

The Intelligent C Language Debugger

Ming Wang, M.S.
Robert Chun, Ph.D.

Computer Science Department
San Jose State University
San Jose, California 95192
mwang@redback.com

ABSTRACT

Most current programming debuggers do not have any built-in intelligence to help programmers perform debugging. When using these non-intelligent debuggers, programmers have to debug their programs statement by statement in order to track down the bugs. There are several disadvantages with the current non-intelligent debuggers. The first disadvantage is that the debugger does not know the variable dependencies of the program being debugged. It does not understand the cause-effect semantics of the program, and how a single error in one line of code can lead to a propagation of errors in later lines of code. The second disadvantage is that the debugger cannot simplify the program being debugged. If programmers are only concerned with some particular variables when debugging the code, the debugger should have the ability to filter out the statements that have no relevancy (i.e., no dependency) with those variables. The third disadvantage is that the debugger cannot help identify the statements which would have a higher probability of causing certain program errors. If the debugger allowed programmers to provide some information regarding what is wrong (and what is right) with the program, it should be able to utilize that information and help programmers narrow down the bugs.

This paper proposes an intelligent debugger which can help programmers to narrow down the bugs in a program. Using test result information provided to it, the intelligent debugger applies an enhanced static slicing algorithm to display a subset of the program containing only the statements that have a variable dependency related to the error condition(s). After slicing, the resulting program becomes more readable and understandable to the user, aiding the debugging process. Also, a variable analyzer is able to compute correctness ratio numbers to assist the user in identifying relatively weak areas of code which have the highest probability of causing certain program errors. Users can then focus on and double check those areas of code to determine if any of them are indeed causing the bugs in the program.

KEYWORDS

Debugger, Program Slicing, Variable Analyzer, Parser, PHP

1. INTRODUCTION

For a complicated program, programmers cannot easily find problems from using a debugger. Most current debuggers are not able to help programmers to simplify the program being debugged based on variable dependency [1,2,3]. Program slicing is a technique that can find variable dependencies for a program. Figure 1 shows the basic concept of program slicing. In this example, the programmer defines five integer variables which are a , b , c , x and y . If he or she is only concerned with the result for variable c (e.g., because it is the only one that is erroneous), the algorithm can slice out the statements: $x=20$, $y=a-x$, and $y=a+x$.

Because these statements do not affect variable c , we can remove them from the program. For example, knowing that only certain variables produce incorrect results on execution, the user can instruct the slicer to eliminate the statements that have no dependency with those specific variables. There are several benefits of using program slicing. The size of the program being debugged can be decreased significantly making it more readable and comprehensible to the programmer [4]. The intelligent C language debugger uses program slicing as the central technique to determine variable dependencies for the given program.

<pre>main(){ int a, b, c, x, y; scanf("%d",&a); scanf("%d",&b); x = 20; if(a > b){ c = a + b; y = a - x; }else{ c = a - b; y = a + x; } printf("c = %d",c); }</pre>	<pre>main(){ int a, b, c; scanf("%d",&a); scanf("%d",&b); if(a > b){ c = a + b; }else{ c = a - b; } printf("c = %d",c); }</pre>
---	--

Figure 1: Example for program slicing

Let the sequence of statements that are executed in the computation of a particular output variable be called the program execution path for that variable. If the programmer has run some tests on the program and ascertained which output variables are computed correctly and which ones are computed incorrectly, then that information can be of use in debugging the code. For any given correct output variable, the program execution path for this variable should be correct. For any given incorrect output variable, the program execution path for this variable should have some problems. The problem(s) can occur at any place along the path to make the result incorrect [5]. Applying these two assumptions on the information provided by the user, the intelligent C language debugger can predict the statements that have a higher probability of causing program errors.

The debugger uses two basic concepts. The first one involves computing the variable dependency between correct variables and incorrect ones. Since the slicer knows the program execution path for all variables in the program, it can use this information to eliminate the correct statements that are on the program execution path for incorrect variables. The second concept is to trace back all possible paths for the remaining variables. Because the program might have conditional blocks, the slicer cannot just simply trace a single path; it needs to trace all possible paths to get a correct prediction.

