Optimization bugs in Compiler

Dip hareshkumar Patel
Computer Science and Engineering
Nirma University
19BCE166

Jalpan Sharmilkumar Patel
Computer Science and Engineering
Nirma University
19BCE177

Parv Mayurkumar Patel
Computer Science and Engineering
Nirma University
19BCE190

***Abstract*—Compilers' essential component is optimization. In particular for safety-critical sectors, bugs in optimizations might have disastrous effects and result in undesired application behaviour. Therefore, a thorough examination of optimization problems should be carried out to assist developers in comprehending and testing compiler optimizations. In order to achieve this, we carry out an empirical investigation to look into the traits of optimization errors in two popular compilers, GCC and LLVM.The findings demonstrate the following five traits of optimization bugs: (1) Except for the C++ component, optimizations are the most bug-prone part of both compilers; (2) The buggiest optimizations in GCC and LLVM, respectively, are the value range propagation optimization and the instruction combine optimization; loop optimizations in both GCC and LLVM are more bug-prone than other optimizations; (3) The majority of mis optimization issues, which account for 57.21% and 61.38%, respectively, are found in both GCC and LLVM; (4) On average, optimization bugs last over five months, and it takes developers 11.16 months to fix a GCC bug and 13.55 months to fix an LLVM bug; numerous confirmed optimization bugs in both GCC and LLVM have been around for a very long time. For researchers and developers, the study provides a thorough understanding of optimization flaws. This might offer developers and academics useful direction for creating compiler optimizations that work better. The studies' findings also imply that we require better methods and equipment for vetting compiler enhancements. Our findings are also helpful for the study of automatic compiler debugging methods, such as automatic compiler bug isolation methods.**

***Index Terms*—component, formatting, style, styling, insert**

I. Compiler Testing for Optimization Sequences of LLVM

Compilers frequently use optimization sequences to boost program speed, however they can also result in serious defects, such compiler crashes. Although several techniques have been created to test compilers automatically, no systematic research has been done to find compiler flaws when using random optimization sequences. Due to the vast array of optimization sequences and testing programs, there are two major obstacles that must be overcome in order to tackle this issue: acquiring representative optimization sequences and choosing representative testing programs. In this work, we suggest CTOS, a brand-new compiler testing technique based on differential testing, to find compiler errors brought on by LLVM optimization sequences. To concurrently record the information about optimizations and their ordering, CTOS first uses the Doc2Vec approach to convert optimization sequences into vectors. In order to build the vector representations of the testing program that concurrently capture program semantics and structure, a method based on the region graph and call connections is created in CTOS. Then, in order to overcome the aforementioned two difficulties, a "centroid"-based selection technique is presented using the vector representations of optimization sequences and testing programs. Finally, CTOS runs each testing program using all of the representative optimization sequences after receiving the testing programs and representative optimization sequences as inputs. An optimization sequence is considered to have caused a compiler issue if a result from a specific testing program differs from the majority of the others. Our analysis shows that CTOS greatly surpasses the baselines in terms of the bug-finding capabilities by up to 24.76% 50.57% on average.studies reported 104 legitimate issues in 5 kinds throughout assessments of LLVM, of which 21 have been confirmed or corrected. The majority of those flaws—57—are incorrect code flaws and crash flaws (24). There are 47 different erroneous optimizations found, 15 of which are loop-related.

Identify applicable funding agency here. If none, delete this.

II. Validating Optimizations of Concurrent C/C++ Programs

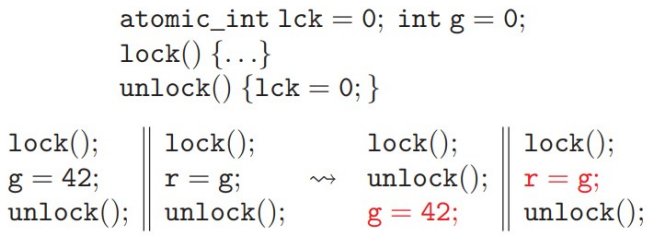


Fig. 1. Unsafe reordering introducing a data race on g

As a very simple example of an incorrect transformation, consider the one shown in Figure 1. The program before the transformation always uses the shared variable g while the lock is held, and so the two accesses to g are ordered. After reordering the g = 42; and the unlock(); statements, however, the store to g is no longer protected by the lock and hence may race with the load on g in the second thread. While reordering g = 42; and unlock(); is correct for sequential programs, it is clearly wrong for concurrent programs because it introduces a data race and, as such, it is forbidden by the C/C++11 standards (ISO/IEC 9899:2011; ISO/IEC 14882:2011).

The validator takes as inputs the programs before and after a set of transformations. It compares them by matching their memory access patterns and reports on whether it could find a matching demostrating that transformation is correct.

