Incremental Interprocedural Pointer Analysis

(M. Tech Project stage II)

Submitted in partial fulfillment of the requirements

of the degree of

Master of Technology

by

Prachee Yogi
(10305077)

under the guidance of

Prof. Uday Khedker

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

June '12

Dissertation Approval Certificate

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

The dissertation entitled **"Incremental Interprocedural Pointer Analysis"**, submitted by **Prachee Yogi** (Roll No: **10305077**) is approved for the degree of **Master of Technology** in **Computer Science and Engineering** from **Indian Institute of Technology, Bombay**.

[illegible]

Prof. Uday Khedkar
Dept CSE, IIT Bombay
Supervisor

[illegible]

Prof. S. Biswas
Dept CSE, IIT Bombay
Internal Examiner

[illegible]

Dr. Bageshri Sathe
Nvidia, Pune
External Examiner

[illegible]

Prof. A Subhramanyam
Dept Maths, IIT Bombay
Chairperson

Place: IIT Bombay, Mumbai
Date: 22nd **June, 2012**

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Prachi

Signature

PRACHEE YOGI

Name of Student

10305077

Roll number

28/06/2012

Date

Acknowledgement

I hereby express my sincere thanks and gratitude towards my Guide Prof. Uday P. Khedker for giving me an opportunity to work on such an interesting project and for his constant help, encouragement and invaluable guidance in all phases of the project. Meetings with him have been a constant source of ideas, and gave me motivation for the project.
I would also like to thanks my family. My parents have always been a source of motivation for me. They have always supported me in stressful situation.
I am extremely thankful to all my friends and people in GRC lab who were always there to discuss with me all my queries. I would like to thank each and every person who have some how contributed to my work and without whom this would not have been possible.

Abstract

Most of the work done till now in pointer analysis sacrifices either flow sensitivity or context sensitivity. For performing pointer analysis precisely, both of them are required. L-FCPA is an approach which performs the pointer analysis obeying both the dimensions. Our measurements done using Spec benchmark shows effective improvement in the pointer analysis by using the liveness based approach. While performing the analysis, most of the information remains same as that of previous iteration. Efforts are wasted in performing the analysis for the same values in further iterations. This can be avoided by propagating only the changes in subsequent iterations so that the pointer analysis can be made faster. This is known as incremental analysis. In this project report, we introduce incremental formulation of liveness based pointer analysis approach.

Contents

1 Introduction **1**

1.1 Motivation 2

1.1.1 Need for Pointer Analysis 2

1.1.2 Issues with Pointer Analysis 2

1.2 Goals and Deliverables 4

1.3 Organization of Report 5

2 Literature Survey **6**

2.1 Points-to Analysis 6

2.1.1 Flow Sensitive Analysis 6

2.1.2 Context Sensitive Analysis 7

2.2 Emami's Pointer Analysis 8

2.2.1 Terminologies Used 8

2.2.2 Pointer Analysis Algorithm 9

2.2.3 Example 10

2.3 Summary 16

3 Liveness Based Pointer Analysis **17**

3.1 Terminologies Used 17

3.2 Data Flow Equations 18

3.3 Prototype Implementation in Prolog 21

3.3.1 Working Example 23

3.4 Empirical Measurement 24

3.4.1 Benchmark Programs 24

3.4.2 Time Measurement 25

3.4.3 Measurement of pointer pair information 26

3.4.4 Measurement of Call string information 26
3.4.5 Summary 27

4 APIs for Liveness Based Pointer Analysis Code 28
4.1 Data-structure used for storing information 28
4.2 Difficulties in writing API for current code 30
4.3 API design 30
4.4 API for pointer information 31
4.4.1 APIs at Vector level 31
4.4.2 APIs at basic block and callstring level 34
4.4.3 APIs at basic block 37
4.5 API for liveness information 40
4.5.1 API at vector index level 40
4.5.2 API at basic block and callstring level 41
4.5.3 API at basic block 42
4.6 Measurements 43

5 Incremental Pointer Analysis 46
5.1 Incremental Analysis 46
5.1.1 Across different iterations in a fixed point computation 46
5.1.2 Across different applications of an analysis 47
5.1.3 Across different fixed point computations of same application 48
5.2 Motivation for incremental pointer analysis 48
5.3 Dataflow Equations 52
5.3.1 Incremental Liveness equation 53
5.3.2 Issues with incremental pointer analysis 53
5.3.3 Ensuring distributivity of incremental points-to analysis 54
5.3.4 Incremental Points-to Analysis equation 54
5.3.5 Working Example 55
5.4 Interprocedural incremental pointer analysis 58
5.5 Implementation 60
5.6 Summary 61

6 Conclusion and Future Work 62
6.1 Conclusion 62

6.2 Future Work 63

A Incremental interprocedural L-FCPA example 64

Chapter 1

Introduction

Pointer Analysis requires computing pointer and pointee information in a program. Pointer Analysis is important as it is required by other data flow analysis. Interprocedural pointer analysis performs the analysis by propagating information between caller-callee procedures and vice-versa. The precise pointer information makes other program analysis more effective. Two important dimensions of pointer analysis precision are *Context Sensitivity* and *Flow Sensitivity.*

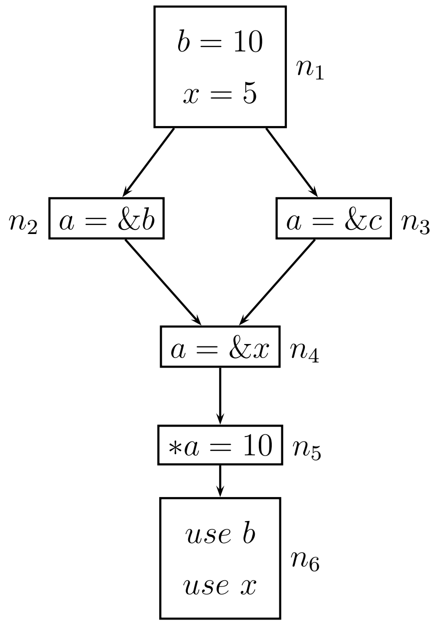
- **Context Sensitive :** It analyzes the data flow information based on the calling context of the procedure whereas context-insensitive analysis merges contexts together, i.e. It does not distinguish between information coming from various procedure call.
- **Flow Sensitive :** A flow-sensitive analysis computes the data flow information at each program point according to the control-flow of a program, whereas a flow-insensitive analysis computes a single solution that conservatively holds for the entire program.

In this report we are dealing with a new approach of pointer analysis for stack data. Pointer analysis is important because pointers allow indirect access to data items. Therefore, pointer information is required for performing other dataflow analysis precisely. Section 1.1 has few examples which show the importance of the pointer analysis while doing other data flow analysis.

1.1 Motivation

1.1.1 Need for Pointer Analysis

In this section we will discuss importance of pointer analysis and its effect on other analysis. The following example shows how pointer information affects other dataflow analysis. In this example, we want to perform constant propagation.



- If constant propagation is performed without considering the pointer information then we get value of x as 5.
- If pointer information, $a \rightarrow x$ is considered, then statement **a = 10* will modify the value of variable x and we get value of x as 10.
- If we have flow-insensitive information ($a \rightarrow \{b,\ c,\ x\}$), then value of b will be modified, which may lead to imprecise constant propagation.

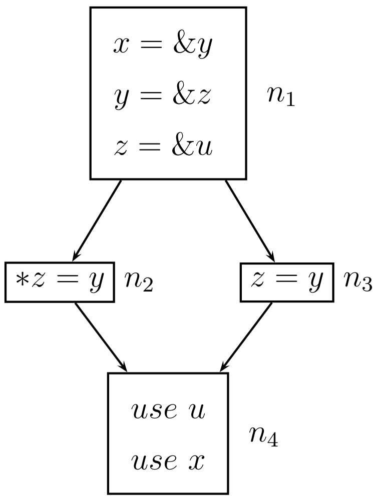
Figure 1.1: Motivating example for importance of pointer analysis

From the above example, we can conclude that precise pointer information is required to perform other data flow analysis correctly.

1.1.2 Issues with Pointer Analysis

Flow and context sensitive pointer analysis gives precise pointer information. Performing pointer analysis with both dimensions become expensive when we have large size of pointer information, thereby affecting the scalability of analysis. All the flow and context sensitive information computed is not useful. Consider the following examples.

- Example 1



Node	Pointer Information	
	In_n	Out_n
1	$\emptyset$	$\{(x,y)(y,z)$ $(z,u)\}$
2	$\{(x,y)(y,z)$ $(z,u)\}$	$\{(x,y)(y,z)$ $(z,u)(u,z)\}$
3	$\{(x,y)(y,z)$ $(z,u)\}$	$\{(x,y)(y,z)$ $(z,z)\}$
4	$\{(x,y)(y,z)$ $(z,u)(u,z)$ $(z,z)\}$	$\{(x,y)(y,z)$ $(z,u)(u,z)$ $(z,z)\}$

Figure 1.2: Motivating example for Liveness based pointer analysis

In the above example, we can observe the following

- The computation of pointer information of variable z at node n3 is unnecessary.
- Computing confluence for pointer information of variable z *and* y is not required at the *In* of node n4.
- Propagating pointer information of variables at out of node n4 is redundant.

Considering pointer information for the variables that are live at a program point can reduce the number of computations done during the analysis. In other words,if we compute pointer information only for the variables that are being used, then number of pointer pairs can be reduced.

- Example 2
Consider the example in Figure 1.3, during second iteration, pointer pair (t,u) and (c,d) are added to the points-to set at in of node n_2. At node n_3, we are modifying pointee of c. Therefore, at the in of node n_3 we require only pointer information of variable c. Computing pointer pair (a,b) is redundant in second iteration. The information generated in the current iteration which was not present in the previous iteration is known as incremental value. Figure 1.3(B) shows the incremental value.

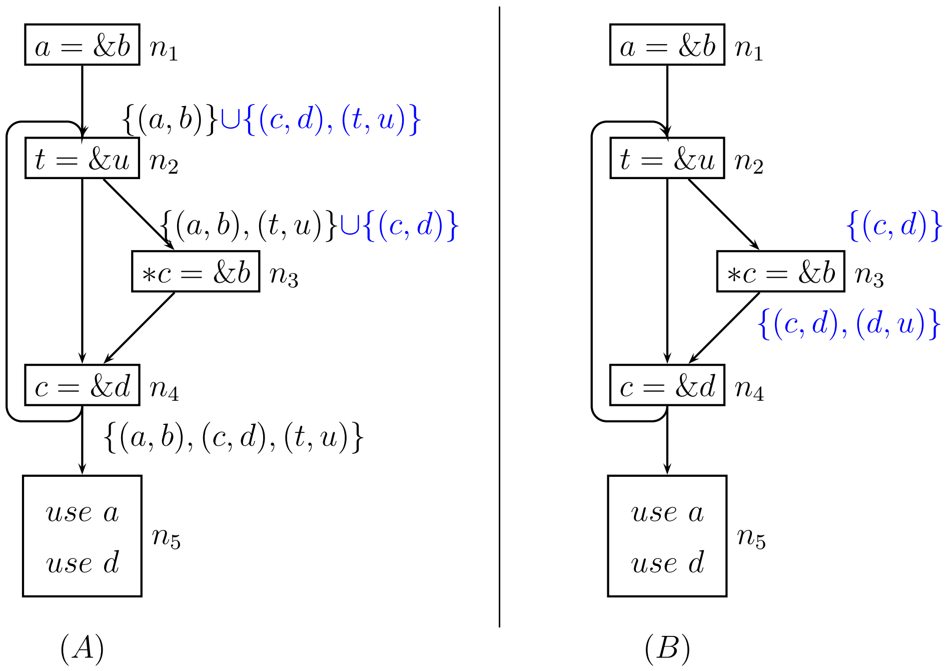


Figure 1.3: Motivating example for Incremental Pointer Analysis

Therefore, propagating the incremental value during the analysis can further reduce the number of computations that are done at a basic block.

1.2 Goals and Deliverables

This project is a part of research which aims at performing the context and flow sensitive pointer analysis for million lines of code efficiently. The Liveness Based Pointer Analysis is implemented in GCC by Prashant. The current goal of this research is as follows :

- Goals
  - To perform empirical measurements on SPEC benchmark for the performance measurement of current implementation of L-FCPA.
  - Implement a prototype model of Intraprocedural Liveness Based Pointer Analysis in prolog.
  - Design and Implement the APIs for current implementation and rewrite the code using these APIs.
  - Simple implementation of Incremental L-FCPA algorithm in GCC.

- Future Goals
  - Design and implementation efficient data structure for storing pointer information.
  - Extending the algorithm for heap variables.
  - Including the L-FCPA in GCC, such that results of L-FCPA can be used by other optimization passes.
  - Scope of improvement in the current implementation of APIs.
  - Improving the incremental L-FCPA implementation to work for large size codes.
  - Study of incremental analysis across different fixed point computation.

1.3 Organization of Report

The rest of the report is organized as follows. Chapter 2 describes about some of the approaches of performing pointer analysis. Chapter 3 describes about a new method of pointer analysis, i.e. Liveness Based Pointer Analysis(L-FCPA) and its performance on SPEC2000 benchmark programs. Chapter 4 contains design of APIs for accessing the data-structure used for storing pointer and liveness information. Chapter 5 encloses the introduction to incremental L-FCPA and its incremental formulation. Chapter 6 includes conclusion and future work.

Chapter 2

Literature Survey

This section contains some of the approaches of performing pointer analysis and some issues related with the approaches. It also covers some basic terminologies used with pointer Analysis

2.1 Points-to Analysis

As discussed earlier two dimensions to perform pointer analysis precisely are : flow sensitivity and context sensitivity.

2.1.1 Flow Sensitive Analysis

Flow Sensitive analysis computes points-to pair for each program point. The pointers to which value are assigned comprises of left location denotes as lhs_n, and the addresses assigned comprises of right location rhs_n. Therefore, for statement such as $lhs_n = rhs_n$ we have[4] :
Here, X represent the incoming pointer information of node n, i.e. information coming from its successors and the right locations of n which depend on X are denoted by $DepLeftL_n(X)$ and $DepRightL_n(X)$. The local effect of n are denoted by $ConstLeftL_n$ and $ConstRightL_n$.

Using these equations we can perform :

May points-to information A pointer variable may point to different addresses along different incoming paths.

