. The variance across optimization levels is different for the two compilers (Figure 3 and Figure 4). In GCC we observe that inconsistencies are more specific to optimization levels than in LLVM. For example, 11.94% of programs with detected issues affect all of GCC's optimization levels but 43.71% affect all of LLVM's. A potential explanation for this difference is that LLVM's optimization pipelines differ less than GCC's.

4.5 How Long-Latent are the Inconsistencies?

We evaluate how the inconsistencies that we detect evolve. To that end, we analyze our test corpus with the last five releases of GCC and LLVM. Several of the detected issues are latent, *i.e.*, they are present in all tested versions (Table 2). For example, 1,700 programs have detected issues with the latest GCC version, of which 900 also have issues in all previous versions. Similarly, for LLVM, we detect 890, 647 of which are latent. Each compiler version also introduced regressions, *e.g.*, the

Table 2. Evolution of optimization inconsistencies over the last five releases of GCC and LLVM. For each compiler we show the number of programs with detected inconsistencies across all optimization levels.

GCC	9.5.0	10.5.0	11.4.0	12.3.0	13.1.1
All	1,383	1,475	1,477	1,345	1,700
Latent	-	1,297	1,253	938	900
Regressions	-	178	51	221	331
Fixed	-	86	55	381	74

LLVM	12.0.1	13.0.1	14.0.6	15.0.7	16.0.4
All	1,884	923	913	892	890
Latent	-	800	738	681	647
Regressions	-	123	43	36	26
Fixed	-	1,084	79	82	61

latest GCC version introduced 331, and LLVM 26. We also observe that several issues are fixed with each version, *e.g.*, GCC 13 fixed 74, and LLVM 16 fixed 61.

4.6 Varied Compiler Components Causing Inconsistencies

The detected inconsistencies are caused by changes in a wide range of compiler components (Table 3). We bisected 89 GCC and 69 LLVM regressions to their offending commits; these include all the unique regressions in our test corpus and the ones found during our testing efforts (Section 4.7). Based on the modified files, we group them into logical compiler components, *e.g.*, alias analysis, loop transformations, peephole optimizations, *etc.*. There are 18 and 16 components touched by these regressions for GCC and LLVM, respectively. This finding highlights our approach's effectiveness at detecting issues in a diverse set of compiler components.

Table 3. Diversity of detected optimization inconsistencies: 18 components in GCC and 16 in LLVM were modified by 89 and 69 regression commits, respectively.

GCC Components	# Commits
Branch Prediction	2
CFG Transformations	9
Constant Propagation	5
Data Dependence Analysis	1
Dead Code Elimination	2
Dead Store Elimination	2
IR Data Structures	3
Inlining	1
Jump Threading	11
Loop Analysis	1
Loop Transformations	12
Pass Management	7
Peephole Optimizations	16
Profile Guided Optimizations	2
Redundancy Elimination	2
Value (Range) Analysis	17
Value (Range) Propagation	12
Value Numbering	3

LLVM Components	# Commits
Alias Analysis	1
Assumption Handling	1
CFG Transformations	4
CSE	1
Dead Store Elimination	1
Dominance based optimizations	1
Escape Analysis	1
Global Value Numbering	2
Interprocedural Analysis	2
Interprocedural Optimization	1
Jump Threading	1
Loop Transformations	11
Pass Management	4
Peephole Optimizations	34
Value (Range) Analysis	9
Value (Range) Propagation	3

4.7 Reported Cases

Table 4. Status of the reported cases

	Confirmed	Fixed	Won't Fix	Unconfirmed	**Total**
GCC	27	12	0	1	**40**
LLVM	13	3	3	0	**19**