A. C11 Semantics

Two accesses are concurrent in an execution if they are not related by the happens-before order. An execution is racy if it has two concurrent accesses to the same location, at least one of which is non-atomic (NA) and at least one of which is a write or update. Programs that have a consistent racy execution have "undefined" semantics; otherwise their semantics is exactly the set of their consistent executions. To illustrate the semantics consider the following example where all variables initially have the value zero.

$$\begin{array}{l} g_{\mathrm{NA}} = 3; \\ X_{\mathrm{REL}} = 1; \end{array} \Bigg\| \begin{array}{r} \mathrm{if}(X_{\mathrm{ACQ}}) \\ g_{\mathrm{NA}} = 4; \end{array}$$

B. Allowed Transformations in C11

We review the memory access reordering and elimination transformations allowed by C11. Memory access introduction is typically incorrect as it may introduce a data race. The only exception is introducing a read adjacent to another access of the same memory location. Safe Reorderings In the sequential setting, two adjacent actions a and b are reorderable (a; b b; a) if they access different locations and they do not have any control or data dependence between them. In the concurrent setting, we must further ensure that no accesses are moved out of critical regions as in Figure 1. The notions of acquire and release actions generalize the notions of acquiring a lock and releasing a lock (Vafeiadis et al. 2015) as explained in Figure 2. Elimination of Redundant Accesses Redundant actions can be eliminated in C11 if they are made redundant by an adjacent memory access (e.g., a store overwritten by another one). If the eliminated access and the justifying access are not adjacent, then the elimination is valid if the access to be eliminated can be moved so as to become adjacent to the one justifying its elimination. Vafeiadis et al. (2015), for example, present these elimination rules for atomic accesses.

Read-after-Read: $\mathsf{R}_X(\ell); C; \mathsf{R}_Y(\ell) \rightsquigarrow \mathsf{R}_X(\ell); C$ is safe if $acquire \notin C$ and $X \sqsupseteq Y$.
Read-after-Write: $\mathsf{W}_X(\ell); C; \mathsf{R}_X(\ell) \rightsquigarrow \mathsf{W}_X(\ell); C$ is safe if $acquire \notin C$ and $X \sqsupseteq Y$.
Overwritten Write: $\mathsf{W}_Y(\ell); C; \mathsf{W}_X(\ell) \rightsquigarrow C; \mathsf{W}_X(\ell)$ is safe if $release \notin C$ and $X \sqsupseteq Y$.

```
1  int c;
2  struct m{
3    unsigned:20;
4    signed a;
5    signed b
6  };
7  struct m d() {}
8  e() {
9    for (; c; c++)
10   d();
11 }
```

```
1  #include<stdio.h>
2  int a = 0;
3  int d() {
4    int e = 2;
5    for (a = 0; a <= 8; a++);
6    return e;
7  }
8  void main() {
9    int f = 0;
10   d();
11   printf("%d\n",a);
12 }
```

Fig. 2. LLVM Bug

C. *Illustrative Examples*

In this subsection, we present two concrete examples to illustrate the compiler bugs in LLVM caused by optimization sequences. Note that these examples in this section only aim to show the phenomenon of the compiler bugs introduced by optimization sequences. Fig. 1a shows a program that triggers a crash bug by the loop-vectorizer2 optimization when the program is optimized by the optimization sequence "-functionattrs -loop-rotate -licm-sroa -loop-vectorize." An assertion fails when loop-vectorizer works on the Intermediate Representation (IR) file optimized by "-functionattrs -loop-rotate -licm -sroa." This bug occurs because the loop invariant operands are scalarized, but they are used outside the loop and should be ignored when computing the scalarization overhead.3 However, when we delete any optimization from functionattrs, loop-rotate, licm, sroa, or change the order, or only use loop-vectorizer to optimize the program, this bug disappears. In addition, there does not exist any bug when the program is optimized by the default standard optimization levels, e.g., O1, O2, and O3 provided by LLVM

III. Approach

The CTOS's framework is depicted in Fig. 3. Differential testing is often the foundation of CTOS [31]. Contrary to other studies, CTOS determines whether there are compiler bugs by comparing the outputs of testing programs optimized by various optimization sequences rather than by comparing the outputs of testing programs compiled by various compilers or optimized by various default optimization levels. In Fig. 3, the terms "front-end" and "back-end" refer to the compiler's front-end and back-end, respectively, which are used to convert a testing program's source code into its LLVM IR and produce executables. Since C programs are utilized as the testing programs in our study, we employ Clang [23], a popular language front-end and compiler driver based on LLVM for the C language family (e.g., C and C++). As the program's front-end, Clang receives the source code and generates the relevant IR. Assembler and linker are only two of the numerous tools used by the back-end to create the final executables from IRs. For ease of usage, we additionally utilize Clang as a

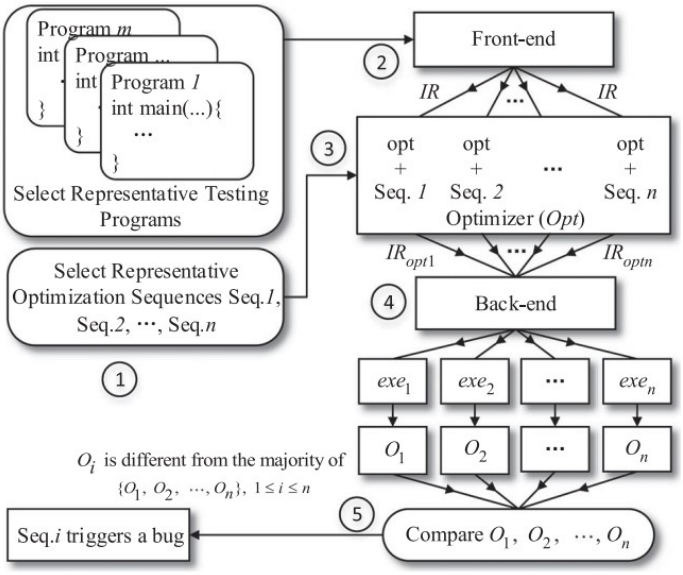


Fig. 3. Framework of CTOS for catching compiler bugs caused by optimization sequences of LLVM.