Left Locations			*Right Locations*		
lhs_n	$ConstLeftL_n$	$DepLeftL_n(X)$	rhs_n	$ConstRightL_n$	$DepRightL_n(X)$
x	$\{x\}$	ϕ	x	ϕ	$\{y \mid (x \rightarrow y) \in X\}$
$*x$	ϕ	$\{y \mid (x \rightarrow y) \in X\}$	$*x$	ϕ	$\{z \mid \{x \rightarrow y, y \rightarrow z\} \subseteq X\}$
			$\&x$	$\{x\}$	ϕ

$$\begin{aligned}
ConstGen_n &= \{x \rightarrow y \mid & & x \in ConstLeftL_n, y \in ConstRightL_n\} \\
DepGen_n(X) &= \{x \rightarrow y \mid & & (x \in ConstLeftL_n, y \in DepRightL_n(X)), \text{ or} \\
& & & (x \in DepLeftL_n(X), y \in ConstRightL_n), \text{ or} \\
& & & (x \in DepLeftL_n(X), y \in DepRightL_n(X))\} \\
ConstKill_n &= \{x \rightarrow y \mid & & x \in ConstLeftL_n\} \\
DepKill_n(X) &= \{x \rightarrow y \mid & & x \in DepLeftL_n(X)\}
\end{aligned}$$

Must points-to information A pointer points to an address along all the incoming paths.

2.1.2 Context Sensitive Analysis

Caller-callee Context is maintained along with the pointer information when performing interprocedural analysis. There are 2 approaches of performing interprocedural dataflow analysis[4]:

- **Functional Approach :** Functional Approach computes summary flow function for all the procedures in call graph. Every call to function is replaced by the summary function. The input to the function is the incoming data flow values coming from caller function. And the outcome of the flow function is the output of the callee.

- **Callstring Based Approach :** Function call sequence is stored in the form of call strings starting with main function. And value is stored along with the call string. For example, if we have a call string $c_1c_2...c_i$ at the start of function f_r is represented by σ, then value-based interprocedural data flow analysis is defined in terms of data flow values that are pair of the form <σ, x>at the calling context σ.

2.2 Emami's Pointer Analysis

The algorithm proposed by Emami is considered as one of the most context and flow sensitive analysis. They use Points-to Abstraction for doing pointer Analysis. In this approach we abstract a real stack location with an abstract stack location using an appropriate name [2]. For example if at a program point p we have a statement as $x = \&y$, then we say that stack location x points-to stack location y. If abstract location $\boldsymbol{x}$ definitely points to $\boldsymbol{x}$ for a calling context, then it is represented by a triplet **(x,y,D)** (similar to must points to in Section 2.1). If abstract location $\boldsymbol{x}$ possibly points to $\boldsymbol{x}$ for a calling context, then it is represented by a triplet **(x,y,P)** (similar to may points to in Section 2.1).

2.2.1 Terminologies Used

L-location is the stack location of variable being referred and **R-location** is the stack location pointed to by variable reference [2]. Following table shows the L-loc set and R -loc set for different variable references made in the program.

Var Ref	L-loc Set	R-loc Set
&a	-	(a,D)
a	(a,D)	(x,d)|(a,x,d) ∈ S
*a	(x,d)|(a,x,d) ∈ S	(y,d1⋀d2)| (a,x,d1)∈S ⋀ (x,y,d2)∈S

Table 2.1: L- location and R-location for points-to set

Invocation Graph represents the call chain in the program. It unfolds the recursive call node by introducing a cycle in the graph. Therefore, in invocation graph, there are 3 types of nodes : Ordinary node*(represents simple function call)*, Recursive node*(represents the first call to a recursive function)* and Approximate node*(represents the recursive call).*

In the example in Figure 2.1, we have a main function calling a recursive function. So invocation graph has a main as a root node and the leaf node(*f1-A*) represents the approximate node for recursive call to function f1. Approximate node and corresponding recursive nodes are connected via dashed line.

```
main()
{
  f1();
}

f1()
{
  if(cond)
     f1();
  else
     f2();
}
f2()
{
  print(...)
}
```

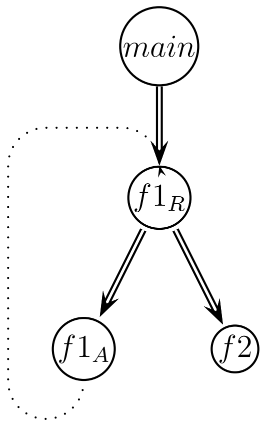


Figure 2.1: Invocation graph for a program

2.2.2 Pointer Analysis Algorithm

Intraprocedural Algorithm

Intraprocedural Algorithm use following definitions for computing points-to information [2].

- **kill_set** : Kills all the points-to set of the variable p if it has definite L-location.

$$kill\_set = \{(p,x,d) | \ (p,D) \in \ L\text{-}locations(lhs(S))\}$$

- **change_set** : If pointer variable p has possible relation in L-location of statement S and incoming information has definite relation with p. Then we change the relationship form definite to possibly points-to.

$$change\_set = \{(p,x,D) | \ (p,P) \in \ L\text{-}locations(lhs(S)) \bigwedge (p,x,D) \in Input\}$$
$$changed\_input = (Input - change\_set) \cup \{(p,x,P) | (p,x,D) \in change\_set\}$$

- **gen_set** : Generate all possible relationship between L-location and R-location.

$$gen\_set = \{(p,x,d1 \bigwedge d2) \mid (p,d1) \in L\text{-}locations(lhs(S)) \bigwedge (x,d2) \in R\text{-}locations(rhs(S))\}$$

$$Output = \{(changed\_input - kill\_set) \cup gen\_set\}$$

Interprocedural Algorithm

When a function call is made then the formal parameters of the caller are mapped with the incoming points-to relationships, global variables retain same points to relationship and Local pointer variables are initialized to NULL. Mapping information is recorded in invocation graph which act as context information for the calling context.
Once function is analyzed with the mapped input, unmapping of mapped value is performed at the call-site.
The nodes of invocation graph are processed in following way :

- **Ordinary node** : Each statement is processed using the equation described in previous section. Input to the function is the mapped input and once the function is analyzed values are unmapped.

- **Recursive node** : Fixed point computation is performed at each recursive node. At each recursive node, an input, an output and a pending list is maintained. The recursive node generalizes the stored input and output tills the value that summarizes all the invocation of recursive function is not obtained.

- **Approximate node** : It approximates the input values for processing the statements in a node. It returns BOTTOM in the output when the current input is the subset of the stored input else it add the current input in the pending list of the recursive node so as to approximate the input for the next computation.

As discussed earlier, this algorithm is considered as one of the most context sensitive algorithm. But we have some examples that proves the algorithm to be context insensitive.

2.2.3 Example

In the given example, pointer pair $\{(a,b)\}$ is generated in function p() and hence the information should only get propagated to inner calls to p(). But the algorithm propagates this information to the main() function also. Following section explains working of algorithm.

Example 1 :

```
    main()
    {
1      b = &c; c = &d;
2      p();
    }

    p()
    {
3      if()
       {
4         a = &b;
5         p();
6         a = *a;
      }
7
    }
```

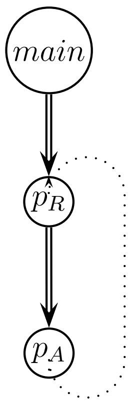


Figure 2.2: Example Program 1 and corresponding invocation graph

We traverse the invocation graph in depth first order and apply the algorithm explained in previous section on every function call. Therefore, first *main* function is invoked. It calls function *p*, which is the next node in invocation graph. Every recursive node is connected with its approximate node and vice-versa.

Let the recursive node p_R be represented as ign_R. Three sets are stored at recursive node. ign_R.storedInput, ign_R.storedOutput, ign_R.pendingList. Initially we have points-to pair (b,c,D) and (c,d,D) as the input to ign_R node. Therefore, Initially we have :
ign_R.storedInput = {(b,c,D),(c,d,D)}
ign_R.storedOutput = {}
ign_R.pendingList = {}
Let *funcInput* represent the input to the function. We have *if* statement as the first statement of the procedure so out of the statement is same as the *funcinput*.

Node	Output
3	{(b,c,D),(c,d,D)}
4	{(b,c,D),(c,d,D),(a,b,D)}
5(p_A)	{}
6	{}
7	{(b,c,D),(c,d,D)}

Table 2.2: Iteration 1 of the algorithm

Statement4 calls for an approximate node in invocation graph. At approximate node we approximate the input and output for the next function call, till the current input becomes equivalent to the input stored at the Recursive node. Therefore in first iteration we add the input to the approximate node in the pending list of corresponding recursive node. Hence, we have :
ign_R.pendingList = {(a,b,D),(b,c,D),(c,d,D)}
and it returns BOTTOM({}) as the output of the call to approximate node.

It checks the value in pending list. If pending list is NULL then it computes the output else it calls the recursive node with the following initialization for *Iteration 2*:
ign_R.storedInput = ign_R.pendingList = {(a,b,D),(b,c,D),(c,d,D)}
ign_R.storedOutput = {}
ign_R.pendingList = {}

In *iteration3*, when computing the output for the approximate node, we have current input equivalent to the stored input at recursive node. Therefore call to approximate node returns the unmapped output values. After the completion of *(*iteration3), pending list becomes empty and hence we start computing output by merging it with the stored output at recursive node and call the recursive function till the stored output becomes equivalent to the current output.
At the end of the analysis we have function output set as {(a,b,P), (a,c,P) ,(a,d,P), (b,c,D), (c,d,D)}. So these value is returned to the callsite of function *p()* in *main*, i.e. points-to information (a,b,P) is returned to main but this information is locally computed in the function *p()*. Therefore can be seen that the algorithm merges the context of recursive call

Node	Iteration 2	Iteration 3	Iteration 4	Iteration 5
	Output	Output	Output	Output
3	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$
4	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$
$5(p_A)$	$\{\}$	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$	$\{(a,b,P),(a,c,P)$ $(c,d,D),(b,c,D)\}$	$\{(a,b,P),(a,c,P),$ (a,d,P) $(c,d,D),(b,c,D)\}$
6	$\{\}$	$\{(a,c,D),(b,c,D)$ $(c,d,D)\}$	$\{(a,d,P),(a,c,P)$ $(c,d,D),(b,c,D)\}$	$\{(a,d,P),(a,c,P)$ $(c,d,D),(b,c,D)\}$
7	$\{(a,b,D),(b,c,D)$ $(c,d,D)\}$	$\{(a,b,P),(a,c,P)$ $(b,c,D),(c,d,D)\}$	$\{(a,b,P),(a,c,P)$ $(a,d,P),$ $(b,c,D),(c,d,D)\}$	$\{(a,b,P),(a,c,P)$ $(a,d,P),$ $(b,c,D),(c,d,D)\}$

Table 2.3: Points-to pair computation

and non recursive call to the function p() and therefore it returns {(a,b)} in the main also. We have another recursive program which shows context insensitivity of the algorithm.

Example 2 :

The Example program in Figure 2.4 performs Reversal of link list. We assume that there is a main function which calls the function *p()*. We are given a link list as shown below :

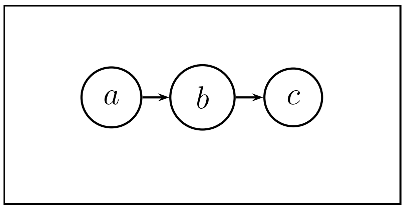

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \end{tikzcd}$$

Figure 2.3: Initial Linked List

Initially the head pointer points to the first element of the link list. We pass the original link list and head pointer information to the function *r()*. Pointer variable n points to the next element in the link list and variable p points to previous element in the link list. Only the information of head pointer and link list pointers are inherited from the caller function. But the algorithm returns the information of pointer p and n also to the caller.

```
    r()
1   {
2      n = *h;
3     *h = p;
4      p = h;
5      if(n ! = NULL)
6      {  h = n;
7         r();
8      }
9      else
10     { p = NULL;
11       n = NULL;
12     }
13     n = *h;
14     *h = p;
15     p = h;
16     h = n;
17  }
```

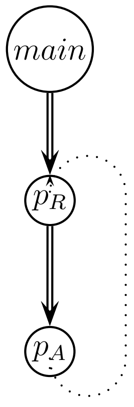


Figure 2.4: Example Program 2 and corresponding invocation graph

After function call to *r()*, we get points-to graph as shown in Figure 2.5. Initialy we have head *h* points-to *a*. After analysing statement 2 we have *n* points-to *b* and statement 3 modifies the points to relation between *a* and *b*. After statement 4 we have *p* points to *a* and *a* points to nothing. The points-to graph after applying algorithm once(1st Iteration) is shown in Figure 2.5.

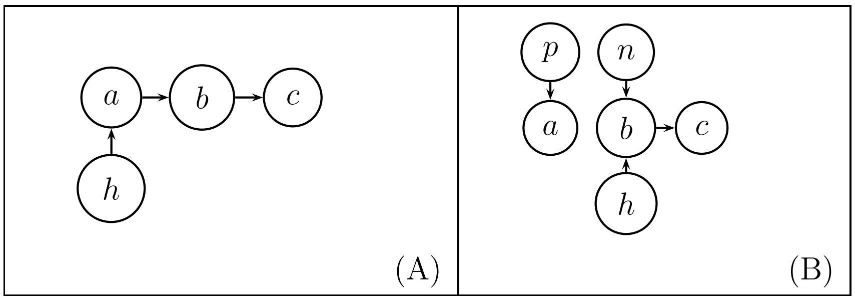


Figure 2.5: (A) Initial link list at start of r() (B)Link List after 1st iteration

So values obtained after applying algorithm successively on recursive node and approximate nodes are shown in Figure 2.6 :

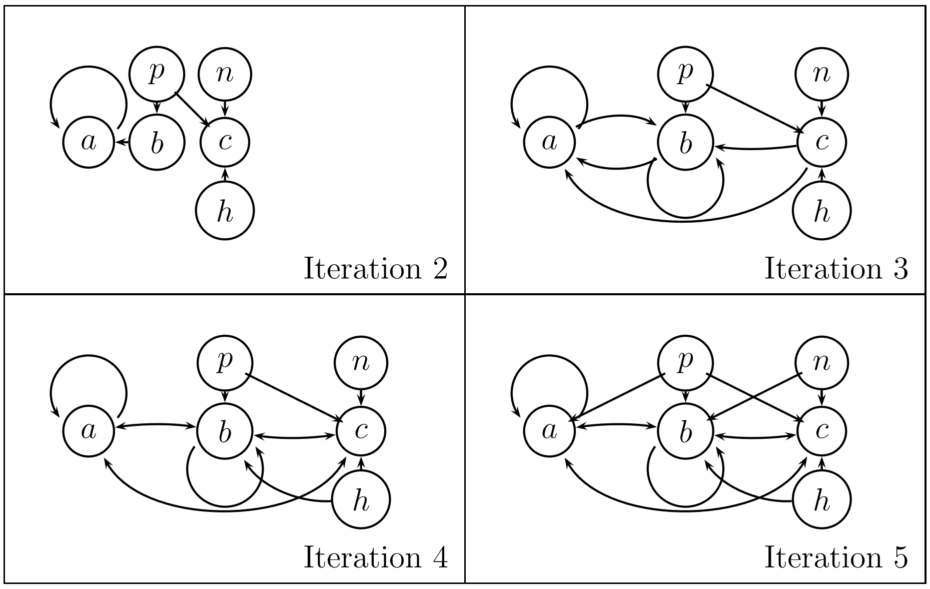


Figure 2.6: Function output after various Iterations

As it is seen from the program n and p within function body and therefore there values exist only for the inner calls to the function. But the result obtained show that function returns the points-to information of the variables n and p to the caller function *main*. From the above two examples it is proved that the emamis algorithm fails for recursive program. This is because it is not able to distinguish between the recursive call and non recursive call to the same function and hence returns the context insensitive information to the non recursive call node.

