An optimized tool for language independent program slicing

SE 801

Submitted By
Fazle Mohammed Tawsif
BSSE 0628

Supervised By
Dr. Kazi Muheymin-Us-Sakib
Professor
Institute of Information Technology
University of Dhaka

Institute of Information Technology
University of Dhaka
Bangladesh
29 October 2017

Approved By

Contents

List of Figures

Chapter 1

Methodology

In this chapter, an approach is proposed that can minimize the compilation error of the language independent program slicing process. It is mentioned in the previous chapter that existing approach can slice programs. Moreover, it tries to compile and execute the sliced programs as soon as it deletes a line from the source code. But it is not the ideal scenario of each time because, in the case of large programs, the amount of compilation error gets too high. Compilation without any type of checking cause this high compilation rate as well as high slicing time. It is probable that execution error can be occurred due to syntactic or semantic error. It is assumed that few statements which are common among the programming languages such as conditional syntax can be omitted from compilation after some checking. In order to overcome the above limitation of the existing approach, an approach namely MLIPS is proposed and developed that can slice programs with less compilation error as well as in less time than previous approach ORBS. The proposed approach is exhaustively described in the following section of this chapter.

1.1 Overview of proposed slicing technique

The internal architecture of MLIPS is shown in figure1.1 The architecture contains six small components where each component performs predefined responsibilities. The information flow and the activity of each component are described below.

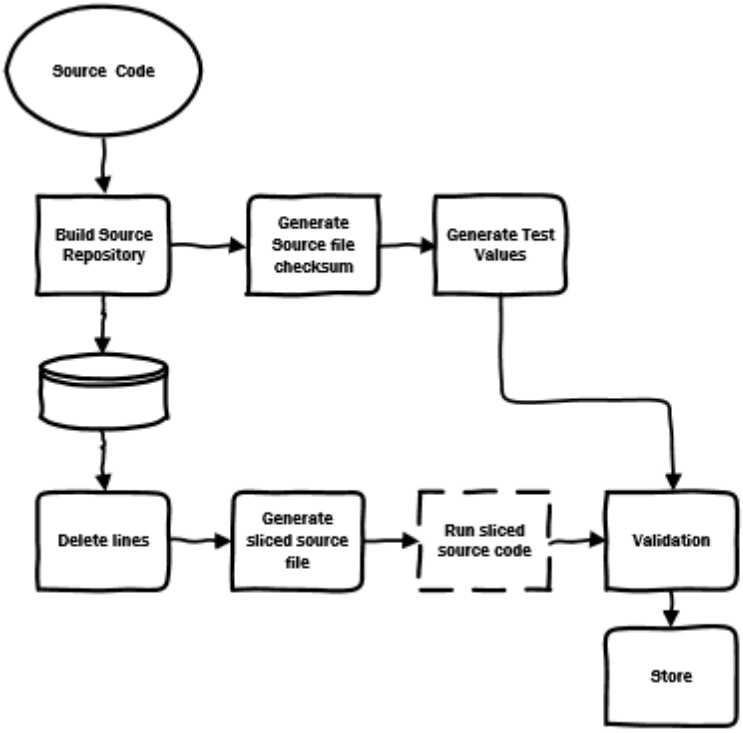

Figure 1.1: Dependence graph of a program

1.1.1 Build Repository

In MLIPS, a list of source code is provided as input which contains program written in different language. From these source codes, a source repository is built with available information such as line number, code contents, file names etc. The repository also maps the source code lines with file names and other counts such as line counts, file counts etc. This repository will be used for parsing and slicing. After slicing the repository codes, it maps the sliced programs with source code files which contains all the valid slices of each source code file.

1.1.2 File checksum

File checksum component is used for file validation. Sliced source file is mapped with a checksum for reuse. It generates a checksum for each file. This checksum is mappad against a hashcode. It is used for identifying same executable bytecode of sliced programs. If any checksum of executable bytecode is matched with the previous

checksum, then the value of previous executable codes are used. Using the previous executable output is considered as cached compilation output. Checksum also helps to identify any changes in the source file by comparing the previous checksum value.