2. RELATED WORK

Slicing, proposed by Mark Weiser in 1981, is a technique that can determine variable dependencies between different statements in a program [6,7,8]. The goal of program slicing is to simplify a complicated program to a less complicated one based on the variables that the programmer is concerned with. Static slicing is one type of slicing technique. It is based only on the structure of a given program (static information), without knowing the (dynamic run time) value for any variables.

3. APPROACH

There are four major components inside the intelligent C language debugger. They are the parser, tokenizer, slicer, and variable analyzer. The parser parses the input program to a token string. Then, the tokenizer reads that string and saves the information to some internal linked lists. Based on information in the linked lists, the slicer executes the static slicing algorithm, and the variable analyzer decides which statement(s) have the highest probability of causing errors.

Here is how the variable analyzer works. It marks the correct variables from top to bottom of the program. Figure 2 shows an example to demonstrate this concept. If we suppose that variable *temp3* is correct in this example, the variable analyzer asks the slicer to provide the program execution path for *temp3*. The slicer determines that statements 39, 41, 42, 44, 45, and 46 are on this program execution path. The variable analyzer scans these statements and realizes that statement 39, 42, and 46 are the statements that define the value for variable *temp3*. Since variable *temp3* is correct, the variable analyzer sets the average correctness ratio to 100 in these statements. In this project, the correctness ratio is a number that measures the percentage of accuracy for a variable. It ranges from 0 to 100. Zero means that this variable has the highest probability of causing the program errors, and one hundred means that this variable has the lowest probability of causing the program errors.

For a correct variable, the variable analyzer sets the correctness ratio to 100. However, for an incorrect variable, the variable analyzer does not set the correctness ratio to zero because of the following reason. For an incorrect input variable, it does not make sense to assume that this variable is incorrect from the beginning to the end of the program. This variable could be correct at the beginning of the program, and could have become incorrect because of a bug somewhere in the program. When the variable analyzer tries to determine the correctness ratio for an incorrect variable, it needs to trace the variable dependency for that variable. When the variable analyzer calculates the correctness ratio for an incorrect or unknown status variable, it needs to know the program execution path for this variable. If an assignment variable is equal to some constant value(s), the variable analyzer has no way to predict the average correctness ratio for this statement. According to our algorithm, the variable analyzer checks to see how many correct and incorrect variables pass through this statement. If many incorrect variables pass through this statement, it probably has a higher chance of causing the program errors. If more correct variables pass through this statement, it probably has a lower chance of causing the program errors. Before the variable analyzer analyzes this type of statement, it first sets the average correctness ratio to 50 (meaning nothing neither good nor bad known about it). Whenever a correct variable passes through this statement, the variable analyzer adds ten to the correctness ratio. Whenever an incorrect

variable passes through this statement, the variable analyzer subtracts ten from the correctness ratio. Note that the emphasis should be on the resulting relative correctness ratios between all statements; not on their absolute values. Table 1 contains an example C program to demonstrate this concept.

```

1  main(){
.
.
38  j = 0;
39  temp3 = 1;
40  temp4 = 1;
41  while(j < degr2){
42      temp3 = temp3 * x1;
43      temp4 = temp4 * x2;
44      j = j + 1;
45  }
46  temp3 = temp3 * coef2;
47  temp4 = temp4 * coef2;
48  fl = temp1 + temp3 + (x1 * coef1) + coef0;
49  f = temp2 + temp4 + (x2 * coef1) + coef0;
.
.
70  }
    
```

Figure 2: Marking correct input variable

Lin	Code	AvgCorrectRati	Correct Ratio	
1	main(){			
2	int a, b,			
3	int w, x,			
4	a = 10;	100%		
5	b = 20;	40%		
6	c = 30;	40%		
7	b = b +	40%	b=40%	c=40%
8	w = a +	70%	a=100%	b=40%
9	x = b +	40%	b=40%	c=40%
10	y = w +	55%	w=70%	x=40%
11	}			

