The Journal of Systems & Software 174 (2021) 110884

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An empirical study of optimization bugs in GCC and LLVM

Zhide Zhou [a,b], Zhilei Ren [a,b,*], Guojun Gao [a,b], He Jiang [a,b,c]

[a] School of Software, Dalian University of Technology, Dalian, China
[b] Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, China
[c] School of Computer Science & Technology, Beijing Institute of Technology, Beijing, China

A R T I C L E   I N F O

Article history:
Received 15 April 2020
Received in revised form 29 October 2020
Accepted 2 December 2020
Available online 14 December 2020

Keywords:
Empirical study
Compiler reliability
Bug characteristics
Compiler optimization bugs
Compiler testing

A B S T R A C T

Optimizations are the fundamental component of compilers. Bugs in optimizations have significant impacts, and can cause unintended application behavior and disasters, especially for safety-critical domains. Thus, an in-depth analysis of optimization bugs should be conducted to help developers understand and test the optimizations in compilers. To this end, we conduct an empirical study to investigate the characteristics of optimization bugs in two mainstream compilers, GCC and LLVM. We collect about 57K and 22K bugs of GCC and LLVM, and then exhaustively examine 8,771 and 1,564 optimization bugs of the two compilers, respectively. The results reveal the following five characteristics of optimization bugs: (1) Optimizations are the buggiest component in both compilers except for the C++ component; (2) the value range propagation optimization and the instruction combine optimization are the buggiest optimizations in GCC and LLVM, respectively; the loop optimizations in both GCC and LLVM are more bug-prone than other optimizations; (3) most of the optimization bugs in both GCC and LLVM are misoptimization bugs, accounting for 57.21% and 61.38% respectively; (4) on average, the optimization bugs live over five months, and developers take 11.16 months for GCC and 13.55 months for LLVM to fix an optimization bug; in both GCC and LLVM, many confirmed optimization bugs have lived for a long time; (5) the bug fixes of optimization bugs involve no more than two files and three functions on average in both compilers, and around 99% of them modify no more than 100 lines of code, while 90% less than 50 lines of code.

Our study provides a deep understanding of optimization bugs for developers and researchers. This could provide useful guidance for the developers and researchers to better design the optimizations in compilers. In addition, the analysis results suggest that we need more effective techniques and tools to test compiler optimizations. Moreover, our findings are also useful to the research of automatic debugging techniques for compilers, such as automatic compiler bug isolation techniques.

1. Introduction

Compilers (e.g., GCC, LLVM) are one of the most important system software. They play an important role in translating source code into machine code. In addition, compilers also provide optimization techniques to improve the performance of programs. Hitherto, optimizations have become a fundamental component of compilers. Hundreds of optimizations have been implemented in compilers. As an example, there are more than 200 and 100 optimizations for GCC and LLVM (Ashouri et al., 2017), respectively. However, with the development of compilers, similar to application software, bugs in optimizations of compilers are inevitably introduced. Optimization bugs can cause unintended behaviors and disasters, especially for safety-critical domains.

Many techniques (Yang et al., 2011; Nagai et al., 2012; Le et al., 2014; Vu et al., 2015a,b; Sun et al., 2016a; Zhang et al., 2017; Liu et al., 2019) have been developed to detect compiler bugs. Especially, Csmith (Yang et al., 2011) is the most used testing program generator for C/C++ compilers, with which more than 400 compiler bugs have been reported.[1] In addition, the techniques (Le et al., 2014; Sun et al., 2016a; Vu et al., 2015a) based on equivalence modulo inputs (Le et al., 2014) have detected more than 1600 compiler bugs.[2] However, there is no study to help developers and researchers understand optimization bugs in compilers. Although an empirical study on compiler bugs has been conducted by Sun et al. (2016b), it focuses on the overall situation on compiler bugs in GCC and LLVM. Besides, not all application developers are aware of compiler optimization bugs in practice. Especially for inexperienced developers (e.g., students),

* Corresponding author at: School of Software, Dalian University of Technology, Dalian, China.
E-mail addresses: cszide@gmail.com (Z. Zhou), zren@dlut.edu.cn (Z. Ren), ggj_gao@mail.dlut.edu.cn (G. Gao), jianghe@dlut.edu.cn (H. Jiang).

1 https://embed.cs.utah.edu/csmith/.
2 https://web.cs.ucdavis.edu/~su/emi-project/.

https://doi.org/10.1016/j.jss.2020.110884
0164-1212/© 2020 Elsevier Inc. All rights reserved.

they always assume that bugs occurred in their programs are introduced by themselves rather than the compiler optimizations. Therefore, for better understanding, detection, and fix of optimization bugs in compilers, we need to further uncover the properties of optimization bugs.

In order to gain a better understanding of optimization bugs in compilers, we conduct an empirical study to investigate their characteristics in this study. Similar to the study conducted by Sun et al. (2016b), we also investigate two mainstream production compilers, GCC and LLVM. We collect 57,591 bugs of GCC and 22,119 bugs of LLVM, and then exhaustively examine 8771 and 1564 optimization bugs of GCC and LLVM, respectively. In particular, we mainly focus on the following five research questions (RQs) to investigate optimization bugs. In these RQs, RQ1 and RQ2 focus on the overall trend of optimization bugs in GCC and LLVM, which can help developers and researchers to understand the evolution history and the distribution of optimization bugs. While RQ3 further provides the type information of optimization bugs to developers and researchers. RQ4 and RQ5 investigate the duration and fix for optimization bugs, which can guide testing and debugging of compiler optimizations.

(1) What is the distribution of optimization bugs?

This RQ investigates the overall evolution history of optimization bugs and the distribution of optimization bugs in components, which can help to understand the importance of optimization bugs for compiler development. The results show that the trend of the optimization bugs in GCC is relatively stable in these years, while it has an increasing trend for LLVM. In addition, optimization bugs account for a higher percentage of the total bugs than the bugs in other components except for the C++ component. Furthermore, most of the optimization bugs are located in the *tree-optimization* component in GCC, and the *Scalar optimization* component for LLVM.

(2) Which optimizations are buggy?

This RQ aims to investigate which optimizations are buggy. We extract 119 and 101 files that are directly related to the optimizations in GCC and LLVM, respectively. Then we study the frequency that each file is changed by the revisions. The results illustrate that the value range propagation optimization and the instruction combine optimization are the buggiest optimizations in GCC and LLVM, respectively. Additionally, the loop optimizations in both GCC and LLVM have more bugs than other optimizations.

(3) What are the types of optimization bugs?

We investigate the types of optimization bugs in this study. According to the compiler testing literatures (e.g., Yang et al., 2011; Le et al., 2014; Vu et al., 2015a,b; Sun et al., 2016a; Zhang et al., 2017), we manually assign a type (i.e., *Misoptimization* (*Mis-opt*), *Crash*, or *Performance*, see Section 2.2 for definitions) to each optimization bug. The results show that most of the optimization bugs in both GCC and LLVM belong to *Mis-opt* bugs, accounting for 57.21% and 61.38% respectively. However, in both compilers, the bug-fixing rate of *Mis-opt* bugs is smaller than those of *Crash* bugs and *Performance* bugs.

(4) How long do optimization bugs live?

This RQ studies the time that optimization bugs live. On average, optimization bugs live over five months, and it takes 11.16 months for GCC and 13.55 months for LLVM to fix an optimization bug. In GCC, the optimization bugs in the *rtl-optimization* component take more time to be fixed, 13.57 months on average; and the developers take 15.07 months to fix *Mis-opt* bugs, which is longer than those of other types of bugs. While the most time-consuming bug fixes belong to the *Interprocedural Optimizations* component in LLVM, 17.77 months on average; and *Performance* bugs take 15.73 months on average to be fixed, which is longer than those of *Mis-opt* bugs and *Crash* bugs in LLVM. In both GCC and LLVM, many confirmed optimization bugs have lived for a long time. On average, the confirmed optimization bugs for GCC have lived for 72.38 months, while 14.38 months for LLVM.

(5) How many files, functions, and code lines are modified to fix an optimization bug?

This RQ investigates the information of bug fixes for optimization bugs. In particular, we study the modification of files, functions, and lines of code to fix optimization bugs. The results show that the bug fixes involve no more than two files and three functions on average in both compilers. In GCC, the bug fixes of the bugs in the tree-optimization component involve more files and functions, 1.81 files and 3.84 functions on average; and the bug fixes involve 2.82 files and 7.18 functions on average for *Performance* bugs, which is larger than those of other types of bugs. In LLVM, the bug fixes of the bugs in the *Interprocedural Optimizations* component involve more files (1.75 files on average), while the bug fixes in *Transformation Utilities* involve more functions (2.67 functions on average); and the bug fixes of *Mis-opt* bugs need to modify more files (1.44 files on average), while the bug fixes of *Crash* bugs touch more functions (2.39 functions on average). In both compilers, around 99% of the bug fixes only modify no more than 100 lines of code, while 90% fewer than 50 lines of code.