2.3 Summary

The two approaches discussed in the chapter fails to give efficient context and flow sensitive pointer analysis method. Simple points-to approach generate a large amount of context and flow sensitive information. It is very difficult to deal with large amount of information and hence the algorithm works inefficiently. Emamis approach is proved to be context insensitive as it is unable to distinguish between the calling context of recursive and non-recursive call made to same function. Also, most of the work done in pointer analysis are either context sensitive or flow sensitive.Hence, in the next chapter, we have discussed a new approach to pointer analysis, which is both context and flow sensitive.

Chapter 3

Liveness Based Pointer Analysis

Liveness Based Pointer Analysis is a flow and context sensitive pointer analysis approach abbreviated as L-FCPA. The main advantage of this approach is that we compute and propagate pointer information only for the variables that are live at a program point. Therefore, propagation of pointer information is done sparsely. This reduces the size of pointer information generated by flow and context sensitive pointer analysis. For computing liveness, strong liveness analysis is performed. Strong liveness is performing liveness in a more strict manner. Every variable that is read is not considered as live. Instead a read value is considered as live only when the variable which refers it becomes live. The methodology discussed in this chapter is from a research paper[3] which is recently accepted in SAS 2012.

3.1 Terminologies Used

This section includes some of the basic terminologies used for defining pointer analysis.
V denotes the variables and P represents the pointer variable such that $\mathsf{P} \subset \mathsf{V}$. A special non-pointer variable *?* represents an undefined location. Points-to information is a set of pointer-pointee pair *(x,y)* where $x \in \mathsf{P}$ and $y \in \mathsf{V}$. Pointer pair *(x,?)* represents that x points to some undefined location, i.e. it does not contain valid address.

Set Arithmetic

A relation R represent all possible pointer information, $R \subseteq \mathsf{P} \times \mathsf{V}$. Let X be some set, then we can define following operation [3] :

$$RX = \{v \mid u \in X \wedge (u,v) \in R\} \tag{3.1}$$

$$R|_X = \{(u,v) \in R \mid u \in X\} \tag{3.2}$$

$$R|_l^+ X = R|_X \cup R|_{RX \cap \mathbf{P}} \tag{3.3}$$

$$R^2 = R \widehat{\circ} R \tag{3.4}$$

For example, if **V** = {a,b,c,?} and **P** = {a,b}. Then R = {(a,b),(b,a),(b,c),(b,?)}. Let X = {b}. Then we have the following result

$$\text{RX} = \{\text{a,c,?}\}$$
$$R|_X = \{(b,a),(b,c),(b,?)\}$$
$$R|_l^+ X = R|_X \cup R|_a = \{(a,b),(b,a),(b,c),(b,?)\}$$

3.2 Data Flow Equations

L-FCPA is formulated in the form of equation shown in Figure 3.1. Here *Ain* and *Aout* represent the Mayin and Mayout points-to information of the program. *Lin* and *Lout* represent strong liveness information at a given program point. Must(R) computes the must points-to information at a program point. It can be observed from the definition that there is no separate fixed point computation required for must points-to information. It can be derived from may points-to information [3].

$\mathit{Pointee}_n$, Def_n, Ref_n,Kill_n are four extractor functions that extracts relevant pointer information from node n. $\mathit{Pointee}_n$ gives all the possible variables which are pointed by a pointer variable. Def_n reads the LHS of statements in node n and gives the variable which are being assigned some value in n. Ref_n returns the variable being referred in node n, i.e. it return the live variables at n. Kill_n returns the variables which are being modified in the node n, i.e. whose liveness is killed in n.

$$Must(R) \quad = \quad \bigcup_{x \in \mathsf{P}} \{x\} \times \begin{cases} \mathsf{V} & (R|_x = \emptyset) \vee (R|_x = (x, ?)) \\ \{y\} & (R|_x = \{(x, y)\}) \wedge (y \neq ?) \\ \emptyset & otherwise \end{cases} \tag{3.5}$$

Extractor Functions	Data Flow Values
$\mathsf{Pointee} \subseteq \mathsf{V}$	$\mathsf{Lin}_n, \mathsf{Lout}_n \subseteq \mathsf{P}$
Kill_n, Def_n, Ref_n	$\mathsf{Ain}_n, \mathsf{Aout}_n \subseteq \mathsf{P} \times \mathsf{V}$

We assume that $x, y \in \mathsf{P}$ and $a \in \mathsf{V}$					
Stmt	Def_n	Kill_n	Ref_n		$\mathsf{Pointee}_n$
			$if\,\mathsf{Def}_n \cap \mathsf{Lout}_n \neq \emptyset$	Other	
$use\ x$	$\emptyset$	$\emptyset$	$\{x\}$	$\{x\}$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\emptyset$	$\emptyset$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$\{y\}$	$\emptyset$	$\mathsf{Ain}_n\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$\{y\} \cup \mathsf{Ain}_n\{y\} \cap \mathsf{P}$	$\emptyset$	$\mathsf{Ain}_n^2\{y\}$
$*x = y$	$\mathsf{Ain}_n\{x\} \cap \mathsf{P}$	$\{Must(\mathsf{Ain}_n)\{x\}\} \cap \mathsf{P}$	$\{x, y\}$	$\{x\}$	$\mathsf{Ain}_n\{y\}$

$$\mathsf{Lout}_n \quad = \quad \begin{cases} \emptyset & n = \mathsf{end} \\ \bigcup_{s \in \mathsf{succ}(n)} \mathsf{Lin}_s & \text{otherwise} \end{cases} \tag{3.6}$$

$$\mathsf{Lin}_n \quad = \quad (\mathsf{Lout}_n - \mathsf{Kill}_n) \cup \mathsf{Ref}_n \tag{3.7}$$

$$\mathsf{Ain}_n \quad = \quad \begin{cases} \mathsf{P} \times \{?\} & n = \mathsf{start} \\ \left(\bigcup_{p \in \mathsf{pred}(n)} \mathsf{Aout}_p \right) \Bigg|_{\mathsf{Lin}_n} & \text{otherwise} \end{cases} \tag{3.8}$$

$$\mathsf{Aout}_n \quad = \quad ((\mathsf{Ain}_n - (\mathsf{Kill}_n \times \mathsf{V})) \cup (\mathsf{Def}_n \times \mathsf{Pointee}_n))\,|_{\mathsf{Lout}_n} \tag{3.9}$$

Figure 3.1: Intraprocedural formulation of L-FCPA [3]

Working example of L-FCPA

Let us consider an example discussed in Section 1.2.1. In Figure 3.2, during first pass of liveness, we have only z live at the in of node n_2 as it is being dereferenced. Pointer information for z is not known yet, lhs of the statement modifies the value of pointee of z. Because of strong liveness y is not considered live at in of n_2 as we dont have liveness information of the pointee of z. So pointer information for y is not generated in the first pass. It is delayed until the liveness of pointee of z is known. Hence we say pointer information is computed lazily. Points-to pair *(z,u)*, reaches at the in of node n_2 and u is

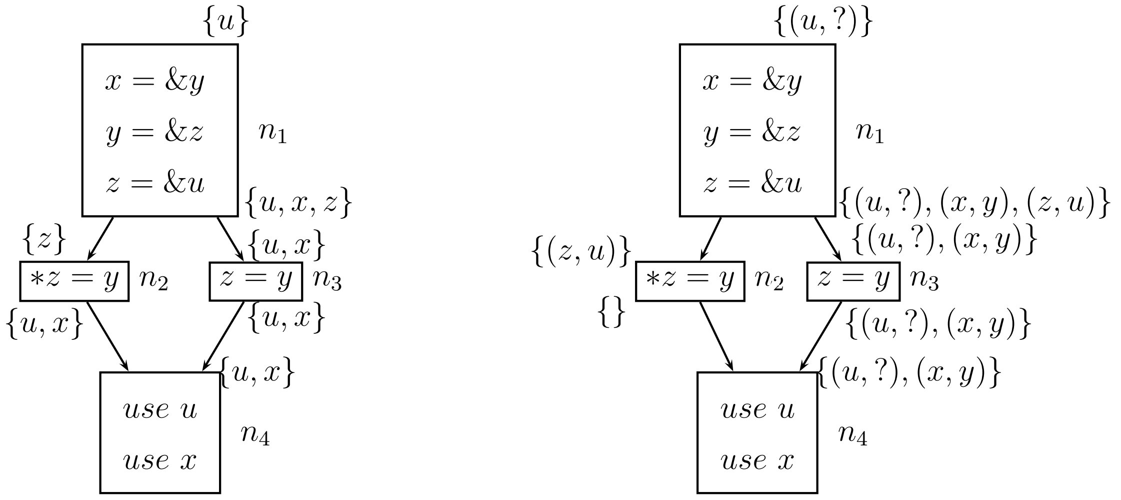


Figure 3.2: First Pass of Strong Liveness and points-to information

live at the out of n_2, therefore y becomes live at the in of node. Because of which in second points to pass we have *(y,z)* information computed. Whereas in node n_3, we still do not have z live at the out of n_3. So points-to pairs of z and y are not propagated to the in of node, i.e. information is propagated sparsely only in the live range of the variable.

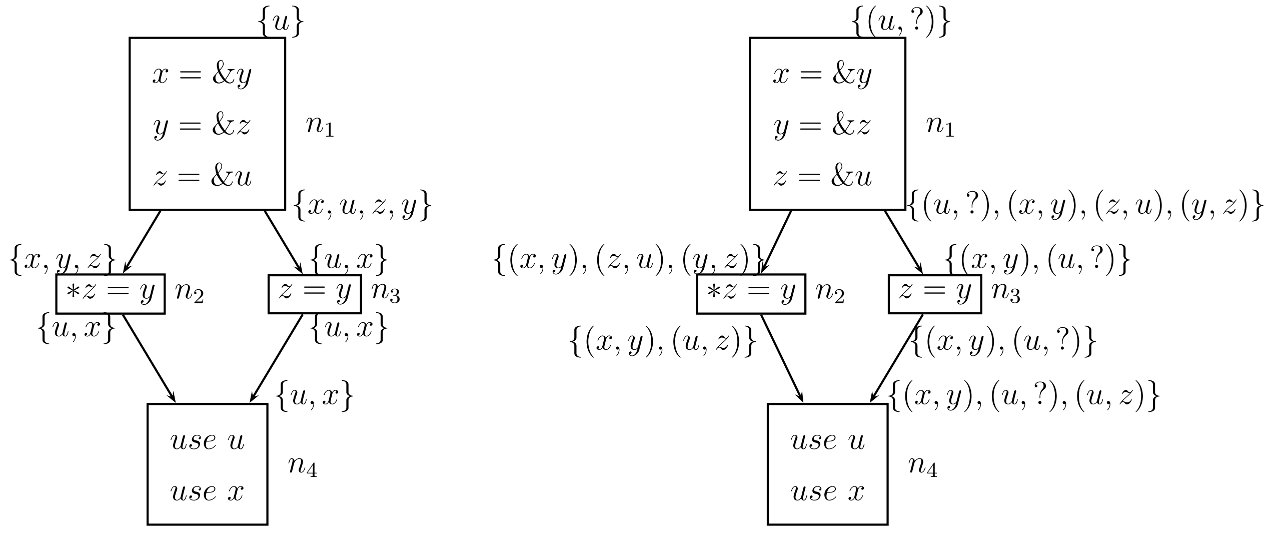


Figure 3.3: Second Pass of Strong Liveness and points-to information

3.3 Prototype Implementation in Prolog

I have implemented the liveness based pointer analysis and strong live analysis at intraprocedural level in prolog. First strong liveness is performed and then pointer information is computed for the variables that are live. For the liveness information of the statements where pointers are dereferenced, we need to first perform pointer analysis for that node. Following are the list of functions defined in order to implement the algorithm :

- Function that represent different pointer statement in a program are :

Function name	**Description**
pointer(p1,x,y)	**p1** represents the pointer statement **x = &y**
asgntoptr(a1,x,ptr(y))	**a1** represents the pointer statement **x = *y**
asgnptr(a1,ptr(x),y)	**a1** represents the pointer statement ***x = y**
asgn(a1,t,y)	**a1** represents the pointer statement **x = y**
use(u1,u)	**u1** represents **use** of pointer variable **u**

- List of functions that give information about the CFG of program :

Function name	**Description**
basicBlock(n,a)	**node n** contains the **assignment a**
edge(n1,n2)	Represents the **edge** between from **node n1 and node n2**.
program([n1,n2,n3])	Program contains nodes n1, n2 and n3.

- Function that computes Points-to information for a given program.

Function name	**Description**
genInfo(N,List,pt(X,Y))	Generates Points to information for X using information in node N and copies it into List.
killptr(A,pt(X,Y),N)	Kill the pointer information in Node N due to assignment A.
ptrin(M,L,pt(X,Y))	Computes the May-points to information at the in of node M.
ptrout(M,L,pt(X,Y))	Computes the May-points to information at the out of node M.

- Function that computes Strong liveness information for a given program.

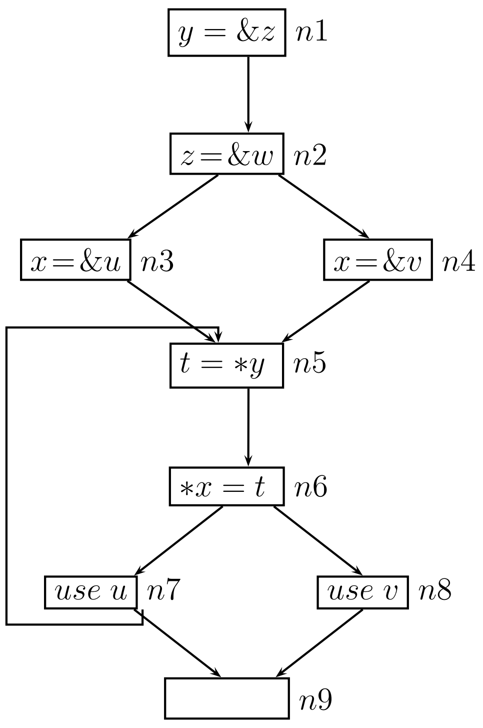
Function name	**Description**
genLiveVar(A,Vari,N)	Strong Live information generated due to assignment A in node N is stored in V.
transpVar1(N,Vari,A)	If information coming from successor is not killed in basic block N by assignment A, it is saved in Vari.
kill(N,V,A)	If information is killed in node N because of assignment A is saved in V.
livein(N,V)	Strong Liveness information at in of node N is saved in Variable V.
liveout(N,V)	Strong Liveness information at out of node N is saved in Variable V.
transpVar(N,Vari,A)	Dereference the pointer and checks the liveness of pointee.

Helper functions are present in file :

- **list** : contains some basic list operations like append, add, etc.
- **setOps** : contains the set Operation function like union, intersect, etc.

3.3.1 Working Example

Following is the program and its corresponding prolog implementation.



```
asgntoptr(a1,t,ptr(y)).  asgn(a3,t,y).
asgnptr(a2,ptr(x),t).    asgn(a4,x,z).
asgn(a5,a,b).            pointer(p1,y,z).
usev(u1,u).              usev(u2,v).
usev(u3,c).              pointer(p2,z,w).
basicBlock(1,p1).        basicBlock(2,p2).
basicBlock(3,p3).        basicBlock(4,a4).
basicBlock(5,a1).        basicBlock(6,a2).
basicBlock(7,u1).        basicBlock(8,u2).
basicBlock(9,a5).
edge(1,2).               edge(2,3).
edge(2,4).               edge(3,5).
edge(4,5).               edge(5,6).
edge(6,7).               edge(6,8).
edge(7,9).               edge(7,5).
edge(6,5).               edge(7,6).
edge(8,9).
program([1,2,3,4,5,6,7,8,9]).
```

Figure 3.4: Example program and its prolog representation

Result

```
In Live[1] is [u,v]
In Pointer[1] is [pt(u,?),pt(v,?)]
Out Live[1] is [u,v,y]
Out Pointer[1] is [pt(u,?),pt(v,?),pt(y,z)]

In Live[2] is [u,v,y]
In Pointer[2] is [pt(u,?),pt(v,?),pt(y,z)]
Out Live[2] is [u,v,y,z]
Out Pointer[2] is [pt(u,?),pt(v,?),pt(y,z),pt(z,w)] . . . . .
```

3.4 Empirical Measurement

The current implementation of L-FCPA is done in GCC-4.6.0 by Prashant and it is being tested by running it on SPEC CPU2006 benchmark[1]. Various measurements are taken and based on the observation we have found the scope of improvements in the current implementation.

3.4.1 Benchmark Programs

We studied some of the benchmark programs, Figure 3.5 is the graph which has number of pointer statements classified according to definition and use. It can be observed that, number of use in assignment(Use_asgn) is large as compared to assignment statements in almost all the benchmarks. Therefore, number of live variables can be reduced by performing strong liveness analysis. Computing points-to pair for only those variables that are live, will reduce number of pointer pairs generated.

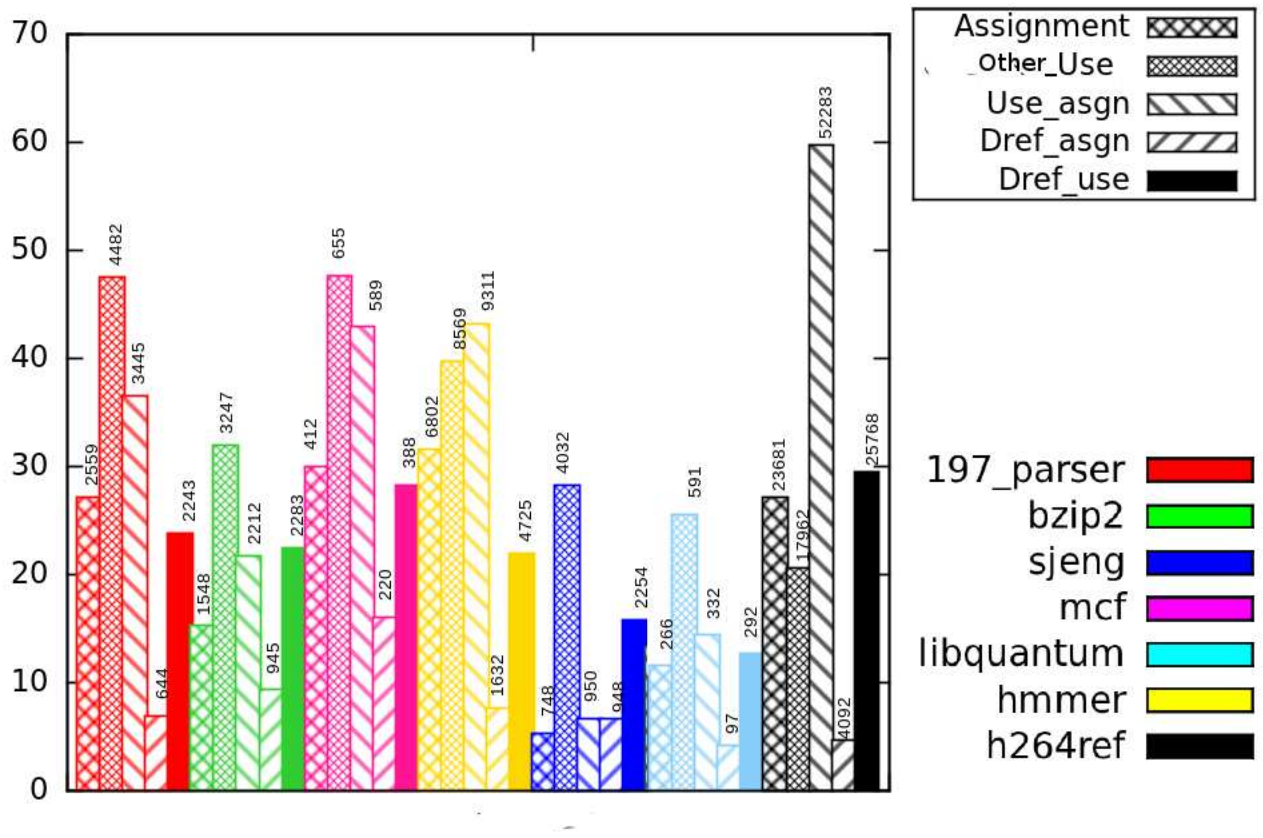


Figure 3.5: Plot of number of pointer use and pointer assignment statements

For eg, if we see statistics of h264ref, number of use in assignment statements (Use_asgn) is very large as compared to number of assignment(Assignment) statements. L-FCPA can avoid the redundant computation of pointer pairs for the variables that are not strongly live. Also, the number of use variables that need to be dereferenced(Deref_Use) is large. If we compute pointer information of these variables only if required, then number of pointer pairs can further reduce.
Remaining section contains the measurements done during the L-FCP analysis. The experimental results proves the observation made by studying the benchmark programs.

3.4.2 Time Measurement

Time of execution was measured for gcc points-to analysis(gpta), simple points-to analysis(spta) and L-FCPA. Following times were observed

Benchmarks	KLOC	L-FCPA		spta	gpta
		liveness	pta		
lbm	0.9	0.55	0.52	1.9	5.2
mcf	1.6	1.04	0.62	9.5	3.4
libquant	2.0	1.8	3.8	5.6	4.8
bzip2	3.7	4.5	4.8	28.1	30.2
parser	7.7	1200	145.6	430000	422.12
sjeng	10.5	858.2	99.0	32000	38.1
hmmer	20.6	90.0	62.9	290000	246.3
gap	35.6	46000	1300	10000	17000

Table 3.1: Time measurement for gpta, spta and L-FCPA

Observations

- It is observed that simple points-to analysis takes more time as compared to gcc pta in most of the cases. It is because lage number of pointer pairs are generated and linked list data structure is used for storing unique points-to information. And hence comparing the points-to information with linked list consumes a lot of time. Hence using some other data structure for storing pointer information, example bitmap or BDDs, can reduce the time complexity of the algorithm.

- In L-FCPA, pointer analysis takes lesser time as it only computes points to information for the variable that are live and propagate the information only for them. Strong liveness further avoid computation of pointer information that is not required.

3.4.3 Measurement of pointer pair information

Number of unique and total pointers are measured and following observations are made :

Benchmark	KLOC	gpta		spta			L-FCPA		
		Total	Unique	Total	Unique	undef	Total	Unique	undef
lbm	0.9	8739	1911	4536	283	224	62	10	2
mcf	1.6	4940	2159	30793	303	64	253	34	10
libquant	2.6	10138	2701	253	119	25	299	49	5
bzip2	3.7	172082	88000	65579	172	38	271	42	27
parser	7.7	66392	19000	28913896	3990	206	9267	265	112
sjeng	10.5	46650	11000	2246656	666	152	4548	194	45
hmmer	20.6	3292342	1900000	6974315	4487	1318	2495	156	74
gap	35.6	274025263	2.5 X 10^9	7039195	1023	248	6650	485	485

Table 3.2: Pointer pair measurement for gpta, spta and L-FCPA.

Observations

- The number of pointer pairs in L-FCPA is less than the number of pointer pairs in spta, i.e. sparse propagation of pointer information gives more precise pointer information.

- Number of unique pointer is very less as compared to the total pointer pairs, therefore we can say that the same information is propagated in the analysis repetitively. Doing the analysis incrementally can be helpful. It will avoid the computation of repetitive information and hence will save some time.

3.4.4 Measurement of Call string information

Call String approach is used in the current implementation in order to achieve context sensitivity. Our observation results show that call string approach helps in performing context sensitive analysis at a very low cost.

Benchmarks	$No.of$ $callsites$	spta		L-FCPA	
		$MaxNo.of$ $context$	$Maxlength$ $ofCS$	$MaxNo.of$ $context$	$Maxlength$ $ofCS$
lbm	33	3	4	3	4
mcf	29	4	5	4	5
libquant	258	55	9	55	9
bzip2	233	140	12	70	12
parser	1123	2648	18	166	10
sjeng	678	4608	12	96	9
hmmer	1292	1066	15	133	10

Table 3.3: Measurement of call string information(these are the maximum values)

Observations
It is observed that the number of callstring is not more than 10 for L-FCPA pta. Calling context reaching a node are considerable and hence processing them is not expensive.

3.4.5 Summary

The observations made during testing showed that the implementation is not scalable beyond 30KLOC. But it can be seen that L-FCPA performance is better than that of gcc pta. The pointer information collected shows that a lot of irrelevant information is generated by the gcc pta. By generating only the required information we can reduce the overhead of pointer analysis. Linked list implementation is used for storing unique points-to information. Accessing the information requires linear traversal of linked list. Therefore it increases the time of analysis. This is the reason because of which the implementation fail beyond 30 KLOC.

It is observed that liveness computation takes a lot of time as compared to pointer analysis. This is because after every dereferencing liveness pass is invoked and hence it decreases the efficiency of the algorithm. The scope of improvement based on the observation can be use of incremental approach and use of better data structure such as BDDs, vectors, bitmaps.

Chapter 4

APIs for Liveness Based Pointer Analysis Code

Current implementation of L-FCPA is the simplest implementation done using linked list. As discussed in previous chapter, the linear traversal of linked list make algorithm inefficient when used with large size codes. Implementing a new efficient data-structure is a tedious task in current implementation, as data structures are not abstracted. Therefore, we have designed the APIs such that, all the data-structures are accessed only through these APIs. Now if we plan to rewrite the code with new efficient data-structures, we need to rewrite only these APIs.

4.1 Data-structure used for storing information

Sorted linked lists are used to store pointer information and liveness information for a basic block. Linked list is simple to implement, but performing insertion and deletion operations in a linked list increases the time complexity of the algorithm. If the size of the list grows exponentially, even binary search would prove expensive.

Data-structure for pointer information

A points-to set at a given program point is a set of LHS variables and its pointee set RHS. Each pointer and pointee can be identified by a unique integer value. Pointee set is represented by a bitmap data-structure. The data structure for storing points-to information is given below.

```
struct pointsto_val_def
{
   unsigned int lhs;
   bitmap rhs;
   struct pointsto_val_def *next;
};
typedef struct pointsto_val_def *pointsto_val;
```

For eg, points-to set $\{a \rightarrow \{b, c\}\}$ is stored as lhs value equal to the unique id of variable a and unique id of pointee b and c are set in bitmap rhs. Since must-points-to information is computed from may information, we need to store only the may-points-to information for each program point. In order to avoid the repetitive storage of points-to set, we store each unique points-to set in `VEC mayinfo` and its index in the `VEC` is used as an identifier of the data flow value. Another `VEC tmpinfo` is used to store the intermediate points-to set obtained from the constraints within a basic block.

```
static VEC(pointsto_val, heap) *mayinfo;
static VEC(pointsto_val, heap) *tmpinfo;
```

Data-structure for liveness information

Liveset is stored in linked list where val is the unique id of the variable.

```
struct df_container
{
   unsigned int val;
   struct df_container *next;
};
typedef struct df_container *df_list;
```

Same set of variables may be live at different program points, so live vars linked list is stored in `VEC liveinfo`, and use its index as a representative liveness info at a program point.

```
static VEC(df_list, heap) *liveinfo;
```

4.2 Difficulties in writing API for current code

As discussed above, the APIs should be such that the representation of data-structures is abstracted. So, we need to design APIs where we can pass simple integer value as parameter. There were few constraints due to which we faced difficulties for designing APIs for the current implementation. So in order to have simple APIs, we did the following changes in the code :

- Initially a variable was identified with its unique id and an offset. With the offset associated with each variable, it was difficult to design APIs with only integer parameters. Therefore, we removed the offset associated with the variable by creating a new variable with offset 0, whenever we encountered a variable with non-zero offset.
- Pointee set was initially represented by a linked list. In order to hide the linked list data-structure, we represented it by a set using bitmaps. So whenever we wanted to access a pointee set, APIs traversed the linked list and created a bitmap set for it. The computation became more expensive as it lead to traversal of linked list multiple times. Hence, we replaced the pointee set linked list by bitmaps.