We reported 40 optimization inconsistencies for GCC and 19 for LLVM, of which 39 and 16 are confirmed or fixed. Table 4 shows the status of the reported cases. We focused on reporting regressions, *i.e.*, inconsistencies that were not present in previous compiler versions but manifest on the current upstream. This approach enables bisecting those issues to a specific commit; by only reporting cases that bisect to different offending commits, we did not encounter any reported cases marked as duplicates. Each reported case is a minimal example that we have reduced with C-Reduce. Roughly, a reduction typically shrinks an original program with thousands of lines of code to about dozens, a reduction ratio of about 100, and it often finishes under a couple of hours. A few reported issues (3) were marked as *wontfix* by the developers because they were caused by phase ordering issues that are difficult to fix, however, this is a small fraction of the reported cases which can be further triaged and ignored by the compiler developers. Listing 4 and Listing 5 show examples of reported cases which were identified with the DCE oracle for each compiler:

- In Listing 4a, LLVM cannot simplify `cond = x == -11; assume(cond == true); rem = -11 % x; return rem == 0;`, the operants of the modulo operator are essentially constant, but LLVM could not leverage this. (The -11 value is the result of casting 24821 to an 8-bit integer, which is then used in the `h = j % k;` statement.)
- In Listing 4b, the issue is that different LLVM passes (`SimplifyCFG` and `EarlyCSE`) handle type-based alias analysis information differently (because the IR reference does not specify which behavior is correct). The call to dead is no longer eliminated because a recent commit caused a phase ordering change.
- In Listing 5a, GCC's new value range propagation framework could not handle address equality checks followed by `__builtin_unreachable()`. Fixed with `f828503eeb7`.
- In Listing 5b, GCC misses a jump threading opportunity and cannot simplify the following condition: `j = PHI <&i, &c>; cond= j == &c || &i == j;` to `cond = true;`. This issue was caused by a change in how value ranges are stored throughout the compilation pipeline, which in turn had an unforeseen interaction with jump threading.

```
static int a = 24821, d;
static int *b = &a, **c = &b;
static int *e(short f, short g) {
 char h;
 if (f) {
  if (g != 24821) __builtin_unreachable();
  int *i = &d;
  unsigned char j = g, k = f;
  h = j % k;
  if (h) i = 0;
  if (b);
  else __assert_fail();
  if (b || i); else dead();
  }
 return 0;
}
int main() { *c = e(a, a); }
```

(a) https://github.com/llvm/llvm-project/issues/63330 : LLVM `-O2` could not simplify the expression `h = j % k` even though the RHS arguments are known constants. Fixed with `7cfc82f`.

```
static int a, d, *b, *e = &a;
static int **c = &b, ***f = &c;
static int *h() { *c = e;
  if ((10^a) == 0) *b = 0;
  else return 0;
  if (b == &a || b == 0) {
    if (b == 0 || b == &a)
      __builtin_unreachable();
    else dead();
  }
  return &d;
}
static void g(int k) {
  int ***i = f; *c = h(); &i || k;
}
static void j() { g(0); }
int main() { j(); }
```

(b) https://github.com/llvm/llvm-project/issues/63124: LLVM `-Os` no longer removes the call to dead due to uncertainty on the semantics of alias metadata and how different passes handle it.

Listing 4. LLVM regression examples found with the DCE oracle. In both cases, the upstream version of LLVM regressed and cannot eliminate the call to dead if additional information is provided (highlighted in red). Previous versions of LLVM could eliminate it.

```c
static int a, c;
static int *b = &a;
static int **d = &b;
__attribute__ ((__noreturn__))
void assert_fail();
int main() {
 int *e = *d;
 if (e == &a || e == &c);
 else {
   __builtin_unreachable();
   assert_fail();
 }
 if(e == &a || e == &c);
 else dead();
}
```

(a) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=109546: GCC's new value range framework missed cases with address equality checks followed by unreachable code. Fixed with f828503eeb7.

```c
static int b, d, f, *c, *e = &d;
static unsigned g;
void a();
int main() {
 g = -19;
 for (; g; ++g) {
  int h = g, *i = &b, **j = &i;
  if (d) {
   int **k = &i; j = &c; *k = &f;
  } else { *e = 0; }
  if (!((h >= -19) && (h <= -1)))
    __builtin_unreachable();
  if (i); else a();
  if (j==&i || j==&c); else dead();
 }
}
```

(b) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=110538: GCC -O2 does not simplify the parent if-statement of dead. Changes in value range storage cause missed jump threading opportunities.