In summary, our study provides a deep understanding of compiler optimization bugs for developers and researchers. The analysis results and findings could be useful to guide developers and researchers to better design optimizations in compilers. For example, in both GCC and LLVM, the loop optimizations are more bug-prone than other optimizations, which may indicate that there are some defects in the design of loop optimizations and the developers should pay more attention to loop optimizations. In addition, the results of our study also suggest that we need more effective techniques and tools to test compiler optimizations. Moreover, our study can provide useful information to the research of automatic debugging techniques for compilers, such as the automatic compiler bug isolation techniques (Chen et al., 2019). For instance, our findings show that developers modify no more than two files and three functions on average to fix optimization bugs in both GCC and LLVM, which may indicate that we can first recognize the buggy optimizations to narrow down the suspect buggy files and functions when we try to isolate the root causes of optimization bugs.

The rest of the paper is organized as follows. Section 2 describes the methodology of our study. Section 3 presents the results of the distribution of optimization bugs. Section 4 shows the buggy optimizations. The types of optimization bugs are presented in Section 5. The analysis results of the time that optimization bugs live are shown in Section 6. Section 7 describes the modification of files, functions, and lines of code for fix optimization bugs. Next, we present the threats to validity and related work in Sections 8–9. Finally, Section 10 concludes our study.

2. Methodology

2.1. Compilers

In this study, we select two mainstream compiler systems, namely, GCC and LLVM, to investigate their optimization bugs. These two compiler systems are open-source and widely used in the industry and academia.

GCC GCC indicates the GNU Compiler Collection,[3] which is a typical three-stage (including front end, middle end, and back end) compiler system.[4] It includes front ends for various programming

[3] https://gcc.gnu.org/.

[4] https://en.wikipedia.org/wiki/Compiler.

languages (e.g., C, C++, Objective-C, Fortran) and back ends for various target architectures (e.g., X86, MIPS, PowerPC, and RISC-V). In addition, it provides more than 200 optimizations (GNU Compiler Community, 2020) to improve program performance. Since the late 1980s, it has been under active development for more than 30 years.

LLVM LLVM is a mature and widely used compiler infrastructure.[5] Similar to GCC, LLVM also is a three-stage compiler system and supports multiple programming languages and multiple target architectures. It provides a collection of modular and reusable compiler and toolchain technologies for arbitrary programming languages. With a common infrastructure, a broad variety of statically and runtime compiled languages (e.g., the family of languages supported by GCC, Rust, Swift, Ruby, Haskell, and WebAssembly) have been implemented based on LLVM. In addition, hundreds of analysis and transformation optimizations have been developed in LLVM (LLVM Compiler Community, 2020). From its beginning in December 2000, LLVM has drawn much attention from both industry and academia.

2.2. Bug sources

We collect bugs of GCC and LLVM from their bug repositories.[6] Similar to the existing study (Sun et al., 2016b), we take the bugs that are confirmed and fixed into consideration in our study. Thus, we can comprehensively understand the characteristics of optimization bugs in GCC and LLVM. For a bug, if its resolution field is set to *fixed*, and the status field is set to *resolved*, *verified* or *closed* in the bug repositories of GCC and LLVM, it is a fixed bug. Different from the fixed bug, developers of GCC treat a bug with *new* status as a confirmed bug.[7] Thus, we say a bug is confirmed in the GCC bug repository, if its status field is set to *new*, and the resolution field is empty. While the confirmed bug of LLVM indicates that its status field is set to *confirmed*, and the resolution field is empty.

Identifying Optimization Bugs In the bug repositories of GCC and LLVM, bugs have been classified according to the components they belong to. In GCC, there are three main intermediate languages to represent the program during compilation: GENERIC, GIMPLE, and RTL,[8] where GIMPLE and RTL are used to optimize the program. GIMPLE is an abstract-syntax-tree-based representation and RTL represents the Register Transfer Language. Thus, for optimization bugs of GCC, developers of GCC usually specify the components of them as the "*tree-optimization*" component or the "*rtl-optimization*" component out of the 52 components in GCC.[9] Therefore, we identify optimization bugs of GCC according to the components "*tree-optimization*" and "*rtl-optimization*" in this study. Similarly, the optimization bugs of LLVM are identified according to the components "*Scalar Optimizations*", "*Loop Optimizer*", "*Transformation Utilities*", and "*Interprocedural Optimizations*" out of 96 components in LLVM. These components related to optimizations in LLVM are set according to the directories that include the buggy optimizations and the functionality of optimizations.[10] For example, the component "Scalar Optimizations" indicates that the buggy optimizations belong to the directory "scalar" in the LLVM project, while the component "Loop Optimizer" includes the optimization bugs that are triggered in loop optimizations.

Identifying Bug Types To investigate the bug type information of optimization bugs, we identify the bug types of optimization bugs. According to the compiler testing literatures, such as literatures (Yang et al., 2011; Le et al., 2014; Vu et al., 2015a; Lidbury et al., 2015; Sun et al., 2016a; Zhang et al., 2017), we divide the optimization bugs into the following three categories:

Crash We say an optimization bug is crash if the optimization crashes for optimizing the code, i.e., internal compiler errors and memory-safety errors for the optimizations.

Misoptimization If the optimized code crashes, terminates abnormally, or produces a wrong output, we say the corresponding optimization bug is misoptimization.

Performance The performance bug of an optimization means that the compiler hangs or the compilation abnormally slows due to the optimization.

To guarantee the correctness of bug types for optimization bugs, the classification of optimization bugs is manually conducted by three authors, and the type of each optimization bug must be agreed upon by at least two authors. In the bug reports of GCC and LLVM, there is a field "Keywords",[11] which provides straightforward information to help us quickly recognize the type of bugs. For example, the value in the field "Keywords" of an optimization bug report is "wrong-code", which indicates that the optimization bug can be classified into *Misoptimization*. However, not all bug reports have an exact value in the field "Keywords" and some bug reports have multiple values in the field "Keywords". In these cases, we need to further read the description and comments of optimization bug reports to assign a type to them.

Identifying Bug Fix Revisions To study the bug fix information of optimization bugs, we identify the revisions of the corresponding fixed optimization bugs as in Sun et al. (2016b). First, we collect the entire revision log from the code repositories using the *git*[12] command "*git log -p –follow –stat*", since GCC and LLVM use *git* as their version control system. Then, for each revision, we check whether this revision is a fix to an optimization bug according to the following three patterns in its commit message. This is because the developers of GCC and LLVM usually mark a bug using one of the three patterns in the commit message (Sun et al., 2016b).

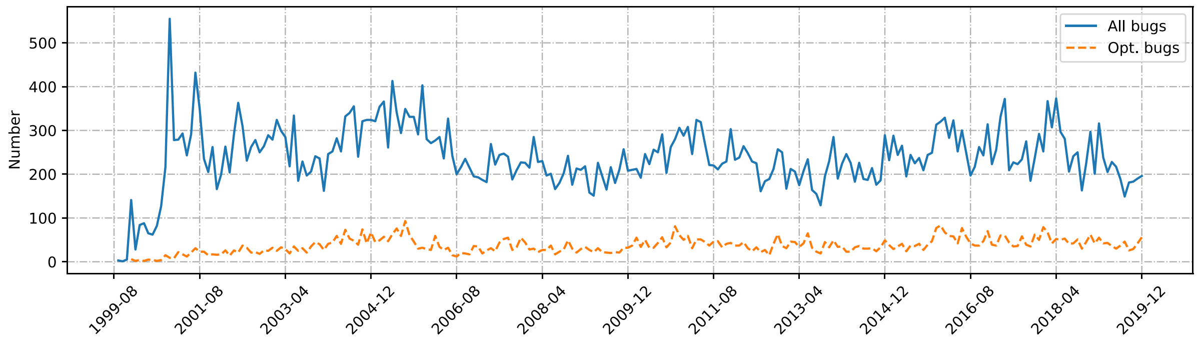
- "PR<bug-id>"
- "PR <bug-id>
- "PR <component>/<bug-id>"

where "PR" means "Problem Report" and <bug-id> stands for the id of the corresponding bug. Note that the literature (Sun et al., 2016b) only contains the second and the third pattern. The reason is that we find some developers of GCC and LLVM also use the first pattern in the commit to show that it is a fix for a bug. For example, the commit[13] in LLVM contains "PR39475", which means that this revision fixes bug 39475.[14] After obtaining the corresponding revision of an optimization bug, we parse it to collect the information about the modified files, functions, and code lines.

