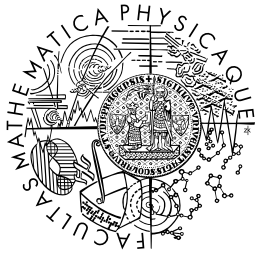

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Denis Leskovar

Automated Program Minimization With Preserving of Runtime Errors

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Pavel Parízek, Ph.D.
Study programme: Computer Science
Study branch: System Programming

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication.

Title: Automated Program Minimization With Preserving of Runtime Errors

Author: Denis Leskovar

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Debugging of large programs is a difficult and time consuming task. Given a runtime error, the developer must first reproduce it. He then has to find the cause of the error and create a bugfix. This process can be made significantly more efficient by reducing the amount of code the developer has to look into. This paper introduces three different methodologies of automatically reducing a given program P into its minimal runnable subset P'. The automatically generated program P' also has to result in the same runtime error as P. The main focus of the reduction is on correctness when operating in a concrete application domain set by this study. Implementations of introduced methodologies written using the LLVM compiler infrastructure are then compared and classified. Performance is measured based on the statement count of the newly generated program and the speed at which the minimal variant was generated. Moreover, the limits of the three different approaches are investigated with respect to the general application domain. The paper concludes with an overview of the most general and efficient methodology.

Keywords: automated debugging, code analysis, syntax tree, statement reduction, clang

Contents

Introduction

TODO: Rewrite the introduction to include goals, fix math fonts.

Automation of routine tasks tied with software development has resulted in a tremendous increase in the productivity of software engineers.

However, the task of debugging a program remains mostly manual chore. This is due to the difficulty of reliably encountering logic-based runtime errors in the code, a task that, to this day, requires the developer's attention and supervision.

Let program P contain a runtime error E that consistently occurs when P is run with arguments A. Since the error E is present at runtime and not compile-time, it can be assumed that syntax wise the code is mostly correct. Therefore, any syntax-based error can be ruled out. This, in turn, leaves us with a set of logical errors $\mathbf{E_L}$. Those include wrongly indexed arrays and calculations that lead to either the incorrect result or an altered control flow of the program. Let $E \in \mathbf{E_L}$. As the generality of errors in $\mathbf{E_L}$ appears too complicated to be solved for all programming languages at once, it is necessary to break the problem down for each programming language. This article is concerned with the logical errors of C and C++. Although C is not a subset of C++, the logical errors made in C can be approached similarly to those in C++. Both languages share mostly comparable constructs. Finding the cause of a logical error in a concrete language requires knowing these constructs, their behavior, and their general handling.

To make finding the cause of an error a systematic approach, one might try removing unnecessary statements in the code, thus minimizing the program. Let P' be a minimal variant of P such that P' results in the same error E as P when run with the same arguments A. If done carefully and correctly, P' represents the smallest subset of P regarding code size, while preserving the cause of the error in that subset. Upon manual inspection, the developer is required to make less of an effort to find the cause in P' as opposed to P.

The minimization of a program can be achieved in numerous ways. In further sections, the article describes and compares three different approaches. The first is based on naive statement removal and its consequences during runtime. The second removes major chunks of the code while periodically testing the generated program's correctness. The third deploys a sequence of code altering techniques, namely slicing and delta debugging.

1. Automated debugging techniques

TODO: Link relevant literature from Slicing of LLVM bitcode (muni.cz) and Bobox Runtime Optimization (cuni.cz)

Debugging can be described as the process of analyzing erroneous code to find the cause of those errors. While most developers see debugging as a manual chore, there were numerous attempts at automating at least some parts of it during the last few decades. The rise in popularity of program analysis resulted in automated error checks for popular programming languages.

While these checks mostly cover only specific cases of potential bugs, such as out-of-range array indexing, they have proven themselves as a useful tool for the developer. In the context of this work, such checks provide a helping hand at a low cost when minimizing a program.

The following sub-chapters will talk about the techniques behind such checks and how they deal with automated debugging.

1.1 Delta debugging

Delta debugging is an iterative approach described by Zeller in 1999. It does not perform any static analysis of the debugged program, as it is not meant to find failures in the code.

Delta debugging instead intends to minimize the debugged program's incorrect input to isolate the input's failure-inducing part. Therefore, it requires the program in question and the specific input and the expected output. In other words, Delta debugging requires a set of test cases, which attempts to minimize and isolate the failure-inducing input. Minimality is defined as follows.