Listing 5. GCC regression examples found with the DCE oracle. In both cases, the upstream version of GCC regressed and cannot eliminate the call to dead if the additional information is provided (highlighted in red).

```c
static int b, c = 8, d, e, f, g = 9;
static char h = 3;
static void a(int, unsigned i) {
   if (!((i >= 1) && (i <= 3421036188)))
     __builtin_unreachable();
}
int main() {
  for (; h; --h) {
    for (; f <= 9; f++) {
      if (d) g = 0;
      if (e) continue;
      a(b, g && c);
    }
    e = d = 0;
    a(0, 3421036188);
  }
}
```

(a) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=112508: a recent change in jump threading causes GCC at -Os to generate a binary that is almost twice as large for this test case.

```c
static short b = -1;
static int c, e, *f = &c;
static char g;
static char a(char h, int i) {
  if (!((i >= -1) && (i <= 0)))
    __builtin_unreachable();
  return h || i ? h : h < 0;
}
static void d(unsigned h) {
  if (!((h >= 0) && (h <= 4095017279)))
    VRMarker();
}
int main(){ *f = b == 0; g=a(c,c);
    d(g); d(4095017279);
    if (e) { b = 0; }
    a(0, b);
}
```

(b) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=112542: a recent change in peephole optimizations made GCC at -O3 unable to infer the value range of h and eliminate the VRMarker() call.

Listing 6. Example regressions found with the Size and VR oracles. In both cases, the GCC regression manifests if the additional information (highlighted in red) is injected.

Listing 6 shows examples detected with the size oracle and the value range based oracles:

- In Listing 6a, GCC `-Os` generates an almost twice as large binary given the injected value range. This is caused by a recent change in a jump threading cost heuristic.
- In Listing 6b, we use the value range oracle to detect whether GCC `-O3` can infer the value range bounds of `h` in function `d`. The value range is inferred correctly (and the call to `VRMarker()` is eliminated) if no additional information is injected, but it is not if the highlighted value range is injected. This is caused by a recent peephole optimization change.

All of our reported cases contain the `__builtin_unreachable()` directive by the design of our approach. However, the root causes of our identified issues are typically orthogonal to the use of `__builtin_unreachable()`, which we utilize as a convenient mechanism for providing extra information to the compiler. We note the following:

- Our approach can be implemented in alternative ways. For example, in Listing 4a, we can detect the same issue (the dead call is not eliminated) by replacing the use of the builtin with a variable assignment, namely `g = 24821;`.
- The root cause analyses and fixes of our reported cases confirm that the majority of the detected issues are indeed not related to the unreachable directive, which merely helps uncover them (thanks to our approach). For example:
  - The fix for Listing 4a improves LLVM's "Correlated Value Propagation" pass to better handle modulo operations whenever `LHS >= RHS`.
  - The root cause analysis of Listing 5b (performed by a GCC developer) distilled the issue down to the fact that in the GCC's GIMPLE IR below:
    ```
    j_24 = PHI <&i(7), &c(3)>
    _2 = j_24 == &c;
    _22 = &i == j_24;
    _23 = _2 | _22; // <= This is always true because j_24 = &i or j_24 = &c
    ```
    GCC 13 can deduce that _23 is always true via dominance analysis and jump threading, but the upstream version cannot.

The active discussions on our reported issues and the detailed analyses by the compiler developers indicate that these issues (1) have diverse root causes (unexpected behaviors between passes, improperly handled cases, *etc.*), and (2) affect a variety of different passes (peephole optimizations, jump threading, partial redundancy elimination, *etc.*). Some of our reports also prompted the compiler developers to uncover and report additional issues because additional shortcomings became evident when they were triaging or fixing our reported issues. One developer commented for example: "I think they [the reports] are useful and help developers to see how changes affect other passes later on. Or if they miss something obvious."