1.1.3 Test Value Generator

Test value generator generates test value output for comparing the output of sliced programs output. This test value is used by validation state. Test value generator uses the initial or base source codes/programs. It compiles and executes these programs to get the test value and then stored for later use.

1.1.4 Line deletion/slicing

Line deletion is the core procedure of language independent program slicing. MLIPS uses this component to delete each line step by step. Line deletion window is incremented using a step function. The value of this step function is incremented after each successful iteration of all line. The maximum value of step function is defined initially. This deletion process is the modified version of the previous implementation of program slicing ORBS. A conditional checking is introduced in this modification to minimize the overall compilation error. It checks each line before delete. And then omits lines which contain conditional statements. This checking step is mentioned previously in the Introduction section of this chapter. Conditional statements such as if, else if, else etc. can be transformed into a generalized syntax after some manipulation. The statements are transformed into generalized form while checking. If the line does not contain any conditional statement, then the line is deleted. After deletion of the line, this source code is compiled and executed. If the compilation and execution are completed without any error, then the output is compared with the test output. The sliced code is stored for later use after a successful comparison of current output and test output. A list of all lines of the source codes is maintained to keep track of deleted lines. This list is hashed after each deletion then mapped with the binary file checksum to use them later as a cached compilation. This hash-checksum mapping is considered as slice cache. On the other hand, the output from the execution of binary file is mapped against each checksum. This output-checksum mapping is considered as result cache. The slice cache and the result cache are used later for cache compilation where deleted lines list hash is same with some previously hashed list. Deletion of line completed after traversing all the existing line. It also excludes lines which are commented. Commented lines are identified by analyzing

general syntaxes for comment in programming languages.

1.1.5 File generation and Execution

The source code of sliced programs are generated. File generation process uses the list of deleted lines which is mentioned in the previous component. It eliminates the lines which are marked as deleted. Then generate source codes with the available lines from the list in a target directory. Execution process is done in two steps. The first step is compilation and the second step is execution. On the first step, generated source files are compiled with the system compiler according to their programming language. The compiler generates some executable bytecode if the compilation is successful or move to next step of deletion. Executable bytecodes are used for generating a checksum. These checksums are used for mapping the output with the executables. This mapping is later considered as cache compilation. Later on the second step, executable files are executed according to their compiler syntax. The output of this execution is stored in the result mapping against the checksum executable bytecodes. In this step, the output state is not considered whether it is error or correct. The output value is stored for later validation.

1.1.6 Validation

Validation process store the valid slices. It compares the sliced program output with the test value. If the sliced program output matched with test value, then that sliced program is stored.

It is seen that the proposed technique comprises five procedures and one algorithm. The five procedures namely buildSourceRepository, generateSlicedFile, compileFile and executeFile, getTestOutput, validate are used to get the required source files, generate sliced flies, running source code and generating test output for validation. The only algorithm namely SliceSource is used to delete source code which is used as a sliced program. Finally, this sliceSource algorithm iterates over the base source code with different deletion window and generate various slice. Valid program slices are preserved.

1.2 Build source repository

As mentioned earlier, program slicing requires source codes of various languages. For this purpose, a source repository is built. Procedure 1 states the repository building process.

Procedure 1: Build source repository

Input: Source code root path

Output: Source repository containing information of valid source codes

```
1  begin
2      repository ⟵ φ;
3      foreach file f ∈ rootfile do
4          if f.type ∈ supported type then
5              fileContent.name ← f.name ;
6              fileContent.lines ← f.lines ;
7              fileContent.lineCount ← f.lineCount ;
8              fileContent.directory ← f.directory ;
9              fileContent.type ← f.type ;
10             repository ⟵ repository ∪ fileContent;
11         else
12             continue;
13         end
14     end
15 end
```

A root path of the source codes is provided as input in procedure 1. It takes all the supported programs in the repository. Each entry of the repository contains file name, content, directory path and line count. Line:4, procedure 1 sort out the supported source files from the directory. Line: 10 includes each supported file entry in the repository.

1.3 Sliced source generation

After each successful slicing, source files are generated from a list of lines where deleted lines are marked. Files are generated excluding those deleted lines. Procedure 2 exhibits the process of file generation.