Definition 1 (Test case)**.** *Let $c_{\mathcal{F}}$ be a set of all changes $\delta_1, \ldots, \delta_n$ between a passing program run $r_{\mathcal{P}}$ and a failing program run $r_{\mathcal{F}}$ such that*

$$r_{\mathcal{F}} = (\delta_1(\delta_2(\ldots(\delta_n(r_{\mathcal{P}}))))).$$

We call a subset $c \subseteq c_{\mathcal{F}}$ a test case*.*

Definition 2 (Global minimum)**.** *A test case $c \subset c_{\mathcal{F}}$ is called a* global minimum *of $c_{\mathcal{F}}$ if $\forall c_i \subseteq c_{\mathcal{F}} : (|c_i| < |c| \implies c_i$ does not cause the program to fail.)*

Global minimum can be interpreted as the smallest set of changes able to make the program fail.

Definition 3 (Local minimum)**.** *A test case $c \subset c_{\mathcal{F}}$ is called a* local minimum *of $c_{\mathcal{F}}$ if $\forall c_i \subseteq c : (c_i$ does not cause the program to fail.)*

Definition 4 (n-minimality)**.** *A test case $c \subset c_{\mathcal{F}}$ is* n-minimal *if $\forall c_i \subseteq c : (|c| - |c_i| \leq n \implies c_i$ does not cause the program to fail.)*

The minimizing Delta debugging algorithm attempts to find a 1-minimal test case.

Delta debugging seems to bet on the premise that large-scale applications are written with automated testing in mind. On the same note, it is the recommended practice to develop programs while at the same time dedicating resources to write tests for that program.

The defined minimality can be used to construct the minimizing algorithm. However, the delta debugging algorithm can be easily and more comprehensively explained without the definition as well.

Algorithm 1 Minimizing Delta Debugging Algorithm

1: $n \leftarrow 2$
2: Split a string S into $\alpha_1, \ldots, \alpha_n$ of equal size.
3: For each α_i, calculate its complement β_i.
4: Run tests on $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$.
5: **if** all tests passed **then**
6: $\quad n \leftarrow 2 * n$
7: $\quad$ **if** $n > |\sigma|$ **then return** the most recent failure causing substring.
8: $\quad$ **else**
9: $\quad\quad$ goto (2).
10: $\quad$ **end if**
11: **else if** $\alpha_i failed$ **then**
12: $\quad n \leftarrow 2$.
13: $\quad \sigma \leftarrow \alpha_i$.
14: $\quad$ **if** $|\sigma| == 1$ **then return** σ.
15: $\quad$ **else**
16: $\quad\quad$ goto (2).
17: $\quad$ **end if**
18: **else** ▷ β_i failed.
19: $\quad \sigma \leftarrow \beta_i$.
20: $\quad n \leftarrow n - 1$.
21: $\quad$ goto (2).
22: **end if**

Additionally, minimizing is not the only approach Delta debugging suggests. A more sophisticated one is isolation. Minimization can be described as removing parts while the failure persists, which means that the output changes are only made in failing iterations. Isolation extends this by adding failure-inducing differences while the program passes tests. This addition results in changes in both the passing and failing iterations.

One can quickly transform the input minimalization of Delta debugging into either source code minimalization or error isolation at both the compile-time and runtime. This transformation can be achieved for the compile-time by first setting the input as the debugged program's source code. Second, it is required to set the expected output to either 'compiled' or 'failed to compile'. Finally, the input is fed into a compiler, for example, GCC, which produces one of the two set outputs.

The runtime variant only differs in two points—first, changing the expected outputs. Second, changing the compiler to a compiler-debugger pipeline so that the source can be compiled and run.

1.2 Static slicing

The first introduced slicing method was static backward slicing. In 1984, Weiser defined a slice with respect to criterion C as a part of a program that potentially affects given variables in a given point.

Definition 5 (Static slicing criterion)**.** *Let $\mathcal{P}$ be a program consisting of program points $P = p_1, \ldots, p_n$ and variables $V = v_1, \ldots, v_m$. Any pair $C = (p_i, V')$, such that $p_i \in P$, $V' \subseteq V$, and $\forall v_i \in V' : v_i$ is present in p_i, is called a* slicing criterion*.*

Slicing is the process of finding such a part of a program. Suggested approaches neglected any execution information and focused solely on observations made by analyzing the code.