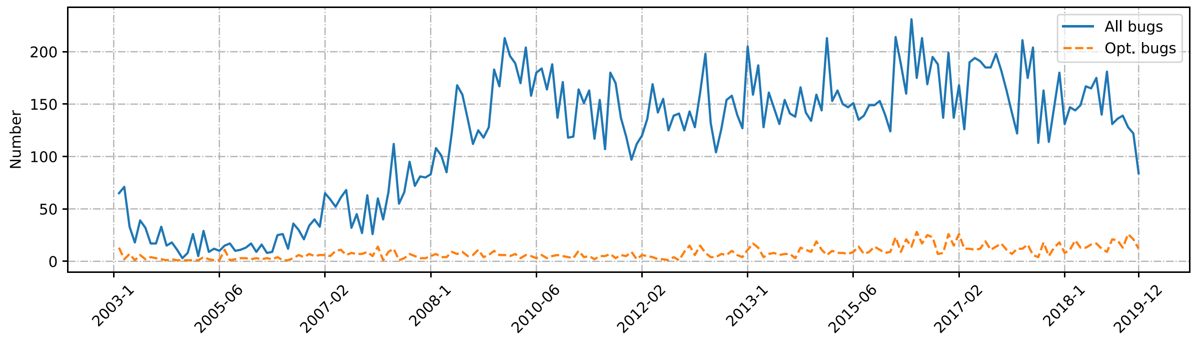
Table 1 shows the bug information used in this study. Generally, we obtain 57,591 GCC bugs and 22,748 LLVM bugs, which account for 66.17% and 52.49% of all bugs in GCC and LLVM, respectively. In particular, we collect 8771 GCC optimization bugs and 1564 LLVM optimization bugs. These optimization bugs account for 76.28% and 55.40% of all optimization bugs in GCC and

5 http://llvm.org/.
6 https://gcc.gnu.org/bugzilla/, https://bugs.llvm.org/.
7 https://www.gnu.org/software/gcc/bugs/management.html.
8 https://gcc.gnu.org/onlinedocs/gccint/Tree-SSA.html#Tree-SSA.
9 https://gcc.gnu.org/bugzilla/describecomponents.cgi?product=gcc.
10 https://bugs.llvm.org/describecomponents.cgi?product=libraries.

11 https://gcc.gnu.org/bugzilla/describekeywords.cgi, https://bugs.llvm.org/describekeywords.cgi.
12 https://git-scm.com/.
13 https://reviews.llvm.org/D53844.
14 https://bugs.llvm.org/show_bug.cgi?id=39475.



(a) GCC



(b) LLVM

Fig. 1. The overall evolution history of the optimization bugs (in months)..

Table 1
Bug information used in this study.

Compiler	Start	End	Total Bugs	Opt. Bugs	Opt. Revisions
GCC	1999–08	2019–12	57,591	8771	3486
LLVM	2003–10	2019–12	22,748	1564	1224

LLVM, respectively, which indicates that the selected optimization bugs are representative. In addition, we manually classify the 10,335 optimization bugs into *Crash*, *Misoptimization*, and *Performance*.

3. What is the distribution of optimization bugs?

This section shows the distribution of optimization bugs. We describe the general statics of optimization bugs in Section 3.1. The distribution of optimization bugs in components is presented in Section 3.2.

3.1. General statistics

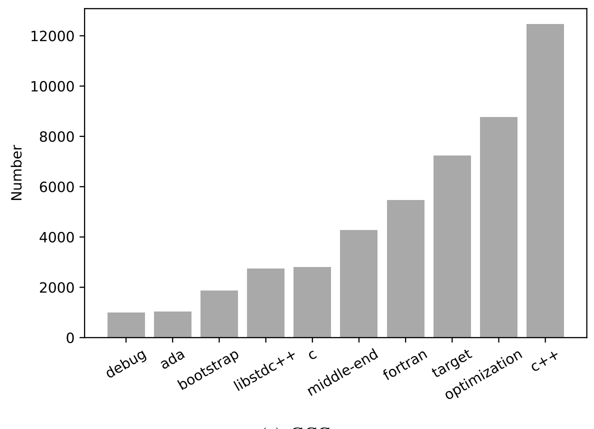
Fig. 1 shows the overall evolution of the optimization bugs for GCC and LLVM. In particular, it shows the number of all bugs and optimization bugs in each month. From Figs. 1(a) and 1(b), we can see that the trend of the optimization bugs is almost consistent with that of all bugs. In the early stage of GCC development (1999 ∼ 2006), GCC gained much attention, which led to increase in the number of bugs. However, the trends of GCC are relatively stable in recent years compared to LLVM. For LLVM, it has become a mature and widely used compiler infrastructure in these years, and has drawn much attention in industry and academia. This results in the rapid growth of the number of LLVM bugs.

To illustrate the importance of optimization bugs, we show the distribution of bugs in compiler components in Fig. 2. There totally have 52 and 96 components in GCC and LLVM, respectively. Similar to the study in Sun et al. (2016b), we also show the top ten buggy components for each compiler. The numbers of bugs in these top ten components account for 82.72% and 69.96% bugs for GCC and LLVM, respectively.

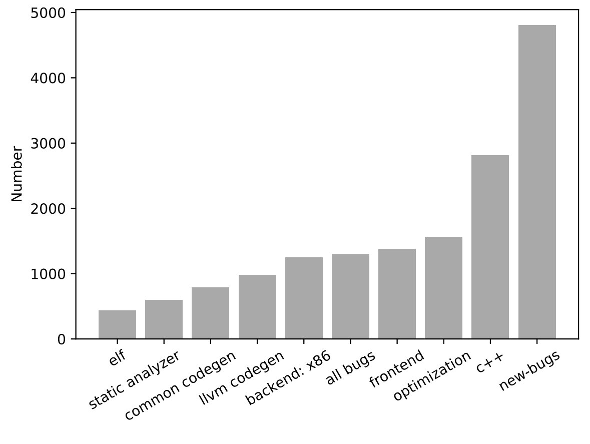
From Figs. 2(a) and 2(b), the bugs of optimization components in both GCC and LLVM account for a high percentage. For GCC, we can see that the buggiest component is C++, containing 21.63% of the 57,591 bugs. Meanwhile, there are four language-dependent components (i.e., C++, Fortran, C, ada) in these ten buggy components. They all belong to the front end of GCC. Apart from the language-dependent components, we can see that the optimization components (including "*tree-optimization*" and "*rtl-optimization*") in GCC are the buggiest components, accounting for around 15.23% of the bugs. For LLVM, the buggiest component is "*new-bugs*", since the developers can submit bug reports without specifying their components (Sun et al., 2016b). Similar to GCC, C++ component in LLVM also has a high percentage of bugs. Besides the C++ and *new-bugs* components, the optimization components (including "*Scalar Optimizations*", "*Loop Optimizer*", "*Transformation Utilities*", and "*Interprocedural Optimizations*") in LLVM are the buggiest components, containing 6.88% of the 22,748 bugs. The percentage of bugs in optimization components of LLVM is lower than that of GCC. The reason for this may be that some optimization bugs are submitted in the *new-bugs* component for LLVM.

3.2. Bugs in optimization components

Tables 2 and 3 show the bugs in each optimization component in GCC and LLVM, respectively. In particular, we present the numbers of bugs with two status (i.e., Fixed and Confirmed) for each optimization component. From Table 2, we can see that most optimization bugs for GCC are contained in the "*tree-optimization*" component, accounting for 65.15% of the 8771 bugs. The reason may be that many optimizations are implemented in the "*tree-optimization*" component. In GCC, there are 85 files that are directly related to the "*tree-optimization*" component. The



(a) GCC



(b) LLVM

Fig. 2. The top 10 buggy components in GCC and LLVM.

Table 2
Bugs in each GCC optimization component.

Component/Status	Fixed	Confirmed	Total
tree-optimization	4868 (85.19%)	846 (14.81%)	5714 (65.15%)
rtl-optimization	2759 (90.25%)	298 (9.75%)	3057 (34.85%)
Total	7627 (86.96%)	1144 (13.04%)	8771

Table 3
Bugs in each LLVM optimization component.

Component/Status	Fixed	Confirmed	Total
Scalar optimizations	930 (96.17%)	37 (3.83%)	967 (61.83%)
Loop optimizer	315 (96.63%)	11 (3.37%)	326 (20.84%)
Interprocedural optimizations	145 (98.64%)	2 (1.36%)	147 (9.40%)
Transformation utilities	122 (98.39%)	2 (1.61%)	124 (7.93%)
Total	1512 (96.68%)	52 (3.32%)	1564

filenames of these files all have the same prefix “*tree-*”. However, the percentage of fixed bugs for the “*rtl-optimization*” component is 90.25%, which is larger than 85.19% for the “*tree-optimization*” component. Besides, we can see that there still are 14.81% and 9.75% bugs that are confirmed but not fixed in “*tree-optimization*” and “*rtl-optimization*” components, respectively.

