Program Input Static Filtering

CSC 766 Spring 2023 (001)

Nathan Vercaemert
CSC
NCSU
Raleigh, NC, USA
nhvercae@ncsu.edu

ABSTRACT

In this study, we present an automated method to determine which parts of a C program's input influence its execution path. By analyzing def-use relations and focusing on key points such as branching points and function pointers, our method identifies the portions of input that are significant for a program's runtime. We implement this technique using the Clang C library. The code processes example programs and extracts variable definitions and uses, enabling the system to understand how input variables affect execution paths. The results show that our approach successfully identifies the relevant input data for various examples, including those involving I/O APIs. This method has potential applications in runtime optimization and adaptation, as it can help predict a program's behavior based on specific input values. Limitations of this study include the current scope of supported I/O APIs. Future work could expand the range of APIs and further automate the process of identifying key points in more complex programs.

CCS CONCEPTS AND KEYWORDS

Compilers, Optimization, Static Filtering, I/O

ACM Reference:

Nathan Vercaemert. 2023. Student. CSC 766 Course Project. *NCSU, Raleigh, NC, UlSA, 3 pages.*

1 Motivation

Predicting the runtime behavior of a program is crucial for optimizing and adapting various aspects of its execution. Inputs to a program determine its behavior on a given machine, but in some cases, only specific parts of the input affect the program's execution path. This project aims to use compiler techniques to identify the parts of the input that determine the execution path of a C program, facilitating better runtime optimizations. By analyzing def-use relations and considering the semantics of I/O and other APIs, we can efficiently identify the input values responsible for the program's runtime behavior. The project focuses on C programs using a specific set of I/O APIs, with the goal of generalizing the approach to other languages and APIs in the future. The motivation behind this project is to enable developers to optimize program performance by understanding the impact of input values on the execution path, leading to more efficient and adaptive software.

2 Challenges

The primary challenge of this project was to accurately identify the input variables that determine the execution path of C programs, which is essential for runtime optimizations and adaptations. Analyzing the def-use relations of the input variables proved to be challenging for certain cases, especially when considering I/O APIs and their semantics. For example, a naive def-use analysis might wrongly conclude that every character in an input string affects the runtime of the program.

The implementation of the project using the Clang C library presented its own set of difficulties. Parsing and traversing the abstract syntax tree (AST) required an in-depth understanding of Clang's API, which

involved handling various cursor types and implementing custom visitor functions. Furthermore, dealing with nested cursors and evaluating expressions required additional recursive functions and logic.

To identify the key points in the code that determined the execution path, manual annotations or special prefixes to the variables were used. Handling more complicated cases, such as the second example in the assignment, required leveraging the semantics of the I/O APIs, adding another layer of complexity to the project.

Lastly, ensuring the project's usability for others required the code to be well-documented, easy to read, and understandable, which necessitated the inclusion of clear comments and structured code organization.

3 Solutions

In the given assignment, we were tasked with identifying the parts of a program's input that determine its execution path. We focused on key points such as branching points and function pointers. To achieve this, we implemented a solution using the Clang library to parse and analyze C code.

The main approach was to traverse the Abstract Syntax Tree (AST) of the program and identify relevant constructs such as binary operators, while and for loops. We utilized Clang's cursor traversal to locate key points in the code. For instance, we detected function calls to 'fgetc' and 'getc' to identify I/O operations. Additionally, we looked for integer literals with a value of 1 (EOF) to infer the use of input in determining the execution path.

To store the information related to variables, their definitions, and their uses, we used a map with variable names as keys and pairs of strings as values. We also employed helper functions such as 'trim', 'get_surrounding_context', and 'print_results' to format and present the final output.

Our solution successfully handles simple C programs and can detect key points in determining the execution path, including I/O operations and branching points. We also provided functionality to work with the SPEC CPU 2017 benchmark suite, allowing for partial marking of key points in more complex programs. This approach enables efficient identification of input variables that determine the execution path of a given C program.

4 Lessons and Experiences

During the project, I gained valuable experience working with Clang's CX Cursor to analyze the control flow and def-use relationships in C code. By implementing a program that identifies the input variables determining the execution path, I developed a deeper understanding of compiler internals and how to leverage compiler APIs. While analyzing the programs, I learned about the def-use relation and its implications in determining runtime behavior, which is essential for runtime optimizations. The challenge of handling complex cases like the second example program taught me the importance of considering the semantics of I/O APIs in my analysis. Working with the SPEC CPU 2017 benchmark suite for final evaluation allowed me to apply my learnings to real-world scenarios, further solidifying my understanding of the techniques involved. In summary, this project provided me with invaluable insights into compiler-based program analysis and runtime behavior prediction, which will undoubtedly prove beneficial in my future endeavors in the field of programming languages and compiler optimizations.

5 Results

In this project, we aimed to analyze C programs to identify parts of the input that determine the execution path. We implemented a tool that uses Clang's libclang library to parse and traverse the AST of the input C code. The tool identifies variables related to file input and their definitions, as well as their use in relation to EOF. Our analysis focused on variables that interact with 'fopen', 'fopen_utf8', 'getc', and 'fgetc' functions. We tested the tool on two example

files: cpp_files.c and convert.c. In the case of cpp_files.c, the tool identified the variable 'f' and its definition and uses. The definition was 'f = fopen (name, "r");', and the uses were in three different 'while' loops. For convert.c, the tool identified the variable 'input', its definition 'input=fopen_utf8(argv[i],"rb");', and its use 'for (c=fgetc(input); c != EOF; c=fgetc(input))'. This information can be helpful in understanding how the program's execution path is affected by file input operations, enabling better optimization and adaptation. It is important to note that more complex cases may require deeper semantic analysis or leveraging the MLIR compiler for greater accuracy.

REFERENCES

[1] LLVM Project. "Clang 14 Documentation." LLVM, LLVM Project, 2021, clang.llvm.org/docs/index.html.