Finding Missed Optimizations through the Lens of Dead Code Elimination

Theodoros Theodoridis
theodoros.theodoridis@inf.ethz.ch
ETH Zurich
Switzerland

Manuel Rigger
manuel.rigger@inf.ethz.ch
ETH Zurich
Switzerland

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT

Compilers are foundational software development tools and incorporate increasingly sophisticated optimizations. Due to their complexity, it is difficult to systematically identify opportunities for improving them. Indeed, the automatic discovery of missed optimizations has been an important and significant challenge. The few existing approaches either cannot accurately pinpoint missed optimizations or target only specific analyses. This paper tackles this challenge by introducing a novel, effective approach that — in a simple and general manner — automatically identifies a wide range of missed optimizations. Our core insight is to leverage dead code elimination (DCE) to both analyze how well compilers optimize code and identify missed optimizations: (1) insert "optimization markers" in the basic blocks of a given program, (2) compute the program's live/dead basic blocks using the "optimization markers", and (3) identify missed optimizations from how well compilers eliminate dead blocks. We essentially exploit that, since DCE heavily depends on the rest of the optimization pipeline, through the lens of DCE, one can systematically quantify how well compilers optimize code. We conduct an extensive analysis of GCC and LLVM using our approach, which (1) provides quantitative and qualitative insights regarding their optimization capabilities, and (2) uncovers a diverse set of missed optimizations. Our results also lead to 84 bug reports for GCC and LLVM, of which 62 have already been confirmed or fixed, demonstrating our work's strong practical utility. We expect that the simplicity and generality of our approach will make it widely applicable for understanding compiler performance and finding missed optimizations. This work opens and initiates this promising direction.

CCS CONCEPTS

• **Software and its engineering → Compilers**.

KEYWORDS

compilers, missed optimizations, testing

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507764

ACM Reference Format:
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 (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507764

1 INTRODUCTION

Both industry and academia have invested decades of effort to enhance compiler optimizations to improve the performance of computer programs [2, 3, 10, 16]. Despite these efforts, optimizing compilers are plagued by performance bugs, also known as missed optimization opportunities [24]. We define a missed optimization opportunity loosely as a case where a compiler produces inefficient code for a source program, for which it could be reasonably expected to produce more efficient code. The reasons for performance bugs are manifold. For example, a developer might forget to implement certain corner cases of an analysis. As another example, changes in one transformation, such as enabling more aggressive loop unswitching, might not consider how a subsequent transformation, such as value propagation, are affected [33]. In both examples, compilers might generate suboptimal code.

Typically, performance bugs are found by users or developers of the compiler when they observe performance bottlenecks in real-world programs. Pinpointing such performance bugs is difficult, time-consuming, and error-prone due to various reasons. A typical approach would be to narrow down the location of the performance bug by gradually reducing the program and either directly measuring the execution time or using a profiler for that purpose. This is often unreliable, because (1) measuring the execution time accurately is often impossible due to noise caused by, for example, non-determinism in process scheduling [5], (2) profilers are not accurate at the instruction or basic block level [9], and (3) unrelated changes in the program might have significant changes on performance, for example, due to the linking order of object files [25]. Consequently, manually pinpointing performance bugs remains an art that typically requires both a combination of measurements and manual analysis of the code.

More systematic ways exist to avoid or detect performance bugs, but they have significant limitations. Compiler developers typically use benchmarking, to monitor the effect of compiler changes on a selection of benchmarks [6]. However, the main use case for benchmarking is to identify regressions and it is inapplicable to identify existing bugs. In addition, due to the limited number of benchmarks, performance bugs in other programs might be overlooked. In terms of automatic testing approaches, Barany proposed comparing static features (*e.g.*, the number of load instructions or the number of

register spills) of the compiled code between compilers for a given source program to find potential missed optimizations [4]. Taneja et al. verified and extended various analysis phases with the help of an SMT solver to improve the performance of the compiled code [31]. While these approaches have successfully identified shortcomings which were subsequently fixed, they only tackle certain instances of the general missed optimizations problem (*e.g.*, in data-flow analyses, register allocation, and instruction selection); techniques that can target a broader spectrum of a compiler's internals are needed to further improve the performance of compiled programs and to prevent regressions in optimizing compilers.

This paper introduces a simple and broadly applicable concept for automatically detecting missed optimization opportunities. Our approach is to instrument code such that we can precisely and deterministically probe the effects of compiler analyses and optimizations. Our core insight is to exploit the interplay of DCE and all the analyses and optimizations on which it depends. For example, a compiler might have to first inline a piece of code and propagate constants before being able to determine that it is never executed and can be removed. In other words, we use DCE as a black-box oracle to check how effective a compiler is in optimizing code and to reveal missed opportunities. More concretely, given a program P, we insert a set of markers $\mathcal{M}$ into the basic blocks of its source code. When P is compiled to machine code, a marker M only remains in the binary *iff* the compiler determined the block in which it is located to be alive; a removed marker indicates that the surrounding block is dead and DCE could successfully be applied. We denote a marker to be alive or dead depending on the status of the surrounding block and denote a marker's liveness as a function $Comp(M) \rightarrow \{dead, alive\}$. We can now apply *differential testing* [23]. Given two compilers $Comp_1$ and $Comp_2$, *e.g.*, GCC and LLVM, we can check whether $\exists M \in \mathcal{M}\ Comp_1(M) \neq Comp_2(M)$; a compiler failing to eliminate a dead marker indicates a missed optimization opportunity *iff* the other compiler could successfully remove it. This technique is also applicable for a single compiler and two optimization levels. Given a compiler with a lower optimization level $Comp_L$ and a higher optimization level $Comp_H$, we can discover missed optimization opportunities by checking whether $Comp_H$ failed to eliminate a dead marker that $Comp_L$ could remove, *i.e.*, $\exists M \in \mathcal{M}$ if $Comp_L(M) \rightarrow \neg Comp_H(M)$ indicates a missed optimization opportunity. In both cases, our technique allows us to identify the location of the missed optimization opportunity, which is the block of the dead marker that could not be eliminated.

DCE in combination with optimization markers satisfies two properties that make it an appropriate oracle: (1) no compiler modifications are needed: finding the dead and alive markers is done by inspecting the generated assembly; (2) DCE is an optimization "sink": its effectiveness depends on a large part of the compiler pipeline. There are other analyses and transformations that can be used in such a differential testing manner. For example, one could compare range analysis results between compilers to uncover weaknesses, however, this would require modifying the target compilers to make such comparisons possible, hindering applicability and adoption, and only a narrow part of the compiler would be tested.

Our empirical analysis demonstrates the usefulness of our approach in uncovering a wide range of missed optimization opportunities. We discovered 2,637 missed optimization opportunities in GCC and 318 in LLVM in a corpus of 10,000 C auto-generated files. We analyzed and reported 53 performance bugs for GCC, out of which 43 were confirmed and 12 were fixed. For LLVM, we reported 31; 19 were confirmed and 11 were fixed. We also found 44 regressions in GCC and 38 regressions LLVM; older versions of these compilers were more successful in optimizing these cases. To investigate the applicability of the technique to find a wide range of performance bugs, we bisected these regressions into 23 unique commits in GCC and 21 in LLVM; in total they affect 34 and 23 unique files in the compilers' codebases, which are related among others to alias analysis, call graph handling, control flow graph analysis, peephole optimizations, loop optimizations, pass management, and value range analysis. Overall, this demonstrates that our technique is highly effective in finding a wide range of relevant performance bugs in state-of-the-art compilers. Our key contributions are:

- A novel, widely-applicable, and effective black-box approach for finding missed optimizations.
- An implementation of our approach and its evaluation via discovering a diverse set of bugs.
- An empirical analysis on how well compilers optimize between optimization levels and in comparison with a hypothetical compiler that can detect all dead code.