4.3 API design

We have designed APIs at three levels.

- **At Basic Block Level** : These APIs are designed to access the dataflow information at a given basic block. Therefore, APIs at this level only contains the basic block as parameter.
- **At Basic Block and Callstrings level** : There can be multiple callstrings reaching a basic block. Therefore, these APIs are designed to access the dataflow information at a given basic block and a callstring at that basic block. Parameters for these APIs will be callstring index and basic block number.

- **At Vector level** : As mentioned above, dataflow information is stored in a vector and its indices are stored with each basic block. These APIs are designed in order to extract information directly from the given vector index. Therefore, we have vector index as the parameter for these APIs.

4.4 API for pointer information

4.4.1 APIs at Vector level

- **GETTER APIs** :

Name	get_pointee_set	It returns the bitmap of set of variables pointed by variable represented by varid at given vec_index.
Return type	bitmap	
Parameters	int vec_index vec_list list unsigned int varid	

Name	get_indirect_pointee_set	It returns the bitmap of set of variables pointed by variable that are being pointed by variable with varid.
Return type	bitmap	
Parameters	int vec_index vec_list list unsigned int varid	

Name	get_must_pointee	It returns the pointer variable pointed by variable with varid at given vec_index.
Return type	unsigned int	
Parameters	int vec_index vec_list list unsigned int varid	

Name	get_varset_having_pointee	It returns the bitmap of set of variables which have pointees at vec_index.
Return type	bitmap	
Parameters	int vec_index vec_list list unsigned int varid	

- **CHECKER APIs** :

Name	has_must_pointee	Returns true, if variable with varid has only one pointee in its points-to set at vec_index.
Return type	bool	
Parameters	int vec_index vec_list list unsigned int varid	

Name	pointer_with_non_empty _pointee_set	Returns true if variable with varid has at least one pointee in its points-to set at vec_index.
Return type	bool	
Parameters	int vec_index vec_list list unsigned int varid	

Name	is_x_in_pointsto_set	Returns true, if variable with varid has pointee information(including NULL pointee) in pointsto set at vec_index.
Return type	bool	
Parameters	int vec_index vec_list list unsigned int varid	

Name	has_undef_pointee	Returns true, if variable with varid has an undef pointee at vec_index.
Return type	bool	
Parameters	int vec_index vec_list list unsigned int varid	

Name	has_only_undef_pointee	Returns true, if variable with varid has only an undef pointee at vec_index.
Return type	bool	
Parameters	int vec_index vec_list list unsigned int varid	

- **MODIFIER APIs** :

Name	set_must_pointee	It updates the points-to set of variable varid with the pointee variable.
Return type	void	
Parameters	int vec_index vec_list list unsigned int varid unsigned int pointee	

Name	add_pointee	It adds a pointer pair to the points-to set stored at vec_index. Returns the index of the updated list.
Return type	int	
Parameters	int vec_index vec_list list unsigned int varid unsigned int pointee	

Name	add_pointee_set	It adds points-to set of a variable with varid to the dataflow information stored at vec_index. Returns the index of the updated list.
Return type	int	
Parameters	int vec_index vec_list list unsigned int varid bitmap pointee_set	

Name	remove_pointee	It removes a pointer pair from the points-to set stored at vec_index. Returns the index of the updated list.
Return type	int	
Parameters	int vec_index vec_list list unsigned int varid bitmap pointee	

Name	remove_pointee_set	It removes points-to set of a variable with varid from the dataflow information stored at vec_index. Returns the index of the updated list.
Return type	int	
Parameters	int vec_index vec_list list unsigned int varid	

- **MERGE APIs** :

Name	merge_pointsto_info	It computes points-to set confluence of dataflow information at index vec_index1 and index vec_index2.
Return type	int	
Parameters	int vec_index1 int vec_index2 vec_list list	

4.4.2 APIs at basic block and callstring level

- **GETTER APIs** :

Name	get_pointee_set_bb_cs	It returns the bitmap of set of variables pointed by variable represented by varid at a given basic block's in or out for callstring callstring_index.
Return type	bitmap	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

Name	get_indirect_pointee _set_bb_cs	It returns the bitmap of set of variables pointed by variables that are being pointed by variable with varid at a given basic block's in or out for a given callstring.
Return type	bitmap	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

Name	get_must_pointee_bb_cs	It returns the pointer variable pointed by variable with varid at a given basic block's in or out for callstring represented by callstring_index.
Return type	unsigned int	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

Name	get_varset_having _pointee_bb_cs	It returns the bitmap of set of variables which have pointees at a given basic block's in or out for a given callstring.
Return type	bitmap	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

- **CHECKER APIs** :

Name	has_must_pointee_bb_cs	Returns true, if variable with varid has only one pointee at a given basic block and call-string.
Return type	bool	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

Name	pointer_with_non_empty _pointee_set_bb_cs	Returns true if variable with varid has at least one pointee at a given basic block and callstring.
Return type	bool	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

Name	is_x_in_pointsto_set_bb_cs	Returns true, if variable with varid has pointee information(including NULL pointee) at a given basic block's in or out for a given callstring.
Return type	bool	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

Name	has_undef_pointee_bb_cs	Returns true, if variable with varid has an undef pointee at a given basic block's in or out for a given callstring.
Return type	bool	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

Name	has_only_undef_pointee_bb_cs	Returns true, if variable with varid has only an undef pointee at a given basic block's in or out for a given callstring.
Return type	bool	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

- **MODIFIER APIs** :

Name	set_must_pointee_bb_cs	It updates the points-to set of variable varid with the pointee variable at a given basic block's in or out for a given callstring.
Return type	void	
Parameters	basic_block bb int callstring_index unsigned int varid unsigned int pointee in_out io	

Name	add_pointee_bb_cs	It adds a pointer pair to the points-to set stored for a given callstring at basic block's in or out.
Return type	void	
Parameters	basic_block bb int callstring_index unsigned int varid unsigned int pointee in_out io	

Name	add_pointee_set_bb_cs	It adds points-to set of a variable with varid to the dataflow information stored for a given callstring at basic block's in or out.
Return type	void	
Parameters	basic_block bb int callstring_index unsigned int varid bitmap pointee in_out io	

Name	remove_pointee_bb_cs	It removes a pointer pair from the points-to stored for a given callstring at basic block's in or out.
Return type	void	
Parameters	basic_block bb int callstring_index unsigned int varid bitmap pointee in_out io	

Name	remove_pointee_set_bb_cs	It removes points-to set of a variable with varid from the dataflow information stored for a given callstring at basic block's in or out.
Return type	void	
Parameters	basic_block bb int callstring_index unsigned int varid in_out io	

4.4.3 APIs at basic block

- **GETTER APIs** :

Name	get_pointee_set_bb	It returns the bitmap of set of variables pointed by variable represented by varid at a given basic block's in or out.
Return type	bitmap	
Parameters	basic_block bb unsigned int varid in_out io	

Name	get_indirect_pointee_set_bb	It returns the bitmap of set of variables pointed by variables that are being pointed by varid at a given basic block's in or out.
Return type	bitmap	
Parameters	basic_block bb unsigned int varid in_out io	

Name	get_must_pointee_bb	It returns the pointer variable pointed by variable with varid at a given basic block bb's in or out.
Return type	unsigned int	
Parameters	basic_block bb unsigned int varid in_out io	

Name	get_varset _having_pointee_bb	It returns the bitmap of set of variables which have pointees at a given basic block bb's in or out.
Return type	bitmap	
Parameters	basic_block bb unsigned int varid in_out io	

- **CHECKER APIs** :

Name	has_must_pointee_bb	Returns true, if variable with varid has only one pointee at a given basic block.
Return type	bool	
Parameters	basic_block bb unsigned int varid in_out io	

Name	pointer_with_non_empty _pointee_set_bb	Returns true if variable with varid has at least one pointee at a given basic block bb's in or out.
Return type	bool	
Parameters	basic_block bb unsigned int varid in_out io	

Name	is_x_in_pointsto_set_bb	Returns true, if variable with varid has pointee information(including NULL pointee) at a given basic block bb's in or out.
Return type	bool	
Parameters	basic_block bb unsigned int varid in_out io	

Name	has_undef_pointee_bb	Returns true, if variable with varid has an undef pointee at a given basic block bb's in or out.
Return type	bool	
Parameters	basic_block bb unsigned int varid in_out io	

Name	has_only_undef_pointee_bb	Returns true, if variable with varid has only an undef pointee at a given basic block bb's in or out.
Return type	bool	
Parameters	basic_block bb unsigned int varid in_out io	

- **MODIFIER APIs** :

Name	set_must_pointee_bb	It updates the points-to set of variable varid with the pointee variable at a given basic block bb's in or out.
Return type	void	
Parameters	basic_block bb unsigned int varid unsigned int pointee in_out io	

Name	add_pointee_bb	It adds a pointer pair to the points-to set stored at basic block bb's in or out.
Return type	void	
Parameters	basic_block bb unsigned int varid unsigned int pointee in_out io	

Name	add_pointee_set_bb	It adds points-to set of a variable with varid to the dataflow information at basic block's in or out.
Return type	void	
Parameters	basic_block bb unsigned int varid bitmap pointee in_out io	

Name	remove_pointee_bb	It removes a pointer pair from the points-to set stored at basic block bb's in or out.
Return type	void	
Parameters	basic_block bb unsigned int varid bitmap pointee in_out io	

Name	remove_pointee_set_bb	It removes points-to set of a variable with varid from the dataflow information stored at basic block bb's in or out.
Return type	void	
Parameters	basic_block bb unsigned int varid in_out io	

4.5 API for liveness information

4.5.1 API at vector index level

Name	is_var_live	Returns true, if variable with varid is in liveset at vector index vec_index.
Return type	bool	
Parameters	int vec_index unsigned int varid	

Name	get_live_set	It returns the bitmap of set of variables that are live in liveset at index vec_index.
Return type	bitmap	
Parameters	int vec_index	

Name	add_var_to_liveset	It adds a variable with varid to the liveset stored at vec_index and returns the index of new list.
Return type	int	
Parameters	int vec_index unsigned int varid	

Name	remove_var_from_live_set	It removes a variable with varid to the liveset stored at vec_index and returns the index of new list.
Return type	int	
Parameters	int vec_index unsigned int varid	

Name	merge_liveness_info	It merges liveness info at vector index vec_index1 and vec_index2 and return the index of new list.
Return type	int	
Parameters	int vec_index1 int vec_index2	

4.5.2 API at basic block and callstring level

Name	is_var_live_bb_cs	Returns true, if variable with varid is in liveset at basic block bb's in or out for a given callstring.
Return type	bool	
Parameters	basic_block bb int callstring_index unsigned int varid in_out inout	

Name	get_live_set_bb_cs	It returns the bitmap of set of variables that are live at basic block bb's in or out for a given callstring.
Return type	bitmap	
Parameters	basic_block bb int callstring_index in_out inout	

Name	add_var_to_liveset_bb_cs	It adds a variable with varid to the liveset stored at basic block bb's in or out for a given callstring.
Return type	void	
Parameters	basic_block bb int callstring_index unsigned int varid in_out inout	

Name	remove_var_from_live_set_bb_cs	It removes a variable with varid to the liveset stored at at basic block bb's in or out for a given callstring.
Return type	void	
Parameters	basic_block bb int callstring_index unsigned int varid in_out inout	

4.5.3 API at basic block

Name	is_var_live_bb	Returns true, if variable with varid is in liveset at basic block bb's in or out.
Return type	bool	
Parameters	basic_block bb unsigned int varid in_out inout	

Name	get_live_set	It returns the bitmap of set of variables that are live at basic block bb's in or out.
Return type	bitmap	
Parameters	basic_block bb in_out inout	

Name	add_var_to_liveset_bb	It adds a variable with varid to the liveset stored at basic block bb's in or out.
Return type	void	
Parameters	basic_block bb unsigned int varid in_out inout	

Name	remove_var_from_live_set_bb	It removes a variable with varid to the liveset stored at at basic block bb's in or out.
Return type	void	
Parameters	basic_block bb unsigned int varid in_out inout	

4.6 Measurements

The implementation of L-FCPA with API is tested by running it on SPEC CPU2006 benchmark.

Time Measurement

Following table contains the time taken(in milliseconds) by L-FCPA and L-FCPA api for the analysis.

Benchmarks	KLOC	L-FCPA	L-FCPA api
lbm	0.9	1.72	10.50
mcf	1.6	3.61	28.29
libquant	2.0	8.72	41.53
bzip2	3.7	18.32	97.30
sjeng	10.5	1028.2	$7.3\text{X}10^6$
hmmer	20.6	849.01	6287.2

Table 4.1: Time measurement for L-FCPA and L-FCPA api

Observations

It can be observed from the above measurements that the L-FCPA implementation with APIs take more time as compared to the old implementation. This is because the number of times linked list is traversed has increased. For eg, if we want to compute pointsto set after a statement. We have an index in `VEC` which stores kill set and gen set. In order to add pointsto set present in gen set, following APIs are called :

- `get_varset_having_pointee` api is invoked, which returns the bitmap of set of variables whose pointee set is there at index in `VEC` (where kill set is stored).
- Then bitmap is iterated and for each variable in set we call `get_pointee_set` API, which again requires a linked list traversal.
- Now, `add_pointee_set` is called for each variable present in gen set and every time linked list is traversed from the start.

Consider an abstract example shown in Figure 4.1 where two sorted linked list are merged. Here `alist` represent the linked list whose elements are to be merged with `mlist` . Here we traverse the linked list only once as during the traversal we add an element to the `mlist`.

```
for(;alist;alist.next){
  for(;mlist;mlist.next){
    if(add_data > main_data)
      increment main list
    if(add_data == main_data)
      increment both lists
    if(add_data < main_data){
      add element in main list and increment both the lists
    }
  }
}
```

```
Merging linked list using APIs :

add_api(new_element add_data){
  for(;mlist;mlist.next){
    if(add_data >= main_data)
      increment main list
    if(add_data < main_data){
      add element in main list and increment both the lists
    }
  }
}
FOR_EACH_ELEMENT_IN_ADD_LIST
    add_api(new_element)
```

Figure 4.1: Merge of two sorted linked list

The code rewritten using API `add_api` is shown in Figure 4.1. It can be observed that for each element of the `alist`, we need to call the `add_api` . Thus increasing the number of linked list traversal.Therefore L-FCPA implementation using APIs take more time than L-FCPA implementation without APIs.