driver to plan and run the back-end tools (such the linker), and it can accept LLVM IRs to produce executables. The LLVM optimizer Opt is in charge of planning and carrying out optimizations. Five steps make up CTOS. (1) Choosing sample optimization sequences and testing programs comes first. (2) After that, a compiler's front-end is utilized to generate the IR file for a specified testing program without doing any optimizations. (3) In the third phase, the IR created in the previous step is optimized using the LLVM optimizer Opt and the chosen optimization sequences. The optimizer Opt will create n optimized IRs (i.e., IRopt1, IRopt2,..., IRoptn) for the IR of a testing program if there are n chosen optimization sequences. (4) A compiler's back-end loads the n optimized IRs in the fourth step to create n executables (i.e., exe1, exe2, ... , exen). (5) Getting the outputs (i.e., O1, O2,..., On) of the n executables and comparing them to see whether there are any flaws is the last step. Although some of the outputs may differ, most of them ought to be the same. Therefore, for the supplied testing program, the ith optimization sequence is assumed to cause a compiler problem if an output Oi differs from the majority of O1, O2,..., On, 1 I n. The first phase is unquestionably the basis of CTOS, as seen in Fig. 3. In general, we can construct testing programs and optimization sequences at random. However, because there are so many testing programs and optimization sequences, this random technique could not be effective. Meanwhile, comparable optimization processes and testing methods may result in a large number of duplicate defects. Therefore, we must choose representative testing programs and optimization sequences. We first introduce the vector representations of optimization sequences and testing programs, respectively, An optimization sequence is a collection of optimizations carried out in a certain order. As a result, optimization sequences that contain comparable optimizations in the same order may result in double-occurrence compiler faults. Additionally, the ability to discriminate between various testing programs depends on the semantics and structural data of a testing program. As a result, testing programs may experience duplicate compiler problems if their semantics and structural information are comparable. A method based on the region graph and call relationships of a program is proposed to represent testing programs as vectors, so that the semantics and structure information of a program can be captured by vectors. This method, known as Doc2Vec, captures optimizations and their orders of the corresponding optimization sequence simultaneously. We assume that comparable optimization sequences and testing programs are close to one another in their respective vector spaces using the vector representations of optimization sequences and testing programs. To choose representative optimization sequences and testing programs, we thus provide a centroid-based selection strategy that maximizes the distances between the chosen representative optimization sequences and testing programs, respectively.

A. *Representation of Optimization Sequences*

An optimization sequence is made up of various optimizations that are performed in a certain order. As a result, the representation of an optimization sequence should accurately reflect the particular optimizations and their order. An optimization sequence intuitively resembles a phrase in spoken language, which is made up of a few words in a certain order. We consider optimization sequences as sentences in this work so that effective sentence representation techniques may be used to convert optimization sequences into vectors. However, several cutting-edge techniques for representing sentences, like the bag-of-words [32], are unable to capture the word order. They are unable to discern between similar-word sentences. We use Doc2Vec [29], a well-liked and often used phrase vector encoding approach, to encode optimization sequences as vectors in order to simultaneously capture optimizations and their ordering of an optimization sequence. A approach for learning continuously distributed vector representations of phrases or documents called Doc2Vec is unsupervised. Word orders are taken into account by Doc2Vec so that sequences with various word orders have various vector representations. Furthermore, variable-length optimization sequences may be created using Doc2Vec for variable-length word sequences.

B. *Representation of Testing Programs*

Another important element that might create compiler errors brought on by optimization sequences is evaluating programs. Different bugs may be brought on by various testing programs. Therefore, in order to increase test efficiency and discover more varied issues, we must use representative testing methods. Our research focuses on identifying compiler flaws brought on by LLVM optimization sequences, which is what led us to choose LLVM IR to build vector representations of testing programs. The LLVM IR is a simple, low-level representation of programs that is expressive, typed, and

extendable [21]. In this section, we provide a method for representing a testing program as a vector based on the region graph and call connections produced by the unoptimized IR. This method allows us to collect program semantics and structural data, which is helpful for choosing representative testing programs. We separate the representation of a function from the representation of the entire program in the vector representation of a testing program. A deep region-first strategy is used to aggregate the vectors of each edge under two constraints in order to generate the vector representation of a function. First, we use the Doc2Vec technique to convert the instruction sequences of each edge in the region graph of a function into vectors. The vector representation of the entire program is created by aggregating all of the function vector representations according to their call connections.

IV. Implementation

Compilers. In our study, we only conduct our experiments on LLVM. The reason is that, as to our knowledge, only LLVM currently can allow developers to adjust the orders of optimizations. For GCC, another mature and widely used compiler in both industry and academia, the orders of optimizations are fixed.6 Although any order of optimizations can be passed as command-line arguments to GCC,the orders of these optimizations cannot affect the behavior of GCC. The same case also occurs for CompCert that is averified and high-assurance compiler for the C language.The fixed order of optimizations is beneficial for the rapid implementation and safety of compilers. However, there is no doubt that the fixed order of optimizations limits the capabilities of compilers to optimize programs for different requirements. In contrast, as a compiler providing support for arbitrary orders of optimizations, LLVM has been widely used to implement many compilers and tools. Our study aims to improve the reliability of optimizations with arbitrary orders for LLVM, which helps to guarantee the correctness of different LLVM-based compilers and tools.