2 ILLUSTRATIVE EXAMPLE

Listing 1a shows an example, which we use to demonstrate how our work enables the discovery of missed optimization opportunities. Note that the program does not rely on any external input or state for its computations. The test case contains two if-statements whose conditions both evaluate to false: `if (d == e)` is false because the pointers `d` and `e` contain different addresses (`&a` and `&b[1]`), `if (c)` is false because it is initialized to 0 and never updated before this check. Dead Code Elimination should remove such blocks.

Dead Code Elimination (DCE) is a compiler transformation that removes unreachable instructions or reachable ones whose results are unused [1]. We refer to such code fragments as "dead". The successful application of DCE depends on the effectiveness of prior compiler phases. For example, a compiler must determine that the pointers `d` and `e` in Listing 1a do not alias to eliminate the body of the `if (d == e)` statement; to eliminate the `if (c)` statement, the compiler must determine that `c`'s value is 0. DCE can also eliminate side-effect free computations whose results are never used, *e.g.*, the dead store `c = 0` at the end of Listing 1a; programmers typically do not write such code, but such patterns can emerge as a result of other compiler transformations. One would expect that compilers remove these dead blocks.

Surprisingly, LLVM and GCC, two state-of-the-art compilers, fail to eliminate both checks in Listing 1a. LLVM eliminates the second if-statement and its body, but not the first one, as shown in Listing 1b: one indication is that the `callq print` instruction was not eliminated. GCC eliminates the first if-statement and its body, however, it generates code for the second as shown in Listing 1c; in addition, it fails to determine that `c = 0;` is a dead store, instead, it generates `movl $0, c(%rip)`.

Existing automated approaches are inapplicable for detecting missed optimizations such as the ones shown in Listing 1. For

Listing 1: Illustrative example. The original test case was much larger and generated by Csmith [37]. The highlighted parts are dead.

(a) Original

```
#include <stdio.h>
char a;
char b[2];
static int c = 0;

int main() {
 char *d = &a;
 char *e = &b[1];
 if (d == e) {
   int f = 0;
   int g = 0;
   for (;f < 10; f++)
     g += f;
   printf("%d", g);
 }
 if (c) {
   b[0] = 1;
   b[1] = 1;
 }
 c = 0;
 return 0;
}
```

(b) LLVM assembly

```
movl    $b+1, %eax
cmpq    $a, %rax
jne     .LBB0_2
pushq   %rax
movl    $.L.str,%edi
movl    $45, %esi
xorl    %eax, %eax
callq   printf
addq    $8, %rsp
.LBB0_2:
xorl    %eax, %eax
retq
```

(c) GCC assembly

```
movl    c(%rip),%eax
testl   %eax, %eax
je      .L2
movw    $257,b(%rip)
.L2:
movl    $0,c(%rip)
xorl    %eax, %eax
ret
```

example, it is unclear how directly comparing the output of multiple compilers [4] would be used, as the generated assembly does not contain any obvious, general, and easily checkable patterns hinting that either compiler missed eliminating the dead pieces of code. A benchmarking-based approach would also not be useful as it is too coarse-grained and the individual parts, such as certain if-statements, are not explicitly targeted.

We tackle the problem of automatically discovering missed optimization opportunities by using optimization markers: we instrument the source code with markers that remain in the resulting assembly *iff* the basic block in which the marker was placed is alive. These markers help us identify when DCE was successfully applied, and successful application of DCE, in turn, indicates that other analyses and optimizations were also applied. We use function calls as markers: we do not provide the bodies of the callees; thus, the compilers cannot analyze them or optimize them away, and thus they only eliminate them if the surrounding basic blocks are dead.

Listing 2a shows the instrumented version of Listing 1a; each function prefixed by `DCECheck` corresponds to a marker, which was inserted for every if-statement and for-loop body. The compilers can only eliminate these markers if they can determine that the

Listing 2: Illustrative example, instrumented. The dead parts are highlighted with light gray. The optimization markers are highlighted with blue, green, and orange

(a) Instrumented

```
#include <stdio.h>

void DCECheck0(void);
void DCECheck1(void);
void DCECheck2(void);

char a;
char b[2];
static int c = 0;

int main() {
 char *d = &a;
 char *e = &b[1];
 if (d == e) {
   DCECheck0();
   int f = 0;
   int g = 0;
   for (;f < 10; f++) {
     DCECheck1();
     g += f;
   }
   printf("%d", g);
 }
 if (c) {
   DCECheck2();
   b[0] = 1;
   b[1] = 1;
 }
 c = 0;
 return 0;
}
```

(b) LLVM assembly

```
movl    $b+1, %eax
cmpq    $a, %rax
jne     .LBB0_2
pushq   %rax
callq   DCECheck0
callq   DCECheck1
...
callq   DCECheck1
movl    $.L.str,%edi
movl    $45, %esi
xorl    %eax, %eax
callq   printf
addq    $8, %rsp
.LBB0_2:
xorl    %eax, %eax
retq
```

(c) GCC assembly

```
movl    c(%rip),%esi
testl   %esi, %esi
jne     .L8
movl    $0, c(%rip)
xorl    %eax, %eax
ret
.L8:
pushq   %rax
callq   DCECheck2
xorl    %edx, %edx
xorl    %eax, %eax
movw    $257,b(%rip)
movl    %edx,c(%rip)
popq    %rcx
ret
```

basic blocks which contain the markers are dead. If LLVM and GCC eliminate different subsets of `DCECheck` markers, then there are markers that can be feasibly eliminated by a compiler but either LLVM or GCC is not eliminating them. In other words, at least one of the compilers is missing an optimization opportunity.

Our approach can detect both missed opportunities in Listing 2a. LLVM is unable to determine that `d == e` and generates code for the first if-statement's body; we can detect this by searching for a call to `DCECheck0` in the assembly code (see Listing 2b). Similarly, GCC is unable to determine that `c == 0` and generates the code for

Listing 3: Reduced test case exposing missed optimization opportunities in LLVM (https://bugs.llvm.org/show_bug.cgi?id=49434).

(a) C code

```
char a;
char b[2];

int main() {
 char *c = &a;
 char *d = &b[1];
 if (c == d)
  DCECheck();
 return 0;
}
```

(b) Assembly

```
leaq  a(%rip), %rax
leaq  b+1(%rip),%rcx
cmpq  %rax, %rcx
jne   .LBB0_2
pushq  %rax
callq  DCECheck
addq   $8, %rsp
.LBB0_2:
xorl %eax, %eax
retq
```

Listing 4: Reduced test cases exposing missed optimization opportunities in GCC (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99357).