TODO: Convert the pseudocode to an easily readable version (i.e. comparison with the non-sliced program).

Algorithm 2 Simple Branching Program

```
1:  x ← 1
2:  a ← read(a)
3:  for i = 1, 2, ..., C do
4:      write(i)
5:  end for
6:  if a mod 2 == 0 then
7:      if a ≠ 0 then
8:          x ← −1 ∗ x
9:      else
10:         x ← 0
11:     end if
12: else
13:     x ← x + 1.
14: end if
15: write(x)
```

Algorithm 3 Static Slice of the Simple Branching Program

```
1:  x ← 1
2:  a ← read(a)
3:  if a mod 2 == 0 then
4:      if a ≠ 0 then
5:          x ← −1 ∗ x
6:      else
7:          x ← 0
8:      end if
9:  else
10:     x ← x + 1.
11: end if
12: write(x)
```

Later that year, Ottenstein and Ottenstein restated the problem as a reachability search in the program dependence graph (PDG). PDG represents statements in the code as vertices and data and control dependencies as oriented edges.

Additionally, edges induce a partial ordering on the vertices. In order to preserve the semantics of the program, statements must be executed according to this ordering.

Edges are, therefore, of two types. First, the control dependency edge specifies that an incoming vertex's execution depends on the outgoing one's execution. Second, the data flow dependence edge suggests that a variable appearing in both the outgoing and incoming edge share a variable, the value of which depends on the order of the vertices execution.

Once the PDG is built, slices can be extracted in linear time with respect to the number of vertices.

TODO: Show how PDG is sliced, from 2.2 Slicing of LLVM bitcode (muni.cz)

However, one can find many potential issues and obstacles when performing data flow analysis. Omitting the interprocedural slicing, as it is not relevant in this paper's context, one is left with pointers and unstructured control flow. While the latter is seldomly used in single-threaded modern programming, the same cannot be said about the former.

Pointers require us to extend the syntactic data flow analysis into a pointer or points-to analysis, which should be performed first. It is necessary to keep track of where pointers may point to (or must point to, in case their address is not reassigned) during the execution. From this knowledge, other data flow edges must be created or changed to accommodate the fact when the outgoing vertex mayhap writes into a memory location possibly used by the incoming vertex.

The analogical approach is then used for control dependency analysis since pointers might alter control flow as well. This change to control flow happens, namely when functions are called using function pointers.

The main advantage of static slicing is that it does not require any run-time information. As program execution can be expensive both time-wise and resource-wise, static slicing offers program comprehension at a low cost. Because static slicing discovers program statements that can affect certain variables, it can remove dead code and be used for program segmentation.

Furthermore, static slicing is used for testing software quality, maintenance, and test, all of which are relevant to this project.

1.3 Dynamic slicing

While the idea of building a program slice prevails, dynamic slicing drastically differs from static slicing in terms of input and the way it is processed.

In 1988, Korel and Laski described a slicing approach that took into consideration information regarding a program's concrete execution. As opposed to static slicing, which builds a slice for any execution, dynamic slicing builds a slice for a given execution of a program. Using information available during a run of the program results in a typically much smaller slice.

TODO: Convert the pseudocode to an easily readable version (i.e. comparison with the non-sliced program).

This decrease in size is mainly due to removing unnecessary branching of control statements and unexecuted statements in general. The slicing criterion now contains a set of the program's arguments in addition to the previous information. The location of the criterion's statement is also specified to avoid vagueness in

Algorithm 4 Dynamic Slice of the Simple Branching Program (for $a = 2$)

1: $x \leftarrow 1$
2: $a \leftarrow \text{read}(a)$
3: $x \leftarrow 0$
4: $\text{write}(x)$

the execution history.

The criterion is therefore defined as follows.

Definition 6 (Dynamic slicing criterion)**.** *Let $\mathcal{H} = (s_{x1}, \ldots, s_{xn})$ be an execution history of a program $\mathcal{P} = (\{s_1, \ldots, s_m\}, V)$, where s_i denotes a statement and V is a set of variables $v_1, \ldots, v_k$. Any triple $C = (h_i, V', \{a_1, \ldots, a_j\})$, such that $h_i \in \mathcal{H}$, $V' \subseteq V$, $\forall v_i \in V' : v_i$ is present in h_i, and $\{a_1, \ldots, a_j\}$ is the input of the program, is called a* slicing criterion*.*

Since dynamic slicing requires the user to run the program, it is typically used in cases where the execution with a fixed input happens regardless. Such cases include debugging and testing. For debugging, dynamic slices must reflect the subsequent restriction: a program and its slices must follow the same execution paths.

