Code Instrumentation using Dynamic Program Analysis Tools

Mr. D. S. Panchal
Department of CS & IT,
Dr. B.A.M.U. Aurangabad
Aurangabad, India.

Dr. S. N. Deshmukh
Department of CS & IT,
Dr. B.A.M.U. Aurangabad
Aurangabad, India.

***Abstract*— in this paper we have studied and analyzed different dynamic analysis tools and code slicing techniques.**

1) Dynamic program analysis tool is a tool used to analyze the program at program execution time. Code Instrumentation using dynamic program analysis tool means inserting extra code into the executed program to collect runtime information or code within a tool which is able to handle the program at program execution time.

2) Program Slicing is an effective for narrowing the focus of attention to the relevant parts of a program during the process of debugging. The two known program slicing methodologies are static and dynamic slicing.

This Paper criticizes the above tools.

Keywords*— Instrumentation; analysis; Executable; Code; slicing; Static; Dynamic; variable;

I. INTRODUCTION

A. *Code Instrumentation:*

Code Instrumentation is a technique to insert the extra code into already developed application executables with the help of program analysis tool known as dynamic program analysis tool i.e. Pin Tool. Pin tool is developed with the help of a framework known as pin. This pin framework provides an API for supporting pin tool development which is created by Intel. With the help of pin tool user can check memory leakages, find programming faults, reduce the extra code redundancy. When you run program within a pin tool, it will stop the program execution at first instruction of the executable and generates ("compiles") new code for the straight line code sequence starting at this instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution Reference number as in [1]. Pin makes this efficient by keeping all of the generated code in memory so it can be reused and directly branching from one sequence to another. Pin can be used to insert C/C++ any code in any places into the executable dynamically. You can either start new process directly from pin tool or attach and detach to an already running process Reference number as in [1]. Like a debugger, Pin can attach to a process, instrument it, collect profiles, and eventually detach. The applications only enter instrumentation overhead during the period that Pin is attached. The ability to attach and detach is a necessity for the instrumentation of large, long-running applications. When developing the pin tool you are actually telling pin how to generate the code from the main executable. It allows tool developer to analyze an application at the program instruction level without any detail information about the underlying instruction set. Pin API is designed as architecture and operating system independent means making source compatible on different architecture and operating system. It supports Linux and Windows executables for IA-32, Intel(R) 64, and IA-64 architectures. As per the need pin tool can access architecture specific details. Pin provides efficient instrumentation by using a just-in-time (JIT) compiler to insert and optimize code. In addition to some standard techniques for dynamic instrumentation systems including code caching and trace linking, Pin uses different techniques like in lining, register re-allocation, run time analysis and instruction scheduling to use instrumentation Reference number as in [2]. Program analysis can be used to find and to detect source code like program testing, program monitored during bugs are found. Dynamic analysis is also used for profiling the program, understanding the program and studies the programming patterns. A user may write instrumentation tools using an API that is rich enough to allow many plug-INS to be source compatible for all the supported instruction sets. Pin allows a tool to insert function calls at any point in the program. It automatically saves and restores registers so the inserted call does not overwrite application registers.

B. *Program Slicing:*

Program slicing is an effective for narrowing the focus of attention to the relevant parts of a program during the process of debugging. The two known program slicing methodologies are static program slicing and dynamic slicing. The inaccuracy is a problem in static slices since they consider all execution that reach the slicing criteria point rather than a fixed execution under which the program is being debugged. Dynamic slicing is precise but expensive due to runtime overhead.

For example, in C programs that make extensive use of pointers, the traditional nature of static data dependency analysis leads to highly inaccurate and considerably larger program slices. Since the main aim of slicing is to identify the subset of program statement that are of interest for a given purpose, the traditional computed, large slices are clearly undesirable. Recognizing the need for accurate slicing, korel and Laski suggested the idea of dynamic slicing. Dynamic program slices are constructed upon users input. It has been shown that the dynamic slices can be considerably smaller than static slices. The importance of slicing extends well beyond debugging of programs. Increasingly applications aimed at improving software quality, reliability, security and performance are becoming candidates for making use of dynamic slicing. Examples of these applications include: detecting spyware that has been installed on systems without the user knowledge, carrying out dependence based software testing, and measuring module structure for the purpose of code restructuring, extracting business logic and business rules from legacy code.

Program slices have been suggested and used in separating business logic from the remaining code and in identifying business rules in large code base. Business rules typically consist of calculations and the conditions/constraints under which the calculations are done. Almost all of earlier work applies static program slicing techniques, which often has high inaccuracy and hence identifies large part of the code pieces, thus necessitating the use of filtering mechanisms to extract rules. This prompts the exploration of dynamic slicing.