From the above example it can be seen that with APIs, computation have become more expensive. Therefore, if we have a data-structure, which can provide direct access to the variable pointee set(eg. hash-maps), then this overhead can be reduced and computation will take less time.

Chapter 5

Incremental Pointer Analysis

In this chapter we will be discussing incremental analysis approach for doing various program analysis and levels of incremental analysis. We will also be introducing incremental L-FCPA formulation.

5.1 Incremental Analysis

As discussed in Chapter 1, incremental analysis means only propagating the incremental values, in other words, applying dataflow equations only on the incremental value. This can reduce the number of computation required in an analysis. Incremental analysis can be done at three levels :

- Across different iterations in a fixed point computation.
- Across different applications of an analysis.
- Across different fixed point computations of same application.

5.1.1 Across different iterations in a fixed point computation

As seen in Figure 1.3, only the new values computed in first iterations are propagated in second iteration. It eliminates the redundant computation and propagation of old values. Therefore incremental analysis can be used to do pointer analysis faster, as now we can avoid propagating and manipulating whole points-to set for the next iteration.

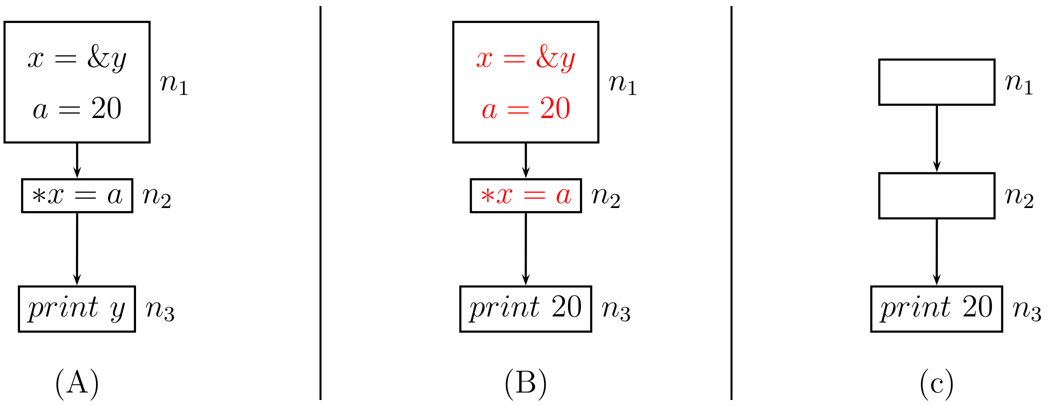


Figure 5.1: Incremental Pointer Analysis across different applications

5.1.2 Across different applications of an analysis

Consider the example shown in Figure 5.1. After performing L-FCPA, we have scope of doing constant propagation. After first liveness pass, we have x live at the out of node n_1, which enables computation of points-to information of x (x $\rightarrow$ y) at node n_1. y is live at the exit of node n_2 and we have x $\rightarrow$ y reaching the entry of node n_2. Therefore, we can directly assign value of variable a to variable y at node n_2. Constant propagation replaces use of variable a at node n_2 by its value 20, i.e. value of variable y also becomes 20. This can be further optimized by directly propagating value of variable y in print statement at node n_3. So the transformed code can be seen in Figure 5.1(B). If we perform dead code elimination on the modified graph, the dead statements in node n_1 and n_2 will be removed. The graph is further modified and can be seen in Figure 5.1 (C). So performing points-to analysis on the modified graph is irrelevant. And the pointer information stored with the modified graph is redundant. Therefore, incremental pointer analysis across different applications is performing the analysis on the modified CFG using only the incremental information.

Issues with incremental analysis across different applications

Incremental analysis across different application is non-monotonic. Consider the example in Figure 5.1, if liveness analysis computes variable y as live at the entry of node n_3. If liveness is performed in modified graph in Figure 5.1(C), we get empty liveset at the entry of node n_3. That is we are moving towards the top of the lattice, i.e, new value is not weaker than the old value. Therefore it is non-monotonic and requires different kind of formulation. In this report, we restrict ourselves to monotonic framework.

5.1.3 Across different fixed point computations of same application

In case of L-FCPA, we have alternate fixed point computations of liveness analysis and points-to analysis. If we observe the time taken by L-FCPA for different benchmark programs in Table 3.1, we can see that time taken by liveness pass in sjeng and parser is more than time time taken by points-to pass. This is because, in the current implementation of L-FCPA, liveness pass is invoked every time a dereferencing of variable is required in a statement during points-to pass. Liveness computation is done from the beginning, every time a liveness pass is invoked. A lot of information that was already obtained in the previous pass is computed again(explained in detail in Section 5.3.5). Therefore, if we have incremental computation across different fixed point computations in the L-FCPA, then we can get rid of redundant computations, there by making the algorithm more efficient.

5.2 Motivation for incremental pointer analysis

Based on experimental result, it can be said that incremental analysis can be useful in L-FCPA for large size code. Following is the graph that gives information about number of pointer pairs at each basic block for L-FCPA.

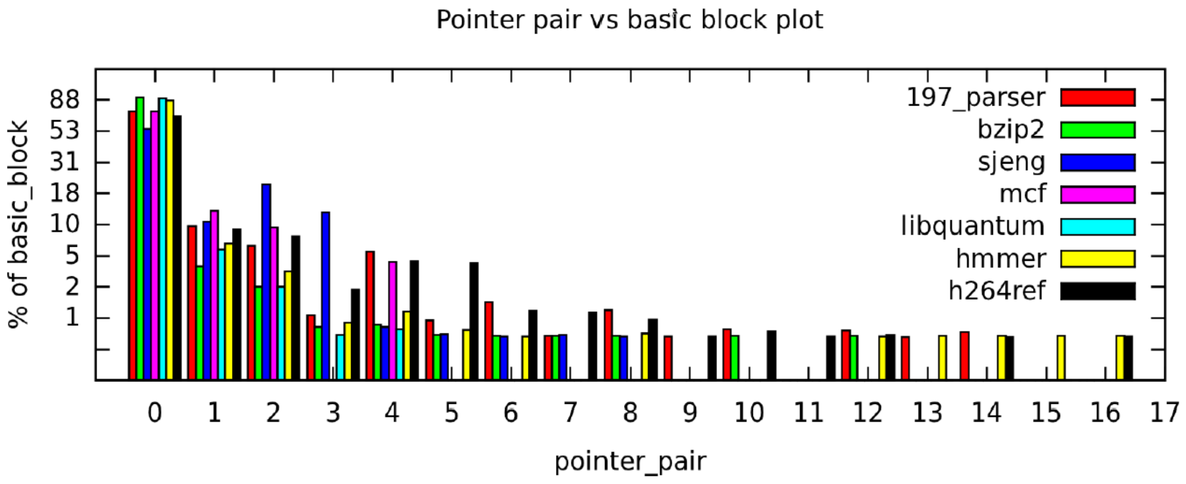


Figure 5.2: (a) Graph for 1 to 17 pointer pairs

In the above plot, it can be clearly seen that maximum number of basic block have 0 pointer pairs. Most of the benchmark program has maximum 10-12 pointer pairs at a given program point. From the Figure 5.2(b) and 5.2(c), it can be observed that, only benchmark program h264ref has more than 17 pointer pairs reaching at a basic block. Figure 5.3 contains the number of pointer statements present in each benchmark. From the two plots we can conclude that, with the increase in the size of code, number of pointer statements also increases. Therefore, with large codes, number of pointer pairs at a basic block will also increase.

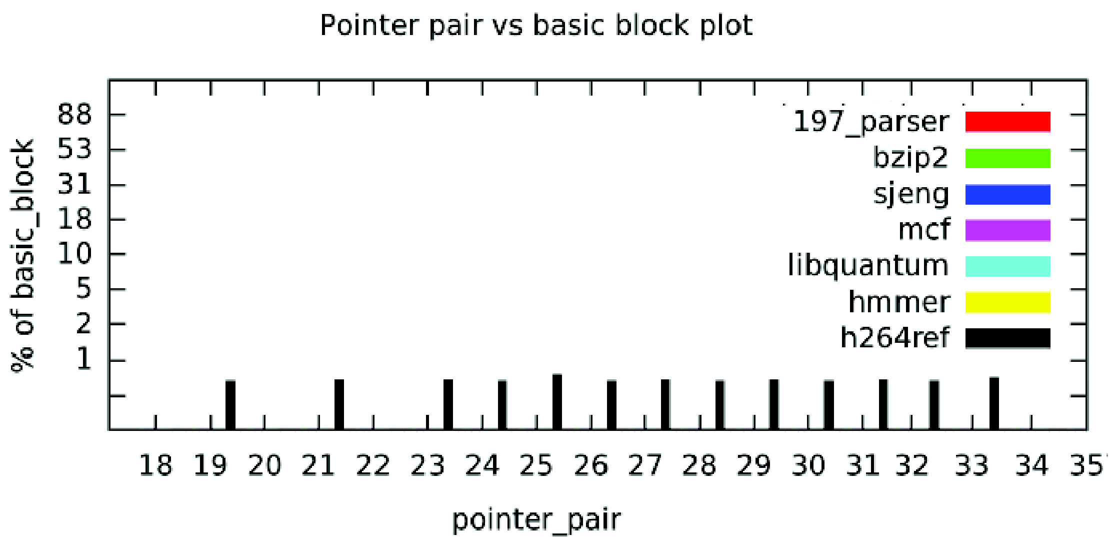


Figure 5.2 (b) Graph for 18 to 35 pointer pairs

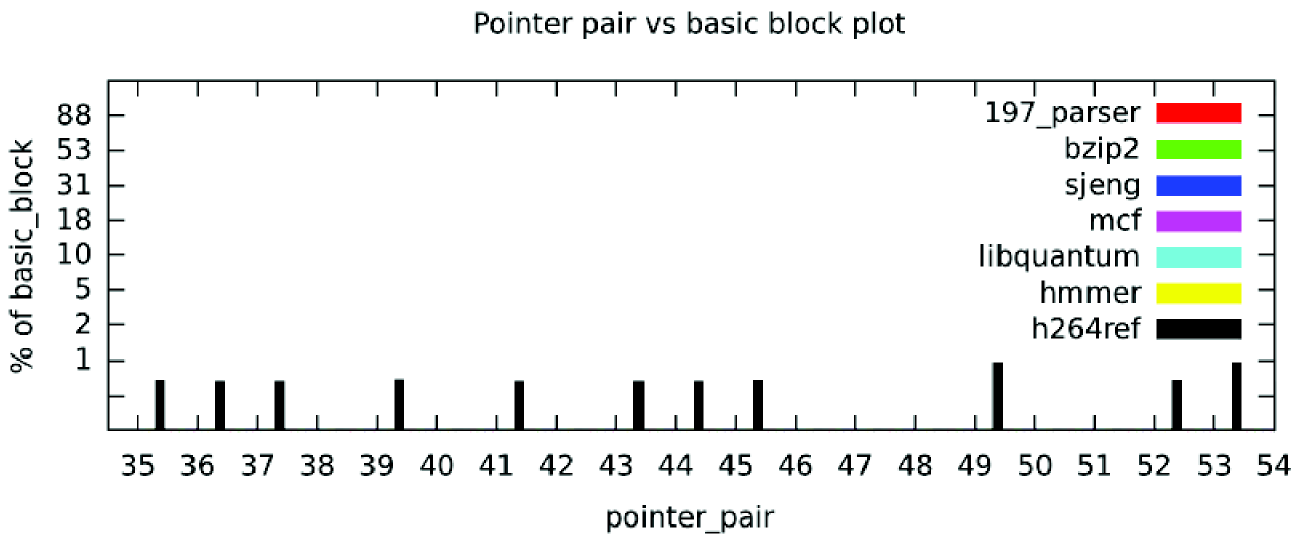


Figure 5.2 (c) Graph for 36 to 55 pointer pairs

Propagating and applying flow function on points-to set having more than 20 pairs, will take more time as compared to small points-to set. If we only propagate the incremental values, in the subsequent iterations, computations will become less expensive. Therefore, performing incremental analysis, for large size code can reduce the time of computation further.

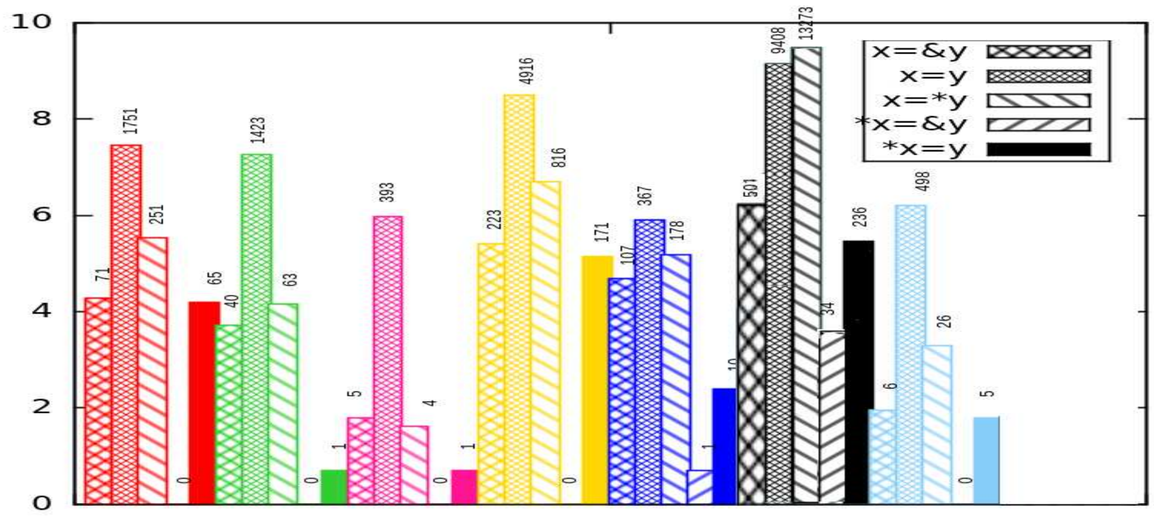
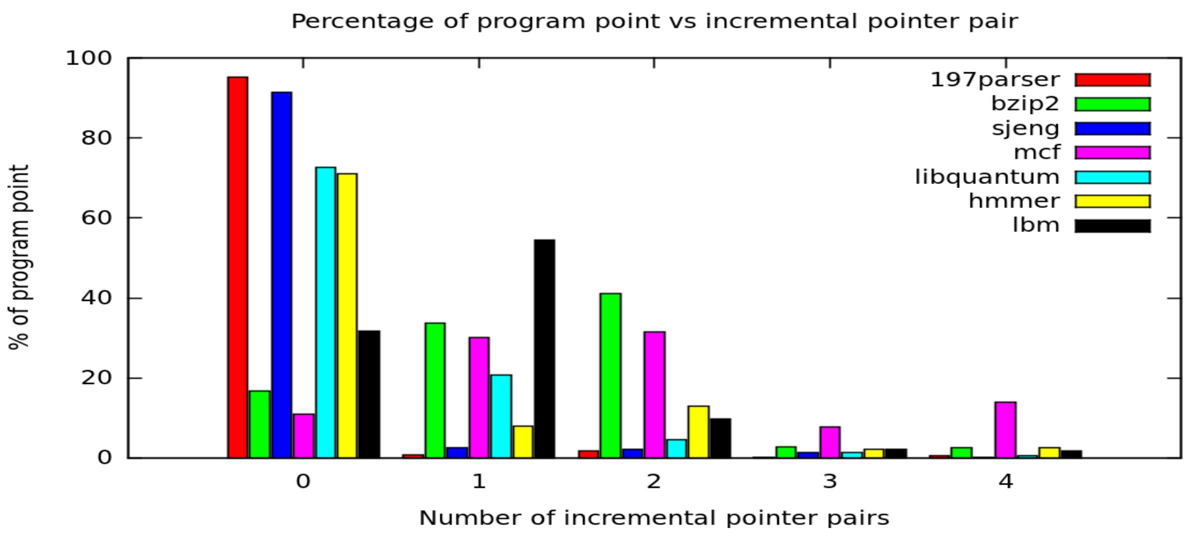