Table 1: Results for average correct ratio and correct ratio

If we suppose that the user has told us that variable *a* is correct and variable *b* is incorrect, the variable analyzer assigns an average correctness ratio of 100 to statement 4 and an average correctness ratio of 40 to statement 5. Because the user does not indicate the status for variable *c* in statement 6, the variable analyzer sets the average correctness ratio to 50 and realizes that variable *b* passes through this statement. It decreases the average correctness ratio from 50 to 40 to indicate that this statement has a higher probability of causing the program errors. In statement 7, the variable analyzer knows that the value of the reference variable *b* comes from statement 5 and the value of the reference variable *c* comes from statement 6. It copies the average correctness ratio from statement 5 to reference variable *b* and the average correctness ratio from statement 6 to reference variable *c*. The variable analyzer follows the same procedure to calculate the rest of the correctness ratio numbers.

4. IMPLEMENTATION

Figure 3 shows the overall design for the intelligent C debugger.

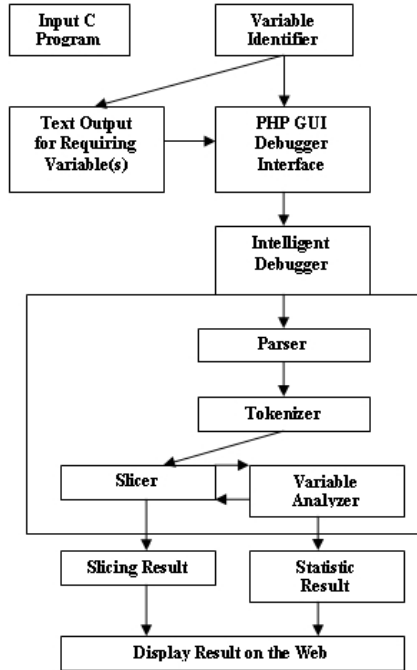


Figure 3: Project Design and Architecture

5. RESULTS

Given test result information on some or all of the variables in a program, the intelligent C debugger is designed to calculate the statements that have a higher probability of causing errors. The best results are obtained if users can provide very accurate and complete information to the debugger regarding which variables are correct or incorrect. Figure 4 shows the test case program, a Newton Raphson algorithm, and the corresponding results. In order to test the effectiveness of the intelligent C debugger, several bugs were deliberately inserted in statements 38, 47, 53, 61, and 66. In the best case, we can suppose that users are able to provide very accurate input information to the debugger. Table 2 is the information provided to the debugger. There are twelve correct input variables identified by users which are variables *dgr1*, *dgr2*, *dgr3*, *cof0*, *cof1*, *cof2*, *cof3*, *i*, *j*, *k*, *l*, and *iteration*. There are three incorrect input variables identified by users which are variables *ini*, *tmp1*, and *dfx*. The rest of the variables (*tmp2*, *tmp4*) are set to an unknown status. According to the results in Figure 4, the debugger clearly indicates the status of each statement by using different colors on a graphical bar. In statement 38, for example, the debugger shows that the correctness ratio is 20%. According to the implementation of the intelligent C language debugger, it correctly catches the bug in statement 38 of the test case program. Table 5 summarizes the correctness rating percentages for the deliberately inserted buggy statements in the test program. According to this table, the intelligent C language debugger marks one of these statements with having a very high chance of causing the program errors, three of the statements as having a high chance of causing program errors, and one of the statements as having a medium chance of causing the program errors. In statement 47, for example, the intelligent C debugger needs to trace variable

dependencies for variables *tmp2* and *tmp1*. In summary, three of the five buggy statements have estimated correctness rates of 33% or less, and all of them were calculated to have correctness rates of 60% or lower. If an iterative, incremental bug removal process is employed (similar to that used in typical debugging cycles with ordinary debuggers), then it is anticipated that subsequent runs of the intelligent debugger with increased test result information on a larger number of variables and fewer remaining bugs will yield better correctness ratio computations.

Sometimes users may not be able to provide very accurate or complete test result information to the intelligent C debugger. Table 4 shows a test case where some of the user input information is not correct. The same testing program in Figure 4 is used. The major difference between the two sets of user inputs is that variable *tmp1* is changed from incorrect to unknown, variable *tmp4* is changed from unknown to correct, variable *j* is changed from correct to incorrect, variable *tmp1* is changed from incorrect to unknown, and variable *dfx* is changed from incorrect to unknown. Based on this information, the debugger provides a very different analysis. For example, for buggy statement 47, the debugger now computes a correctness ratio of 41%, which is higher than the 33% computed earlier. Buggy statement 66 shows a correctness ratio of 65%, which is also higher than the 51% previously computed. When users specify incorrect or incomplete information to the debugger, it cannot correctly trace the variable dependencies for some statements in the program. For complicated programs, variables usually depend on each other. In these situations, the intelligent C language debugger cannot provide accurate results to users. Table 5 lists the correctness ratio for the buggy statements based on less accurate user input.

Correct Variables	<i>i, j, k, l, dgr1, dgr2, dgr3, cof0, cof1, cof2, cof3, iteration</i>
Incorrect Variables	<i>ini, dfx, tmp1</i>
Unknown Status Variables	<i>tmp2, tmp4</i>

Table 2: Users provide accurate input to debugger

Statement Number	Correct Rate %
38	20%
47	33%
53	33%
61	60%
66	51%

Table 3: Correctness rate percentages for accurate input

Correct Variables	<i>i, k, l, dgr1, dgr2, dgr3, cof0, cof1, cof2, cof3, iteration</i>
Incorrect Variables	<i>j, ini, tmp2, tmp4</i>
Unknown Status Variables	<i>dfx, tmp1</i>

Table 4: Users provide less accurate input to debugger

Statement Number	Correct Rate %
38	30%
47	41%
53	41%
61	100%
66	65%

Table 5: Correctness percentages for less accurate input

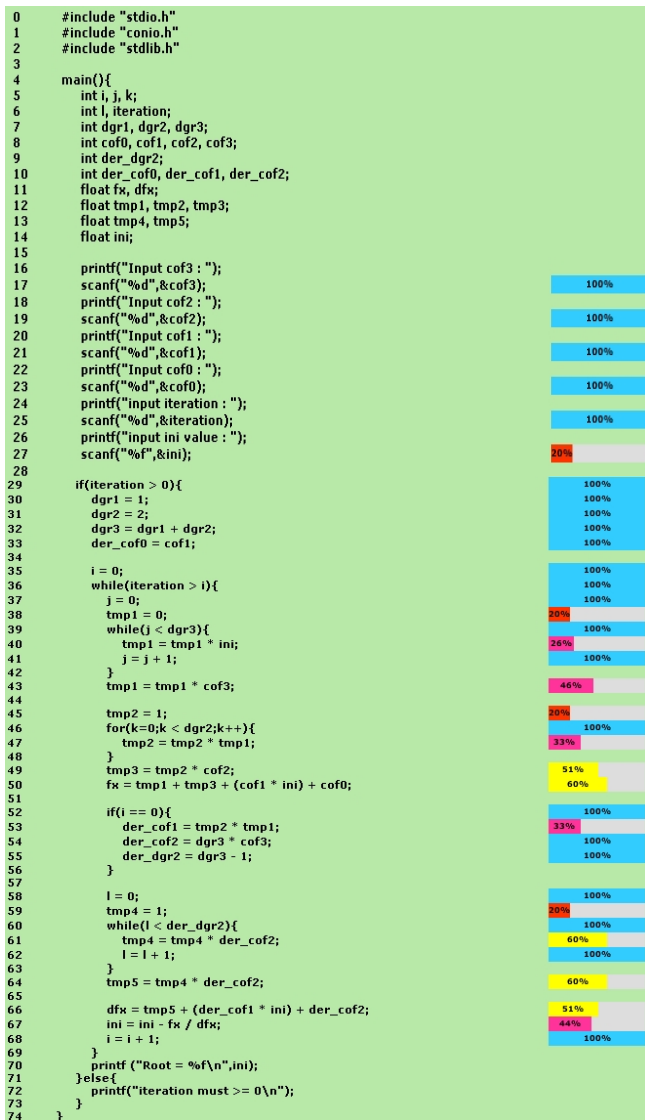


Figure 4: Testing program and the corresponding results

6. CONCLUSION AND FUTURE WORK

Static slicing can be a very useful tool to incorporate into debuggers. The intelligent C language debugger uses slicing as a key technique to analyze the program execution paths for variables in the program. The current slicer can handle assignment statements, I/O statements, *if-else* statements, *for* statements, and *while* statements. Another important feature in the slicer is that it can handle nested structures of conditional statements. This is an improvement over Weiser's static slicing algorithm which is not able to clean up empty conditional blocks. In figure 1, for example, Weiser's algorithm does not specify how to handle the empty *else* conditional block after slicing out statements $c = a - b$ and $y = a + x$. The slicer in the intelligent C language debugger is augmented to be able to detect the presence of an empty conditional block. When this situation happens, the slicer developed for this project cleans up the empty conditional blocks and their corresponding variables.

The variable analyzer is the second important tool for the intelligent C language debugger. When understanding the program execution paths for each variable in a program, the debugger uses the variable analyzer to analyze variable dependencies. In this step, the debugger actually combines two pieces of information together to predict the bugs in a program. The first part is the execution test result information provided by the user, and the second part is the information from the variable analyzer. By combining these two types of information together, the debugger can predict the statements that have a higher probability of causing program errors.

For future work, some parts of the intelligent C language debugger can be improved. Since the C language supports many different data types including *char*, *int*, *long*, *double*, or *long double*, the first step in improving the debugger is to add support for more of these data types which are often found in real world programs. The second way to improve the debugger is to support arrays. Many programmers employ arrays in their code. The third way to improve the debugger is to support subroutines in a C program. Subroutines help to simplify and modularize source code, making it more readable to the human being. Most programmers use subroutines in their code. If the intelligent C language debugger is to solve real world complicated programs, it needs to support subroutines to achieve this goal.

The process of debugging code can be laborious and frustrating for programmers. An intelligent debugging tool could provide much needed assistance to the programmer. If users can provide the execution test results for at least some of the variables of an erroneous program to the intelligent C language debugger, the debugger can help programmers to narrow down the bugs in the source code. For complicated programs, this tool can be especially useful because it can help programmers to focus their debugging efforts on just the relevant (most potentially buggy) parts of code.

REFERENCES

- [1] Rebaudengo, M., Sonza, M., Torchiano, M., Violante, M. (1999). Soft-error Detection through Software Fault-Tolerance Techniques. International Symposium on Defect and Fault Tolerance in VLSI System, Page 210.
- [2] Weyuker, E. (1993). More Experience with Data Flow Testing. IEEE Transactions on Software Engineering, Vol 19, No 9.
- [3] Laski, J., Korel, B. (1983). A Data Flow Oriented Program Testing Strategy. IEEE Transaction on Software Engineering, Vol SE-9, No 3.
- [4] Korel, B., Rilling, J. (1998). Program Slicing in Understanding of Large Programs. 6th International Workshop on Program Comprehension, Page 145.
- [5] Nakajo, T., Kume, H. (1991). A Case History Analysis of Software Error Cause-Effect Relationships. IEEE Transactions on Software Engineering, Vol 17, No 8.
- [6] Weiser, M. (1981). Program Slicing. In Proceedings of the Fifth International
- [7] Weiser, M. (1982). Programmers use slices when debugging. CACM, 25(7): 446 – 452.
- [8] Weiser, M. (1984). Program Slicing. IEEE Transaction on Software Engineering, 10:352 – 357.