Most of the work is done in static slicing while dynamic slicing is only in theory. Dynamic slicing was considered costly for computation of slice but gives precise results. As now-a-days memory is costly and processors are faster, we focused on dynamic slicing. We aim to study various techniques of dynamic slicing as a part of these code instrumentation techniques:

- To decide the best techniques of dynamic slicing and implementation, measure of this technique could be comparison of static to dynamic slicing in terms of
  i) Length of accuracy of slice
  ii) Time to get slice

- Find all paths to implement this dynamic slicing technique and compare them to get the best one.
- Given a slicing criterion to find all slices that can affect this criterion (because lot of input variables can point to the same output variables) reference number as in [9].

II. Dynamic Program Analysis

Dynamic program analysis for runtime verification can be used for many purposes, such as security or safety policy monitoring, debugging, testing, verification, validation, profiling, fault protection, behavior modification (e.g. recovery) etc. In C programming many times user defined variables will not use on entire application code, these variables consuming large amount of memory which are called wastage of memory i.e. memory space having no use. effect is memory get consumed and decrease the performance of application program execution, at that time we require extra memory for program execution on execution time which is not get back from operating system after program execution and user cannot change the executed code so we require a technique to insert code when program is executing. For that, user can use a technique known as Code Instrumentation using dynamic program analysis tool and program slicing which is able to narrowing the focus of attention to the relevant parts of program during process of debugging. Using these techniques we can collect runtime information of program such as unused variables space, programming faults, logging information, programming speed etc. And improve the performance of entire program.

III. PROGRAM INSTRUMENTATION

Program instrumentation using dynamic program analysis tool is used to improve the programming speed, generating traces, finding the programming fault and resolve them, delete unwanted source code and insert new source code into program executables and program slicing is to identify the subset of program statement as per the execution need, the traditional computed, large slices are clearly undesirable. Recognizing the need for accurate slicing, korel and Laski suggested the idea of dynamic slicing. Dynamic program slices are constructed upon users input. It has been shown that the dynamic slices can be considerably smaller than static slices. By using these two techniques we can improve whole performance of application program without any deadlocks or obstacles by analyzing and recording the runtime behavior of program executables.

IV. PIN TOOL

Pin tool can be used to insert C/C++ arbitrary code in arbitrary places in the dynamically executed executable; you can either start a new process directly from pin tool (our own program that uses Pin framework) or attach to an already running process. When developing the pin tool, you're actually telling Pin how to generate the code from the main executable: i.e. the code addition/modification processes. When you run a program within a pin tool, it will stop the program execution at first instruction and modify the code generation process. Then it will generate the code it will later execute by using one of the following modes:

i) TRACE_AddInstrumentFunction API call.
ii) INS_AddInstrumentFunction API call.
iii) IMG_AddInstrumentFunction API call.
iv) RTN_AddInstrumentFunction API call.

1) *System Design and Approach*

To understand functioning of dynamic program analysis, capture memory leakages in application program. Memory leak occur when a program consume some memory for its execution but unable to release it back to the main memory, A memory leak decrease the performance of software application due to reducing the available memory size. Due to low available memory system program get slow down or some of the application program get failed. The aim is to detect the memory leakages and collect required information to fix the memory leakage problems for efficiently and easily.

A. Memory Leak detection

Dynamic program analysis process starts by instrumenting the binary file. This instrumentation will inject few call back methods into the binary file. These methods will get invoked when a particular condition is met. This instrumented binary is executed in a test environment which monitors the program behavior. The inputs to the Program are given at its runtime. The program execution is follows a desired path depending on the input given and calling the instrumented methods to perform specified actions. This action involves writing a log containing required information on to a file. This log is then passed to an analyzer program which then computes and generates a report showing the analysis results. This analyzer contains a data structure used to store the information and perform operations and then generate the report. The instrumentation can also be done on source code which requires a compilation phase to generate a binary which will be run in test environment and program will be analyzed in the same manner as explained above.