4.8 Comparison with DCE Marker Based Differential Testing

One important question is whether refining programs by injecting dead code and value range information assists in detecting more and different issues. That is, can we detect the same issues without the refinement step? One way to answer this question is to compare our approach with the differential testing approach proposed by Theodoridis *et al.* [31] that uses DCE markers. We do this by attempting to detect the issues reported in Section 4.7 using only DCE markers without refining the input programs.

Ours is a metamorphic testing approach, thus we cannot directly compare the two methods. However, we bisected each regression in Section 4.7 to the commit that introduced it, *i.e.*, we have two compilers for each issue: C_{bad} that the exhibits the regression and C_{good} that does not. While not guaranteed, since C_{bad} and C_{good} are one commit apart, if we detect an issue with them via differential testing, then it is likely that it is the same regression that we detected with our approach.

For each regression and its corresponding C_{bad} and C_{good}, we perform two tests:

(1) We use the Csmith program that triggered the issue with our approach. We remove all injected information and only add DCE markers. If C_{bad} eliminates fewer markers than C_{good}, we have identified the regression.
(2) If the previous test fails, we use an additional corpus of 10,000 Csmith programs (we generate a new corpus for each regression), which we instrument with only DCE markers and perform the same differential test.

If we find a difference in either of the two tests, then we have detected the issue, *i.e.*, DCE marker based differential testing suffices to detect it and no refinement is needed. Out of the 59 reported cases, we found that 11 can be detected using only DCE markers. This demonstrates that, in practice, our approach detects issues that DCE marker based differential testing does not. Thus, besides being conceptually different and orthogonal, refining programs with additional information is also clearly beneficial for detecting many additional optimization-related issues.

4.9 Discussion

Takeaways. In this work, we systematically investigate the problem of optimization inconsistency in compilers and devise an effective approach for identifying such issues. We also provide concrete evidence of the usefulness of our approach: the inconsistencies that we report are not just contrived examples related, *e.g.*, to magic constants in heuristics, but relevant issues that compiler developers appreciate and fix. Our approach helps reveal unexpected interactions between the various analyses and transformations in compilers, and identifies many missed optimization opportunities.

Using Different Kinds of Information. Our current approach leverages `__builtin_unreachable` as a general mechanism to annotate two types of properties: dead branches and value ranges. We use this to demonstrate our general concept and technique, which can be extended in various ways. For instance, we can express different classes of properties via the `if(C) __builtin_unreachable();` construct. Example properties include whether two pointers never refer to the same address (C: `ptr1 == ptr2`), relationships among program variables (such as C: `v1 > v2`), whether a variable is a power of 2 (C: `__builtin_popcount(v) != 1`), explicit loop bounds (C: `loop_iter > LOOP_BOUND`), *etc.*. Besides using `__builtin_unreachable`, there are other alternatives. For example, given a variable `v` with a singleton value, say 10, we may capture this information in the refined program as `x = 10`. Note that how to refine a program is orthogonal to how to detect whether or not a refined program manifests any optimization inconsistencies.

Alternative Oracles. Our approach is general and parametrizable over any oracle that can compare two compiler's outputs. An interesting oracle would be runtime performance, *i.e.*, detecting cases where a refined program is slower than the original. In this work, we instantiated our approach with several oracles to demonstrate its effectiveness and utility. To this end, we focused on oracles that are deterministic, compile-time, and workload-independent, while runtime performance is both workload- and platform-dependent, which we leave for interesting future work.

Using Non-closed Programs. Our approach can also be used on non-closed programs given fixed inputs. Similarly to the case with closed programs, we can derive additional information via program execution (Section 3.2). The derived additional information is only guaranteed to be valid for the given fixed inputs, however, this does not invalidate our approach. Injecting the derived information "instructs" the compiler that we only expect the program to work on these inputs.

