SHORT-PAPER

Quality Assurance for Non-trivial Systems: Use Case GCC Plugins

NIMANTHA KARIYAKARAWANA, KU Leuven, Leuven, Vlaams-Brabant, Belgium

Published: 11 September 2024

ISSTA '24: 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis
September 16 - 20, 2024
Vienna, Austria

Conference Sponsors:
SIGSOFT

Quality Assurance for Non-trivial Systems: Use Case GCC Plugins

Nimantha Kariyakarawana
KU Leuven
Leuven, Belgium
nimantha.kariyakarawana@kuleuven.be

Abstract

There are software systems that function without human intervention. Such systems work according to preset rules, algorithms, or data. In this doctoral research, we aim to explore the testing-related challenges associated with such systems with GCC plugins as our use case. The GCC compiler family is widely known especially as a compiler for C and C++. GCC allows extensions to be added through the GCC Plugin API. GCC plugins execute at the compiler level and influence the compiler at different compiling stages. Plugins depend on the GCC internal data structures such as AST, GIMPLE, Basic Blocks and RTL. Therefore, it is difficult to test the correctness of a GCC plugin. Testing is essential for ensuring the quality and the correctness of the plugins. The attempts made in the past are insufficient. They depend on testing the plugins from a black-box perspective. Internal issues may remain hidden when only testing from a black-box perspective is considered. Testing the plugins from both white-box and black-box perspectives is essential to guarantee their quality. We intend to shed light on the complexity and challenges of GCC plugin testing. We propose a four-tiered approach for GCC plugin testing. The approach consists of static and dynamic testing techniques. The main challenge is white box testing. This research aims to address the challenge and provide a solution by utilising logs focusing on higher test coverage. We intend to utilise the findings of the study with other comparable systems.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; **Error handling and recovery**; **Software evolution**.

Keywords

GCC, GCC-Plugin, Compiler, Frameworks, Testing, Black box testing, White box testing, Hybrid testing, Static testing, Dynamic testing, Logs, Analysis, Test coverage, RTL, GIMPLE, AST

ACM Reference Format:
Nimantha Kariyakarawana. 2024. Quality Assurance for Non-trivial Systems: Use Case GCC Plugins. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3650212.3685557

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3685557

1 Introduction & Problem Description

The GNU project developed a compiler collection as the GNU Compiler Collection or as its popular form *GCC*. GCC supports multiple programming languages. However, it is mostly known to be the compiler for C and C++ [12]. Despite being a powerful compiler, GCC has limitations. Such limitations were overcome by modifying the compiler source. It was a difficult task because modifying the compiler source required a thorough understanding of the GCC source. With the introduction of the plugin API with GCC 4.5, third-party developers were able to create custom modules as extensions [5]. The plugin API provided access to GCC internals such as Abstract Syntax Tree (AST), GIMPLE, Register Transfer Language (RTL), and Basic Blocks (BB). This eliminated the need for compiler source modification. GCC plugins are written as separate programs and compiled into a library that the GCC compiler can load during the compilation process [8].

GCC plugins often are part of a bigger framework which intends to perform a certain task [9] [7]. The functionality of such frameworks depends on the output of the GCC plugin. The quality of such frameworks depends on the quality of the GCC Plugins [6] [3]. GCC plugins work closely with the compiler. Therefore, GCC plugins can compromise the compiler. Thus, the plugins should be tested rigorously [2].

De Blaere et al. put forward a GCC plugin that is part of the bigger framework that focuses on data flow error detection. The GCC plugin is the heart of the framework and ensuring it meets quality standards is essential to guranteeing the quality of the framework [4]. Plugins like De Blaere et al. motivate us to investigate systematic testing approaches for GCC plugins.

The current software testing frameworks for C and C++ test the applications on the application run-time. Using such frameworks for testing an application that executes at the compiler run-time is not straightforward. The libraries, header files, and packages utilised in run-time application testing frameworks do not work directly in the GCC plugins. GCC plugins operate in a restricted environment with limited access to external libraries and runtime features. They cannot rely on dynamic memory allocation or complex runtime behaviour. They must follow the rules set by the compiler to avoid unexpected behaviour. GCC plugins interact with GCC internals such as BB, AST, GIMPLE, and RTL. However, it can be argued, given that GCC plugins are written in C or C++, instead of compiling them as a shared object file (.so), compiling them as a regular C or C++ application would allow treating them as regular run-time C++ application. In theory, it is possible. However, GCC plugins execute at different stages of the compilation based on how they are configured. Mocking the behaviour of a GCC compiler is a complex and near-impossible task due to its intricate internals, platform-dependent behaviour, and reliance on macros. The GCC performs various optimisations and interacts with system libraries.

Therefore, creating the necessary data structures or mocking the GCC behaviour required for GCC plugin testing is impossible at run time. Therefore, available practices and frameworks are insufficient to ensure the quality of the GCC plugins. There is an evident gap in the area of GCC plugin testing. We plan to develop a semi-automated testing strategy "GCC-PT4" (code name) based on static and dynamic testing techniques. It consists of a four-step strategy towards GCC plugin testing. Static testing will ensure software specification is followed in the implementation. Dynamic testing will be focused on a black-box testing step, a white-box testing step and a keep-alive testing step. Except for the white-box testing, we plan to utilise existing techniques. Our focus is mainly on developing a white-box testing technique for GCC plugins and utilising the findings for systems that have similar complications with testing.