Optimizations In our study, C programs are used as the inputs of LLVM. Thus we mainly focus on testing the machine independent optimizations of LLVM that are useful for the C programming language. We currently do not consider the optimizations for object-oriented programming languages (e.g., C++ and Objective-C), profile guided optimizations, and results visualization of optimizations. Finally, 114 optimizations are selected.8 In LLVM, each optimization may depend on certain other optimizations as the preconditions. LLVM provides a mechanism to manage the dependencies.

Bug Types In this study, it mainly find the following five types of compiler bugs caused by optimization sequences of LLVM. (1) Crash. The optimizer Opt of LLVM crashes when optimizing the IR of a program. (2) Invalid IR. In LLVM, each optimization takes in the valid IR of a program as input, and its outputs also should be a valid IR. However, invalid IR may be generated by some optimizations due to the interacation among optimizations. In our evaluation,tune on the option "-verify-each11" of the optimizer Opt to verify whether the output IR is valid after every optimization. If the IR is invalid after an optimization, the optimizer Opt will be stopped and will output some error messages. (3) Wrong code. The optimized IR produced by the optimizer Opt may contain different semantics to the original program, which makes the corresponding executable produce wrong outputs, or occur segmentation faults or floating point exceptions. (4) Performance. When the out-of-memory of the optimizer Opt occurs, we treat it as performance bugs. It could slow the compilation of programs. In the worstcase, the computer system may be jammed due to the performance bug. Generally, we set the maximum size of the memory of an optimizer process to 4 GB, since it is sufficient to optimize a testing program using 4 GB memory in most cases. Thus if themaximum memory of the optimizer Opt is greater than 4 GB, the optimizer will be stopped like a crash bug. We also try not to limit the size of the memory, but it has the same results as the 4 GB limitation. (5) Code generator bug. The code generator in the backend is used to generate the assembly code of a program from the corresponding IR. However, the IR of a program optimized by some optimizations may trigger some bugs in the code generator. Currently, we only find one bug for this type, it makes the code generator not emit a machine instruction and stops the code generator.

	Average								Min. total bugs	Max. total bugs	*P*-value	Effect size (A_{12})
	TP.	Crash	WC.	Inv. IR	Perf.	CGB	Total bugs	*imp.*				
CTOS(*RP+RS*)	113.2	6.4	1.0	1.4	1.0	0.0	9.8	33.67%	7	12	<.001	0.955
CTOS(*RP+SS*)	100.7	6.7	1.7	1.0	0.9	0.0	10.3	27.18%	8	12	.001	0.905
CTOS(*RP+2W*)	9.2	7.2	1.3	0.4	1.0	0.0	9.9	32.32%	8	11	<.001	0.960
CTOS(*RPS+RS*)	28.8	5.2	1.0	1.4	1.0	0.1	8.7	50.57%	7	10	<.001	1.000
CTOS(*RPS+SS*)	27.5	5.4	1.4	1.2	0.9	0.0	8.9	47.19%	7	11	<.001	0.980
CTOS(*RPS+2W*)	3.9	5.6	1.3	0.9	1.0	0.0	8.8	48.86%	6	11	<.001	0.990
CTOS(*RPSc+RS*)	61.4	5.7	0.9	1.3	1.0	0.1	9.0	45.56%	7	12	<.001	0.965
CTOS(*RPSc+SS*)	53.0	5.7	1.7	1.4	1.0	0.1	9.9	32.32%	9	11	<.001	0.980
CTOS(*RPSc+2W*)	5.5	6.6	1.3	1.2	1.0	0.1	10.2	28.43%	8	13	<.001	0.915
CTOS(*SP+RS*)	122.4	6.6	1.4	1.2	1.0	0	10.1	29.70%	9	11	<.001	0.960
CTOS(*SP+2W*)	8.6	6.8	1.8	0.6	1.0	0	10.2	28.43%	8	12	<.001	0.935
CTOS(*SPS+RS*)	117.8	5.8	1.1	1.1	1.0	0	9.0	45.56%	7	10	<.001	1.000
CTOS(*SPS+SS*)	102.4	6.9	2.0	0.6	1.0	0	10.5	24.76%	9	12	.001	.0930
CTOS(*SPS+2W*)	8.2	7.0	1.5	0.5	1.0	0	10.0	31.00%	8	12	<.001	0.955
CTOS	109.9	7.4	3.5	1.2	1.0	0	13.1	–	11	15	–	–

TP.: Testing Programs, Inv. IR: Invalid IR, WC.: Wrong Code, CGB.: Code Generator Bug.

Fig. 4. Results of CTOS and its Variants

V. Optimization bugs in GCC and LLVM

Compilers, like GCC and LLVM, are among the most crucial pieces of system software. They are crucial for converting source code into machine code. Compilers additionally offer optimization methods to enhance programme performance.Until now, compilers have been fundamentally dependent on optimizations. Compilers have implemented countless optimizations. For instance, GCC and LLVM each include more than 200 and 100 optimizations, respectively.Compiler optimization flaws, like those in application software, must eventually be introduced as compilers develop. Particularly in safety-critical fields, optimization errors might result in undesired actions and catastrophes. To find compiler problems, numerous methods have been developed. With more than 400 reported compiler faults, Csmith is the most popular testing programme generator for C/C++ compilers. Additionally, more than 1600 compiler errors have been found using techniques based on equivalence modulo inputs. There is, however, no research to assist researchers and developers in understanding