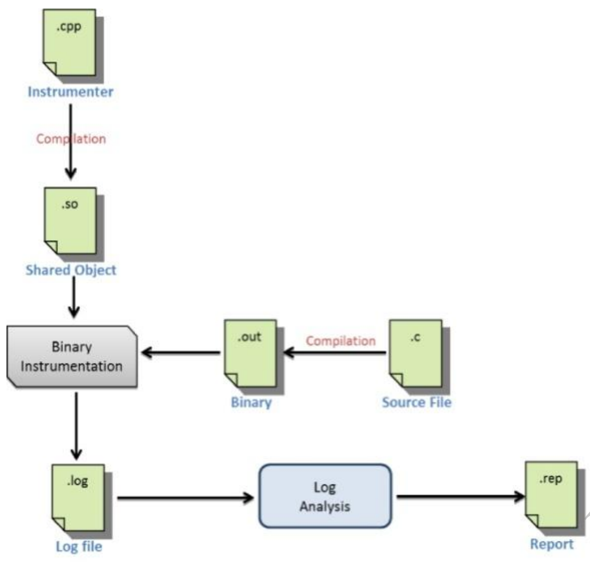


Fig. 1. Memory Leak Detection Process.

2) *Analysis of Data to be collected*

Program instrumentation should generate a log specific to the inputs given. This log must have all the information about memory allocation, modification and release. When a call to free is done, we have to match the memory address with previously allocated address therefore memory address is needed in the log. Also, needed to calculate total amount of memory leakage all the parameter values passed to the function along with its return value must be logged. For the location of defects, instrumentation should add where in the Source program the memory call occurred. In case of binary instrumentation, only instruction address is not sufficient as location, therefore actual location (le-name and line no) can be logged using the debugging information from binary.

3) *Instrumentation for memory leakages.*

To detect memory leak on function level for that instrumentation is used. All the memory related functions like malloc(), calloc(), realloc() and free() are instrumented using the technique method wrappers. These function calls are instrumented by registering callback methods before and after every function call. Appropriate number of parameters is passed to the callback methods. Also, to start instrumentation, main () function is also wrapped. Figure 1 shows the program flow due to instrumentation.

4) *Log Analysis*

To analyze the log, an allocation table is to be maintained reflecting all details like address, size, and location about each memory allocation. Entries are added for logs generated by malloc() and calloc(). Entries are marked as "freed" for logs generated by free() for matching the memory address. After building data structures for the entire log, search for the entries which are not marked as "freed" and declare them as memory leaks To avoid program crash due to invalid free(), analysis of free() should be done while instrumenting only. Statistical analysis includes the percentage of memory allocation which was freed during program run, total memory leakage in bytes, distribution of memory leaks within multiple files, etc.

B. Uninitialized Variable Detection

Instrumentation should collect detailed data about each variable which may consist of location where variable is declared, storage type and default values, use and define points of it along the path. In case of arrays, only variable name is not sufficient, index is also required to be logged. For composite variables, name of variable along with the internal name of the field is necessary to locate the exact uninitialized variable. If a variable is written (initialized) through pointer, then the pointer pointing to it must also be logged.

1) *Instrumentation for uninitialized variable detection*

To detect uninitialized variables, one way is to track all the reads and writes happening in the program. If source code instrumentation is used, instrumenting code (extra code to log the read/write) has to be added at each type of assignment. If binary instrumentation is used, individual instruction is to be instrumented to check whether it is reading or writing to some variables, arrays, composite variables (records/structures) and pointers. Also scope of variables should be taken into consideration. To detect un-int variable at source level, both the source code and binary code instrumentation techniques are used. Source code instrumentation allows finding out all the variable names used in the program as well as their addresses. On the other hand, binary instrumentation allows to keep a track of every read/write happening to those variables.

2) *Log Analysis for Uninitialized variable*

To detect uninitialized variables, first a table of initialization is maintained having entries of all the variables. Global variables and other variables with default values are marked as initialized at start. When a variable is written, it is also marked as initialize. If a variable is read before getting written, it is uninitialized. The report generated should contain all information about the fault to make it easier for developer to fix the fault.

V. Dynamic Slicing

A. *Executable Dynamic Slices*

An executable dynamic slice is a slice that can be executed and it preserves a value of a variable of interest. An executable dynamic slice of program P on slicing criterion C is any syntactically correct and executable program P that is obtained from P by deleting zero or more statements and when executed on program input x produces an execution trace Tx for which there exist the similar execution position q such that the value of yq in Tx equals the value of yq′ ' in Tx′' The existing algorithms for the computation of dynamic slices can be classified as trace-based (execution trace) algorithm and non-trace based algorithms.