Figure 5.3: Graph showing types of statement in each benchmark

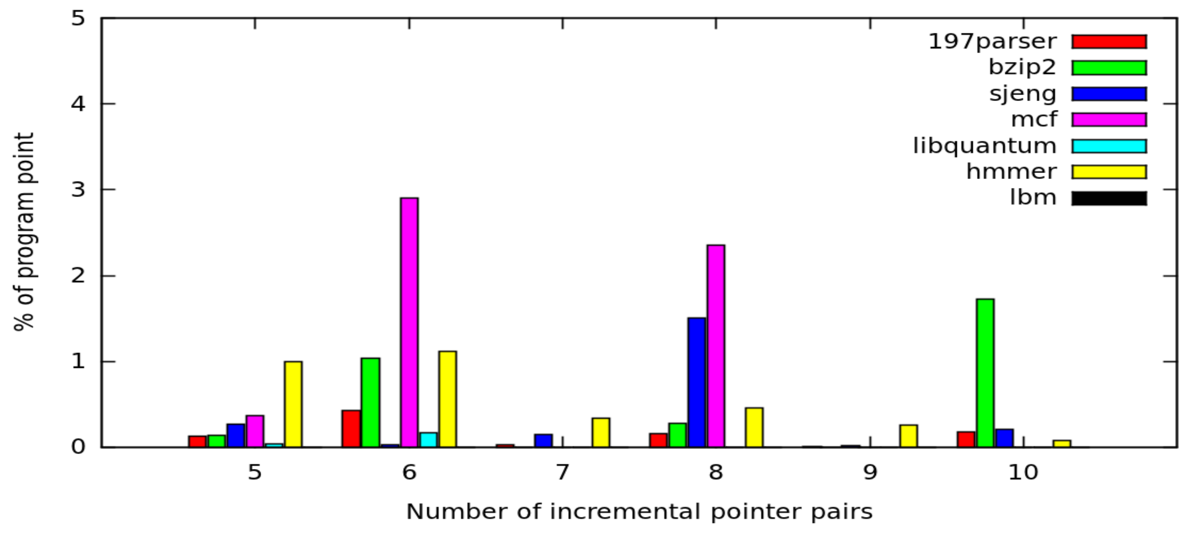
From Figure 5.3, it can be observed that we have significant number of x = &y types of statement. Propagating this information in each iteration will require some computation time. So, if we use incremental analysis, we can avoid such redundant propagation of information in subsequent iterations.

Incremental information in L-FCPA

The number of incremental pointer pairs was computed at a program point by difference in number of pointer pairs in old points-to set(in previous iteration) and new points-to set(in current iteration). The plot in Figure 5.4 and Figure 5.5 shows the percentage of program point(y-axis) that have x number of incremental pointer pairs(x-axis).



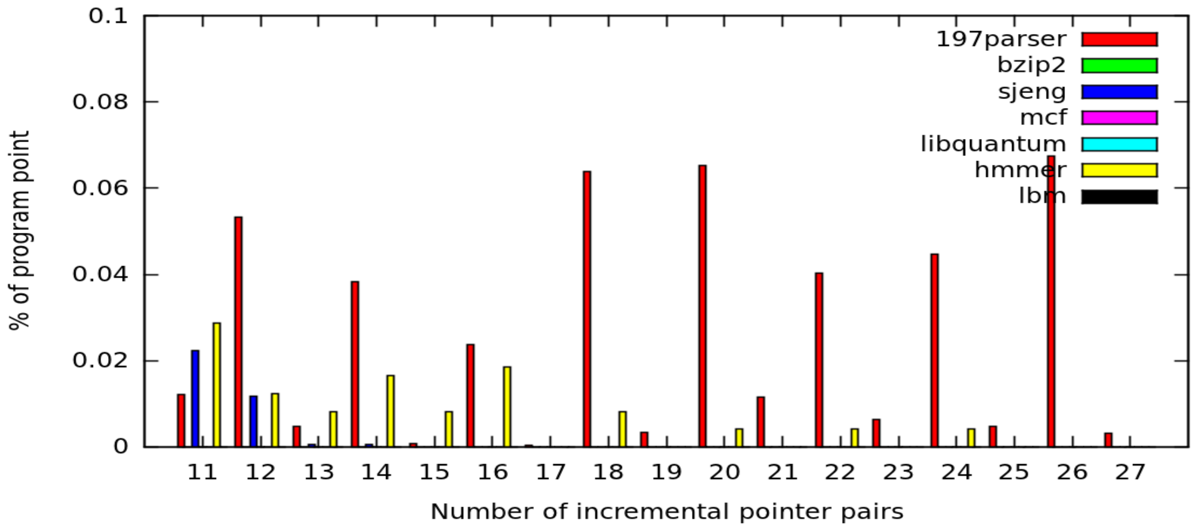
(A)



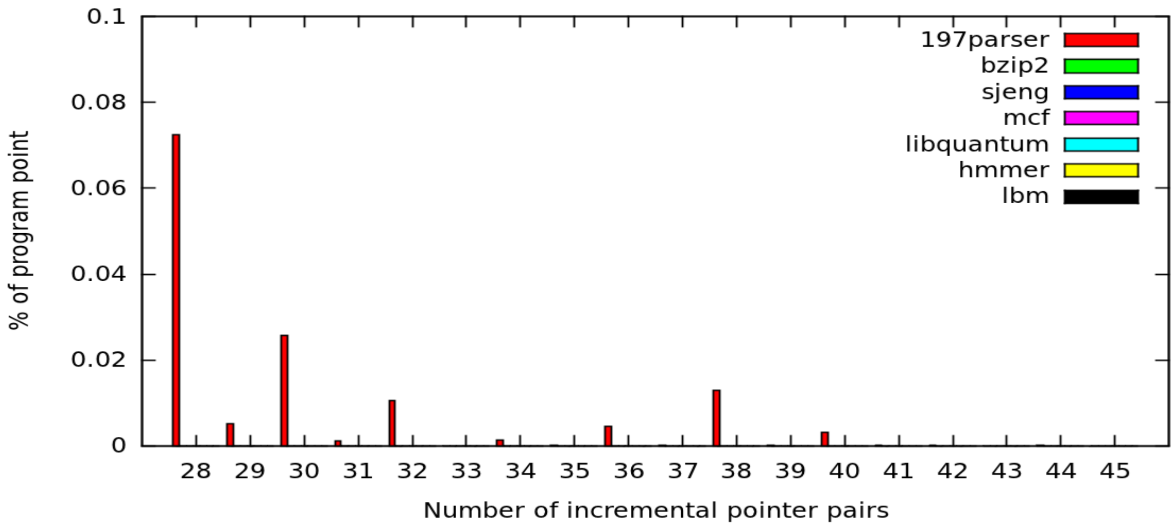
(B)

Figure 5.4: Percentage of pregram point vs number of incremental pointer pair

The plot in Figure 5.4 shows that that around 95% of program point in 197parser has 0 incremental value. As the number of incremental points-to pair increases, % of program point having those values decreases significantly. The plot in Figure 5.5 shows that there are less than 1% of basic blocks that have more than 10 incremental pointer pairs. That is, we can say that computation required for computing and propagating pointer pair can be reduced by propagating only the incremental value. Thus, doing increment analysis can reduce the number of computations done at a program point.



(A)



(B)

Figure 5.5: Percentage of pregram point vs number of incremental pointer pair

5.3 Dataflow Equations

This section contains formulation of incremental L-FCPA. These definitions are taken from the older draft of the L-FCPA paper [3] but is removed from the paper that was submitted.

5.3.1 Incremental Liveness equation

In, incremental liveness analysis we first compute the $\Delta \mathit{Lout}_n$ and $\Delta \mathit{Lin}_n$ for a given node n(Δ represents incremental values). Then we add the incremental value in the accumulated liveness set Lin_n and Lout_n. Here, $\Delta \mathit{Ref}_n$ is calculated from the $\Delta \mathit{Ain}_n$(incremental points-to set) value computed at node n.

$$\Delta \mathit{Lout}_n = \begin{cases} \emptyset & n = \mathit{end} \\ \left(\bigcup\limits_{s \in \mathit{succ}(n)} \Delta \mathit{Lin}_s \right) - \mathit{Lout}_n & \text{otherwise} \end{cases} \tag{5.1}$$

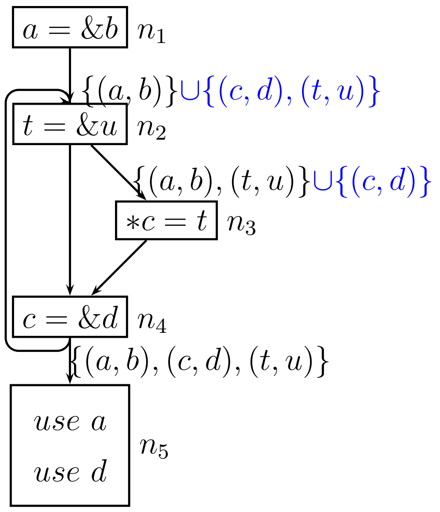
$$\Delta \mathit{Lin}_n = ((\Delta \mathit{Lout}_n - \mathit{Kill}_n) \cup \Delta \mathit{Ref}_n) - \mathit{Lin}_n \tag{5.2}$$

$$\mathit{Lout}_n = \begin{cases} \emptyset & n = \mathit{end} \\ \mathit{Lout}_n \cup \Delta \mathit{Lout}_n & \text{otherwise} \end{cases} \tag{5.3}$$

$$\mathit{Lin}_n = \mathit{Lin}_n \cup \Delta \mathit{Lin}_n \tag{5.4}$$

5.3.2 Issues with incremental pointer analysis

Incremental Pointer analysis is non-distribute. Consider the example in Figure 5.6.



At node n_3

- $\triangle X$ = {(c,d)} F($\triangle X$) = {(c,d)}
- X = {(a,b),(t,u)} F(X) = {(a,b),(t,u)}
- F($X \cup \triangle X$) = {(a,b),(t,u),(c,d),(d,u)}
- F($X \cup \triangle X$) $\neq$ f($\triangle X$)$\cup$f(X)

Figure 5.6: Incremental L-FCPA is non-distributive

5.3.3 Ensuring distributivity of incremental points-to analysis

With L-FCPA, widening[1] is performed for achieving distributive incremental analysis. We need to widen the incremental set for only two kind of statements in order to have correct result.

- For **x = *y** all the points-to pair for pointer variable y should be known at the start of node containing the statement. So if we have this statement in a node then along with incremental information we should check for the pointer information for y in the incoming information and include them in the incremental points-to set. This can be done by performing following operations :
  $\triangle \mathit{Ain}_n = \triangle \mathit{Ain}_n \cup (\triangle \mathit{Ain}_n \cup \mathit{Ain}_n)|^{+}_{l\,y}$
- For ***x = y** we need to include the pointer information for variable x and y. So we can say the incremental set is widened over x and y so as to improve the correctness of outcome after the iteration. This is can be done by following operation.
  $\triangle \mathit{Ain}_n = \triangle \mathit{Ain}_n \cup (\triangle \mathit{Ain}_n \cup \mathit{Ain}_n)|_{l\,(x,y)}$

For example in Figure 5.6 we have statement $*c = t$ and after 1st iteration, we get (c,d) as incremental information. So widening enable us to include (t,u) information also.

5.3.4 Incremental Points-to Analysis equation

While computing $\Delta \mathit{Ain}_n$ and $\Delta \mathit{Aout}_n$, extractor function uses the accumulated Lin_n and Lout_n values because pointer variable mint have been found live in earlier iterations. $\Delta \mathit{Def}_n$ and $\Delta \mathit{Pointee}_n$ are computed from the incremental value $\Delta \mathit{Ain}_n$. Widening of $\Delta \mathit{Ain}_n$ is done, in order to achieve distributivity.