(a) C code

```
static int a = 0;

int main() {
  if (a) {
    DCECheck();
  }
  a = 0;

  return 0;
}
```

(b) Assembly

```
movl  a(%rip), %ecx
testl %ecx, %ecx
jne   .L8
movl  $0, a(%rip)
xorl  %eax, %eax
ret
.L8:
pushq %rax
call  DCECheck
xorl  %eax, %eax
movl  $0, a(%rip)
popq  %rdx
ret
```

the second if-statement's body (see Listing 2c); the generated code contains a call to `DCECheck2`. The sets of the eliminated markers are {`DCECheck2`} for LLVM and {`DCECheck0`, `DCECheck1`} for GCC. They both fail to seize optimization opportunities. Note that LLVM's generated code also contains `DCECheck1`, but we can ignore it and focus on `DCECheck0` which is likely the *primary* missed opportunity (see Section 3.2): the for-loop containing `DCECheck1` is nested within the if-statement body containing `DCECheck0`.

The reduced and reported test cases are shown in Listing 3 and Listing 4. **Listing 3a**: this test case reveals that LLVM's EarlyCSEPass, a phase that is applied early in the pipeline and tries to eliminate redundant instructions, cannot simplify `&a == &b[1]` to `false`. Interestingly, by changing the indexing from `b[1]` to `b[0]`, EarlyCSEPass manages to simplify it and the dead block is eliminated. **Listing 4a**: Removing the `a = 0` assignment after the if-statement helps the compiler recognize that `a` is constant and it can constant-fold and eliminate it. This test case reveals two issues in GCC: (a) GCC's global variable value analysis is not flow-sensitive [7] and cannot deduce that `a` does not change until the assignment; (b) GCC fails to apply dead store elimination: storing 0 to `a` does not have any effect (`movl $0, a(%rip)`). We utilized our optimization-marker-based technique to find and report these bugs; the GCC developers confirmed the second one. While this section aims to convey the intuition behind our core approach, the subsequent section presents our technique in detail.

3 DETECTING MISSED OPTIMIZATIONS

We present a highly effective black-box approach for finding missed optimization opportunities, *i.e.*, cases where a compiler unexpectedly fails to optimize code. Our core idea is to utilize DCE, whose effectiveness depends on other optimizations being applied, for detecting such missed opportunities. In order to realize this idea, we propose optimization markers: if a compiler, C_1, can eliminate a marker via DCE, but another compiler, C_2 cannot, then this indicates that C_2 is missing an optimization opportunity. As we demonstrate in Section 4, such missed opportunities are caused by interactions between DCE and a variety of other analyses and optimizations.

Figure 1 illustrates the steps of our approach. In step ①, we insert optimization markers to each basic block of an input test case. Since we perform the instrumentation on a source-code level using function calls, the approach is general and applicable to test any compiler. In step ②, we compile the instrumented test case with two different compilers; the result is multiple potentially differently-optimized versions. Instead of two different compilers, we can use a single compiler and multiple optimization levels to identify when higher-optimization levels fail to utilize optimization opportunities discovered by lower-optimization levels. In step ③, we compare the set of remaining markers in each binary; if the sets differ, then at least one compiler failed to eliminate a piece of code that the other managed to. In step ④, we eliminate the so-called non-*primary* markers, which we define as markers whose elimination potentially depends on other non-eliminated markers (see Section 3.2 for more details). The output of our approach is a set of bug-inducing test cases that can be further triaged and automatically reduced using known techniques [38]. Section 3.1 details steps ① through ③, which are the core of our approach. Section 3.2 details step ④.

3.1 Approach

Instrumentation In step ①, we insert optimization markers into the source code of the test case to help us detect if DCE was successfully applied. Markers allow us to do this in a black-box and general way without relying on or changing the internals of a compiler. If we place a marker in the source code and it remains in the generated assembly, then either it is in an alive part of the program or the compiler failed to eliminate it; if it was identified as *dead*, we should be unable to find it in the assembly. Optimization markers can be implemented in many ways such as function calls, compiler builtins, inline assembly, or writes to global variables.

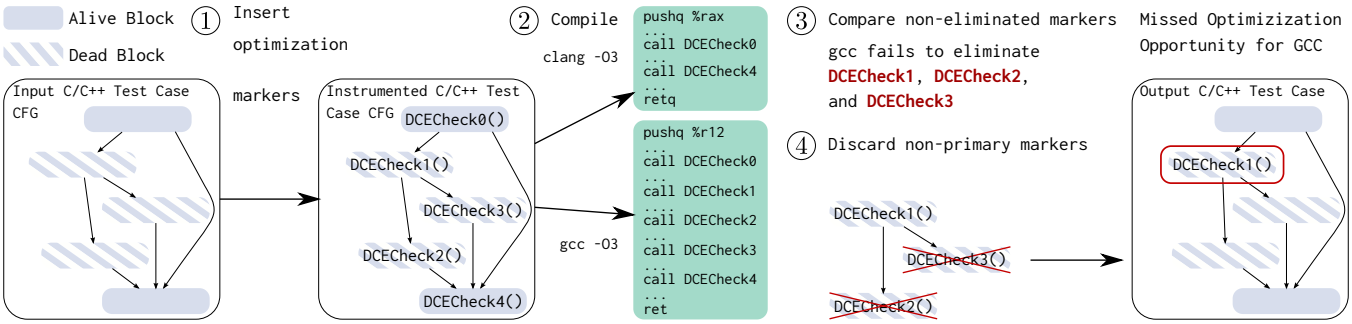

Figure 1: Overview of our approach for Missed Optimization discovery.

We implement optimization markers with function calls. We insert calls to functions whose bodies are not provided; thus, the compiler cannot analyze or inline them. Consequently, these markers can only be eliminated if they are dead, *i.e.*, they can never be executed. To check if a marker is eliminated, we can search for the corresponding instruction, *e.g.*, `callq DCECheck0`[1]: if the corresponding call instruction is not present, the marker was eliminated.

An optimization marker is only eliminated if the basic block which contains it is dead: a compiler cannot analyze whether a marker has any effect on the input program's execution. Consequently, we can use optimization markers to detect which basic blocks in a program are dead.[2] For example, we can check that `DCECheck2` in Listing 2b and `DCECheck0` in Listing 2c were eliminated because the corresponding markers are not present; thus the if-statement bodies in Listing 2a are dead.

Missed optimization opportunity detection In step ②, we compile the instrumented test case with multiple compilers to generate multiple potentially differently optimized binaries. The compilers can be a combination of one or more distinct compilers, *e.g.*, LLVM and GCC, and multiple optimization levels. As we discuss in Section 4, markers eliminated at lower optimization levels are not necessarily eliminated at higher ones; thus using multiple optimization levels is beneficial.

In step ③, we detect missed optimization opportunities by comparing the set of alive markers using multiple compilers (*i.e.*, using differential testing [23], which exploits N-version programming [18]). If a marker M, given a compiler $Comp$, is present in the generated assembly, then $Comp$ determined the marker's surrounding basic block to be alive, thus M is an alive marker: $Comp(M) = alive$. If the marker is dead, $Comp(M) = dead$, the surrounding block was eliminated via DCE. Given the set of *alive* and *dead* markers, we determine the existence of potential missed opportunities via differential testing.

We perform differential testing for missed optimization opportunity detection in the following manner. Let $\mathcal{M}$ be the set of markers of an instrumented test case, and $Comp_1$ and $Comp_2$ the two compilers: if $\{M : M \in \mathcal{M} \wedge Comp_1(M) \neq Comp_2(M)\} \neq \emptyset$ then at least one compiler failed to eliminate one or more markers which are actually dead. The set of missed markers for $Comp_1$ is: $\{M : M \in \mathcal{M} \wedge Comp_1(M) = alive \wedge Comp_2(M) = dead\}$, *i.e.*, the markers that $Comp_2$ eliminates but $Comp_1$ does not. In principle, it may be infeasible for a compiler to determine that a certain piece of code is dead, however, these missed markers are eliminated by $Comp_2$, thus they are feasible missed opportunities for $Comp_1$.

[1]Depending on the backend they may also be code-generated as jump instructions.
[2]We note that the inserted optimization markers may impact how a compiler optimizes the instrumented code vs. the original uninstrumented code, but this does not affect the effectiveness of our technique.

3.2 Primary Missed Optimization Opportunities

In step ④, we exclude discovered missed optimization opportunities that are potentially caused by other missed opportunities. Eliminating redundancy is useful both for understanding the capabilities of state-of-the-art compilers, as well as for reporting cases to compiler developers. To that end, we rely on a function's Control Flow Graph (CFG) to detect such secondary missed opportunities based on whether their predecessors are also missed.

A basic block can be dead because it is dominated by other dead blocks. We call such dead blocks *secondary*. For example, the nested if-statement in Listing 5 is dead because the outer if-statement condition (`expr1`) evaluates always to false; if a compiler fails to eliminate the outer one, and as a result the inner one too, then we want to ignore the latter and investigate why the former is missed. In general, we want to focus on the root causes of missed dead code elimination opportunities.

We identify secondary missed dead blocks by checking their CFG predecessors. If a dead block is missed but one or more of its predecessors are also dead and missed, then it is likely that the latter is the root cause: if the compiler was able to eliminate the predecessor(s) then it would have likely eliminated the successor. This does not necessarily hold, but it helps focus triaging efforts on missed opportunities which may resolve the secondary ones.

We use the term *missed primary dead basic block* to denote non-eliminated dead blocks whose predecessors are either alive or detected dead. More formally: let $G = (V, E)$ be an (inter-procedural) control-flow graph. The predecessors of node v are $pred(v) : \{u | u \rightarrow v \in E\}$. Each $v \in V$ is labeled *dead* or *live*, *i.e.*, $l(v) = dead$ or *live*. Thus, $V = V_{dead} \sqcup V_{live}$. A compiler C may miss to detect some of the V_{dead} nodes, but we assume it will not misidentify any of the live nodes in V_{live} as *dead*. Each $v \in V_{dead}$ is also labeled *detected* or *missed*, *i.e.*, $C(v) = missed$ or *detected*. For each v such that $C(v) = missed$, we decide whether it is primary or not.

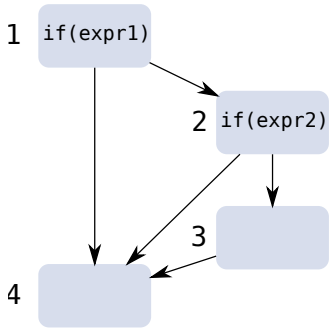

Figure 2: Control Flow Graph of Listing 5.

Listing 5: Nested Dead Code Example

```
1  void foo(){
2    if (expr1){//expr1 is always false
3      //Dead Code
4      if (expr2){//expr2 can be false
5        //or true
6        //Also Dead Code
7      }
8    }
9    //...
10 }
```

Definition. *Missed Primary Dead Block. A node v, with $C(v) = missed$, is primary iff: $\forall u \in pred(v),\ l(u) = live$ or $C(u) = detected$.*

For example, basic blocks 2 (B2) and 3 (B3) in Figure 2 (Listing 5) are dead because `expr1` is always false and therefore the branch in block 1 will always jump to block 4. If a compiler does not eliminate B2 and B3, that is, $C(2) = missed$ and $C(3) = missed$, then we want to focus on the root cause, *i.e.*, why was B2 missed? If B2 was properly detected and eliminated then B3 should also have been eliminated. B2 is a missed primary dead block because $C(2) = missed$, $pred(2) = \{1\}$, and $l(1) = live$. In contrast, B3 is not because $C(3) = missed$, $pred(3) = \{2\}$, and $C(2) = missed$. On the other hand, if B2 was properly eliminated but B3 not, the latter would become a primary missed block: $C(3) = missed$, $pred(3) = \{2\}$, and $C(2) = detected$. A primary missed dead block is guaranteed to be interesting in the sense that there are no neighboring missed dead blocks that are causing its misidentification.

We define *primary missed optimization opportunities* and *markers* as those that involve a primary missed dead block.

4 EVALUATION

We evaluate the effectiveness and practical utility of our approach. We use two compilers in our experiments: LLVM (up to commit `3cc38703d5ab`) and GCC (up to commit `92acae5047e`). We first compare LLVM's and GCC's DCE effectiveness against a theoretically ideal compiler with respect to the missed optimization opportunities our approach uncovers (see Section 4.1). We then demonstrate the practical utility of our approach by discovering a variety of missed optimization opportunities in both GCC and LLVM (see Section 4.2). We finally present and discuss some of the reported missed opportunities (see Section 4.3).

Test programs We use a corpus of 10,000 programs generated by Csmith [37], a random C program generator. Csmith programs are self-contained, do not require inputs, and have large dead parts; this makes Csmith-derived programs ideal for testing the effectiveness of our approach. While such auto-generated programs may not represent real-world ones, the bugs they help uncover are often relevant to real-world programs [19, 37]. As a concrete example, a missed optimization opportunity that we discovered with our approach and reported (Listing 9f) had previously been reported by developers of GCC. In addition, new programs can be generated on demand, meaning that Csmith can be used as part of a continuous effort for discovering missed opportunities and regressions.

Implementation We implemented our approach as a tool in approximately 500 lines of C++ code using LLVM's LibTooling for the instrumentation, and approximately 500 lines of shell script and Python code for the generation of test programs, differential testing, reductions, and bisections. We instrument C and C++ source-level structures that roughly correspond to basic blocks, *e.g.*, if-then-else blocks, loop bodies, switch case statements, switch default statements, and function bodies after conditional returns.

Experimental environment We used an AMD Ryzen Threadripper 3990X based system running Ubuntu 20.04 for our experiments. Generating the test cases, instrumenting them, executing them to derive the actual set of dead and alive blocks, and performing differential testing across all compiler configurations required around an hour; all of these steps are very quick and can be performed in parallel for each test case. Reducing a discovered missed opportunity with C-Reduce [28] typically required 4 to 8 hours; bisecting a regression required a similar amount of time as it requires re-building large parts of the compiler under test multiple times.

4.1 LLVM and GCC's Optimization Effectiveness

We evaluate how state-of-the-art compilers would compare against an ideal one, which eliminates all dead code. Our test cases are deterministic and do not require inputs; therefore, dead code observed during one execution is dead for all executions. We determine the dead and alive blocks of an instrumented test case by executing it: Executed markers indicate alive blocks, while the remaining markers are dead. Thus, we can determine how well an ideal compiler would perform with respect to DCE on the given test cases.

Dead block prevalence Out of the 3,109,167 instrumented blocks, 89.59% are dead and 10.41% are alive. The significant percentage of dead blocks is expected, given that the programs were randomly generated. It is also beneficial for our analysis since it allows targeted testing of DCE.

Compilers eliminate the majority of dead blocks Both GCC and LLVM at `-O3` eliminate the majority of the dead markers, 94.40% and 95.69%, respectively. In particular, the percentage of non-eliminated *primary* dead markers is 1.53% (42,478) for GCC and 1.37% (38,194) for LLVM. Higher optimization levels expectedly eliminate more dead blocks (see Table 1 and Table 2). The compilers'

Table 1: Higher optimization levels eliminate, as expected, a larger percentage of dead blocks.

Optimization Level	% dead blocks that are missed	
	GCC	LLVM
`O0`	85.21%	83.82%
`O1`	8.18%	5.20%
`Os`	5.94%	4.75%
`O2`	5.66%	4.35%
`O3`	5.60%	4.31%

Table 2: Higher optimization levels eliminate, as expected, a larger percentage of primary dead blocks.

Optimization Level	% dead blocks that are primary missed	
	GCC	LLVM
`O0`	15.30%	4.75%
`O1`	1.76%	1.47%
`Os`	1.56%	1.43%
`O2`	1.53%	1.38%
`O3`	1.53%	1.37%

Listing 6: Reduced missed opportunities for both GCC and LLVM. The dead parts are highlighted with light gray.

(a)

```
static int a = 0;
int main() {
  if (a) DCECheck();
  a = 1;
}
```

(b)

```
static int a,*b = &a;
int main() {
  if (*b) DCECheck();
  b = 0;
}
```

Listing 7: LLVM could previously eliminate `dead()` at both `-O2` and `-O3`, but after a regression the latter stopped eliminating the call. The dead parts are highlighted with light gray.

```
int a, b, c;
int main() {
  b = 0;
  while (a) while (c) if (b) dead();
  return 0;
}
```

front ends already perform a basic form of DCE and even at `-O0`, GCC eliminates 14.79% and LLVM 16.18% of the dead blocks. At `-O1` and above, both compilers eliminate more than 90% of the dead blocks, and the differences between `-O2` and `-O3` are in the order of 0.06% for GCC and 0.04% for LLVM. These differences enable discovering a variety of missed opportunities (Section 4.2).

Examples Listing 6 shows two reduced examples with dead markers that both GCC and LLVM fail to eliminate. The case in Listing 6a is very similar to Listing 4a, with the only difference being in the last assignment in `main`: `a = 1` instead of `a = 0`; GCC fails to eliminate both for the same reason as explained in Section 2: its flow-insensitive global value analysis prevents it from propagating the initial value of `a` to the if-statement check. LLVM can eliminate the version with `a = 0`; however, it cannot eliminate the marker with `a = 1`. This is an old regression. LLVM up to version `3.7.1` eliminated the markers in both versions, but starting from version `3.8`, it can no longer propagate the initial value of `a` to the if-statement check if it does not remain constant. Similarly, both compilers are unable to remove the marker in the second case shown in Listing 6b, as they cannot deduce that `a` remains constant and fail to propagate its value via `b` to the if-statement.

4.2 Practical Utility

We evaluate the practical utility of our approach, *i.e.*, whether it can discover a diverse set of missed optimization opportunities in state-of-the-art compilers, in two scenarios: (a) running both LLVM and GCC at `-O3` and checking if they eliminate a different subset of markers, and (b) running the same compiler at `-O1`/`-O2` versus `-O3` and checking for cases where the higher optimization level misses opportunities that the lower one seizes. To this end, we check which optimization markers are eliminated in the generated assembly.

Between GCC and LLVM Differential testing between GCC and LLVM at `-O3` reveals many missed opportunities for both of them. They eliminate a different subset of markers: GCC can eliminate 3,781 that LLVM misses, but LLVM eliminates 39,723 that GCC fails to eliminate. Out of these 396 and 4,749 are *primary* missed optimization opportunities. In principle, both compilers should be able to eliminate the missed opportunities; the fact that the other compiler succeeds shows that it is feasible.

Note that the discovered missed opportunities are not necessarily unique, *i.e.*, the same root cause might be the source of multiple missed opportunities. We deduplicate cases after reducing them and before reporting them to compiler developers. We subsequently investigate the diversity of the discovered missed opportunities: we show that they relate to various compiler components.

Between optimization levels Differential testing across different optimization levels in the same compiler reveals additional missed opportunities. Lower optimization levels sometimes lead to better results: GCC fails to eliminate 308 dead markers at `-O3` but it manages to eliminate them at `-O1` or `-O2`; similarly, LLVM misses 456. Out of these, 24 and 54 are *primary*. Some of these missed opportunities are regressions; for example, LLVM could previously eliminate the call to `dead` in Listing 7 at `-O2` and `-O3`, but after a recent change of the loop unswitching implementation, LLVM eliminates the call only at `-O2`.

Table 3: The commits introducing missed DCE opportunities affect a diverse set of LLVM components.

Component	# Commits	# Files
Alias Analysis	1	1
Instruction Operand Folding	2	1
Jump Threading	1	1
Loop Transformations	1	1
Pass Management	2	2
Peephole Optimizations	7	10
SSA Memory Analysis	2	1
Target Info	1	2
Value Constraint Analysis	1	1
Value Propagation	4	2
Value Tracking	1	1

Missed optimization diversity We evaluate the diversity of the discovered missed opportunities by categorizing them based on the compiler components they relate to. We use offending commits, *i.e.*, changes in a compiler that introduced these missed opportunities. We find the offending commits in the following manner: (a) we locate a previous compiler version in which a missed call is detected, and (b) we bisect between this version and the current latest development version. Doing this allows us to check which of the compilers' file changes trigger each case and therefore which component (*e.g.*, alias analysis or pass management) the missed opportunities most likely relate to. We only consider *primary* `-O3` regressions. We categorize the files touched by each offending commit into compiler component categories.

LLVM. Out of the 54 *primary* missed dead markers which manifest at `-O3` but not at `-O2/-O1`, 38 are regressions. Bisecting these led to 21 different unique commits. The offending commits affect a variety of LLVM's components, *e.g.*, alias analysis, value propagation, peephole optimizations, loop optimizations, and the pass manager; 11 components spread across 23 files are affected (Table 3).

GCC. Out of the 308 primary missed dead markers which manifest at `-O3` but not at `-O2/-O1`, 44 are regressions. Bisecting these led to 23 different unique commits. These offending commits affect a variety of GCC's components such as control-flow and callgraph analyses, constant propagation, inter-procedural optimizations, inlining, value numbering, and loop transformations: 16 components spread across 34 files are affected (Table 4).

This demonstrates that our approach can target and uncover missed opportunities in a variety of compiler components.

4.3 Reported Bugs

Reporting missed optimization opportunities and receiving feedback from the compiler developers helps us assess the utility of our approach. We reported 53 GCC missed opportunities for GCC and 31 for LLVM. Table 5 shows the status of the reported cases. Out of

Table 4: The commits introducing missed DCE opportunities affect a diverse set of GCC components.

Component	# Commits	# Files
Alias Analysis	3	1
C-family Frontend	1	4
Common Subexpression Elimination	3	2
Constant Propagation	4	2
Control Flow Graph Analysis	1	2
Copy Propagation	1	1
Inlining	3	2
Interprocedural Analyses	1	1
Interprocedural SRoA [13]	1	1
Jump Threading	1	3
Loop Transformations	3	2
Pass Management	2	2
Peephole Optimizations	1	1
Target Info	1	1
Value Numbering	3	2
Value Propagation	6	7

53 GCC reported cases, 43 were confirmed as unique and 12 fixed. Out of 31 LLVM reported ones, 19 were confirmed and 11 fixed. Out of the 5 bugs marked as duplicates in GCC, one was previously reported by a different developer; this demonstrates that our approach can find relevant missed opportunities. We received very positive feedback from the compiler developers and one replied:

> "That was a very interesting discovery! Looking forward to more!"

Test case reduction Reducing the discovered cases is helpful for compiler developers in understanding and addressing them. We used C-Reduce [28] for reductions; the "interestingness" check, *i.e.*, whether a reduced candidate should be kept or discarded, is the same as before: one compiler eliminates a dead call and another one does not. We use compiler warnings, sanitizers [29], and whenever possible CompCert's C interpreter [22] to detect and reject invalid code or code with Undefined Behavior (UB) [34, 35]; while this does not guarantee the absence of UB, it makes manually inspecting the reduced case easier.

Table 5: Missed Optimizations Reported, Confirmed, Marked Duplicate, and Fixed

	GCC	LLVM
Reported	53	31
Confirmed	43	19
Marked Duplicate	5	0
Fixed	12	11

Listing 8: Example LLVM missed DCE opportunities. The dead parts are highlighted with light gray.

(a) https://bugs.llvm.org/show_bug.cgi?id=49773: The dead call was eliminated at `-O2` but not at `-O3`. Confirmed bug.

```
static int a;
int **b;
int c;
int d;
int e;

int main() {
 if (a) {
   dead();
 }
 while (e) {
  while (a++) {
   while (c) {
    *b = &d;
   }
  }
 }
 return 0;
}
```

(b) https://bugs.llvm.org/show_bug.cgi?id=49731: The dead call was eliminated at `-O1` but not at `-O3`. Fixed with `611a02cce509`.

```
static long a = 78240;
static int b, d;
static short e;
static short c(short f
    ,
               short h)
                   {
 return h == 0 ||
        (f && h == 1)
            ?
        0 : f % h;
}
int main() {
 short g = a;
 for (b=0; b<1; b++) {
  e = a;
  d = c((e == a)^g, a)
      ;
 }
 if (d) {
   dead();
   for (; a; a++);
 }
 return 0;
}
```

Examples of reported bugs To further highlight the diversity of the uncovered missed opportunities, we discuss a small selection. Listing 8 shows two missed opportunities in LLVM, Listing 9 shows six missed opportunities in GCC.

- **Listing 8a**: A regression which involves enabling more extensive loop unswitching caused LLVM to miss eliminating the `dead` call. Prior versions turned the first load of `a` into 0, but the new interactions between loop unswitching and constant propagation prevent this.
- **Listing 8b**: LLVM could not eliminate the `dead` call after a recent regression affecting `-O3`. Modulo operations on constant ranges of the form $[X, X+1)\ \%\ [Y, Y+1)$ could not be simplified. This was an omission as this was possible for other operations such as `and`. Fixed with commit `611a02cce50`.
- **Listing 9a**: GCC was missing the relation $X << Y\ != 0 \rightarrow X != 0$. Fixed with `5f9ccf17de7`.
- **Listing 9b**: GCC failed to eliminate the `dead` call at `-O3`. It optimized `main` to `return 0;` at `-O1`, but a SRA copy of `d` was created [13] but not eliminated (cleaned up).
- **Listing 9c**: GCC could not determine that `d` does not alias `b` at `-O3` and as a result the dead call was not eliminated. Interestingly, this was not an issue at `-O1`. Fixed with `d1d01a66012`.
- **Listing 9d**: GCC at `-O3` did not properly clean up the IR after removing the dead store to `c`. A leftover phi node confused the value range propagation (VRP) and jump threading passes, which thread through dead code. GCC's older jump threaders were not able to find any threading opportunities and as a result the dead call was previously eliminated, *i.e.*, this is a regression. Fixed with `113860301f4`.
- **Listing 9e**: GCC failed to eliminate the call to `dead` at `-O3`, but not at `-O1`. It vectorized the loop at `-O3`, but using unsigned long as the internal type for the vectorized pointer data. This prevented constant folding and eventually DCE. Fixed with commit `7d6bb80931b`.
- **Listing 9f**: GCC could not determine that, irrespective of the index value, the same constant is loaded from `b`, which prevents constant-folding and subsequently dead code elimination. This is a duplicate of #80603 which was previously discovered by GCC's developers. This shows that our approach can discover "real-world" performance bugs.

4.4 Discussion

Capabilities and limitations Our approach can uncover many kinds of missed optimizations as evidenced by our evaluation (*e.g.*, see Table 3 and Table 4). In theory, DCE can directly capture all missed optimizations related to data-flow analyses: any data-flow analysis result can be converted to an `if(expr)` check which can be eliminated by DCE. In practice, our approach captures many additional transformations that interact with data-flow analyses, *e.g.*, peephole optimizations or loop transformations, since they affect analysis results upon which the effectiveness of DCE depends. On the other hand, our approach cannot detect missed optimizations that have no impact on DCE, such as vectorization or register allocation. Although we target C and C++, our approach is general and applicable to any source language and can be used to test any static compiler, however, its applicability to dynamic compilers is more limited since interpreters typically record whether a block is executed; if it is not, it might not be compiled at all.

Control-flow versus data-flow based DCE Our approach is applicable to both two types of dead code, control-flow based and data-flow based. Our instrumentation captures control-flow based dead code by instrumenting basic blocks (see Listing 2a). Data-flow based dead code concerns useless definitions based on def-use information. A useless definition fails to be eliminated mostly due to spurious def-use information, *i.e.*, spurious uses residing in missed dead blocks. Thus, our approach is general and captures both types.

DCE as an optimization oracle Our approach relies on DCE to measure a compiler's optimization effectiveness. A potential concern could be that compilers only selectively apply DCE, which would limit the applicability of our approach. While it has been shown that transformations, such as DCE, might negatively affect performance in special cases, such as when they affect the binary's layout [25, 26]), there seems to be a broad consensus that DCE is an optimization that should be applied whenever possible, because it has no obvious negative side-effects. This is in contrast to many

Listing 9: Example cases uncovering a diverse array of GCC missed DCE opportunities. In all cases GCC fails to either eliminate the call to `dead` (`extern void dead(void)`) in general, or it is able to eliminate at `-O1` but not at `-O3`. The dead parts are highlighted with light gray.

(a) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=102546: GCC could not deduce that `X « Y != 0` implies `X != 0`. as a result the dead call was not eliminated. Fixed with `5f9ccf17de7`.

```
static int a;
static char b, c, d;
int main() {
  int f = 0;
  for (; f <= 5; f++) {
    bar();
    b = b && f;
    d = f << f;
    if (!(a >= d || f)) {
      dead();
    }
    c = 1;
    for (; c; c = 0)
    ;
  }
}
```

(b) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100034: GCC at `-O3` optimized `main` to just `return 0;` however, it failed to eliminate an unused interprocedural SRA copy [13] of `d`. GCC at `-O1` did not have this issue. Confirmed bug.

```
static int a, b, f, g;
static int d() {
  while (g) f = 0;
  while (1) dead();
}
static void c() { d (); }
void e() {
  while (b) {
    if (!a) continue;
    c();
  }
}
int main() {
  e();
  return 0;
}
```

(c) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100051: GCC failed to determine that `d` does not alias `b` at `-O3` (but it could at `-O1`) and as a result, the dead call was not eliminated. Fixed with `d1d01a66012`.

```
int a, c, *f, **d = &f;
char b;
static void e() {
  if ((2 ^ b) == 0) dead();
}
int main() {
  if (a) {
    b = 0;
    int *g = &c;
    *g = 0;
    f = *d;
    *d = f;
    e();
  }
  return 0;
}
```

(d) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=102703: GCC at `-O3` did not properly clean up the IR after removing the dead store to `c` which confused the value range propagation and jump threading passes, as a result the dead call was not eliminated. Fixed with `113860301f4`.

```
static int a, b;
static short c;
int main() {
  for (; a; ++a) {
    unsigned short d = a;
    c = d >= 2 ? 0 : 2;
    if (!(b | d) && d)
      dead();
  }
}
```

(e) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99776: GCC eliminated the call at `-O1`; at `-O3` the loop was vectorized, however, pointer arrays are vectorized as unsigned longs and this lead to a type mismatch which prevented constant folding. Fixed with `7d6bb80931b`.

```
static int a[2];
static int b;
static int *c[2];
int main() {
  for (b = 0; b < 2; b++) {
    c[b] = &a[1];
  }
  if (!c[0]) dead();
  return 0;
}
```

(f) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99419: Rediscovered bug: GCC failed to determine that no matter the index the same constant, 0, is loaded from `b`, this prevented constant folding and dead-code eliminating the call.

```
int a;
static int b[2] = {0,0};

int main() {
  if (b[a]) {
    dead();
  }
  return 0;
}
```

other compiler optimizations which may decrease the resulting binary's performance if applied when they should not, *e.g.*, deciding whether to vectorize a loop is not a simple binary decision [27].

Optimization marker alternatives Optimization markers can also be as compiler builtins, inline assembly, or writes to global variables. For example, a write to a global variable instead of a function call would be eligible for elimination *iff* it is never executed; to determine if such a write is actually eliminated, it is necessary to determine if the corresponding `mov` instruction exists in the generated assembly. We chose functions because they do not require modifying any compiler (unlike introducing a new builtin), and they also allow us to record their execution by linking a test case with appropriate function implementations.

Uncovering missed optimizations in practice A potential concern might be that developers are not interested in addressing the missed optimizations. However, the response from the compiler developers who confirmed and fixed our reported bugs was highly

positive (see quote in Section 4.3): out of the 62 confirmed reported bugs, 23 were fixed. We also rediscovered a missed optimization that was previously reported by compiler developer (see Listing 9f); this further demonstrates the practical relevance of our approach.

In this work, we used a single compiler version for LLVM and GCC to discover missed optimization opportunities. However, differential testing can be applied to different versions of the same compiler. For example, the latest development branch can be continuously tested against its previous release to monitor for new regressions. Specific commits, *e.g.*, ones that involve significant changes, can also be stress tested against their earlier commits. Both are practical ways to utilize and adopt our work.

Future directions We demonstrated our approach by instrumenting source code elements that roughly correspond to basic blocks. Several extensions are possible that do not rely on the existence of dead blocks. For example, inserting checks of the form `if (v != C) DCECheck();`, where `v` is a program variable and C a constant, which can, *e.g.*, be derived by running the program and recording v's value(s). This can facilitate testing specific analysis, *e.g.*, such checks can be inserted after loops to test scalar evolution. Another possibility would be to directly generate code that contains challenging dead code elimination opportunities instead of using instrumented existing or auto-generated code.

Applicability of our approach While we evaluated our approach using Csmith-generated programs, missed optimizations can be found using other programs as well. We primarily relied on Csmith to determine how well state-of-the-art compilers compare to a hypothetical optimal compiler; this was possible because the generated programs are deterministic and require no inputs, and, thus we could compute the actual set dead and alive blocks. Note that Csmith generates sequential programs, but our approach is general and not restricted to sequential code: if compiler `A` eliminates a piece of dead code in a parallel program and compiler `B` does not, then B has missed an optimization opportunity.

5 RELATED WORK

Differential testing The most directly related work is by Barany [4] on directly comparing elements in the generated assembly of different compilers,*e.g.*: number of memory loads and stores, register copies, and floating-point operations. In contrast, our work relies on dead code elimination to uncover a broader range of missed optimizations (as demonstrated in Section 4.3). Barany's work requires crafting targeted assembly code matcher for differential testing, whereas in our approach we do not know in advance what kind of missed optimizations our optimization markers will uncover, therefore the two approaches are not directly comparable.

Missed opportunities in equivalent code Hashimoto et al. proposed randomly generating equivalent programs arithmetic expressions for differential testing [12]; Gong et al. used source-level loop transformations to discover missed opportunities [11]. Both of these approaches target specific compiler components unlike our approach which is more generic.

Data-flow analysis testing Taneja et al. proposed a white-box approach based on SMT-solvers for exposing issues in various data-flow analyses of LLVM such as integer ranges: the solver helps to check if the compiler's result are imprecise [31]. Our approach does not directly target such analyses but instead focuses on the overall optimization results; in addition, our approach is black-box and it does not require any additional effort to support more compilers.

Benchmarking Benchmarking is the standard practice for detecting regressions in compilers. Popular benchmark suites for C/C++ include SPEC 2017 [6] and PolyBench [17]. Benchmarking can reveal the existence of regressions, track improvements, and identify performance bottlenecks on specific benchmarks with specific inputs. However, it is inapplicable for pinpointing the exact changes that caused performance bugs and it cannot be used to discover existing performance bugs; both of which our work tackles.

Uncovering performance bugs in generic software Several approaches have been proposed for finding performance bugs and/or regressions in user applications. Approaches include finding pathological program inputs [21], synthesizing programs that use certain functions and expose performance bottlenecks [32], or focusing on certain kinds of targets such as database management systems [15]. Empirical studies that explored performance bugs in the "real-world" also exist [14]. To our knowledge, none of such previous performance bug works focused specifically on compilers.

Dynamic analysis for performance bug discovery Several dynamic approaches for exposing missed optimizations exist. Deadspy [8] identifies dead stores by dynamically detecting successive writes to the same memory locations without a read between them. Runtime value numbering [36] identifies computation redundancies that compilers failed to eliminate. CIDetector [30] detects dead and redundant stores as well as redundant loads. There is an overlap in the class of missed optimizations found by our approach. However, our approach is designed to be more general and static, and it does not specifically target these analyses/optimizations.

Automated compiler testing Many automated approaches for finding correctness bugs in compilers exist. Examples include among others the development of Csmith which heavily is used for differential testing on compilers [37], and EMI is an approach that derives programs that are equivalent given a certain input used for compiler testing [19, 20]. Such efforts have been very successful in uncovering hundreds of correctness bugs in compilers. Neither of these approaches can be used for finding performance bugs. However, for the realization of our approach, we relied on previously-developed tools, such as Csmith and C-Reduce.

6 CONCLUSION

We have proposed a new approach for uncovering missed optimization opportunities in compilers. We uncovered, analyzed, and reported a diverse set of missed optimizations involving many compiler components: out of the 84 reported bugs, 62 were confirmed or fixed. We performed our investigation though the lens of dead code elimination: we were able to systematically quantify how well compilers optimize code by instrumenting programs with *optimization markers* and exploiting that DCE heavily depends on the rest of the optimization pipeline.We expect our methodology to help further understand the optimization capabilities of state-of-the-art compilers and to aid the discovery of missed optimizations and regressions. This work is the first step towards a promising direction for future work.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains the code and dataset we used for our experiments, as well as scripts to generate the numbers and tables of our evaluation. Specifically, it includes (a) the corpus of randomly generated programs that we used in Section 4's evaluation; (b) scripts for generating a new corpus and validating the existing one; (c) our LLVM-based optimization marker instrumenter; (d) scripts for generating the missed optimization statistics presented in Section 4; (e) the full list of submitted bug reports with links to the respective compiler bug trackers; (f) end-to-end examples that led to bug reports. Everything is packaged and pre-built as a docker image. A standard X86 Linux machine running docker is necessary to evaluate this artifact.

A.2 Artifact Checklist (Meta-information)

- **Data set:** Autogenerated (Csmith) C programs
- **Run-time environment:** Linux
- **Hardware:** X86
- **Output:** Statistics on missed optimizations for GCC and LLVM
- **How much disk space required (approximately)?:** 15G
- **How much time is needed to prepare workflow (approximately)?:** A few minutes to download and import the docker image.
- **How much time is needed to complete experiments (approximately)?:** A few minutes to validate the existing results, or up multiple hours to generate new results.
- **Publicly available?:** Yes
- **Code licenses:** MIT
- **Archived (DOI):** 10.5281/zenodo.5870184

A.3 Description

A.3.1 How to Access. The artifact can be downloaded from https://zenodo.org/record/5870184.

A.3.2 Hardware Dependencies. A standard X86 computer.

A.3.3 Software Dependencies. Docker.

A.4 Installation

```
tar xf ASPLOS22-DCE-Artifact.tar.gz
cat dce-artifact-image.tar|docker import - dce_artifact
```

A.5 Experiment Workflow

(1) Generate random C programs with Csmith.
(2) Instrument them with optimization markers.
(3) Generate the ground truth, *i.e.*, which markers are actually dead and which alive; generate for each compiler and optimization level the set of eliminated and non-eliminated markers.
(4) Compare the results to identify missed dead markers.

A.6 Evaluation and Expected Results

The existing corpus can be used to regenerate the results of Section 4. We provide scripts to perform all the steps in Section A.5 and generate the Tables and numbers in Section 4 (the instructions are in README.md).

REFERENCES

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[2] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42. https://doi.org/10.1145/3197978

[3] David F Bacon, Susan L Graham, and Oliver J Sharp. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)* 26, 4 (1994), 345–420. https://doi.org/10.1145/197405.197406

[4] 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

[5] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–27. https://doi.org/10.1145/3133876

[6] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42. https://doi.org/10.1145/3185768.3185771

[7] David Callahan. 1988. The program summary graph and flow-sensitive interprocedual data flow analysis. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. 47–56. https://doi.org/10.1145/53990.53995

[8] Milind Chabbi and John Mellor-Crummey. 2012. Deadspy: a tool to pinpoint program inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 124–134. https://doi.org/10.1145/2259016.2259033

[9] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. 2010. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 42–52. https://doi.org/10.1145/1772954.1772963

[10] Armando Fox, Michael Hsiao, James Reed, and Brent Whitlock. [n.d.]. A Survey of General and Architecture-Specific Compiler Optimization Techniques.

[11] Zhangxiaowen Gong, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, Alexander Veidenbaum, Alexandru Nicolau, et al. 2018. An empirical study of the effect of source-level loop transformations on compiler stability. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. https://doi.org/10.1145/3276496

[12] Atsushi Hashimoto and Nagisa Ishiura. 2016. Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs. *IPSJ Transactions on System LSI Design Methodology* 9 (2016), 21–29. https://doi.org/10.2197/ipsjtsldm.9.21

[13] Martin Jambor. 2010. The new intraprocedural Scalar Replacement of Aggregates. In *GCC Developers' Summit*. 47.

[14] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance 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, 77–88. https://doi.org/10.1145/2254064.2254075

[15] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70. https://doi.org/10.14778/3357377.3357382

[16] Sesha Kalyur and GS Nagaraja. 2016. A survey of modeling techniques used in compiler design and implementation. In *2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*. IEEE, 355–358. https://doi.org/10.1109/CSITSS.2016.7779385

[17] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2018. Polybench: The first benchmark for polystores. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 24–41. https://doi.org/10.1007/978-3-030-11404-6_3

[18] J. C. Knight and N. G. Leveson. 1986. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Trans. Softw. Eng.* 12, 1 (jan 1986), 96–109. https://doi.org/10.1109/TSE.1986.6312924

[19] 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

[20] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

[21] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM*

SIGSOFT International Symposium on Software Testing and Analysis (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[22] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[23] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[24] Tipp Moseley, Dirk Grunwald, and Ramesh Peri. 2009. OptiScope: performance accountability for optimizing compilers. In *2009 International Symposium on Code Generation and Optimization*. IEEE, 254–264. https://doi.org/10.1109/CGO.2009.26

[25] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

[26] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14. https://doi.org/10.1109/CGO.2019.8661201

[27] Vasileios Porpodas and Timothy M Jones. 2015. Throttling automatic vectorization: When less is more. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 432–444. https://doi.org/10.1109/PACT.2015.32

[28] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346. https://doi.org/10.1145/2254064.2254104

[29] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (ATC 12)*. 309–318.

[30] 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*. 1–12. https://doi.org/10.1145/3392717.3392754

[31] 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

[32] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2018. Synthesizing Programs That Expose Performance Bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 314–326. https://doi.org/10.1145/3168830

[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*. 147–156. https://doi.org/10.1145/1128022.1128042

[34] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *Proceedings of the Asia-Pacific Workshop on Systems*. 1–7. https://doi.org/10.1145/2349896.2349905

[35] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 260–275. https://doi.org/10.1145/2517349.2522728

[36] Shasha Wen, Xu Liu, and Milind Chabbi. 2015. Runtime value numbering: A profiling technique to pinpoint redundant computations. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 254–265. https://doi.org/10.1109/PACT.2015.29

[37] 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*. 283–294. https://doi.org/10.1145/2254064.2254075

[38] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes* 27, 6 (2002), 1–10. https://doi.org/10.1145/587051.587053