1.4 Summary

While the described program minimizing and debugging approaches have been formulated more than two decades ago, there have not been nearly enough successful attempts at implementing them.

With each approach having its clear positives and negatives, it would be interesting to see how they handle program minimization. When cleverly used, a combination of these methods might result in a reasonably fast and inexpensive algorithm for the reduction of program size.

2. Compilers and analysis tools

In the previous chapter, the reader was introduced to a branch of program analysis. The techniques discussed above focused on both the static and runtime side of program analysis.

Regardless of whether these approaches have been implemented, it was required to find a suitable tool for source code manipulation for two reasons. First, any external tool output might require altering the input source code based on its output. Second, if implementing any code reducing algorithm would have to occur, one would need a sophisticated code modifying framework.

Due to these reasons, an analysis of compilers and tools for C and C++ was conducted. The goal of the analysis is to pick the most practical tool available. Required criteria include frequent upkeep of the framework, an existing user base, and the ability to manipulate some abstract representation of the code.

The representation boiled down to an abstract syntax tree (AST). AST embodies the syntactic structure of the code, regardless of the code's language. A vertex of an AST represents a construct of the code while not being concrete with the programming language's details. This generality is perfect for C and C++'s chosen domain, as both languages only differ syntax-wise in minor details.

TODO: Add more text about AST.

Below are the findings concerning the most important candidates.

2.1 GCC

A well known C and C++ compiler, the GNU Compiler Collection is an extensive open source project. As popular as GCC is, it does not provide the features an analysis-tool-building developer needs.

For the sake of building such tools, a compiler front end is used. Due to an old design, it is difficult to work with either the front end or the back end of GCC alone. Besides, the compiler implicitly makes optimizations that destroy any parallels between the source code and the AST. Therefore, the AST has to be treated as an entirely different object rather than an abstraction of the code. Most of the compiler's source code representation is unintuitive and hard to pick up for anyone not actively contributing to GCC.

As far as AST manipulation is concerned, the compiler allows the user to dump the structure into a text representation. However, due to the difficulties mentioned above, it can hardly be used.

TODO: Add GCC AST text dump.

These issues result in a seldom-used variant that offers nearly no developer-friendly features. An upside is that GCC allows the user to visualize the AST. However, that is hardly a useful feature in the context of this paper.

2.2 Clang

Thanks to LLVM, the widespread compiler infrastructure, the Clang project has provided a compiler front end not only for C and C++ but also for CUDA,

OpenCL, and other languages. The extend of Clang as a compiler front end is so vast that it covers both the C++ standard and the unofficial GNU++ dialect.

The project does not include just the front end but also a static analyzer and several code analysis tools, which are now commonly used in IDE's as syntax and semantic checks.

This description of Clang foreshadows its friendliness to analysis tool developers. The fact that the front end runs on a common intermediate language also indicates that openly working with abstract code representations is supported.

TODO: Add more general text based on what I write in the Clang chapter.

2.3 Summary

While the chapter only highlighted two significant candidates, the analysis looked at a plethora of tools. Those, however, were not able to compete feature-wise due to the sheer size and extent of GCC and Clang.

It would seem that parsing multiple programming languages into an abstract representation requires a common intermediate language, in which the representation is stored. Having an intermediate language is not always possible for several reasons, including licensing and old architecture. The compiler giant GCC seems to suffer from precisely that. Additionally, since the Clang project is being contributed to regularly, resulting in as many as five releases per year, it pulls in a more significant developer community.

Therefore, Clang is the favorite source code altering tool for this project. In the following chapter, the relevant parts of the Clang project will be broken down and explained.

3. Title of the second chapter

An example citation: Anděl [2007]

TODO: Remove the mock citation.

3.1 Title of the first subchapter of the second chapter

3.2 Title of the second subchapter of the second chapter

Conclusion

Bibliography

J. Anděl. *Základy matematické statistiky.* Druhé opravené vydání. Matfyzpress, Praha, 2007. ISBN 80-7378-001-1.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment