International Journal of Latest Research in Engineering and Technology (IJLRET)
ISSN: 2454-5031
www.ijlret.com || Volume 05 - Issue 03 || March 2019 || PP. 54-65

Program slice technology based on pointer analysis

LIU Hai qing[1] , LI Zhi qiao[1]
[1]*North China Electric Power University, China*

Abstract: With the explosive growth of computer programs and data, it becomes difficult to analyze large amounts of code and complex structures by manpower alone, and it is necessary to slice the program. Based on the classical slicing technique, this paper proposes an improved pointer analysis program slicing method. Firstly, the data dependency and control dependencies of the program are built. Secondly, the constructed dependencies are analyzed by pointers. The set of variables that the pointer may point to at runtime is counted. Finally, the Calpointer algorithm is used to implement the program slice. Experimental results show that pointer analysis reduces code size, reduces program slicing time, and improves slice efficiency and accuracy.
Keywords: Program slicing; pointer analysis; data flow; data dependence; control dependence

I. INTRODUCTION

Software maintenance is an important part of the software life cycle. But in many cases, software designer is different from the operation maintainer and it is difficult for the software maintainer to accurately and specifically understand the entire software code in a short period of time. Therefore, how to accurately locate the feature information and quickly clarify the program architecture has a certain meaning. In the large software development phase, the "modular design" method is often followed. Also in the software maintenance phase, in order to improve the software maintenance efficiency, a similar "modular" decomposition program method -- program slice is used. Program slicing is a method proposed by Mark Weiser[1] for analyzing programs. The program slice finds and filters out the statements that may be affected or affected by the points of interest in the program, and breaks up the largeprogram into smaller parts. The set of program statements is the result of the program slice. The program slices obtained by the decomposition are then analyzed to make the large-scale computer program easier to construct, understand and maintain.

Regarding the program slicing, there is some literature on the slicing method. The literature [1] starts from the new method of constructing the summary side, and introduces the IFDS algorithm to propose the inter-process program slicing based on the information flow analysis. The intra-process slicing refers to a program. The slice is calculated internally; if there are multiple processes in a program, you need to use inter-process slices. This article uses SDGSlicer, InfoSlicer, IFDSSlicer three tools to achieve program slicing. However, these three compatibility problems cause the program to be inefficient in parallel and consume a lot of time. In the literature[2], an unreachable path detection method based on program slice and symbol execution is given. First, the program is statically sliced, the slice results are constrained, the unreachable path is filtered, and then dynamically sliced. Finally, the dynamic and static slice results are combined to realize the unreachable path detection. Literature [3] uses a technique that adds an exception handling structure to the construction of the graph, enabling better program slicing. These program slice results have an exception handling structure, but the program control graph SDG in this method is more complex than the general SDG control chart and requires further optimization. In [4],the paper proposes a program slicing method based on information flow analysis, which can process the program with more variable characteristics, and as a result reduces the errors caused by the sporadic factors. However, this method increases the slicing cost of the program slice, which complicates the result of the program slicing. Inliterature [5], an inter-process slicing algorithm based on id UCF's missing structure is proposed. It is not necessary to calculate data dependencies, control dependencies, and related function call information that are not related to slicing, but this method increases the slicing time and ultimately results in a longer time. In [6], a dynamic program slicing method based on forward computation is proposed. This method uses the definition variable influence set of the current execution statement, calculates its direct dynamic dependency and calculates the dynamic slice in the current execution statement. However, when the above slicing method is used for some C programs with jump instructions, the jump control will cause the program control flow to change, causing the related statements in the slice to be cut off. In the end, the program slice results are not comprehensive enough to cover all the statements under the program slice standard.

International Journal of Latest Research in Engineering and Technology (IJLRET)
ISSN: 2454-5031
www.ijlret.com || Volume 05 - Issue 03 || March 2019 || PP. 54-65

Aiming at this problem, this paper proposes a program slicing method based on pointer analysis, using pointer analysis to analyze its data structure and control structure, and using slicing algorithm Calpointer to slice the program, so that the slice result of the program can completely cover the program related content. This method not only effectively reduces the program size, but also increases the true statement in the program fragment after splicing by nearly double, improving the accuracy of the program.

II. SOURCE CODE PREPROCESSING

1.1Build control dependencies

Assuming that two instructions $S_1$and S_2, S_1 in the program determine whether S_2 is executed as a control command, S_2 control depends on S_1, and the relationship between commands S_1 and S_2 is called control dependency. In this paper, we use the post-dominant boundary algorithm to construct the control dependence [2]. For the given node x in the control dependency graph CFG, we calculate the post-dominant boundary of x by scanning the post-dominant tree of the dependency graph. To ensure that each node in the program has control dependencies, many nodes in the CFG appear linear using the CFG graph, using the post-boundary advantage for actual inference.

Suppose $PDF(x)$ is used to represent the post-domination boundary of node x, $pred$ (x) represents the predominant of nodes in CFG, chd (x) represents child node x in the post-dominant tree, and $x \subset_{pd} y$ indicates that x is y Dominating nodes, ipdom (X) is the direct dominating node of the node , then:

$$PDF_{local}(x) = \{y \in pred(x) | x \not\subset_{pd} y\} \tag{1}$$

$$PDF_{up}(x) = \{y \in PDF(x) | ipdom(x) \not\subset_{pd} y\} \tag{2}$$

The above formula shows that if a node X is not dominated by X, then when it belongs to PDF(x), then $PDF_{local}(x) \subseteq PDF(x)$ can be derived. For any node K in the CFG, we can find the common node of the post-dominated boundary of the post-dominant boundary of k and the sub-node of the post-dominant tree. The post-dominant boundary of x can be represented by a combination of a collection PDF and a set of child nodes:

$$PDF(x) = PDF_{local}(x) \bigcup \bigcup_{z \in chd(x)} PDF_{up}(x) \tag{3}$$

The algorithm for calculating the post-dominant boundary needs to access the post-dominant tree of the program. In this paper, the sliced platform LLVM is used to calculate the dominating tree. The direct post-dominant edge of the LLVM block is regarded as dependent on the basic block in the graph, and the post-dominated boundary is calculated. The sample source program is shown in Table 1:

Table 1 Sample source program

```
#include <assert.h>
#include <stdio.h>
long int fact(int x)
{ long int r = x;
while (--x >=2)
r *= x;
return r;}
int main(void)
{ int a, b, c = 7;
while (scanf("%d", &a) > 0) {
assert(a > 0);
printf("fact: %lu\n", fact(a));}
return 0;}
```

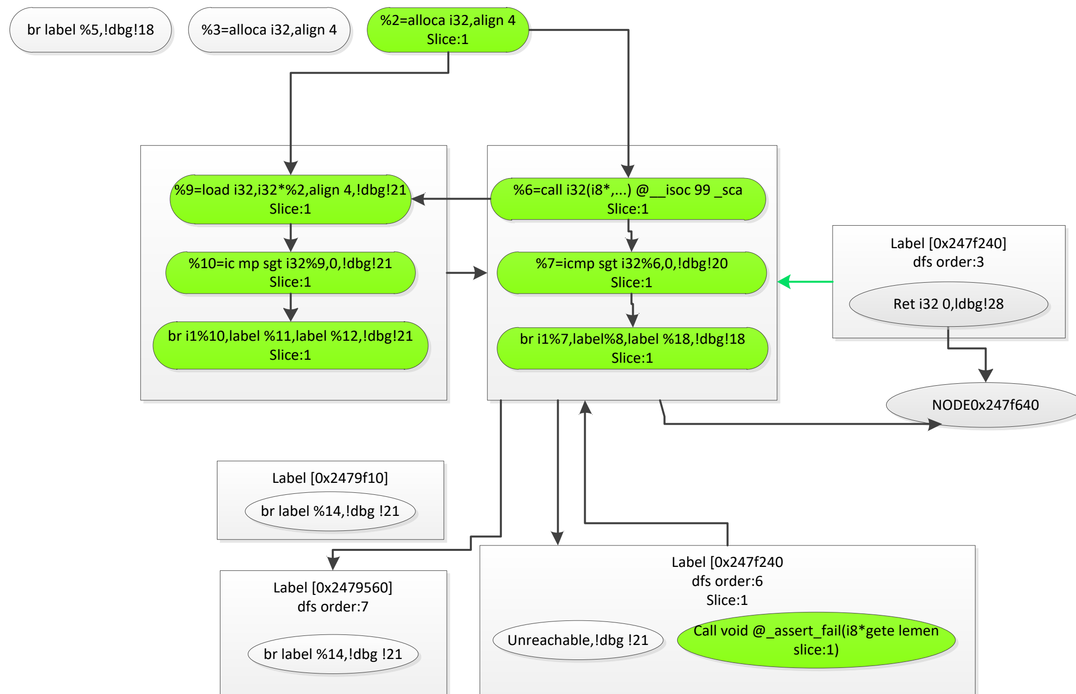

Fig 1 Program dependency graph construction

The corresponding program dependency graph is shown in Figure 1. The green box in the above figure refers to the part of the program that has control dependence on the slice criteria, and the gray part is the unrelated part.

1.2 Build data dependencies

Data dependency refers to the relationship between a piece of data, some data that has been processed before due to the structure of the program. The data dependent edge is merged into the dependency graph by other types of edges, but since LLVM retains the defuse edge for top-level variables and uses indirect data dependency construction of variables, it is necessary to calculate the indirect data dependency of the address-based variable. Pointers are an efficient means of obtaining variable addresses in the C/C++ language, so pointer analysis is required before determining data dependencies. The pointer points to the address information of the memory during program execution and sets a point for each pointer, classifying the address information as stream sensitive and stream insensitive. Stream sensitivity refers to pointing a pointer to content after all relevant programs have been executed in sequence. Collect pointer-related statements from the program, and express the collection's inclusions as equations, and convert these equations into graphs for analysis. The relationship between the point and the information edge is converted into a storage shape graph (SSG). Using C language syntax to analyze reference pointers, the rules for defining SSG are shown in Table 2:

Table 2 Example of SSG operation rules

p=&a	$p \rightarrow a$
q=p	$\forall x$ if $p \rightarrow x$ then $q \rightarrow x$
q=*p	$\forall x$ if $p \rightarrow x$ then ($\forall y$ if $x \rightarrow y$ then $q \rightarrow$ y)

In the table, p is a pointer. When p=&a appears in the program, it indicates that it is necessary to add an edge from node p to node a. When q, p is a pointer and q=p appears in the code, you need to add the edge of q to

the point of the p pointer. However, multiple dereferences will be created in the same way, where dereferencing is the return of the value stored in the memory address, and the introduction of the temporary variable converts multiple dereferenced programs into a single dereference, providing a dereference rule. The same decomposition is done for the dereference on the left, and then these rules are referenced to simply add the edges as procedural rules.

Use the relevant statements in the flow insensitive analysis iterator, then add the edges of the SSG according to the rules until you can't add new edges, then analyze the program. By assigning an SSG to each point in the program, we can get information about the valid points in the program. The pointer information of the statement is calculated, the data stream is analyzed, and the information is passed to the successor of the statement in the CFG. And then the success or passes the new state to the subsequent successor and continues to cycle until it reaches a steady state. This method uses the same rules to calculate pointer information and finally maps those statements to the program flow graph. The algorithm process that the build program depends on is shown in Table 3:

Table 3 Build dependency algorithm

Input: SDG, CG of program P, slice standard <s0> (s0 is the statement in program P)
Output: summary edge collection
declare $S_1$is a collection of P calls directly or indirectly;
    S_2 is a collection of programs called directly or indirectly by P;
    S_3 is the program collection;
Initialization: $S_1=S_2=S_3=\{P\}$;
  begin
    for each $Q \in S_1$
$\{R_1,R_2...R_n\} \leftarrow$Find all the programs that call Q from CG;
3) $S_1 \leftarrow S_1 \cup \{R_1,R_2...R_n\}$;$S_3 \leftarrow S_3 \cup \{R_1,R_2...R_n\}$;
4)   endfor
5)   for each $X \in S_2$
6) $\{Y_1,Y_2...Y_m\}$;$\leftarrow$find all the programs called by X from CG;)
$S_2 \leftarrow S_2 \cup \{Y_1,Y_2...Y_m\}$; $S_3 \leftarrow S_3 \cup \{Y_1,Y_2...Y_m\}$;
8)endfor
9) call calpointer(S_3);
end

1.3 Pointer analysis optimization and pointer subgraph

In C/C++ programs, pointeris a common data structure that exists widely in various programming languages, but when two or more variables access the same storage address by pointers, the dependencies between variables are ignored because they point to the same address, resulting in incomplete slice results. Therefore, pointer analysis is necessary. Pointer analysis is the use of static analysis to get a collection of variables that a pointer can point to at runtime when compiling a program. The essence of pointer analysis is data stream analysis. Its main purpose is to statically obtain the pointer of the program at runtime to point to the information. The obtained pointer analysis result is the basis of the program slicing.

Before the pointer analysis, a pointer subgraph (PS) needs to be constructed for the program. PS is generated by deleting all nodes in the dependency graph CFG of the analyzed program that are not related to pointer analysis.

The pointer analysis framework of this paper can analyze the rules of the shared summary edge for different pointers, then select the relevant nodes from the SSG, and add new summary edges according to the information in these nodes. If the nodes are different, the analysis results will be different. the summary side represents the transfer dependency of the parameters in the program. For example, if the formal parameter x is used to calculate the value returned from the procedure in the formal parameter y, then there is a summary edge between the actual parameters corresponding to x and y.

From Figure 1, all nodes not related to pointer analysis are deleted, and the pointer subgraph is obtained, as shown in Figure 2:

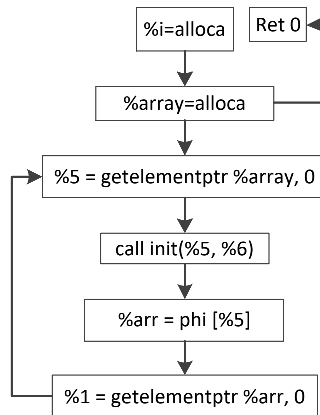

Fig 2 Pointer subgraph PS

To solve the domain-sensitive problem, a pair of (n, off) is used to represent a pointer, where n is the node from the PS that allocates the memory (the node associated with the alloca instruction or the function that calls the allocation) dynamic memory, and off is the memory offset represented by the 64-bit number 10. Using the offset to change the SSG, each node of the SSG is divided into portions that correspond to different offsets in the memory. In order to maintain point-to-information for each offset in the SSG node, a sparse mapping is used instead of an explicit enumeration of all possible offsets, since the range of possible offset values in a node can be large.

When parsing a program pointer, each pointer is also a variable, so it can be referenced, and there can be a summary edge in the SSG between any two nodes. Therefore, top-level variables are not suitable for storing shape graphs because they are simply memory valuesor calculations for a point in the program. At the same time, since nodes are not created for the top-level variables in the SSG, and the same top-level variables in the SSG are also in the PS of the program, in order to track the pointers in the memory, an SSG is established using the corresponding nodes of the PS. The analysis works with the PS node and performs queries and modifications to the SSG as needed, storing the results in a PS node corresponding to the pointer in the LLVM. Combining the pointer sub-picture of Figure 2 with the SSG extended by offset results in Figure 3.

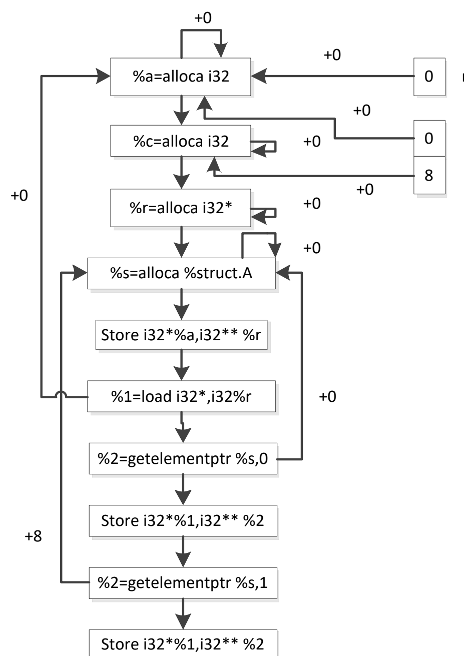

Fig3 Improved SNP submap with SSG

The pointer node is then used to process the PS node and select the relevant node of the SSG with the actual point-to-information based on the operation associated with the node. The pointer analysis framework uses virtual methods to get the memory objects covered by the actual analysis. This method returns a node from a SSG with a pointer, which refers to information related to the PS node being processed. Then use this method to implement the rules for the summary side. The pointer subgraph PS of this paper is generic and can be generated from any other program representation (such as C or assembler), so the pointer analysis of this article can be used in any language as long as PS can be generated.

III. SLICE METHOD BASED ON POINTER ANALYSIS

After the initial source is obtained, the source program is first preprocessed using LLVM (the underlying virtual machine) and converted to LLVM IR. The LLVM IR language is an intermediate language specific to the underlying virtual machine LLVM. The LLVM code representation is used in a set of abstract instructions to provide critical advanced information for program analysis while enhancing the applicability of the language.

In a C program, if there is a jump function or a long jump function, it may cause the slicing algorithm to cut off the unrelated rows of the called row of the cut line in the entire program when slicing a particular row. The slice's track is different from the original program's state track, causing the call to the procedure not to return to the calling site. Therefore, we need to slice the program using the program segmentation method based on pointer analysis, so that all the statements controlled by the non-return call point depend on the jump statement executed in the program, and can get the complete slice content.

3.1 Slice process based on pointer analysis

The source code is first processed using the LLVM compilation framework, processed as an intermediate form of LLVM IR, and then using pointers and unstructured control flow analysis programs. Among them, the pointer variable is different from the common variable, the pointer variable holds the address of another variable, and the common variable only represents a certain address. Therefore, if you directly analyze the static value of the program when analyzing the pointer variable, it is likely to cause problems in the analysis of the pointer variable when the pointer variable is dereferenced [9]. In the pointer analysis of the program, it is necessary to simplify the pointer variable in the program, convert it into a first-level pointer, and then perform data flow insensitive analysis of the pointer, and obtain the pointer set of the pointer.

After preprocessing the program, using the Calpointer algorithm to construct the summary edge, the Calpointer algorithm can reduce the time complexity of calculating the summary side. The slice method based on the algorithm Calpointer not only improves the slice efficiency, but also ensures that the slice accuracy is not affected. The general process of slicing is shown in Figure 4:

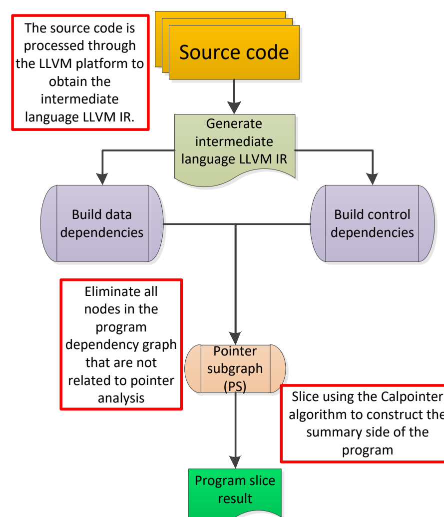

Fig 4 Slicing system flow

3.2 Slicing algorithm

In a static program slice, the slice result is not affected by the input value. The process of the slice depends only on the static information in the program, so the slice result is incomplete and cannot cover all the relationships in the program. In the previously established system dependency graph, it is assumed that there is a procedure Q of the program. In order to obtain the node formal-in that each formal-out node depends on, each formal-out node in the process Q needs to be sliced. Then, in order to construct the dependent-side of actual-in to actual-out, according to the correspondence between the formal-in/out and actual-in/out nodes obtained in the first step, get the dependencies between the two nodes and get the passed dependent edges. The summary edge calculation of process Q may be affected by the summary edge calculation of the module called by Q. Therefore, globally, the solution of the summary edge corresponding to each process is an iterative process. When all the summary edges of the call are stable, the calculation terminates.

This paper proposes to use the Calpointer algorithm to synthesize the summary edge and perform program slicing. From the whole program, the summary side calculation of process Q may be affected by the calculation of the summary side of the module called by Q. The solution process corresponding to the summary side of each process in the program is an iterative process, when all the summary edges of the call are stable. The calculation will be terminated. The algorithm is shown in Table 4:

Table 4 Calpointer algorithm for pointer analysis

Algorithm (Calpointer)
procedure pointer(s_3)
declare S_v to be the slice result set of $v_{formal-out}$;
S_{vi} is a collection of all input parameters in S_v;
Sum Form is a set of (formal-out, formal-in);
Sum Act is a set of (actual-out, actual-in);
Initialization: Sum Form=Sum Act= S_v =Ø;
begin
Repeat
W={Z1,Z2...Zn}←Obtain the program from s_3 according to the inverse topological order of CG;
$s_3 \leftarrow s_3 \cup$ W;
for each $Z \in W$ doCFG //After deleting all the nodes in the Z set that are not related to the pointer analysis, the pointer subgraph of Z is obtained;
for each $v_{formal-out} \in Z$ do
if (S_v ==Ø)
S_v ←Invoking the in-process slicing algorithm to slice the $v_{formal-out}$ program;
for each $u_{formal-in} \in S_{vi}$ (S_{vi} ←Take all the input parameters from S_v ;)
Sum Form←Sum Form $\cup \{(v_{formal-out}, u_{formal-in})\}$;
for each $(v_{formal-out}, u_{formal-in}) \in$ Sum Form

Sum Act←Sum Act ∪ $\{(x_{actual-out}, y_{actual-in})\}$;

Until s_3 =Ø;

End

The execution steps of the Calpointer algorithm are as follows:

(1). Find all direct or indirect calls and called assemblies in program P, where the assembly calling P is {X} and the call to P is {Y}.

(2). When the pointer program performs the first iteration, first extract the program X from the CG, and then perform in-process slicing on the formal-out node of X to obtain the corresponding formal-in node.At the same time, according to the information in the sum form obtained, the binary relationship between actual-out and actual-in at the calling point in P is obtained, and the Sum Act is updated.

(3). When the pointer(S_{proc}) program performs the second iteration, it also performs in-process slicing on the formal-out in the program P, and updates the Sum Form, and then uses the Sum Form information to obtain the binary relationship between actual-out and actual-in at the calling point in C , and update the Sum Act. Then the third and fourth iterations are performed in sequence until S_{proc} is empty, the iteration is stopped, and the algorithm ends.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

The experimental environment used in this article is 8G memory, Intel(R) Core(TM) i7-3940XM, 64-bit Win10 physical machine, completed in 64-bit Ubuntu virtual environment.

The source code is processed using the LLVM compilation framework and converted to the LLVM IR intermediate form. The source program shown in Table 1.1 above is preprocessed and converted to the intermediate format IR. The interception part is shown in Figure 5:

```
; ModuleID = 'test.bc'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str.1 = private unnamed_addr constant [6 x i8] c"a > 0\00", al
@.str.2 = private unnamed_addr constant [7 x i8] c"test.c\00", a
@__PRETTY_FUNCTION__.main = private unnamed_addr constant [15 x
@.str.3 = private unnamed_addr constant [11 x i8] c"fact: %lu\0A

; Function Attrs: noinline nounwind optnone uwtable
define i64 @fact(i32) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i64, align 8
  store i32 %0, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = sext i32 %4 to i64
  store i64 %5, i64* %3, align 8
  br label %6

; <label>:6:                                      ; preds = %10,
  %7 = load i32, i32* %2, align 4
  %8 = add nsw i32 %7, -1
  store i32 %8, i32* %2, align 4
  %9 = icmp sge i32 %8, 2
  br i1 %9, label %10, label %15

; <label>:10:                                     ; preds = %6
  %11 = load i32, i32* %2, align 4
  %12 = sext i32 %11 to i64
  %13 = load i64, i64* %3, align 8
  %14 = mul nsw i64 %13, %12
  store i64 %14, i64* %3, align 8
  br label %6

; <label>:15:                                     ; preds = %6
  %16 = load i64, i64* %3, align 8
  ret i64 %16
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 7, i32* %4, align 4
  br label %5

; <label>:5:                                      ; preds = %14,
  %6 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inb
  %7 = icmp sgt i32 %6, 0
```

Fig5 Intermediate format IR program

After the pointer analysis of the program is performed, the slice is as shown in Table 5:

Table 5 Program slice results

```
; ModuleID = 'test.sliced'

source_filename = "test.c"

Targetdatalayout =

"e-m:e-i64:64-f80:128-n8:16:32:64-S128"

target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1

@.str.1 = private unnamed_addr constant [6 x i8] c"a > 0\00", align 1

@.str.2 = private unnamed_addr constant [7 x i8] c"test.c\00", align 1

@__PRETTY_FUNCTION__.main = private unnamed_addr constant [15 x i8] c"int
main(void)\00", align 1

@.str.3 = private unnamed_addr constant [11 x i8] c"fact: %lu\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable

define i32 @main() #0 {

entry:  ret i32 0 }

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-
math"="false""disable-tail-calls"="false""less-precise-fpmad"="false""no-frame-pointer-
elim"="true""no-frame-pointer-elim-non-leaf""no-infs-fp-math"="false""no-jump-
tables"="false""no-nans-fp-math"="false""no-signed-zeros-fp-math"="false""no-trapping-
math"="false""stack-protector-buffer-size"="8""target-cpu"="x86-64""target-
features"="+fxsr,+mmx,+sse,+sse2,+x87""unsafe-fp-math"="false""use-soft-float"="false" }

!llvm.module.flags = !{!0}

!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}

!1 = !{!"clang version 5.0.0 (tags/RELEASE_500/final 337178)"}
```

Among them, the time complexity of the Calpointer algorithm to obtain the dependent edge is $O(P \times n \times (V + E))$, the space complexity is $O(n^2 \times V)$, where n represents the maximum output shape parameter of a single program, V is the number of program dependent graph vertices, E is the number of edges in the program dependency graph, P represents The number of statements in the program.

After the first stage pointer analysis, the result of the pointer analysis reduces the code size, and the time spent by the system for pointer analysis is offset, so that the subsequent optimization effect is better, and the optimization efficiency speed is improved. In addition, because the number of vertices in the program dependency graph and the number of edges in the program dependency graph are reduced, the time complexity is greatly reduced compared with the previous one after adding the pointer analysis, and the space complexity is increased. Improve the efficiency of the slice without affecting the accuracy of the slice. At the same time, because the time cost of program pointer analysis is very small, the pointer analysis reduces the program slicing

time by reducing the time of program analysis. On the one hand, the time complexity of the algorithm is generally linear and the efficiency is relatively high; on the other hand, after the pointer analysis The result is a reduction in the size of the program code, making subsequent slicing processes faster, thus offsetting the cost of pointer analysis.

Experiment with the following typical procedures, as shown in Table 6:

Table 6 Comparison of slice time for different programs

Program	Quantity			Time(ms)	
	Number of classes	Number of methods	Number of lines of code	Pointer optimization	Unused pointer optimization
compiler	74	803	10245	80	91
compress	63	767	10590	43	58
derby	60	776	9849	124	134
sparse	79	786	10526	68	82
validation	67	775	10691	115	124
scimark.large	61	760	10673	125	142
scimark.small	57	753	9958	94	97
sunflow	69	793	10044	81	87
serial	75	781	10194	97	102
mpegaudio	76	798	10539	55	59
crypto	77	809	10198	69	75

Optimization comparison Figure 6 is as follows:

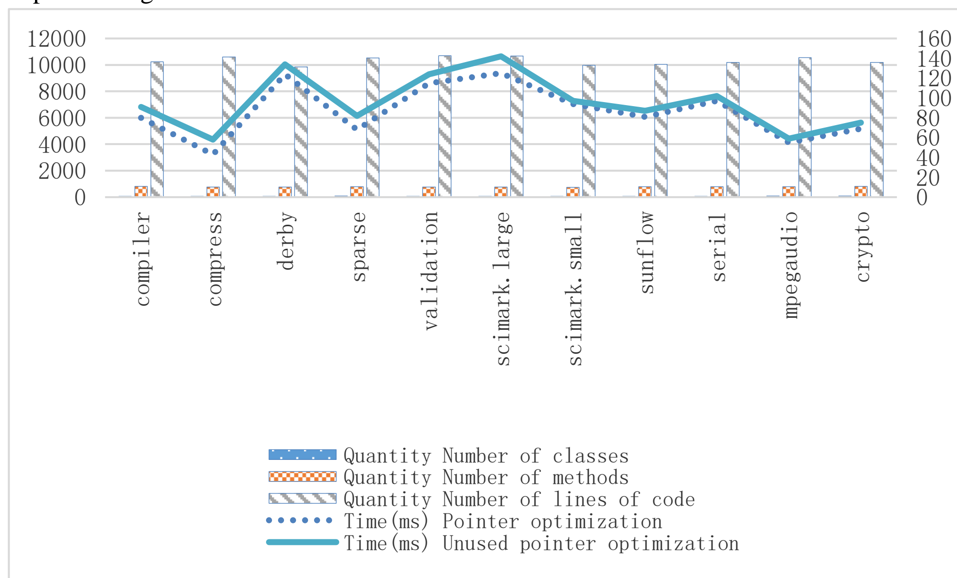

Fig 6 Optimization comparison chart

This article uses the CPA checker tool to verify the plausibility of the slicing method. Using the C benchmark test, when looking for errors in a computer program, classify the categories by type and understand the errors that may be included in the program, such as signed integer overflows or memory leaks. On CPA check, separate slice and individual pointer analysis control flow slices are compared. In the benchmark, there are several possible outcomes: true, failed, unknown, timeout, and error. Among them, “true” means that the

program does not detect an error, "failed" indicates that there is an error in the program, "unknown" is unable to determine whether there is an error in the program, time refers to the response time, and "error" refers to the number of times the error occurs. As long as there is any error in the verification process, the error result will be returned.

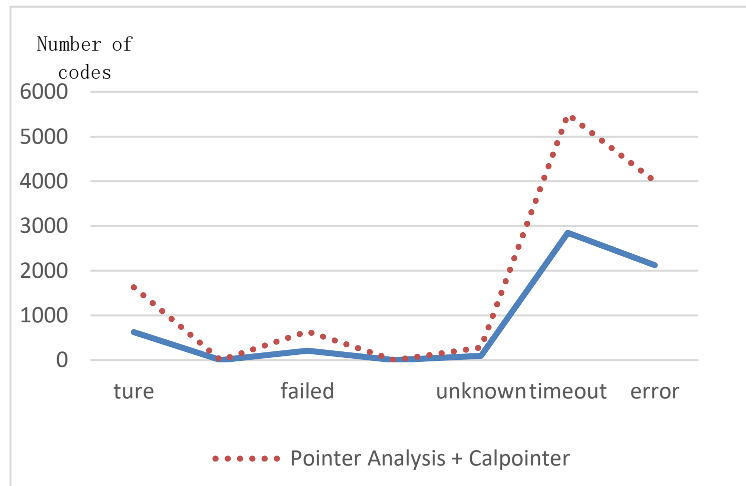

Fig 7 Optimization comparison chart

As can be seen from Figure 7, after adding the pointer analysis, the result of the program slice is optimized, and the true statement in the slice result is nearly doubled. Experiments on different programs as follows show that after adding the pointer analysis, not only the slicing time is shortened, but also the accuracy of the slicing is improved to some extent.

V. CONCLUSION

This paper proposes to apply the pointer analysis method to the slicing technology, which can accurately and effectively slice the program to reduce the slicing time. Before the program is sliced, the pointer analysis method is used to analyze the program structure, which can effectively avoid the final error or the incomplete slice result caused by the overlap of multiple variable access positions in the program caused by the pointer.

Then use the Calpointer algorithm to slice the program. Comparing the obtained program slice results with the results of program slices without pointer analysis, it can be clearly found that after adding the pointer analysis, the time of the program slice is shortened, and the number of correct sentences in the program slice result is improved. Therefore, it can be concluded that this paper combines pointer analysis with program slicing to optimize the results of program slicing.

REFERENCES

[1] Xu Chenchen. Research on static program slicing method based on LLVM [D]. Nanjing University of Posts and Telecommunications, 2017.

[2] WANG Hong-da,XING Jian-chun,SONG Wei,YANG Qi-liang.Static BPEL program slicing based on program dependency graph[J].journal of Computer Applications,2012,32(08):2338-2341.

[3] Xu Manqing. Unreachable path detection method based on program slice and symbol execution [D]. Nanjing University of Posts and Telecommunications, 2016.

[4] Hao Jie. A program slicing diagram construction method with exception handling structure [J]. Volkswagen Technology, 2012, 14 (02): 18-20.

[5] Yao Jiabao. A program slicing method based on information flow analysis [D]. Jilin University, 2013.

[6] SU Xiao-hong, GONG Dan-dan, WANG Tian-tian, MA Pei-jun. A new fast algorithm for static slicing between processes[J].Journal of Harbin Institute of Technology,2015,47(05):25-31.

[7] Wang Xingya, Jiang Shujuan, Pei Xiaolin, Shao Haoran A Dynamic Program Slicing Method Based on Forward Calculation [J] Computer Science, 2014, 41(01): 250-253 + 278.

[8] Jiang Gang,Li Zhaopeng.Design and Implementation of Program Slice in C Program Analysis Tool[J].Mini-micro Systems,2018,39(03):401-405.

[9] Liu Fei. Research on pointer analysis technology for error detection [D]. Southeast University, 2015.

[10] Guo Wei,He Yanxiang,Zhang Huanguo,Hu Ying,Gamila Shatal.An Improved Pointer Security Analysis Algorithm[J].Journal of Wuhan University(Science Edition),2010,56(02):170-174.

[11] Xiangyu Zhang, R. Gupta and Youtao Zhang, "Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams," Proceedings. 26th International Conference on Software Engineering, 2004, pp. 502-511.

[12] S. Jiang, R. Santelices, M. Grechanik and H. Cai, "On the Accuracy of Forward Dynamic Slicing and Its Effects on Software Maintenance," 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, Victoria, BC, 2014, pp. 145-154.

[13] Rapoport M., Lhoták O., Tip F. (2015) Precise Data Flow Analysis in the Presence of Correlated Method Calls. In: Blazy S., Jensen T. (eds) Static Analysis. SAS 2015. Lecture Notes in Computer Science, vol 9291. Springer, Berlin, Heidelberg

[14] Jiang Gang. Design and implementation of program slicing and transformation in C analysis tools [D]. University of Science and Technology of China, 2017.

[15] Zhang Yingzhou,Xu Chenchen,Yu Shurong.A Parametric Improved SDG Program Slice Method[J].Journal of Nanjing University of Posts and Telecommunications(Natural Science Edition), 2017,37(06):75-80+89.

[16] Wu You. Design and Implementation of Dynamic Data Dependence Analysis Tool for C Program Based on LLVM [D]. Jilin University, 2016.