For LLVM, we can see that the buggiest optimization component is “*Scalar Optimizations*” from Table 3, containing 61.83% of the 1564 bugs. However, the percentages of fixed bugs for the *Interprocedural Optimizations* and “*Transformation Utilities*” components are higher than other components. This may be that the optimizations implemented in these two components are much simpler than those of other components. The major optimizations of LLVM are implemented in the “*Scalar Optimizations*” components, there are 77 files in the corresponding directory in LLVM. The overall rate of fixed optimization bugs in LLVM is 96.68%, which is higher than GCC’s 86.96%. The reason may be due to the fine-grained modular design of LLVM.

4. Which optimizations are buggy?

This section presents information about buggy optimizations in GCC and LLVM. Instead of listing the concrete optimizations, we show the files that implement the optimizations because it is hard to understand all optimizations of GCC and LLVM. To obtain the files of buggy optimizations, we parse the commit logs to find the commits for the revisions of the optimization bugs. We utilize the pattern listed in Section 2.2 to check whether a revision is for an optimization bug. Then we extract the changed files, and manually check whether a file is used to implement an optimization. In this way, we totally extract 1589 and 348 files for GCC and LLVM, of which 119 and 101 files are directly used to implement optimizations in GCC and LLVM, respectively. From these files, we can indirectly learn about the buggy optimizations, since the names of these files, to some extent, can reflect the function of the optimizations implemented in these files. Tables 4 and 5 show the top 30 buggy files related to the optimizations of GCC and LLVM. We give each file a description according to the documents of GCC and LLVM. The frequency of each file is the number of revisions that modify this file to fix bugs.

From Table 4, we can see that the buggiest files used to implement optimizations of GCC belong to the “*tree-optimization*” component. The file “tree-vrp.c” is the buggiest file, which has been modified by at least 143 revisions related to optimization bugs. The value range propagation optimization is implemented in the file “tree-vrp.c”. In addition, most optimizations in these 30 files focus on optimizing a single function, rather than the interprocedural optimization for the whole program. There is only the file “ipa-cp.c” that is used to implement the interprocedural constant propagation optimization in these 30 files. This may indicate that the optimizations in GCC for optimizing a single function need to be further tested. Specifically, there are five files (i.e., “tree-loopdistribution.c”, “tree-parloops.c”, “tree-ssa-loop.c”, “tree-ssa-loop-ivcanon.c”, and “tree-ssa-loop-im.c”) that focus on the loop optimizations, which may suggest that the developers should pay more attention on the loop optimizations in GCC.

For LLVM, we can see that the buggiest file is “InstructionCombining.cpp” that implements the instruction combine optimization from Table 5. Note that the frequency of the file “InstructionCombining.cpp” is higher than that of other files in Table 5. This is because there are many files that are used to implement the instruction combine optimization and the name of these files all have the same prefix “InstCombine”, so we add all the frequency of these files to the frequency of “InstructionCombining.cpp”. Similar to GCC, the files used to implement loop optimizations have more bugs than other files from Table 5. There are nine files (i.e., “LoopVectorize.cpp”, “LICM.cpp”, “LoopStrengthReduce.cpp”, “LoopIdiomRecognize.cpp”, “LoopSimplify.cpp”,

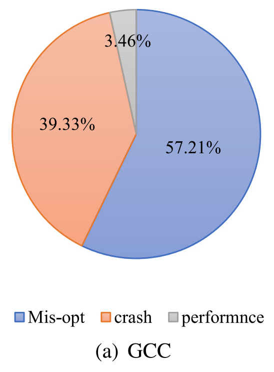
Table 4
Top 30 buggy files related to the optimizations of GCC.

File	Description	Freq.	File	Description	Freq.
tree-vrp.c	Value range propagation	143	tree-ssa-phiopt.c	PHI node optimizations	36
tree-ssa-sccvn.c	Value numbering	92	tree-ssa-dom.c	Dominator optimizations	36
combine.c	Instruction combination	83	tree.c	Tree nodes operations	35
tree-ssa-pre.c	Partial redundancy elimination	81	tree-ssa-strlen.c	Optimization for string length	30
tree-cfg.c	Split critical edges	69	tree-ssa-loop.c	Loop optimization	30
tree-sra.c	Scalar replacement of aggregates	61	tree-ssa-math-opts.c	Math optimizations	29
tree-ssa-structalias.c	May-alias optimization	50	tree-ssa-loop-ivcanon.c	Canonical induction variable creation	29
tree-ssa-reassoc.c	Reassociation	50	tree-ssa-loop-im.c	Loop invariant motion	28
tree-ssa-ccp.c	Conditional constant propagation	50	ipa-cp.c	Interprocedural constant propagation	28
passes.c	Pass manager	50	ifcvt.c	If conversion	25
tree-loop-distribution.c	Loop distribution	42	tree-ssa-dce.c	Dead code elimination	24
tree-ssa-forwprop.c	Forward propagation of single-use variables	41	tree-ssa-copy.c	Conditional copy propagation	24
tree-if-conv.c	If-conversion for vectorizer	40	tree-eh.c	Lower exception handling control flow	24
tree-parloops.c	Autoparallelization	39	cfgexpand.c	Translate GIMPLE trees to RTL	24
tree-ssa.c	Enter static single assignment form	36	tree-ssa-ifcombine.c	Combine conditional expression to simplify control flow	23

Table 5
Top 30 buggy files related to the optimizations of LLVM.

File	Description	Freq.	File	Description	Freq.
InstructionCombining.cpp	Combine instructions	238	InlineFunction.cpp	Inline of function	12
LoopVectorize.cpp	Loop vectorize	58	SimpleLoopUnswitch.cpp	Simple loop unswitch	11
SimplifyCFG.cpp	Peephole optimize the CFG	32	Reassociate.cpp	Reassociate commutative expressions	11
SLPVectorizer.cpp	superword-level parallelism vectorization	32	LoopSimplify.cpp	Loop simplify	11
SROA.cpp	Scalar replacement of aggregates	25	Local.cpp	Local transformations	10
LICM.cpp	Loop invariant code motion	24	LoopUnroll.cpp	Loop unroll	9
IndVarSimplify.cpp	Induction variable simplify	21	LCSSA.cpp	Loop-closed SSA	9
GlobalOpt.cpp	Global variable optimization	18	JumpThreading.cpp	Jump threading	9
SCCP.cpp	Sparse conditional constant propagation	17	GVNHoist.cpp	Global value numbering based expression hoist	9
GVN.cpp	Global value numbering	17	LoopRotation.cpp	Loop rotation	8
SimplifyLibCalls.cpp	Library calls simplifier	16	Inliner.cpp	Inline	8
NewGVN.cpp	New global value numbering	15	MemCpyOptimizer.cpp	Memcpy calls optimization	7
DeadStoreElimination.cpp	Dead store elimination	14	LoopUnswitch.cpp	Loop unswitch	7
LoopStrengthReduce.cpp	Loop strength reduction	13	ConstantHoisting.cpp	Constants to hoist	7
LoopIdiomRecognize.cpp	Loop idiom recognizer	13	PromoteMemoryToRegister.cpp	Promote memory to register	6

"LoopUnroll.cpp", "LCSSA.cpp", "LoopRotation.cpp", and "LoopUnswitch.cpp") related to loop optimizations, accounting for 30% of the 30 files. Thus, more test may be further conducted for loop optimizations in LLVM. In addition, the files (i.e., "GVN.cpp", "NewGVN.cpp", and "GVNHost.cpp") related to the global value numbering optimization are all buggy. This may indicate that



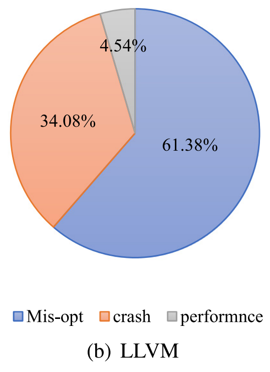


Fig. 3. The percentage of optimization bugs for each bug type in GCC and LLVM.

Table 6
Bugs of each type in GCC's optimization components.

Component	Mis-opt	Crash	Performance	Total
tree-optimization	3294 (57.65%)	2227 (38.97%)	193 (3.38%)	5714 (65.15%)
rtl-optimization	1724 (56.40%)	1223 (40.01%)	110 (3.60%)	3057 (34.85%)
Total	5018 (57.21%)	3450 (39.33%)	303 (3.45%)	8771

Table 7
Bugs of each type in LLVM's optimization components.

Component	Mis-opt	Crash	Performance	Total
Scalar Optimizations	599 (61.94%)	317 (32.78%)	51 (5.27%)	967 (61.83%)
Loop Optimizer	182 (55.83%)	129 (39.57%)	15 (4.60%)	326 (33.71%)
Interprocedural Optimizations	99 (67.35%)	45 (30.61%)	3 (2.04%)	147 (9.40%)
Transformation Utilities	80 (64.52%)	42 (33.87%)	2 (1.61%)	124 (7.93%)
Total	960 (61.38%)	533 (34.08%)	71 (4.54%)	1564

there are some design flaws for the global value numbering optimization in LLVM.

5. What are the types of optimization bugs?

5.1. General statistics

Bug types are an important aspect to understand the quality of a compiler. Through the analysis of the bug types for optimization bugs, we can further know the flaws of the optimizations in GCC and LLVM. In this study, we manually assign a type to each optimization bug according to the compiler testing literatures (e.g., Yang et al., 2011; Le et al., 2014; Vu et al., 2015a; Lidbury et al., 2015; Sun et al., 2016a; Zhang et al., 2017). Generally, the types of optimization bugs can be divided into three categories, i.e., *Crash*, *Misoptimization* (*Mis-opt* for short), and *Performance*. The details of these three bug types can be found in Section 2.2.

Fig. 3 shows the percentage of each type of optimization bugs in GCC and LLVM. From Figs. 3(a) and 3(b), it is clear that most optimization bugs are *Mis-opt* bugs in both GCC and LLVM, accounting for 56.15% and 68.03% of the total optimization bugs for GCC and LLVM, respectively. The percentage of *Mis-opt* bugs in LLVM is larger than that of GCC. This may be caused by the insufficient test of LLVM. However, the percentage of *Crash* bugs in GCC is larger than that of LLVM. The reason for this may be that the code of GCC is implemented by C programming language, while C++ for LLVM. Although the developers of GCC, to some extent, have refactored the code of GCC using C++ programming language, there is still much work to be done. In both GCC and LLVM, the percentage of *Performance* bugs is very small compared to *Mis-opt* bugs and *Crash* bugs. One possible explanation is that testers and users may do not pay much attention to *Performance* bugs, since *Performance* bugs are hard to reproduce in a different environment. In most cases, *Performance* bugs may be caused by the setting of the user's system, such that the developers can hardly reproduce them in their system. In addition, we cannot rule out that *Performance* bugs may rarely occur in essence compared to other types of bugs.

Table 8
The status of optimization bugs for each bug type in GCC.

Type\Status	Fixed	Confirmed	Total
Mis-opt	3965 (79.02%)	1053 (20.98%)	5018 (57.21%)
Crash	3390 (98.26%)	60 (1.74%)	3450 (39.33%)
Performance	272 (89.77%)	31 (10.23%)	303 (3.45%)
Total	7627 (86.96%)	1144 (13.04%)	8771

Table 9
The status of optimization bugs for each bug type in LLVM.

Type\Status	Fixed	Confirmed	Total
Mis-opt	919 (95.73%)	41 (4.27%)	960 (61.38%)
Crash	523 (98.12%)	10 (1.88%)	533 (34.08%)
Performance	70 (98.59%)	1 (1.41%)	71 (4.54%)
Total	1512 (96.68%)	52 (3.32%)	1564

5.2. Bugs of each type in optimization components

Tables 6 and 7 show the information about optimization bugs of each type in the optimization components of GCC and LLVM, respectively. From Table 6, we can see that the percentages of *Mis-opt* bugs, *Crash* bugs, and *Performance* bugs in both *tree-optimization* component and *rtl-optimization* component are similar. The numbers of bugs of these three types account for about 56%, 40%, and 3% of the 5714 and 3057 bugs in the two optimization components, respectively. Similar to the number of bugs in the *tree-optimization* component and *rtl-optimization* component, the number of bugs of each type in *tree-optimization* component is almost twice as large as that of the *rtl-optimization* component.

Similar cases also occur in LLVM. However, the percentage of *Crash* bugs in the *Loop Optimizer* component is larger than those of other components. There are 129 *Crash* bugs in the *Loop Optimizer* component, accounting for 39.57% of the 326 bugs. In addition, the percentage of *Performance* bugs in the *Scalar Optimizations* is also larger than those of other components, accounting for 5.27%. This may suggest that the developers of LLVM should conduct more tests for different types of bugs.

5.3. Status of optimization bugs for each bug type

To investigate the status of optimization bugs for each bug type, we statistic the number of optimization bugs according to their types and status. Tables 8 and 9 show the information about

Table 10
The average months that optimization bugs live for each optimization component and each bug type in GCC.

	Fixed	Confirmed
tree-optimization	9.79	68.99
rtl-optimization	13.57	82.02
Mis-opt	15.07	73.87
Crash	6.43	40.45
Performance	13.06	83.58

Table 11
The average months that optimization bugs live for each optimization component and each bug type in LLVM.

	Fixed	Confirmed
Scalar optimizations	13.97	9.49
Loop optimizer	10.70	25.0
Interprocedural optimizations	17.77	66.0
Transformation utilities	12.62	1.5
Mis-opt	14.11	13.41
Crash	12.26	7.7
Performance	15.73	134.0

the status of each bug type for optimization bugs in GCC and LLVM. In general, the bug-fixing rate of LLVM is larger than that of GCC. For LLVM, 96.68% optimization bugs are fixed, while 86.96% for GCC. Besides, from Tables 6, 7, 8, 9, we can see that the majority of optimization bugs in both GCC and LLVM are *Crash* and *Mis-opt* bugs. However, the bug-fixing rate of *Crash* bugs is larger than those of *Mis-opt* bugs for both GCC and LLVM. The bug-fixing rates of *Crash* bugs are 98.26% and 98.12% for GCC and LLVM, respectively. While the bug-fixing rates are only 79.02% and 95.73% for *Mis-opt* bugs of GCC and LLVM, respectively. This may be because of the difficulty for analyzing and locating the root causes of *Mis-opt* bugs. For *Crash* bugs, developers can leverage the backtrace information to analyze the root causes of bugs, while only limited information could be used to help developers logically understand the root cause of a *Mis-opt* bug.

In addition, for *Performance* bugs, LLVM has a higher bug-fixing rate than GCC. There are 10.23% *Performance* bugs that are confirmed but not fixed for GCC. However, only one confirmed *Performance* bug for LLVM is not fixed, accounting for 1.41% of 71 LLVM *Performance* bugs. Besides, LLVM also has a better bug-fixing rate of *Mis-opt* bugs compared to GCC. There are 95.73% *Mis-opt* bugs that are fixed, but only 79.02% for GCC. One possible reason is due to the better modular design of LLVM, such that developers can quickly find the root causes of *Performance* bugs and *Mis-opt* bugs.

6. How long do optimization bugs live?

This section investigates the time that optimization bugs live for GCC and LLVM. Specifically, we focus on the time to fix bugs and the time that confirmed bugs live until Dec. 31, 2019. Fig. 4 shows the months that optimization bugs live for GCC and LLVM. From Figs. 4(a) and 4(b), we can see that most fixed bugs can be processed in about 24 months in both GCC and LLVM. In general, developers of GCC need 11.16 months for fixing an optimization bug on average. While the confirmed optimization bugs of GCC live for 72.38 months on average. For the 7627 fixed bugs, 6633 and 3419 of them can be fixed in 24 months and one month, account for 86.97% and 44.83%, respectively. In LLVM, the distributions of the time for the two types of bugs are similar to GCC. The average months for the fixed bug and the confirmed bug are 13.55 and 14.63, respectively. There are 1218 and 777 optimization bugs of LLVM that can be fixed in 24 months and one month, accounting for 80.56% and 51.39%, respectively. Besides, it is clear from Fig. 4 that the time for confirmed bugs of LLVM is shorter than GCC. The time that the confirmed bugs live is 14.63 months on average for LLVM, while it is 72.38 months for GCC. This is because there are some confirmed bugs of GCC that have lived for a long time. For example, GCC bug 5738[15] is confirmed at Apr. 2002, but it is still not fixed until Dec. 2019.

Tables 10 and 11 present the average months that optimization bugs live for each optimization component and each bug type in GCC and LLVM. From Table 10, we can see that the average months for the bugs in the *rtl-optimization* component are longer than those of the *tree-optimization* component. This is because that optimizations implemented in the *rtl-optimization* component are more complex than those in the *tree-optimization* component. In addition, *Mis-opt* bugs take longer time than other types of bugs. For example, developers of GCC need an average of 15.07 months for fixing a *Mis-opt* bug, while it is 6.43 months for a *Crash* bug on average. The reason for this may be that it is difficult to analyze and locate the root causes of *Mis-opt* bugs. To alleviate this difficult, Chen et al. (2019) present a method based on effective witness test program generation to automatically isolate the root causes of compiler bugs.

For LLVM, we can see that the most time-consuming bugs belong to the *Interprocedural Optimizations* component from Table 11. The average months for fixed bugs and confirmed bugs in the *Interprocedural Optimizations* component are 17.77 and 66.0, respectively. This is because, on the one hand, the interprocedural optimizations are hard to understand by developers, since they are used to optimize the whole program that may include many functions. On the other hand, it is difficult to analyze the root causes of the bugs in *Interprocedural Optimizations* component. For a target program with many functions, if it triggers a bug for an interprocedural optimization, any function of it may contain the code to trigger the bug. Besides, instead of *Mis-opt* bugs, the average months for *Performance* bugs in LLVM are longer than those of other types of bugs. For instance, it takes average of 15.73 months to fix a *Performance* bug. Especially, the LLVM performance bug 3082[16] is confirmed at Nov. 2008, but it has lived about 134.0 months until Dec. 2019. This may be due to the difficulty to locate the real reason for *Performance* bug. For example, the developers take a long time to discuss the root cause of LLVM bug 10584[17] and propose patches for this bug.

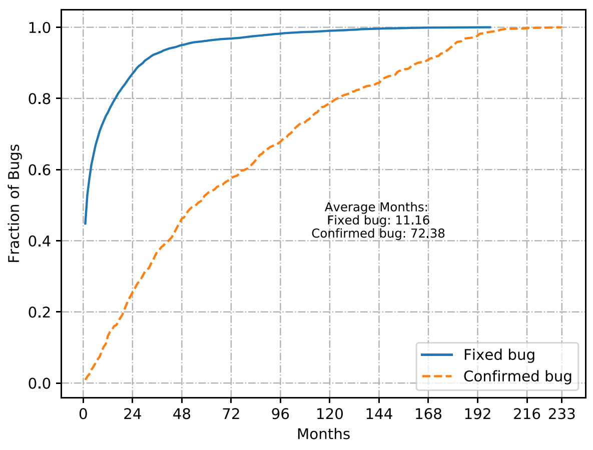
7. How many files, functions, and code lines are modified to fix an optimization bug?

This section investigates the number of files, functions, and code lines for fixing optimization bugs in GCC and LLV-M. With a better understanding of bug fixes of optimization bugs, developers of compilers can make better design decisions of optimizations. In addition, our study also provides useful guidance for automatic debugging techniques (e.g., Chen et al., 2019) for compiler bugs. We parse patches in the commit log, and extract the information about the modified files, functions, and code lines. Note that we exclude the files that are used to test GCC and LLVM, since these files do not impact the functionality of GCC and LLVM but they contain many code changes. Table 12 shows the average number of files, functions, and code lines for fixing optimization bugs in GCC and LLVM. In general, the developers of GCC need to modify more files and functions to fix optimization bugs than LLVM. However, to fix optimization bugs in LLVM, the developers of LLVM need to add or delete more code than GCC.

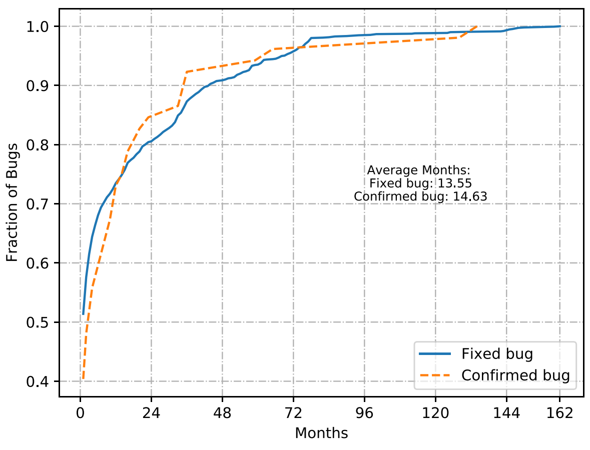
15 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=5738.

16 https://bugs.llvm.org/show_bug.cgi?id=3082.

17 https://bugs.llvm.org/show_bug.cgi?id=10584.



(a) GCC



(b) LLVM

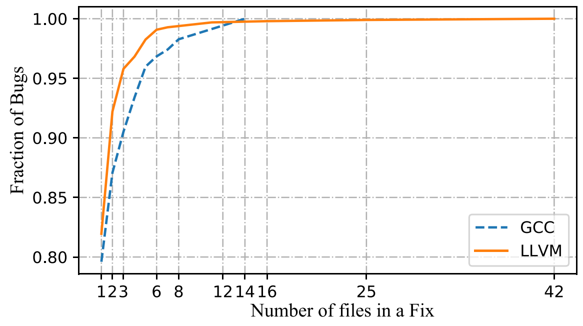
Fig. 4. The months that optimization bugs live for GCC and LLVM.

Table 12
The average number of files, functions, and code lines for fixing optimization bugs in GCC and LLVM.

	Files	Functions	Code lines	
			Add	Delete
GCC	1.66	3.07	25.11	10.10
LLVM	1.43	2.45	40.15	18.25

Table 13
The statistics of the number of files in bug fixes for each optimization component and each bug type in GCC and LLVM.

	Mean	Median	SD	Min	Max
GCC:tree-optimization	1.81	1.00	1.92	1	14
GCC:rtl-optimization	1.41	1.00	1.63	1	14
GCC:Mis-opt	1.70	1.00	1.73	1	12
GCC:Crash	1.53	1.00	1.70	1	14
GCC:Performance	2.82	2.00	3.64	1	14
LLVM:Scalar optimizations	1.41	1.00	1.95	1	42
LLVM:Loop optimizer	1.34	1.00	1.06	1	11
LLVM:Interprocedural optimizations	1.75	1.00	2.81	1	25
LLVM:Transformation utilities	1.52	1.00	1.56	1	11
LLVM:Mis-opt	1.44	1.00	1.59	1	25
LLVM:Crash	1.43	1.00	2.37	1	42
LLVM:Performance	1.32	1.00	0.83	1	6



(a) The empirical cumulative distribution of the number of files modified in a bug fix.

	Mean	Median	SD	Min	Max
GCC	1.66	1.00	1.82	1	14
LLVM	1.43	1.00	1.88	1	42

(b) The statistics of the number of files in bug fixes.

Fig. 5. The number of files modified in bug fixes for GCC and LLVM.

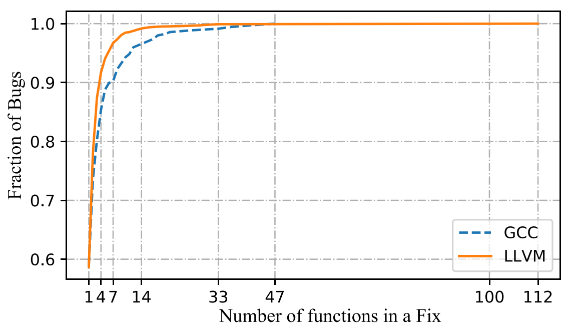
7.1. Number of files

Fig. 5 shows the relation between the number of files modified in a fix and the fraction of bugs. From Fig. 5(a), we can see that around 90% of the optimization bugs involve at most 3 files in GCC and LLVM. Specially, more than 79% of the GCC optimization bugs and 81% of the LLVM optimization bugs only involve one file. Moreover, most optimization bugs in both GCC and LLVM can be fixed by modifying no more than 14 files. Fig. 5(b) shows the summary statistics (including mean, median and standard deviation (SD)). On average, the developers need to modify 1.66 and 1.43 files to fix an optimization bug in GCC and LLVM, respectively. This may indicate that both GCC and LLVM all have a good modular design of optimizations, such that most of the optimization bugs only touch a small portion of the compiler code.

Table 13 presents statistics of the number of files in bug fixes of each optimization component and each bug type in GCC and LLVM. For GCC, the optimization bugs in the *tree-optimization* component involve more files than the *rtl-optimization* component. The mean of the *tree-optimization* component is 1.81, while it is 1.41 for the *rtl-optimization* component. For each type optimization bugs in GCC, *Performance* bugs involve more files. The mean, median, and SD of *Performance* bugs are 2.82, 2.00, and 3.64, which are much larger than those of *Mis-opt* bugs and *Crash* bugs. For LLVM, the optimization bugs in the *Interprocedural Optimizations* touch more files than other optimization components. In addition, the developers need to modify more files to fix *Mis-opt* bugs compared to *Crash* bugs and *Performance* bugs, 1.44 files on average.

7.2. Number of functions

Fig. 6(a) shows the empirical cumulative distribution of the number of functions modified in a bug fix. From Fig. 6(a), around 96% of the GCC optimization bugs and 99% of LLVM involve at most 14 functions. In particular, more than 58% of the optimization bugs in GCC and LLVM only involve one function. Moreover, the bug fixes with 4 functions cover more than 85% and 90% of the optimization bugs in GCC and LLVM, respectively. Fig. 6(b) presents the statistics of the number of functions in bug fixes. Generally, GCC needs to modify more functions than LLVM, 3.07 functions on average.

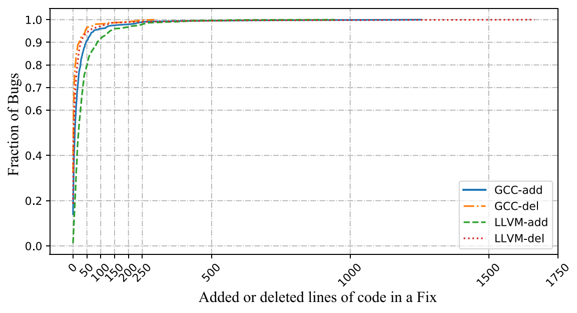


(a) The empirical cumulative distribution of the number of functions modified in a bug fix.

	Mean	Median	SD	Min	Max
GCC	3.07	1.00	5.27	1	47
LLVM	2.24	1.00	4.39	1	112

(b) The statistics of the number of functions in bug fixes.

Fig. 6. The number of functions modified in bug fixes for GCC and LLVM.



(a) The empirical cumulative distribution of the number of code lines modified in a bug fix.

	Mean	Median	SD	Min	Max
GCC-add	25.11	7.00	84.14	0	1256
GCC-del	10.10	1.00	30.70	0	293
LLVM-add	40.15	20.00	68.68	0	945
LLVM-del	18.25	3.00	85.12	0	1665

(b) The statistics of the number of code lines in bug fixes.

Fig. 7. The number of code lines modified in bug fixes for GCC and LLVM.

Table 14
The statistics of the number of functions in bug fixes for each optimization component and each bug type in GCC and LLVM.

	Mean	Median	SD	Min	Max
GCC:tree-optimization	3.84	2.00	6.23	1	47
GCC:rtl-optimization	1.85	1.00	2.76	1	26
GCC:Mis-opt	2.92	1.00	4.78	1	41
GCC:Crash	2.95	1.00	5.20	1	47
GCC:Performance	7.18	2.0	9.87	1	36
LLVM:Scalar optimizations	2.20	1.00	4.99	1	112
LLVM:Loop optimizer	2.20	1.00	2.34	1	24
LLVM:Interprocedural optimizations	2.32	1.00	3.75	1	33
LLVM:Transformation utilities	2.67	1.00	3.80	1	24
LLVM:Mis-opt	2.18	1.00	2.84	1	33
LLVM:Crash	2.39	1.00	6.37	1	112
LLVM:Performance	2.04	1.00	1.76	1	9

Table 14 displays the statistics of the number of functions in bug fixes of each optimization component and each bug type in GCC and LLVM. For GCC, the optimization bugs in *tree-optimization* component and *Performance* bugs involve more functions. Specifically, the mean of the number of functions for fixing a *Performance* bug is 7.18, while it is only 2.92 for *Mis-opt* bugs. However, in contrast to GCC, the *Crash* bugs touch more functions in LLVM, 2.39 functions on average.

7.3. Lines of code

In addition to the files and the functions, we also investigate the lines of source code in bug fixes. In our study, we treat code and comments in a uniform way, that is, the information about code and comments are all considered when we investigate the lines of code in bug fixes. This is because that comments play an important role in software developments. Comments can not only improve the readability and maintainability of source code, but also provide a significant resource for software reuse (Lu et al., 2019). In addition, GCC and LLVM are two mature and widely used compilers. All source code of GCC and LLVM are implemented according to strict code standards,[18] which guarantees the quality of code and comments. Thus, under the premise that both the source code and the comments are of high quality, we believe it is reasonable to consider all information about the code and the comments for investigating the lines of code in bug fixes. Fig. 7 shows the relation between the number of code lines modified in a fix and the fraction of bugs. As shown in Fig. 7(a), most of the bug fixes add or delete fewer than 250 lines of code in GCC and LLVM. Around 99% of the bug fixes only modify no more than 100 lines of code, while 90% less than 50 lines of code. On average, as Fig. 7(b) shows, the developer of LLVM need to add or delete more lines of code to fix an optimization bug. The means of the added and deleted lines of code in LLVM are 40.15 and 18.25 respectively, while they are 25.11 and 10.10 for GCC. In addition, the median for GCC is smaller than LLVM. Especially, the median of the added lines of code in GCC is only 7, while it is 20 for LLVM. The reason may be that LLVM has drawn much attention from both industry and academia, such that many developers devote to enhance LLVM.

Table 15 shows the statistics of the number of code lines added/deleted in bug fixes of each optimization component and each bug type in GCC and LLVM. For GCC, the trend of the added/deleted lines of code in the bug fixes is the same as the function modification. For example, the bug fixes in the *tree-optimization* component involve more functions than the *rtl-optimization* component (see Table 14). Similarly, the bug fixes in the *tree-optimization* component also need to add/delete more lines of code. On average, the developers add/delete 33.15/14.16 lines of code to fix an optimization bug in the *tree-optimization*, and the median is 9.00/2.00. However, in LLVM, although the bug fixes in the *Interprocedural Optimizations* component involve more functions than other optimization components, they contain fewer modification of lines of code. The mean is 33.42/9.57 to add/delete lines of code in the *Interprocedural Optimizations* component, while it is 45.31/30.55 for the *Transformation Utilities* component.

8. Threats to validity

Threats to Internal Validity. The threats to internal validity mainly lie in the correctness of the examination methodology. Firstly, we collect the optimization bugs from the bug repositories of GCC and LLVM according to the predefined optimization components they belong to. However, not all optimization bugs are

18 https://www.gnu.org/prep/standards/standards.html, https://llvm.org-/docs/CodingStandards.html.

Table 15
The statistics of the lines of code added/deleted in bug fixes for each optimization component and each bug type in GCC and LLVM.

	Mean	Median	SD	Min	Max
GCC:tree-optimization	33.15/14.16	9.00/2.00	105.12/37.61	0/0	1256/293
GCC:rtl-optimization	12.28/3.62	4.00/1.00	21.61/11.01	0/0	138/77
GCC:Mis-opt	26.12/8.87	7.00/1.00	101.83/26.79	0/0	1256/293
GCC:Crash	19.61/9.62	7.00/1.00	41.29/30.20	0/0	387/231
GCC:Performance	92.00/36.55	28.00/12.00	176.91/64.75	0/1	643/236
LLVM:Scalar optimizations	40.85/17.10	18.00/3.00	76.49/66.13	0/0	945/1231
LLVM:Loop optimizer	38.81/20.77	23.50/3.00	46.84/126.06	1/0	271/1665
LLVM:Interprocedural optimizations	33.42/9.57	22.00/4.00	41.05/13.97	1/0	236/69
LLVM:Transformation utilities	45.31/30.55	22.00/5.00	72.56/125.39	0/0	404/69
LLVM:Mis-opt	45.20/16.34	23.00/4.00	77.66/78.78	0/0	945/1665
LLVM:Crash	30.07/15.55	16.00/2.00	47.81/66.15	0/0	444/1046
LLVM:Performance	50.91/61.62	17.00/10.00	70.22/197.81	0/0	271/1231

in the predefined optimization components. Thus, our data may not contain all the optimization bugs of the selected compilers. For example, the developers of LLVM can submit bug reports without specifying their components. Some optimization bugs of LLVM may belong to the *new-bugs* component, which is not an optimization component. Secondly, we manually assign a type to each optimization bug according to its comments. However, it is difficult to obtain the type of a bug only according to the comments for some special cases. To reduce this threat, we ask three authors of this study to evaluate the comments until we reach a consensus. Thirdly, we parse the commit logs to collect the information of the modification of files, functions, and lines of code in bug fixes. However, the results may not be precise. For example, we may not extract a few functions due to the complex syntax of C/C++. To alleviate this threat, we manually check the results, and try our best to make our program automatically extract precise results. We believe that the results of this study are credible.

Threats to External Validity. The threats to external validity mainly lie in the representativeness of the chosen compilers. We select GCC and LLVM in this study. Both GCC and LLVM are mature and widely used compilers written in C/C++. They have been used to implement many programming languages (e.g., C++, Ada, Fortran) for various architectures, and many effective optimization algorithms have been implemented. We believe that GCC and LLVM can well represent most of the traditional compilers, and the optimizations implemented in them can also represent most of the optimizations used by other compilers. However, the selected compilers in this study may not well reflect the compilers based on the virtual machine (e.g., Just-In-Time compilers for Java) or interpreters (e.g., Python). While both Just-In-Time compilers and interpreters also contain many optimizations, we thus believe that the results of this study may also be useful for the design and the implementation of Just-In-Time compilers and interpreters.

9. Related work

The most relevant work are the studies (Sun et al., 2016b; Marcozzi et al., 2019). Sun et al. (2016b) present the first empirical study on compiler bugs. They examine about 50K bugs and 30K bug fix revisions of GCC and LLVM, and show four characteristics of compiler bugs, such as the distribution of bugs in source files. Similar to the study (Sun et al., 2016b), we also study the compiler bugs of GCC and LLVM. However, we focus on the optimization bugs, since optimizations are important for modern compilers. Our study complements the work conducted by Sun et al. (2016b), such that the developers and researchers can further understand compiler bugs. The second most relevant work is the study conducted by Marcozzi et al. (2019). In Marcozzi et al. (2019), the authors study the practical impact of fuzzer-found compiler bugs on real-world applications. Although the authors show that only a very small part of miscompilation bugs in a mature compiler can influence the real-world application, compiler fuzz testing is also an important technique to improve the quality of compilers. Our study provides an insight into compiler optimization bugs, which may help the researchers to design more efficient fuzz testing methods for compilers.

Besides the work (Sun et al., 2016b; Marcozzi et al., 2019), there are still many empirical studies on bugs. Chou et al. (2001) present a study of operating system errors. They collect approximately one thousand errors by applying automatic, static, compiler analysis to the Linux and OpenBSD kernels, and show that device drivers have much more bugs than other components of the kernels. Li et al. (2006) study bug characteristics in modern open-source software. They analyze around 29, 000 bugs of Mozilla and Apache using natural language text classification techniques. They find that semantic bugs are the dominant root causes and security bugs have an increasing trend the open-source software. Lu et al. (2008) investigate the real world concurrency bug characteristics. They randomly collect 105 real-world concurrency bugs from MySQL, Apache, Mozilla, and OpenOffice, and examine concurrency bug patterns, manifestation, and fix strategies of these bugs. Their findings are useful for concurrency bug detection, testing, and concurrent programming language design. Sahoo et al. (2010) conduct an empirical study on bugs that affect reproducibility at the production site. They analyze 266 reported bugs of six server applications, such as MySQL, Apache, and OpenSSH, and provide several implications on automatic bug diagnosis tools based on their findings. Song and Lu (2014) present an empirical study to understand performance bugs in open source projects, and show that the statistical debugging technique can well diagnose performance problems.

For better understanding bugs in machine learning systems, Thung et al. (2012) analyze bugs of Apache Mahout, Lucene, and OpenNLP. They divide the bugs into three categories based on their characteristics, and provide the relationship between bug categories and bug characteristics. Zhang et al. (2018) conduct an empirical study on the characteristics of deep learning defects. They collect 175 TensorFlow coding bugs from GitHub issues and StackOverflow questions, and analyze the symptoms and root causes of these bugs. Islam et al. (2019) analyze characteristics of bugs in deep learning software. They collect 2716 high-quality posts from Stack Overflow and 500 bug fix commits of five open-source deep learning libraries (e.g., Tensorflow, Theano, and Torch), and analyze many characteristics of these bugs, such as the types of bugs, root causes of bugs, and the impacts of bugs.

Unlike other empirical studies, our work focus on optimization bugs in compilers, which may be a complement for other studies to better understand the corresponding bugs. For example, there are also many optimizations in some deep learning frameworks, such as TensorFlow (Abadi et al., 2015) and PyTorch (Paszke et al., 2017). Most of the algorithms for these optimizations are the same as those in compilers. Our findings may also provide useful guidance toward better designing and testing optimizations in deep learning frameworks.

10. Conclusion

In this study, we present an empirical study of optimization bugs in GCC and LLVM. We collect 57,591 GCC bugs and 22,748 LLVM bugs, and then exhaustively examine 8771 and 1564 optimization bugs of GCC and LLVM, respectively. In particular, we have shown the characteristics of compiler optimization bugs including (1) the distribution of optimization bugs over time and the distribution in components; (2) the optimizations related to the optimization bugs; (3) the types of optimization bugs; (4) the time that optimization bugs live; and (5) the bug fixes information of optimization bugs.

The results illustrate that (1) optimizations are the buggiest component except for the C++ component in both GCC and LLVM; (2) the buggiest optimization in GCC is the value range propagation optimization, while it is the instruction combine optimization for LLVM; in both GCC and LLVM, the loop optimizations are more bug-prone than other optimizations; (3) more than half of optimization bugs are *Mis-opt* bugs, which account for 57.21% and 61.38% of the total optimization bugs in GCC and LLVM, respectively; (4) optimization bugs in both compilers live over five months, and the average months for fixing an optimization bug are 11.16 and 13.55 for GCC and LLVM respectively; many confirmed optimization bugs in GCC and LLVM have lived for a long time; (5) developers modify no more than two files and three functions on average to fix an optimization bug in both compilers, and around 99% of the bug fixes modify less than 100 lines of code, while 90% less than 50 lines of code.

Our study gives the compiler developers and researchers insight into understanding optimization bugs. This could guide the developers and researchers to better design, test, and debug the optimizations in compilers.

CRediT authorship contribution statement

Zhide Zhou: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Zhilei Ren:** Conceptualization, Validation, Data curation, Resources, Writing - review & editing, Project administration, Funding acquisition. **Guojun Gao:** Software, Formal analysis, Investigation, Data curation, Writing - review & editing, Visualization. **He Jiang:** Resources, Writing - review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments, which help us a lot to improve our paper. This work is supported in part by the National Natural Science Foundation of China under Grant Nos. 61772107, 61722202, 62032004, and 62072068.

References

Abadi, Martín., Ashish Agarwal, P.B., et al., 2015. Tensorflow: Large-scale machine learning on heterogeneous systems. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Ashouri, A.H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J., 2017. MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. ACM Trans. Archit. Code Optim. 14 (3), 29:1–29:28.

Chen, J., Han, J., Sun, P., Zhang, L., Hao, D., Zhang, L., 2019. Compiler bug isolation via effective witness test program generation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 223–234.

Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D., 2001. An empirical study of operating systems errors. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. SOSP '01, Association for Computing Machinery, New York, NY, USA, pp. 73–88.

GNU Compiler Community, 2020. Gcc's option summary. URL https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html.

Islam, MdJohirul., Nguyen, G.P.R., Rajan, H., 2019. A comprehensive study on deep learning bug characteristics. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 510–520.

Le, V., Afshari, M., Su, Z., 2014. Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14, ACM, New York, NY, USA, pp. 216–226.

Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C., 2006. Have things changed now? An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability. ASID '06, Association for Computing Machinery, New York, NY, USA, pp. 25–33.

Lidbury, C., Lascu, A., Chong, N., Donaldson, A.F., 2015. Many-core compiler fuzzing. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '15, ACM, New York, NY, USA, pp. 65–76.

Liu, X., Li, X., Prajapati, R., Wu, D., 2019. DeepFuzz: Automatic generation of syntax valid C programs for fuzz testing. In: Proceedings of the AAAI Conference on Artificial Intelligence.

LLVM Compiler Community, 2020. LLVM's analysis and transform passes. URL https://www.llvm.org/docs/Passes.html.

Lu, S., Park, S., Seo, E., Zhou, Y., 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XIII, Association for Computing Machinery, New York, NY, USA, pp. 329–339.

Lu, Y., Zhao, Z., Li, G., Jin, Z., 2019. Learning to generate comments for API-based code snippets. In: Software Engineering and Methodology for Emerging Domains. Springer Singapore, Singapore, pp. 3–14.

Marcozzi, M., Tang, Q., Donaldson, A.F., Cadar, C., 2019. Compiler fuzzing: How much does it matter? Proc. ACM Program. Lang. 3 (OOPSLA).

Nagai, E., Awazu, H., Ishiura, N., Takeda, N., 2012. Random testing of C compilers targeting arithmetic optimization. In: Workshop on Synthesis and System Integration of Mixed Information Technologies. SASIMI 2012. pp. 48–53.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A., 2017. Automatic differentiation in pytorch. URL https://pytorch.org/.

Sahoo, S.K., Criswell, J., Adve, V., 2010. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. ICSE '10, Association for Computing Machinery, New York, NY, USA, pp. 485–494.

Song, L., Lu, S., 2014. Statistical debugging for real-world performance problems. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '14, Association for Computing Machinery, New York, NY, USA, pp. 561–578.

Sun, C., Le, V., Su, Z., 2016a. Finding compiler bugs via live code mutation. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016, ACM, New York, NY, USA, pp. 849–863.

Sun, C., Le, V., Zhang, Q., Su, Z., 2016b. Toward understanding compiler bugs in GCC and LLVM. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ISSTA 2016, ACM, New York, NY, USA, pp. 294–305.

Thung, Ferdian., Shaowei Wang, D.L., Jiang, L., 2012. An empirical study of bugs in machine learning systems. In: 23rd IEEE International Symposium on Software Reliability Engineering. ISSRE 2012, Dallas, TX, USA, November 27-30, 2012, IEEE Computer Society, pp. 271–280.

Vu, L., Chengnian, S., Zhendong, S., 2015a. 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. OOPSLA 2015, ACM, New York, NY, USA, pp. 386–399.

Vu, L., Chengnian, S., Zhendong, S., 2015b. Randomized stress-testing of link-time optimizers. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. ISSTA 2015, ACM, New York, NY, USA, pp. 327–337.

Yang, X., Chen, Y., Eide, E., Regehr, J., 2011. Finding and understanding bugs in c compilers. In: ACM SIGPLAN Notices, Vol. 46. (6), ACM, pp. 283–294.

Zhang, Y., Chen, Y., Cheung, S.-C., Xiong, Y., Zhang, L., 2018. An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2018, Association for Computing Machinery, New York, NY, USA, pp. 129–140.

Zhang, Q., Sun, C., Su, Z., 2017. Skeletal program enumeration for rigorous compiler testing. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2017, ACM, New York, NY, USA, pp. 347–361.