2 Proposed Approach

Our Aim. is to develop a white box testing technique for GCC plugins. Traditionally, metrics such as line coverage, branch coverage, statement coverage and path coverage are used to understand if the white box testing efforts are effective. Moreover, they help understand areas which require more focus to improve the functionality of the developed application. A single execution cycle in a GCC plugin would not guarantee all the paths are covered. Therefore, to cover all the paths, a source code that triggers all the paths should be compiled with the plugin to achieve proper coverage. Higher coverage is necessary because it serves as an indicator of the robustness and reliability of the software. Furthermore, higher coverage indicates a large amount of the source code has been tested. However, due to the complexities involved with GCC, it is a challenging task.

Case for Logging. Based on our experience so far, logging is our best option towards monitoring the GCC plugin. Logs provide developers with visibility into the inner workings of a software system. With increased logging, we can extract valuable information related to the execution of a GCC plugin at the compiler level. Extracted logging information later can be analysed to extract further coverage metrics. Even though higher coverage does not guarantee bug-free software, it would be an indicator of how much of the software has been tested.

Source that triggers all the paths. Another challenge relies on creating a source that executes all the paths within a GCC plugin. How to create a source or multiple sources that would trigger all the paths remains a challenging task. We plan to utilise applications such as *Csmith* to effectively generate applications that could potentially trigger all the execution paths in a GCC plugin.

Current progress. The authors are currently working on this. The development of the logger for GCC plugins is complete. Existing loggers were unusable due to their focus on logging for run-time applications. Extraction of the logs and analysis towards achieving whitebox testing with coverage as the indicator is currently in the works. There are similar systems in which users do not have direct control over the input. The outcome of this study will be further utilised to enhance the transparency and reliability of such systems.

3 Related Work

Vankeirsbilck et al. provide instructions on how to use a program counter-based technique to develop a regression test. They created a Python script to get the hardware target's program counter and changed eight carefully chosen case studies. This is a black-box testing technique [10]. With randomly generated input data, Vogt et al.'s plugin to eliminate run-time dependence on specific data sets was assessed. By comparing the data produced by the accelerator hardware with the software's computations, the accuracy has been confirmed. They have used manual testing in conjunction with a black-box testing strategy. [11]. Kumar et al. used black-box testing with six micro-kernels to verify their CIT plugin. These micro-kernels included references to dynamically allocated memory, function parameters by reference, and global variables. They validated the CIT system using a customized multinational test application, leveraging the known dependency tree and functionality of the plugin for effective validation. [7]. Aldea et al. validated their study on a Plugin-Based Compiler Pass for Thread-Level Speculation in OpenMP by conducting fifty grouped regression tests. These tests, based on black-box testing methodologies, assessed the plugin's correctness without examining its internal functionality. The tests were designed to ensure the plugin's reliability before real-world implementation [1].

References

[1] Sergio Aldea, Alvaro Estebanez, Diego R Llanos, and Arturo Gonzalez-Escribano. 2014. A new GCC plugin-based compiler pass to add support for thread-level speculation into OpenMP. In *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings 20*. Springer, 234–245. https://doi.org/10.1007/978-3-319-09873-9_20

[2] Junjie Chen and Chenyao Suo. 2022. Boosting compiler testing via compiler optimization exploration. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–33. https://doi.org/10.1145/3508362

[3] Charlie Curtsinger and Emery D Berger. 2013. Stabilizer: Statistically sound performance evaluation. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 219–228. https://doi.org/10.1145/2451116.2451141

[4] Brent De Blaere, Elias Verstappe, Jens Vankeirsbilck, and Jeroen Boydens. 2021. A Compiler Extension to Protect Embedded Systems Against Data Flow Errors. In *2021 XXX International Scientific Conference Electronics (ET)*. IEEE, 1–6. https://doi.org/10.1109/ET52713.2021.9580074

[5] Sandeep Koranne. 2010. *Handbook of open source tools*. Springer Science & Business Media. https://doi.org/10.1007/978-1-4419-7719-9

[6] Rohit Kumar and Ann Gordon-Ross. 2015. An automated high-level design framework for partially reconfigurable FPGAs. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 170–175. https://doi.org/10.1109/IPDPSW.2015.99

[7] Sumeet S Kumar, Anupam Chahar, and Rene van Leuken. 2013. Cit: A gcc plugin for the analysis and characterization of data dependencies in parallel programs. In *Proceedings of the International Conference Design of Circuits and Integrated Systems*. Citeseer, 1–6. https://sps.ewi.tudelft.nl/pubs/Kumar_DCIS_2013.pdf

[8] GCC GNU ORG. 2023. *Plugins building (GNU Compiler Collection (GCC) Internals)*. https://gcc.gnu.org/onlinedocs/gccint/Plugins-building.html

[9] Liang Peng, Chengyong Wu, Jörn Rennecke, Grigori Fursin, et al. 2010. Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation. In *2nd International Workshop on GCC Research Opportunities (GROW'10)*.

[10] Jens Vankeirsbilck, Jonas Van Waes, Hans Hallez, and Jeroen Boydens. 2019. Automated Regression Testing of a GCC Toolchain used on Embedded CPU Programs. In *2019 IEEE XXVIII International Scientific Conference Electronics (ET)*. IEEE, 1–4. https://doi.org/10.1109/ET.2019.8878623

[11] Markus Vogt, Gerald Hempel, Jeronimo Castrillon, and Christian Hochberger. 2015. GCC-plugin for automated accelerator generation and integration on hybrid FPGA-SoCs. *arXiv preprint arXiv:1509.00025* (2015). https://doi.org/10.48550/arXiv.1509.00025

[12] William Von Hagen. 2011. *The definitive guide to GCC*. Apress. https://doi.org/10.1007/978-1-4302-0219-6

Received 2024-07-07; accepted 2024-07-22