1) *Execution trace based*

In the trace based algorithms an execution trace of the program is first recorded and the computation of a dynamic slice is performed on the recorded execution trace. During recording of the execution trace different types of information is recorded depending on the algorithm. Typically, an execution trace contains information about the statement executed and variables defined or used at this statement (especially, variables used or defined that cannot be determined by static analysis, e.g. addresses of array elements used or defined).

a) *Program dependence graph based algorithm*

This is precise algorithm for dynamic slicing. The algorithm uses the concept of data and control dependencies to compute dynamic program slices. Dynamic dependencies between actions are captured by two types of dependencies: data dependence and control dependence. The data dependence captures the situation where one action (node) assigns a value to an item of data and the other action (node) uses that value. Control dependence captures the dependence between test nodes and nodes that have been chosen to be executed by these test nodes. The program dependence graph of a program has one node for each simple statement(assignment, read, write etc., as opposed to compound-statement like if-then-else, while-do etc.) and one node for each control predicate expression (the condition expression in if-then-else, while-do etc.) and two types of directed edges for data dependence and control-dependence.

Once a program is executed and its execution trace collected, precise dynamic slicing typically involves two tasks: pre-processing which builds dependence graph by recovering dynamic dependences from the programs execution trace and slicing which computes slices for given slicing requests by traversing the dynamic dependence graph. In Zhang and Gupta proposed three precise dynamic slicing algorithms that differ in the degree of preprocessing they carry out prior to computing any dynamic slices. The full preprocessing (FP) algorithm builds the entire dependence graph before slicing. The no preprocessing (NP) does not perform any preprocessing but rather during slicing it uses demand driven analysis for recovering dynamic dependencies and caches the recovered dependencies for potential future reuse. Finally the limited preprocessing (LP) algorithm performs some preprocessing to first increase the execution trace and then during slicing it uses demand driven analysis to recover the dynamic dependences from the compacted execution trace.

b) *Dynamic Dependence Graph*

Hiralal and Horgon proposed concept of Dynamic Dependence Graph(DDG) in which node is created for each occurrence of a statement in the execution history, with outgoing dependence edges to only those statements (their specific occurrences) on which this statement occurrence is dependent Reference number, as in [18]. Every node in the new dependence graph will have at most one outgoing edge for each variable used at the statement. Once we have constructed the dynamic dependence graph for the given execution history, we can easily obtain the dynamic slice for a variable, var, by first finding the node similar to the last definition of var in the execution history, and then finding all nodes in the graph reachable from that node. Dynamic Dependence Graph for the program in following figure the test case (N=3, X=-4, 3, -2). Nodes in bold give the Dynamic slice for this test-case with respect to variable Z at the end of execution.

```
Begin
        S1:     read(N);
        S2:     I=1;
        S3:     While(I<=N)
                do
        S4:     Read(X);
        S5:     If(X<0)
                then
        S6:             Y=f1(X);
                else
        S7:             Y=f2(X);
                end_if;
        S8:     z:=f3(Y);
        S9:     WRITE(z);
        S10:    I:=I+1;
                end_while;
                end
```

Fig. 2. Example

The size of a Dynamic Dependence Graph (total number of nodes and edges) is, in general, unbounded. This is because the number of nodes in the graph is equal to the number of statements in the execution history, which, in general, may depend on values of run-time inputs. So instead of creating a new node for every occurrence of a statement in the execution history, create a new node only if another node with the same transitive dependencies does not already exist. This new graph is known as the Reduced Dynamic Dependence Graph (RDDG). The Reduced Dynamic Dependence Graph for the Program in Figure 2 for the test case (N=3, X=-4, 3, -2) is shown in Fig. 4.