5 RELATED WORK

Automated Testing for Missed Compiler Optimizations. Several works that focus on automatically finding missed compiler optimizations exist. CIDetector [29] operates directly on binaries and detects dead stores and redundant loads and stores. Barany [3] proposed a differential testing method that relies on hand-written assembly matchers that identify cases where one compiler generates better code than another. The work of Theodoridis *et al.* [31], another differential testing approach, uses dead code elimination as an oracle to identify missed optimizations between compilers. Our work also uses dead code elimination as an oracle, but unlike the mentioned works, we do not use differential testing to identify issues, our approach is a metamorphic testing [5] one: we use a single compiler and test its behavior by injecting additional information about the input program.

Automated Compiler Testing for Miscompilations. Most work on automatic compiler testing focuses on finding miscompilations [4]. Random program generators such as Csmith [34] and YarpGen [19] have been extensively used to find correctness bugs via differential testing. Several approaches have been proposed to improve the effectiveness of randomized testing such as EMI [16] which mutates programs in a semantic preserving way given fixed inputs. Similarly to our approach, Sun *et al.* [28] investigate dynamically identifying variable properties at specific program locations to enable transformations that don't alter the program's overall behavior. Another example of using program dynamic information to improve compiler testing is the work of Even-Mendoza *et al.* [12] which identifies and removes safe math wrappers generated by Csmith. The goal of our approach is to find optimization inconsistency issues that are unrelated to miscompilations.

Providing Additional Information to the Compiler. Doerfert *et al.* [10] proposed a framework that explores the performance impact of providing additional static analysis annotations to the compiler. Similar to our work, the hints are provided via builtins. However, those hints are optimistic and are not necessarily correct, whereas the information we provide is guaranteed to be correct. Moreover, the aims of the two works are different: improving performance via additional information versus identifying optimization inconsistencies due to additional information.

Hiding Information from the Compiler. Siso *et al.* [27] proposed a framework for evaluating the auto-vectorization capabilities of compilers by hiding information such as loop bounds and array attributes. They showed that hiding information is usually detrimental to auto-vectorization, however, they also found cases where less information led to better code. Another example of hiding information from the compiler is inserting "dead by construction" code blocks [17] to enable EMI-based testing for OpenCL compilers. Our work is orthogonal to these approaches; we do not hide information from the compiler, but rather provide additional information.

6 CONCLUSION

We have developed an automated testing approach for detecting optimization inconsistencies, *i.e.*, cases where given additional information about an input program, a compiler generates worse code. Our approach derives additional information by running an input program, refines it using the derived information, and tests if the compiler's output is worse on this refined version. Our systematic analysis shows that optimization inconsistencies are prevalent in compilers such as GCC and LLVM, and that they are caused by a wide range of their components. We uncovered and reported a diverse set of such cases: out of the 59 reported issues, 55 were confirmed or fixed. We expect our methodology to open up a new direction for understanding unexpected interactions between compiler components and improving compilers.

ARTIFACT

Our archived artifact on Zenodo [32] contains all the necessary code and tools for identifying optimization inconsistencies. It also contains instructions, scripts, and the dataset needed for reproducing the paper's systematic evaluation. The code implementing the tracking and refinement processes (as explained in Section 3) is also available in the program markers GitHub repository.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.* 26, 4 (dec 1994), 345–420. https://doi.org/10.1145/197405.197406

[3] Gergö Barany. 2018. Finding Missed Compiler Optimizations by Differential Testing. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 82–92. https://doi.org/10.1145/3178372.3179521

[4] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (feb 2020), 36 pages. https://doi.org/10.1145/3363562

[5] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (jan 2018), 27 pages. https://doi.org/10.1145/3143561

[6] Khushboo Chitre, Piyus Kedia, and Rahul Purandare. 2022. The Road Not Taken: Exploring Alias Analysis Based Optimizations Missed by the Compiler. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 153 (oct 2022), 25 pages. https://doi.org/10.1145/3563316

[7] Keith D. Cooper, Mary W. Hall, and Linda Torczon. 1992. Unexpected Side Effects of Inline Substitution: A Case Study. *ACM Lett. Program. Lang. Syst.* 1, 1 (mar 1992), 22–32. https://doi.org/10.1145/130616.130619