compiler optimization problems. Sun et al(2016b) .'s empirical study on compiler bugs does focus on the general state of compiler issues in GCC and LLVM, though.In addition, not all programmers are aware of compiler optimization flaws in real-world situations. Particularly for novice developers (e.g., students), They consistently believe that any flaws in their programmes are the result of their own mistakes rather than compiler optimizations. We must therefore further elucidate the characteristics of optimization bugs in order to improve understanding, identification, and repair of optimization bugs in compilers. We perform an empirical study to look at the characteristics of optimization bugs in compilers in order to better understand them. We also look into two popular production compilers, GCC and LLVM, much like the research done by Sun et al. We compile 57,591 GCC bugs and 22,119 LLVM problems, and we thoroughly evaluate 8771 optimization bugs in GCC and 1564 optimization bugs in LLVM, respectively. To analyse optimization flaws, we particularly pay attention to the following five research questions (RQs). RQ1 and RQ2 in these RQs concentrate on the overall trend of optimization defects in LLVM and GCC, respectively, which can aid academics and developers in comprehending the evolution and dispersion of optimization bugs. While RQ3 also gives researchers and developers the kind of information about optimization flaws. RQ4 and RQ5 look into how long optimization defects last and how to fix them, which can help with compiler optimization testing and debugging.

A. *How are optimization bugs distributed?*

In order to comprehend the significance of optimization bugs for compiler development, this RQ looks into the general progression history of optimization flaws as well as the distribution of optimization problems in components. The findings demonstrate that whereas LLVM has seen an increase in optimization problems over the years, GCC has had a rather constant trend over the same period. Furthermore, with the exception of the C++ component, optimization errors make up a larger portion of all bugs than do bugs in other components. Additionally, the GCC tree-optimization component and the LLVM scalar optimization component are where the majority of optimization issues are found.

B. *Which optimizations are buggy?*

This RQ looks into which optimizations are bugs. We remove 119 and 101 files that are specifically relevant to the GCC and LLVM optimizations, respectively. The frequency of modifications to each file is next examined. The findings show that the most problematic optimizations in GCC and LLVM, respectively, are the value range propagation optimization and the instruction combine optimization. Furthermore, compared to other optimizations, the loop optimizations in LLVM and GCC both contain more problems.

C. *What are the types of optimization bugs?*

In this study, we look into many optimization bugs. We manually classify each optimization bug as either Misoptimization (Misopt), Crash, or Performance (see Section 2.2 for definitions), in accordance with 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). According to the findings, Mis-opt issues account for the majority of optimization bugs in both GCC and LLVM, accounting for 57.21% and 61.38%, respectively. Mis-opt bugs are, however, fixed at a rate that is lower in both compilers than that of Crash and Performance bugs.

D. *How long do optimization bugs live?*

This RQ investigates the lifespan of optimization bugs. On average, optimization defects persist for more than five months, and fixing them takes 11.16 months for GCC and 13.55 months for LLVM. In GCC, optimization issues in the rtl-optimization component take longer to fix, taking an average of 13.57 months, and developers take longer to fix Mis-opt bugs, taking an average of 15.07 months. The Interprocedural Optimizations component of LLVM takes the longest to fix defects, taking an average of 17.77 months. Performance bugs take the longest to fix, taking an average of 15.73 months, longer than Mis-opt bugs and Crash bugs in LLVM. Numerous proven optimization problems have existed for a very long period in both LLVM and GCC. The confirmed optimization flaws have typically been present in GCC for 72.38 months and in LLVM for 14.38 months.

E. *How many functions, files, and lines of code must be changed to address an optimization bug?*

This RQ looks into information about optimization bug patches. We focus on changing files, functions, and lines of code to correct optimization errors. The findings reveal that, on average, both compilers' bug patches only affect two files and three functions. In GCC, the average bug repairs for bugs in the tree-optimization component involve 1.81 files and 3.84 functions, while the average bug remedies for Performance bugs, which are greater than those of other sorts of bugs, involve 2.82 files and 7.18 functions. In LLVM, the bug fixes for the Interprocedural Optimizations component require the modification of more files (1.75 files on average), whereas the bug fixes for Transformation Utilities require the modification of more functions (2.67 functions on average). Similarly, the bug fixes for Misopt and Crash require the modification of more files (1.44 files and 1.39 functions, respectively). 90% of problem patches in both compilers change fewer than 50 lines of code, while 99% change no more than 100 lines.

VI. Methodology

A. *Compiler*

In this study, we select GCC and LLVM, two popular compiler systems, to examine their optimization problems. These two open-source compiler systems are widely employed in both business and academics. GCC GCC stands for the GNU Compiler Collection3, a standard three-stage compiler system that consists of the front end, middle end, and back end.It offers back ends for many target architectures such X86, MIPS, PowerPC, and RISCV as well as front ends for different

programming languages like C, C++, Objective-C, and Fortran. Additionally, it offers more than 200 improvements to enhance programme efficiency (GNU Compiler Community, 2020).It has been in active development for more than 30 years, starting in the late 1980s. LLVM The compiler infrastructure LLVM is a well-established and popular one. Similar to GCC, LLVM is a three-stage compiler system that supports a variety of target architectures and programming languages. For random programming languages, it offers a bundle of modular, reusable compiler and toolchain technologies. A wide range of statically and runtime compiled languages, including those supported by GCC, Rust, Swift, Ruby, Haskell, and WebAssembly, have been created based on LLVM and share a similar architecture. Additionally, LLVM has created thousands of analysis and transformation optimizations (LLVM Compiler Community, 2020). Since its launch in December 2000, LLVM has attracted a lot of interest from both business and academics.

B. Bug sources

We gather bugs from the bug repositories of LLVM and GCC. In line with the previous study (Sun et al., 2016b), we consider the confirmed and corrected issues in our analysis. As a result, we have a thorough understanding of the traits of LLVM and GCC optimization problems. In the bug repository of GCC and LLVM, a bug is considered fixed if its resolution field is set to fixed and its status field is set to resolved, verified, or closed. GCC developers treat a bug with a new status as a confirmed bug in contrast to a bug that has been fixed. In the GCC bug repository, an issue is considered confirmed if its status field is set to new and its resolution field is empty. While the LLVM confirmed bug says that the resolution field is empty and the status field is set to confirmed. Bugs in Optimization Identification According to the components they are a part of, bugs have been categorised in the bug repositories of GCC and LLVM. GENERIC, GIMPLE, and RTL are the three primary intermediate languages used by GCC to represent the programme during compilation, with GIMPLE and RTL being utilised to optimise the programme. The Register Transfer Language is represented by RTL, while GIMPLE is an abstract-syntax-tree-based representation. Thus, out of the 52 components in GCC, developers of optimization issues typically identify the component as either the ”tree-optimization” component or the ”rtl-optimization” component. 9 Therefore, in this study, we discover GCC optimization flaws based on the components ”tree-optimization” and ”rtl-optimization.” Out of 96 components in LLVM, the ”Scalar Optimizations,” ”Loop Optimizer,” ”Transformation Utilities,” and ”Interprocedural Optimizations” components are where the optimization problems are found. These LLVM optimization-related components are configured in accordance with the folders that contain the functional and buggy optimizations. 10 For instance, the component ”Loop Optimizer” contains the optimization bugs that are produced by loop optimizations, whereas the component ”Scalar Optimizations” indicates that the buggy optimizations belong to the directory ”scalar” in the LLVM project.

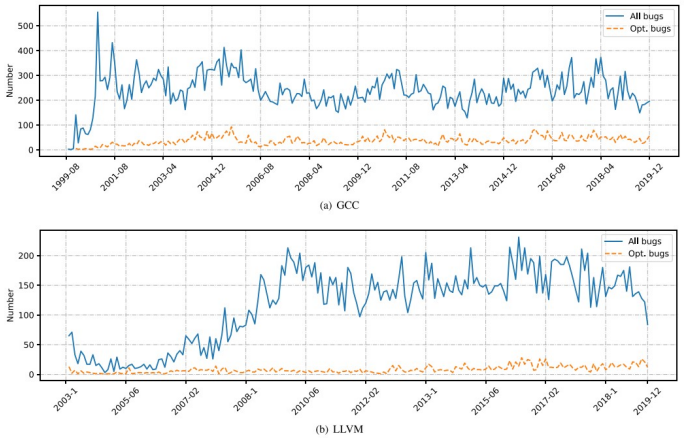


Fig. 5. evolution history of the optimization bugs (in months)

TABLE I
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

VII. WHAT IS THE DISTRIBUTION OF OPTIMIZATION BUGS?

This section shows the distribution of optimization bugs.

A. General statistics

demonstrates the general development of the optimization bugs for LLVM and GCC. It specifically displays the monthly totals for both general and optimization defects. The trajectory of the optimization bugs is almost consistent with the trend of all bugs, as can be shown in Figs. 1(a) and 1(b). Early on in its development (1999–2006), GCC attracted a lot of attention, which increased the amount of defects. In contrast to LLVM, GCC's trends have been more stable in recent years. In recent years, LLVM has developed into a mature and extensively used compiler infrastructure, garnering a lot of interest from both business and academics. As a result, the number of LLVM problems is growing quickly. We display in Figure 2 the distribution of bugs in compiler components to highlight the significance of optimization bugs. In total, GCC and LLVM have 52 and 96 components, respectively. We also display the top ten buggy components for each compiler, similar to the research by Sun et al. For GCC and LLVM, the percentages of problems in these top 10 components are 82.72% and 69.96%, respectively. The defects of optimization components in both GCC and LLVM account for a significant portion, as seen in Figs. 2(a) and 2(b). The component with the most problems in GCC, accounting for 21.63% of the 57,591 bugs, is C++. These ten flawed components contain four language-dependent components, including C++, Fortran, C, and Ada.

They are all a part of the GCC's front end. The optimization components in GCC, such as "tree-optimization" and "rtl-optimization," are the most problematic ones, accounting for about 15.23% of the defects, in addition to the language-dependent ones. The most problematic part of LLVM is "new-bugs," as developers can report defects without mentioning specific components (Sun et al., 2016b). Like GCC, LLVM's C++ component has a significant proportion of flaws. The optimization components in LLVM, which include "Scalar Optimizations," "Loop Optimizer," "Transformation Utilities," and "Interprocedural Optimizations," are the most bug-ridden components, accounting for 6.88% of the 22,748 defects, along with the C++ and new-bugs components. Compared to GCC, LLVM has a lower percentage of errors in its optimization-related components. This might be because LLVM's new-bugs component has received certain optimization bugs.

B. Bugs in optimization components

The problems in each optimization component of LLVM and GCC are displayed in Tables 2 and 3, respectively. For each optimization component, we specifically show the amount of defects with two statuses (i.e., Fixed and Confirmed). According to Table 2, which shows that 65.15 percent of the 8771 optimization problems for GCC are in the "treeoptimization" component. The "tree-optimization" component's" extensive implementation of optimizations may be the cause. The "tree-optimization" component is directly related to 85 files in GCC. The problems in each optimization component of LLVM and GCC are displayed in Tables 2 and 3, respectively. For each optimization component, we specifically show the amount of defects with two statuses (i.e., Fixed and Confirmed). According to Table 2, which shows that 65.15 percent of the 8771 optimization problems for GCC are in the "treeoptimization" component. The "tree-optimization" component's" extensive implementation of optimizations may be the cause. The "tree-optimization" component is directly related to 85 files in GCC.

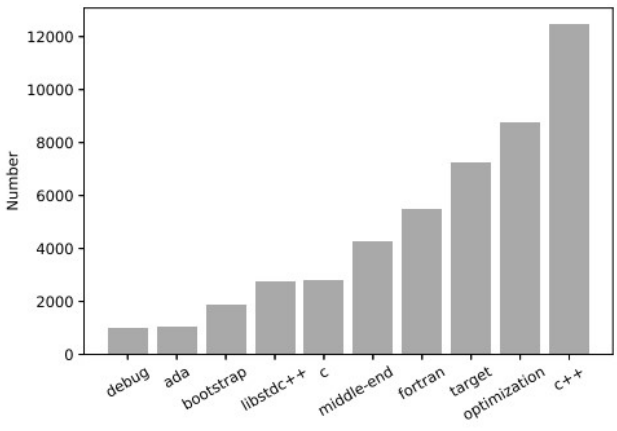
VIII. Which optimizations are buggy?

Information on problematic optimizations in LLVM and GCC is provided in this section. It is difficult to comprehend all of GCC and LLVM's optimizations, thus rather than detailing the specific optimizations, we display the files that implement the optimizations. We read the commit logs to discover the commits for the revisions of the flawed optimizations in order to acquire the files of such optimizations.

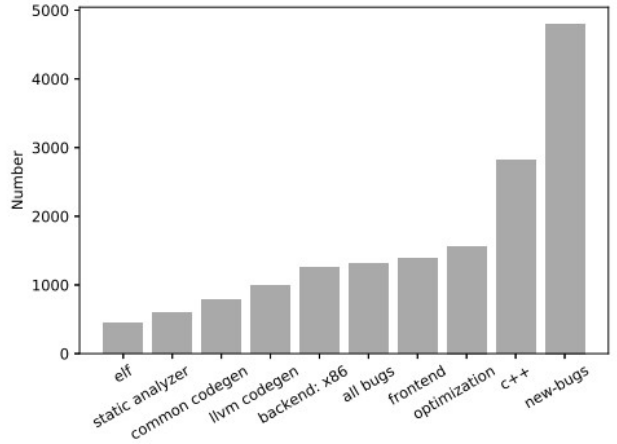
TABLE II
BUG IN EACH GCC OPTIMIZATION COMPONENT

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

To determine whether a change is for an optimization bug, we use the pattern described in Section 2.2. After extracting the modified files, we manually determine whether a file is being used to carry out an optimization. Thus, we completely extract 1589 and 348 files for GCC and LLVM, respectively, of which 119 and 101 files are utilized to implement optimizations in those programs. Since the names of these files may, in part, represent the purpose of the optimizations done in them, we can infer information about the flawed optimizations indirectly from them. The top 30 problematic files pertaining to LLVM and GCC optimizations are displayed in Tables 4 and 5. We describe each file in accordance with the GCC and LLVM documents. Each file's frequency indicates how many times it has been modified to correct errors. Table 4 reveals that the "tree-optimization" component contains the buggiest files



(a) GCC



(b) LLVM

Fig. 6. The top 10 buggy components in GCC and LLVM.

TABLE III
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 optimization	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

used to apply GCC optimizations. The most problematic file is "tree-vrp.c," which has undergone at least 143 changes with optimization-related problems. The file "tree-vrp.c" contains the value range propagation optimization. Furthermore, rather than interprocedural optimization for the entire program, the majority of the optimizations in these 30 files concentrate on optimizing a particular function. The interprocedural constant propagation optimization is implemented in just one of these 30 files, ipa-cp.c. This might be a sign that more testing is necessary for GCC's single-function optimization optimizations. Five specific files—"tree-loopdistribution.c," "tree-parloops.c," "tree-ssaloop.c," "tree-ssa-loop-ivcanon.c," and "tree-ssa-loop-im.c"—concentrate on loop optimizations, which may be a sign that developers of GCC should pay more attention to this area.