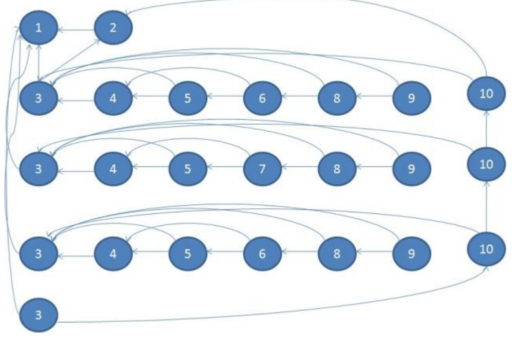


Fig. 3. Dynamic Dependence Graph for figure 2

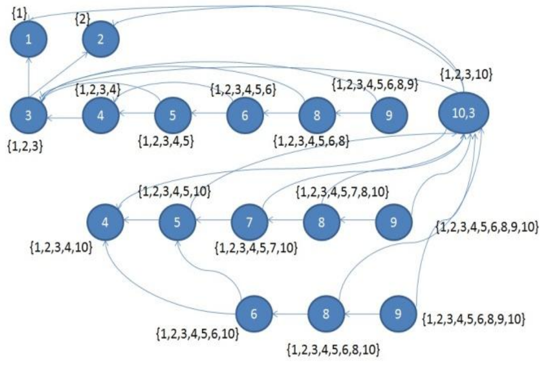


Fig. 4. Reduced DDG for Fig. 3

2) *Non-execution Trace Based*

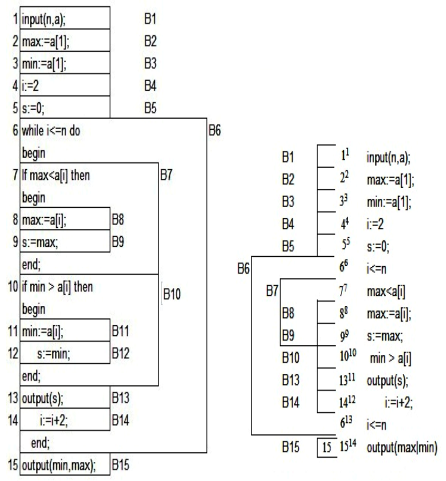
In non-trace based algorithms an execution trace is not recorded and dynamic slices are computed during program execution.

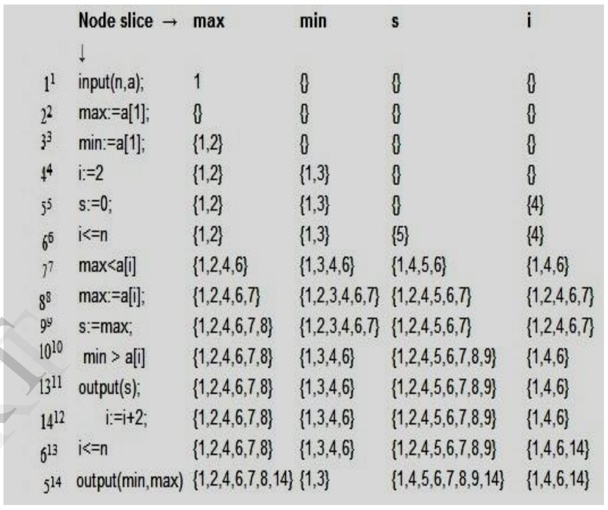
a) *Forward Algorithm*

The forward algorithm starts from the first statement in the program and proceeds “forward” with program execution and at the same time performs the computation of dynamic slices for program variables along with the program execution. The following are two sample condition used by the forward algorithm to compute dynamic slices during program execution.

1. If value of variable v has not been modified during execution of block B, and none of the statements executed inside of block B belong to the dynamic slice of variable v, the block is not included in the dynamic slice for variable v at the exit from B.
2. A dynamic slice for a variable modified by an assignment is a union of dynamic slices of all variables used at the assignment. Refer fig. 5 for example

Fig. 5. Block Diagram


```
1  input(n,a);        B1
2  max:=a[1];         B2
3  min:=a[1];         B3
4  i:=2               B4
5  s:=0;              B5
6  while i<=n do      B6
   begin
7  If max<a[i] then   B7
   begin
8  max:=a[i];         B8
9  s:=max;            B9
   end;
10 if min > a[i] then B10
   begin
11 min:=a[i];         B11
12    s:=min;         B12
   end;
13 output(s);         B13
14    i:=i+2;         B14
    end;
15 output(min,max);   B15
```



	Node slice → ↓	max	min	s	i
1^1	input(n,a);	1	{}	{}	{}
2^2	max:=a[1];	{}	{}	{}	{}
3^3	min:=a[1];	{1,2}	{}	{}	{}
4^4	i:=2	{1,2}	{1,3}	{}	{}
5^5	s:=0;	{1,2}	{1,3}	{}	{4}
6^6	i<=n	{1,2}	{1,3}	{5}	{4}
7^7	max<a[i]	{1,2,4,6}	{1,3,4,6}	{1,4,5,6}	{1,4,6}
8^8	max:=a[i];	{1,2,4,6,7}	{1,2,3,4,6,7}	{1,2,4,5,6,7}	{1,2,4,6,7}
9^9	s:=max;	{1,2,4,6,7,8}	{1,2,3,4,6,7}	{1,2,4,5,6,7}	{1,2,4,6,7}
10^{10}	min > a[i]	{1,2,4,6,7,8}	{1,3,4,6}	{1,2,4,5,6,7,8,9}	{1,4,6}
13^{11}	output(s);	{1,2,4,6,7,8}	{1,3,4,6}	{1,2,4,5,6,7,8,9}	{1,4,6}
14^{12}	i:=i+2;	{1,2,4,6,7,8}	{1,3,4,6}	{1,2,4,5,6,7,8,9}	{1,4,6}
6^{13}	i<=n	{1,2,4,6,7,8}	{1,3,4,6}	{1,2,4,5,6,7,8,9}	{1,4,6,14}
5^{14}	output(min,max)	{1,2,4,6,7,8,14}	{1,3}	{1,4,5,6,7,8,9,14}	{1,4,6,14}

Fig. 6. Forward Algorithm Result

b) *Static Program Dependence Based Algorithm*

In this algorithm the program dependence graph of the program is first derived. The program dependency graph captures the data and control dependencies in the program. During program execution the edges (representing data or control dependence) of the program dependence graph that occurred during program execution are marked. After the program execution, the algorithm traverses the program dependence graph only along the marked edges to find a dynamic slice as in figure: 7.

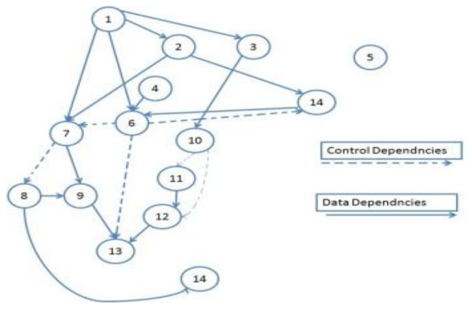
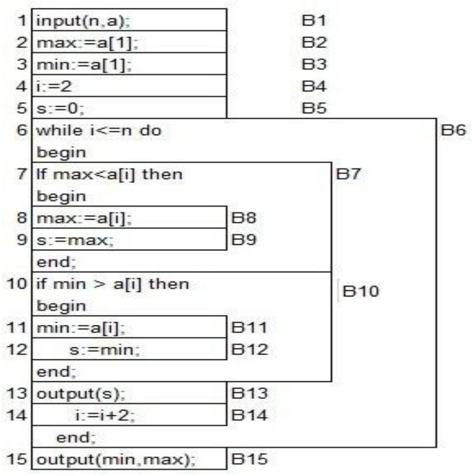


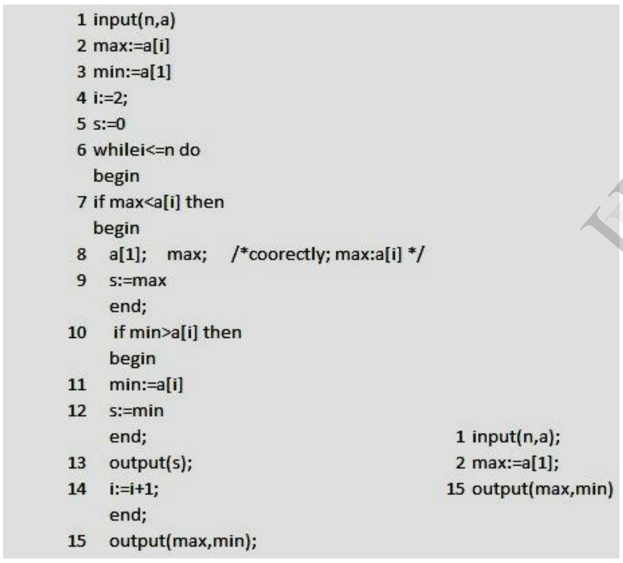
Fig. 7. Static Algorithm



```
1  input(n,a);
2  max:=a[1];
3  min:=a[1];
4  i:=2
5  s:=0;
6  while i<=n do
   begin
7  If max<a[i] then
   begin
8  max:=a[i];
9  s:=max;
   end;
10 if min > a[i] then
   begin
11 min:=a[i];
12     s:=min;
   end;
13 output(s);
14     i:=i+2;
     end;
15 output(min,max);
```

B. Issues in Implementation of Non Executable Dynamic Slice

For a given slicing criterion C=(x,yq), a non-executable dynamic slice contains statements that “influence” the variable of interest Yq during program execution on input x. There is no hypothesis about the execution of the dynamic slice and preservation of the value of variable y. In most cases non-execution dynamic slices cannot be executed. Figure 8 shows a non-executable trace.



```
1 input(n,a)
2 max:=a[i]
3 min:=a[1]
4 i:=2;
5 s:=0
6 whilei<=n do
   begin
7 if max<a[i] then
   begin
8    a[1];   max;   /*coorectly; max:a[i] */
9    s:=max
     end;
10    if min>a[i] then
     begin
11   min:=a[i]
12   s:=min
     end;                           1 input(n,a);
13   output(s);                     2 max:=a[1];
14   i:=i+1;                       15 output(max,min)
     end;
15   output(max,min);
```

Fig. 8. Non Execution Trace

C. Implementation of Program Slicing

Approach III (DDG) and Approach IV (RDDG) reference number, as in [14] and are stated as approach 1 and approach 3 in Algorithm section. In Addition, we developed modified algorithm called xDDG which is better than DDG in terms of time of compute slice and no. Of nodes, and is better than RDDG in terms of memory of dependence graph.

1) Problem with WET

The Whole Execution Trace (WET) is a unified representation that holds full execution history including, control flow, value, address and dependence (data and control) histories. WET is essentially a static representation of the program that is labeled with the dynamic profile information. This provides a direct access to all of the relevant profile information associated with every execution instance of every statement.

If conditional statement is present in program, WET is not in order of execution of statements. During conditional statements in program, the order of trace is as follows:

i) First trace is of first function statement.
ii) Next trace is of statement next to conditional statement then of statement next to if-then-else.
iii) After that there is trace of 1st statement of else part.
iv) Then the trace of remaining statement starts from beginning.
v) Depending on the first condition, first statement in true block reappears in trace. While the false block statement doesn’t.

That is why we need to consider the trace in which the dependencies node is yet to occur. We need to keep a list of visited nodes otherwise we will go into infinite loop because of problem with WET.

2) Algorithms

Here are the algorithms to construct dependence graphs from execution trace and get dynamic slice. The three algorithms differ in terms of dependence graph.

a) Approach 1 (DDG):

i) Using Whole Execution Trace, we construct nodes.
ii) Dependencies for each node in trace are stored. Thus, Dynamic Dependence Graph (DDG) is obtained.
iii) To compute dynamic slice for given statement, we traverse constructed DDG from given trace until all dependencies are resolved. The list of all visited nodes is out-putted as dynamic slice.

b) Approach 2 (xDDG): We have designed this algorithm. This is possible only because node of dependence graph is considered as tuple instead of line number.

i) Initially, construct nodes same as in DDG using WET.
ii) Check every time whether new constructed node has same dependencies as earlier nodes.
iii) If yes, combine the nodes.
iv) After the last trace, extended DDG (xDDG) is constructed.
v) Start traversing nodes from the similar trace no. Of given statement.
vi) Visit each node and similar dependency nodes.
vii) Keep a list of visited nodes.
viii) For each node repeat step 6 only if the node is not in visited nodes.

c) Approach 3 (RDDG): Same as xDDG algorithm, only difference is:

i) Instead of just storing the dependencies, we store the whole slice for each similar node.
ii) So we also need to maintain a list of nodes whose dependencies are yet to occur.
iii) And only those nodes present in the slice of required node and above list are resolved.

We need to maintain a visited nodes list during resolution to prevent infinite loop.

Example:

```
 #include<stdio.h>
 int main()
{
int i, x, z, n=3;
i=0;
while(i<n){
scanf("%d",&x);
if(x<0){
y=x+x;
} else {
y=x*x;
}
z=y+2;
printf("%d",z);
i=i+1;
}
}
```

Fig. 9. A C example

VI. CONCLUSIONS

Code instrumentation using Dynamic Program Analysis tools helps to reduce development effort and eases extension. i.e. pin tool is easy to use and handle inserting the code into the running program application code and attach or detach to running application process. You can follow each instruction, function, system call, etc. By using Pin you have a complete control over the dynamically executing program. Therefore, you can use it in various tasks. Like in c/c++ programming and attach or detach to running application process. And improve the whole performance of application process by changing its behavior by inserting the code dynamically on runtime application program.

Program slicing is a useful and needful technique for today's programming world, because consuming memory by executables is very much loss of time and decrease in speed also. Therefore programming slicing algorithm Like DDG in terms of number of nodes and time to compute dynamic slice, RDDG in terms of memory of dependence graph are applicable. For programs of with very long time executions the forward approach of dynamic slice computation has been proposed Reference number as in [11].

REFERENCES

[1] http://software.intel.com/sites/landingpage/pintool/docs/49306/Pin/html/

[2] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, StevenWallace Vijay, Janapa Reddi, Kim Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", Intel Corporation, University of Colorado.

[3] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, Moshe Bach, "Dynamic Program Analysis of Microsoft Windows Applications", Intel Corporation, University of Virginia.

[4] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal,"Analyzing Parallel Programs with Pin", intel.

[5] Patrick Wood, "A Survey of Performance Analysis Tools", pwood@wustl.edu.

[6] Quan Jia, "Introduction to the Pin Instrumentation Tool", George Mason University, Mar. 27 2013.

[7] Prashanth P. Bungale, Chi-Keung Luk, "PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation", Intel Corporation, Hudson, MA and Harvard University, Cambridge, MA, Intel Corporation, Hudson, MA.

[8] Aamer Jaleel, Chi-Keung Luk, Bobbie Manne, Harish Patil "Using the Pin Instrumentation Tool for Computer Architecture Research", Intel® Corporation, June 17, 2006.

[9] http://www.sciencedirect.com/science/article/pii/S0950584998000895

[10] http://www0.cs.ucl.ac.uk/staff/mharman/exe1.html

[11] Bogdan Koral, Jurgen Rilling, "Application of Dynamic Slicing in Program Debugging", Department of Computer Science Illinois Institute of Technology, Chicago, IL 60616, USA.

[12] Xiangyu Zhang and Rajiv Gupta, "Whole Execution Traces", In proceeding of IEEE/ACM 37th International symposium on Microarchitecture, pages 105-116, Portland, Oregan, December 2004.

[13] Klaus Havelund, "Dynamic Program Analysis", Kostrel Technology, Palo Alto, California, USA.

[14] http://resources.infosecinstitute.com/pin-dynamic-binary-instrumentation-framework/

[15] Aditya Khamparia, Saira Banu J., "Program Analysis with Dyanmic Instrumentation Pin and Performance Tools", Computer Science and Engineering Institute of Technology, SCSE, Vellore, India.

[16] Anthony M. Sloane, "Generating Dynamic Program Analysis Tools", Department of Computer Science, James Cook University, QLD, Australia, tony@cs.jcu.edu.au.

[17] Herbert Ritch, Harry M. Sneed, "Reverse Engineering via Dynamic Analysis", Software Engineering Service, D-8012 Ottobrunn, Germany.

[18] Hiralal Agrawal and Joseph R. Horgan, "Dynamic Program Slicing", In proceeding of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI 90, PP.246-256, (1990)).

[19] http://onlinelibrary.wiley.com/doi/10.1002/swf.41/full

[20] Markus Mock, Darren C. Atkinson, "Improving Program Slicing with Dynamic Points-To Data", Department of Computer Science & Engineering , University of Washington.

[21] Jiri Slaby, Jan Strejeck, and Marek Tritik, "Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution", Faculty of Informatics, Masaryk University Botanicka 68a, 60200 Brono, Czech Republic.

[22] Tevi Devor, "Pin: Intel's Dynamic Binary Instrumentation Engine Pin Tutorial", Intel Corporation.

[23] Bas Cornelissan , Andy Zaidmain, Arie van Deursen, Leon Moonen, Rainer Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis", Student, Member IEEE, Member IEEE Computer Society.

[24] Markus Denker, Orla Greevy, Michele Lanza, "Higher Abstraction for Dyanamic Analysis", Software Composition Group, University of Berne, Switzerland, Faculty of Informatics University Lugano Switzerland.

[25] Mayur Naik, " Static and Dyanamic Program Analysis : Synergies and Applications", Intel Labs Berkely, CS 243, Standford University, March 9, 2011.

[26] Walter Binder, Philippe Moret, Danilo Ansaloni, Aibek Sarimbekov, Akira Yokokawa, eric Tanter, "Towards a Domain-Specific Aspect Language for Dynamic Program Analysis", Faculty of Informatics University of Lugano-Switzerland, PLEIAD Laboatory, Computer Science Department(DCC), University of Chile-Chile.

[27] Zhiqiang Lin, "CS 6V81-05: System Security and Malicious Code Analysis Dynamic Binary Instrumentation", Department of Computer Science, University of Texas at Dallas, january 30th, 2012.

[28] Walter Binder, "Higher-Level Abstractions for Instrumentation-based Dynamic Program Analysis", Dynamic Analysis Group, Faculty of Informatics University of Lugano.

[29] Nicholas Nethercote, “Dynamic Binary Analysis and Instrumentation”, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, United Kingdom, PP. 43-48, November 2004.

[30] Xingyu Zhang, Rajiv Gupta, “Whole Execution Traces”, The Unversity of Arizona, Department of Computer Science Tucson, Arizona 85721.

[31] Xingyu Zhang, Rajiv Gupta , “Cost Effective Dynamic Programming Slicing”, Department of Computer Science, The University of Arizona Tucson, Arizona 85721.