[8] Christian Couder. 2008. Fighting regressions with git bisect. *Online: The Linux Kernel Archives* 4, 5 (2008). https://www.kernel.org/pub/software/scm/git/docs/git-bisect-lk2009.html

[9] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-Based Alias Analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 106–117. https://doi.org/10.1145/277650.277670

[10] Johannes Doerfert, Brian Homerding, and Hal Finkel. 2019. Performance exploration through optimistic static program annotations. In *34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16–20*. Springer, 247–268. https://doi.org/10.1007/978-3-030-20656-7_13

[11] Timur Doumler. 2022. *P1774R8: Portable assumptions*. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1774r8.pdf

[12] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2021. Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative about Undefined Behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1219–1223. https://doi.org/10.1145/3324884.3418933

[13] Zhangxiaowen Gong, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, Alexander Veidenbaum, Alexandru Nicolau, and Josep Torrellas. 2018. An Empirical Study of the Effect of Source-Level Loop Transformations on Compiler Stability. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 126 (oct 2018), 29 pages. https://doi.org/10.1145/3276496

[14] William H. Harrison. 1977. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering* 3 (1977), 243–250. https://doi.org/10.1109/TSE.1977.231133

[15] Ken Kennedy. 1979. *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division.

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[17] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

[18] Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring Missed Optimizations in WebAssembly Optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and*

Analysis (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 436–448. https://doi.org/10.1145/3597926.3598068

[19] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (jun 2023), 22 pages. https://doi.org/10.1145/3591295

[20] Rahim Mammadli, Marija Selakovic, Felix Wolf, and Michael Pradel. 2021. Learning to Make Compiler Optimizations More Effective. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 9–20. https://doi.org/10.1145/3460945.3464952

[21] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. 1991. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/113445.113447

[22] Ankush Phulia, Vaibhav Bhagee, and Sorav Bansal. 2020. OOElala: Order-of-Evaluation Based Alias Analysis for Compiler Optimization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 839–853. https://doi.org/10.1145/3385412.3385962

[23] 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* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[24] Manuel Rigger, Stefan Marr, Bram Adams, and Hanspeter Mössenböck. 2019. Understanding GCC Builtins to Develop Better Tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 74–85. https://doi.org/10.1145/3338906.3338907

[25] Paul B. Schneck. 1973. A survey of compiler optimization techniques. In *Proceedings of the ACM Annual Conference* (Atlanta, Georgia, USA) *(ACM '73)*. Association for Computing Machinery, New York, NY, USA, 106–113. https://doi.org/10.1145/800192.805690

[26] 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

[27] Sergi Siso, Wes Armour, and Jeyarajan Thiyagalingam. 2019. Evaluating Auto-Vectorizing Compilers through Objective Withdrawal of Useful Information. *ACM Trans. Archit. Code Optim.* 16, 4, Article 40 (oct 2019), 23 pages. https://doi.org/10.1145/3356842

[28] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. (2016), 849–863. https://doi.org/10.1145/2983990.2984038

[29] Jialiang Tan, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2020. What Every Scientific Programmer Should Know about Compiler Optimizations?. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 42, 12 pages. https://doi.org/10.1145/3392717.3392754

[30] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3368826.3377927

[31] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 697–709. https://doi.org/10.1145/3503222.3507764

[32] Theodoros Theodoridis and Zhendong Su. 2024. *PLDI 2024 Artifact for "Refined Input, Degraded Output: The Counterintuitive World of Compiler Behavior"*. https://doi.org/10.5281/zenodo.10808465

[33] Sid-Ahmed-Ali Touati and Denis Barthou. 2006. On the Decidability of Phase Ordering Problem in Optimizing Compilation. In *Proceedings of the 3rd Conference on Computing Frontiers* (Ischia, Italy) *(CF '06)*. Association for Computing Machinery, New York, NY, USA, 147–156. https://doi.org/10.1145/1128022.1128042

[34] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Received 2023-11-16; accepted 2024-03-31