Procedure 2: Generate sliced source file

Input: A list of deleted lines from initial source code

Output: Sliced source codes in a target directory

```
1  begin
2      Lines;
3      DeletedLines;
4      for j ← 1 to length(Lines) do
5          FileName ← Lines [j].fileName ;
6          content ← Lines [j].content ;
7          if DeletedLines [j] == false then
8              append(FileName, content) ;
9          end
10     end
11 end
```

Procedure 2 takes a list of lines and list of deleted lines as input. Generates source files with the sliced codes as output. Line 5, 6 take name and line content from the list. If the line is not deleted then it is appended to the file which occurs in Line 8. This procedure is used every time a line is deleted. Source files are generated for further compilation and execution.

1.4 Compilation and Execution

Procedure 3 illustrates the compilation process. Source code compilation is used in two steps in slicing process. Initially, the compilation process is used in test value generation, later this process is used after a line deleted. After successful compilation, bytecode is generated for execution. If any compilation fails it returns false to the slicing process.

Procedure 3: Compile source file

Input: A list of source code

Output: Generate executable bytecode

Result: Return true if successful otherwise false if error

```
1  begin
2      Files;
3      foreach file f ∈ Files do
4          Compiler ← GetCompiler(f.type) ;
5          Error ← Compiler.compile(f) ;
6          if Error == false then
7              return true ;
8          else
9              return false ;
10         end
11     end
12 end
```

Procedure 3 takes source code as input and produce executable bytecode as output. On the Line: 4, appropriate compiler is obtained using file extension type. Line: 5, compiles the file and preserve the output. If the compilation is successful then it returns true otherwise false. Beside compilation process, execution process also follows the same procedure 3. The main difference from compilation process is, it takes executable bytecodes as input and generate a value after execution as output.

1.5 Test value generation

The test value is used for validation of the sliced program. Test generation is completed in the very first stage of slicing process. Test generation is completed in two steps. The first step is used for source file compilation and second step for execution.

Procedure 4: Generate Test Output

Input: A list of initial source code

Output: Value or list of values of the executed source codes

```
1  begin
2      Files;
3      Output;
4      ExecutableFiles;
5      foreach file f ∈ Files do
6          Error ← compile(f) ;
7          if Error == true then
8              return FAIL ;
9          else
10             ExecutableFiles ← ExecutableFiles ∪ getExecutableExtention(f) ;
11         end
12     end
13     foreach file exf ∈ ExecutableFiles do
14         Error ← exf ;
15         if Error == true then
16             Output ← append(Output, FAIL) ;
17         else
18             Output ← append(Output, OutputValue) ;
19         end
20     end
21     return Output;
22 end
```

Test generation process takes initial source codes as input. Line 5 to 11 in procedure 4 is for compilation and line: 13 to 19 for execution. The output of this step is stored which is stored for later validation process.

1.6 Slice validation

Procedure 5: Validate sliced programs

Input: Output value of sliced source code and Initial code outout value

Output: Returns true if validation successful, otherwise false

Result: True indicate that slice need to be stored otherwise eliminate the sliced program

```
1  begin
2      TestValue;
3      SlicedOutput;
4      isSame ← check(SlicedOutput, TestValue) ;
5      if isSame == true then
6          store() ;
7          return true ;
8      else
9          return false ;
10     end
11 end
```

Algorithm 6: Validate sliced programs

Input: Source program, $P = p_1, ..., p_n$ and maximum deletion window size, δ

Output: slice, S , of P

```
1  begin
2      O ← Setup(P)
3      V ← Execute(Build(P))
4      S ← O
5      repeat
6          deleted ← False i ← 1 while  i ≤ length(S) do
7              builds ← False
8              for i ← 1 to δ do
9                  if ¬ContainsConditional(s_m in(length(S), i + j − 1)) then
10                     S′ ← S - s_i, ...., s_m in(length(S), i + j − 1)
11                 else
12                     continue
13                 end
14                 B′ ← Build(S′) if B' built successfully then
15                     builds ← True break
16                 end
17             end
18             if builds then
19                 V′ ← Execute(B') if  V = V' then
20                     S ← S′ deleted ← True
21                 end
22             else
23                 i ← i + 1
24             end
25         end
26     until ¬ deleted;
27     return S
28 end
```

Bibliography