TABLE IV
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 V
BUGS OF EACH TYPE IN LLVM 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 optimization	99 (67.35%)	45 (30.61%)	3(2.04%)	147 (9.40%)
Transformation utilities	80(64.52%)	42 (1.61%)	71(4.54%) 124 (7.93%)	
Total	960 (61.38%)	533 (34.08%)	71(4.54%)	1564

IX. WHAT ARE THE TYPES OF OPTIMIZATION BUGS?

A. General statistics

An significant factor in determining a compiler's quality is the type of bugs it produces. We can learn more about the shortcomings of the GCC and LLVM optimizations by analyzing the different sorts of optimization bugs. According to the compiler testing literature, we manually classify each optimization fault in this study. Generally speaking, there are three forms of optimization bugs: Crash, Misoptimization (abbreviated as "Mis-opt"), and Performance. Section 2.2 contains information on these three bug categories in detail. The percentage of each sort of optimization bug in LLVM and GCC is shown in Fig. 3. According to Figs. 3(a) and 3(b), which represent 56.15% and 68.03% of all optimization defects for GCC and LLVM, respectively, Mis-opt bugs account for the majority of optimization bugs in both GCC and LLVM. LLVM has a higher percentage of Mis-opt bugs than GCC does. This can be due to LLVM's inadequate testing. However, compared to LLVM, GCC has a higher percentage of Crash defects. This might be because LLVM uses C++, whereas GCC uses the C programming language to implement its code. Although the GCC developers have used the C++ programming language to reorganize the GCC code to some extent, there is still considerable work to be done. Performance bugs make up a very small portion of bugs in GCC and LLVM compared to Mis-opt bugs and Crash bugs. One explanation is that since performance defects are challenging to reproduce in diverse environments, testers and users may not pay much attention to them. Performance defects are frequently brought on by system settings on the user's end, making it difficult for developers to duplicate them on their end. Additionally, we can't completely rule out the possibility that performance defects are actually more uncommon than other types of bugs.

TABLE VI
THE STATUS OF OPTIMIZATION BUGS FOR EACH BUG TYPE IN GCC

Type	Fixed	Confimed	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 VII
THE STATUS OF OPTIMIZATION BUGS FOR EACH BUG TYPE IN GCC

Type	Fixed	Confimed	Total
Mis-opt	919 (95.73%)	41 (4.27%)	960 (67.38%)
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

B. 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.

C. Status of optimization bugs for each bug type

We quantify the amount of optimization bugs according to their types and status in order to look into the status of each problem category. The status of each problem class for optimization bugs in GCC and LLVM is shown in Tables 8 and 9. In general, LLVM fixes bugs more frequently than GCC does. In comparison to GCC, 96.68% of optimization

problems in LLVM have been fixed. In addition, Tables 6, 7, 8, and 9 show that Crash and Mis-opt issues make up the majority of optimization bugs in both LLVM and GCC. For both GCC and LLVM, the bug-fixing rate for Crash problems is higher than that for Mis-opt bugs. For GCC and LLVM, the bug-fixing rates for Crash problems are 98.26% and 98.12%, respectively. While the bug-fixing rates for GCC and LLVM's Mis-opt bugs are just 79.02% and 95.73%, respectively. This might be due to how challenging it is to identify and analyze the fundamental causes of Mis-opt problems. Developers can utilize backtrace data to examine bug causes for crash bugs, but very limited data can be used to help developers rationally understand the bug causes for mis-opt bugs. Additionally, LLVM has a greater bugfixing rate than GCC for performance-related problems. For GCC, there are 10.23% confirmed performance problems that have not yet been resolved. Only one of 71 LLVM Performance issues, or 1.41% of them, has been officially confirmed as unfixed. Additionally, when compared to GCC, LLVM has a higher bugfixing rate for Mis-opt problems. Only 79.02% of GCC problems have been fixed compared to 95.73% of Mis-opt bugs. One explanation could be that LLVM's superior modular design makes it easier for developers to quickly identify the causes of Performance bugs and Mis-opt bugs.

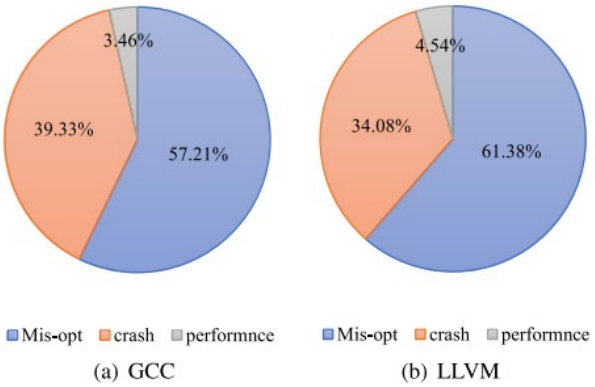


Fig. 7. The percentage of optimization bugs for each bug type in GCC and LLVM.

REFERENCES

[1] G. Fursin et al., "Milepost GCC: Machine learning enabled selftuning compiler," Int. J. Parallel Program., vol. 39, no. 3, pp. 296– 327, 2011.

[2] J. Ansel et al., "OpenTuner: An extensible framework for program autotuning," in Proc. 23rd Int. Conf. Parallel Archit. Compilation, 2014, pp. 303–316.

[3] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl., 2012, pp. 147–162.

[4] D. B. Loveman, "Program improvement by source-to-source transformation," J. ACM, vol. 24, no. 1, pp. 121–145, 1977.

[5] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," ACM SIGMICRO Newslett., vol. 13, no. 4, pp. 125–133, 1982.

[6] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," ACM SIGMICRO Newslett., vol. 13, no. 4, pp. 125–133, 1982.

[7] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," ACM SIGMICRO Newslett., vol. 13, no. 4, pp. 125–133, 1982.

[8] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation, 2017, pp. 347–361

[9] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in Proc.

[10] ACM Int. Conf. Softw. Eng., 2017, pp. 700–71 A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles, Techniques, and Tools (2nd Edition)," Addison-Wesley Longman Publishing Co., Inc., 2006

[11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In POPL'11, pages 55–66. ACM, 2011.

[12] K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In POPL'12, pages 509–520. ACM, 2012.

.