[1]This widening is different from the widening operator in static program analysis. Widening operator ∇ is used to enforce termination if we have infinite ascending lattice chains. But in incremental L-FCPA, widening is used to enlarge the incremental points-to set at a program point.

$$\Delta \mathit{Ain}_n = \begin{cases} \emptyset & n = \mathit{start} \\ \left(\bigcup_{p \in \mathit{pred}(n)} \Delta \mathit{Aout}_p \right) \Big|_{\mathit{Lin}_n} - \mathit{Ain}_n & \text{otherwise} \end{cases} \tag{5.5}$$

$$\mathit{Temp} = \mathit{Ain}_n \cup \Delta \mathit{Ain}_n$$

$$\Delta \mathit{Ain}_n = \begin{cases} \Delta \mathit{Ain}_n \cup \mathit{Temp}|^{+}_{l\{y\}} & n \text{ is } x = *y \text{ and } \left(\mathit{Temp}|^{+}_{l\{y\}} \cap \Delta \mathit{Ain}_n\right) \neq \emptyset \\ \Delta \mathit{Ain}_n \cup \mathit{Temp}|_{l\{x,y\}} & n \text{ is } *x = y \text{ and } \left(\mathit{Temp}|_{l\{x,y\}} \cap \Delta \mathit{Ain}_n\right) \neq \emptyset \\ \Delta \mathit{Ain}_n & \text{otherwise} \end{cases} \tag{5.6}$$

$$\Delta \mathit{Aout}_n = ((\Delta \mathit{Ain}_n - (\mathit{Kill}_n \times \mathbf{V})) \cup (\Delta \mathit{Def}_n \times \Delta \mathit{Pointee}_n))|_{\mathit{Lout}_n} - \mathit{Aout}_n \tag{5.7}$$

$$\mathit{Ain}_n = \begin{cases} \mathbf{P} \times \{?\} & \mathit{start} \\ \mathit{Ain}_n \cup \Delta \mathit{Ain}_n & \text{otherwise} \end{cases} \tag{5.8}$$

$$\mathit{Aout}_n = (\mathit{Aout}_n \cup \Delta \mathit{Aout}_n) \tag{5.9}$$

5.3.5 Working Example

This section contains detailed working of intraprocedural incremental L-FCPA. Consider the example in Figure 5.6.

Node	First Liveness Pass							
	Iteration 1				Iteration 2			
	Δ *Lin*	Δ *Lout*	*Lin*	*Lout*	Δ *Lin*	Δ *Lout*	*Lin*	*Lout*
n_1	{d,c}	{a,d,c}	{d,c}	{a,d,c}	∅	∅	{d,c}	{a,d,c}
n_2	{a,d,c}	{a,d,c}	{a,d,c}	{a,d,c}	∅	∅	{a,d,c}	{a,d,c}
n_3	{c}	{a,d}	{c}	{a,d}	∅	∅	{c}	{a,d}
n_4	{a,d}	{a,d}	{a,d}	{a,d}	∅	{c}	{a,d}	{a,d,c}
n_5	{a,d}	∅	{a,d}	∅	∅	∅	{a,d}	∅

Table 5.1: First pass of incremental liveness analysis

If we observe the Table 5.1, we will find that in second iteration, we have only incremental information {c} at exit of node n_4. It can be clearly seen that propagating the incremental information, reduces the computations done in the second iterations.

Node	First Points-to pass							
	Iteration 1				Iteration 2			
	Δ *Ain*	Δ *Aout*	*Ain*	*Aout*	Δ *Ain*	Δ *Aout*	*Ain*	*Aout*
n_1	(d,?),(c,?)	(a,b),(d,?),(c,?)	(d,?),(c,?)	(a,b),(d,?),(c,?)	∅	∅	(d,?),(c,?)	(a,b),(d,?),(c,?)
n_2	(a,b),(d,?),(c,?)	(a,b),(d,?),(c,?)	(a,b),(d,?),(c,?)	(a,b),(d,?),(c,?)	(c,d)	(c,d)	(a,b),(d,?),(c,?),(c,d)	(a,b),(d,?),(c,?),(c,d)
n_3	(c,?)	∅	(c,?)	∅	(c,d),(c,?)	∅	(c,d),(c,?)	∅
n_4	(a,b),(d,?)	(a,b),(d,?),(c,d)	(a,b),(d,?)	(a,b),(d,?),(c,d)	∅	∅	(a,b),(d,?),(c,d)	(a,b),(d,?)
n_5	(a,b),(d,?)	∅	(a,b),(d,?)	∅	∅	∅	(a,b),(d,?)	∅

Table 5.2: First pass of incremental points-to analysis

In Table 5.2, we have incremental points-to information *(c,d)* at the entry of node n_2, which is propagated to its successor. If we do not perform incremental computations, then at the entry of node n_3, we need to propagate all the pointer pairs that are obtained at the exit of node n_2, there by increasing the time of computation.

Node	Second Liveness pass							
	Iteration 1				Iteration 2			
	Δ *Lin*	Δ *Lout*	*Lin*	*Lout*	Δ *Lin*	Δ *Lout*	*Lin*	*Lout*
n_1	{d,c}	{a,d,c}	{d,c}	{a,d,c}	∅	∅	{d,c}	{a,d,c}
n_2	{a,d,c}	{a,d,c,t}	{a,d,c}	{a,d,c,t}	∅	∅	{a,d,c}	{a,d,c,t}
n_3	{a,d,c,t}	{a,d}	{a,d,c,t}	{a,d}	∅	∅	{a,d,c,t}	{a,d}
n_4	{a,d}	{a,d}	{a,d}	{a,d}	∅	{c}	{a,d}	{a,d,c}
n_5	{a,d}	∅	{a,d}	∅	∅	∅	{a,d}	∅

Table 5.3: Second pass of incremental liveness analysis

If we observe the dataflow values in second pass of incremental liveness analysis, shown in Table 5.3, we will find that most of the values computed in second pass were already computed in first pass. This increases the time of computation. If we have incremental computation across different fixed point computations in L-FCPA, then we can avoid redundant computations of dataflow values.

Widening operation is performed at entry of node n_3 because of statement $*c = t$. In second

Node	Second Points-to pass							
	Iteration 1				Iteration 2			
	Δ *Ain*	Δ *Aout*	*Ain*	*Aout*	Δ *Ain*	Δ *Aout*	*Ain*	*Aout*
n_1	(d,?),(c,?)	(a,b),(d,?),(c,?)	(d,?),(c,?)	(a,b),(d,?),(c,?)	$\emptyset$	$\emptyset$	(d,?),(c,?)	(a,b),(d,?),(c,?)
n_2	(a,b),(d,?),(c,?)	(a,b),(d,?),(c,?),(t,u)	(a,b),(d,?),(c,?)	(a,b),(d,?),(c,?),(t,u)	{(c,d)}	{(c,d)}	(a,b),(d,?),(c,?),(c,d)	(a,b),(d,?),(c,?),(t,u),(c,d)
n_3	(c,?),(t,u)	$\emptyset$	(c,?),(t,u)	$\emptyset$	(c,d),(c,?),(t,u)	(d,u)	(c,d),(c,?),(t,u)	(d,u)
n_4	(a,b),(d,?)	(a,b),(d,?),(c,d)	(a,b),(d,?)	(a,b),(d,?),(c,d)	(d,u)	(d,u)	(a,b),(d,?),(d,u)	(a,b),(d,?),(c,d),(d,u)
n_5	(a,b),(d,?)	$\emptyset$	(a,b),(d,?)	$\emptyset$	(d,u)	$\emptyset$	(a,b),(d,?),(d,u)	$\emptyset$

Table 5.4: Second pass of incremental points-to analysis

iteration, we add points-to information of c and t at the entry of n_3. But as discussed in case of incremental liveness, in second pass of incremental points-to analysis also, a lot of redundant computations are done.

Incremental computation across different fixed point computations in L-FCPA

The formulation of incremental dataflow equations for L-FCPA allows us to do the incremental computations across different fixed point computation in L-FCPA. If we perform incremental computation across fixed point computations in second points-to pass of L-FCPA, we get the dataflow values as shown in Table 5.5.

From the above table, it can be observed that, at every program point F($X \cup \triangle X$) = F($\triangle X$)$\cup$F(X) holds, where Δ X is the incremental value and X is the accumulated value.

Node	Iteration 1			
	∆ *Ain*	∆ *Aout*	*Ain*	*Aout*
n_1	∅	∅	(d,?),(c,?)	(a,b),(d,?),(c,?)
n_2	∅	(t,u)	(a,b),(d,?),(c,?),(c,d)	(a,b),(d,?),(c,?),(t,u),(c,d)
n_3	(c,?),(t,u),(c,d)	(d,u)	(c,?),(t,u),(c,d)	(c,d),(c,?),(t,u),(d,u)
n_4	(d,u)	(d,u)	(a,b),(d,?),(d,u)	(a,b),(d,?),(c,d),(d,u)
n_5	(d,u)	∅	(a,b),(d,?),(d,u)	∅

Table 5.5: Second pass of incremental points-to analysis across different fixed point computations

5.4 Interprocedural incremental pointer analysis

Callstring approach is used for performing context sensitive incremental Liveness based pointer analysis. In this approach, we perform representation of the callstring whenever two callstrings reaches the start of the procedure with same dataflow value. And we regenerate the represented callstring at the end of the procedure. With incremental approach we have two set of values associated with each basic block, incremental value and accumulated value. Whenever incremental value is ∅, we need not propagate it further. So for incremental analysis, we do representation and regeneration in the following manner.

Representation :
If we have callstrings (σ_1, x_1, y_1) and (σ_2, x_2, y_2), reaching the start of a basic block, where x_1 and x_2 are incremental value and y_1 and y_2 are the accumulated values. Then σ_1 is represented by σ_2 if x_1 is equal to x_2 and y_1 is equal to y_2.

Regeneration :
If we have (σ_1, x_1^1, y_1^1) reaching the endblock. Then we regenerate σ_2 with incremental dataflow value x_1^1 and accumulated value as y_1^1.

Example
The working of incremental interprocedural analysis is explained through the example shown in Figure 5.7. Dataflow value is represented by set (λ, {x}, {y}), where x is the incremental value and y is the accumulated value. First we will perform incremental liveness analysis on the above supergraph.

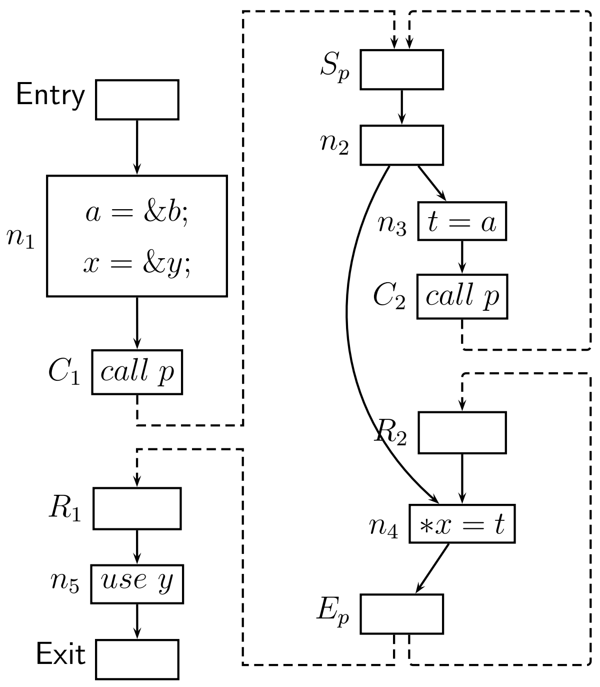


Figure 5.7: Supergraph of a recursive program

In first pass we will have variable x live at the entry of node of n_4, which enables generation of points-to pair $x \rightarrow y$ in first pass of incremental L-FCPA.Since, variable y is live at the exit of node n_4 and $x \rightarrow y$ reaches the entry of node n_4, variable t becomes live with callstring c_1 *and* c_1c_2. So dataflow value reaching the exit of node n_2 in first iteration is $\{(c_1,\{x,t\},\{x,t\}),(c_1c_2,\{x,t\},\{x,t\})\}$. When this dataflow value reaches the exit of node n_3, variable t is killed and variable a becomes live at entry of node n_3. The incremental live set ({x,a}) at node n_3 is propagated to the exit of node n_2. So the dataflow value at the exit of node n_2 will become $\{(c_1,\{a\},\{a,x,t\}),(c_1c_2,\{a\},\{a,x,t\})\}$, i.e. we have only variable {a} in the incremental set. Now this incremental value is propagated to its predecessors. The detailed working of this algorithm is shown in appendix A.1.

In second pass of liveness first we have dataflow (c_1,{y},{y}) reaching the end node of procedure p(E_p). This dataflow value becomes (c_1,{x,t},{x,t}) at the entry of node n_4. Now at node R_2, new callstring is generated. So we have new dataflow value (c_1c_2,{x,t},{x,t}) reaching at E_p. Since incremental and accumulated values are different for callstring c_1 and c_1c_2, we cannot perform representation. In subsequent iteration, we will have ($c_1c_2c_2$,{x,t},{x,t}) reaching the node E_p and since the incremental and accumulated dataflow value is same as callstring c_1c_2, we represent callstring $c_1c_2c_2$ by c_1c_2.

When the callstring c_1c_2 reaches the node S_p for the first time, the dataflow values associated with it are (c_1c_2,{x,t},{x,t}). We perform regeneration of callstring $c_1c_2c_2$ with the dataflow values of callstring c_1c_2 at the node S_p. The value of callstring c_1c_2 gets modified at node n_3 and we have a new set of incremental dataflow value (c_1c_2,{a},∅) reaching the node S_p. We now compute the accumulated live set at node S_p and the new dataflow value obtained is (c_1c_2,{a},{a,x,t}). After computing accumulated set we perform the regeneration of callstring $c_1c_2c_2$ at node S_p.

5.5 Implementation

Current L-FCPA implementation is extended to support the incremental interprocedural pointer analysis. First incremental liveness analysis was implemented and then incremental pointer analysis. For incremental pointer analysis, data structure for storing the dataflow information at a program point is modified, a new field was introduced which stores the incremental dataflow value at that point. It is also an index in the `VEC` . The data structure that stores context sensitive information at a program point is :

```
struct cs_dfa
{
   int linfo, cs_index;
   int pinfo; /*accumulated pointer information*/
   int incr_pinfo /* incremental pointer information */
   bool new_lvdf, new_ptadf;
   struct cs_dfa *next;
};
typedef struct cs_dfa *csdfa_info;
```

The algorithm works in following steps :

- Pointer information is computed at the out of a basic block based on the constraints in the basic block and is stored in `incr_pinfo`.
- Final incremental data flow values at out are computed by removing the pointer information in `pinfo` from `incr_pinfo`.

- Propagate the incremental pointer information `incr_pinfo` to the in of all the successor basic blocks and add the successors to the worklist.
- If `incr_pinfo` is less than 0(incremental value is $\emptyset$), then we do not propagate the information to its successors and the successor nodes are not added to the worklist.
- Final incremental value is computed at the in of the successor by removing the information in `pinfo` from `incr_pinfo.`
- Compute the accumulated pointer information `pinfo` from `incr_pinfo` at the in and out of the basic_block.
- For callstring representation, value of both `pinfo` and `incr_pinfo` are compared.
- Regenerated callstring's `pinfo` and `incr_pinfo` is same as that of represented callstring's `pinfo` and `incr_pinfo.`

Currently, we have two implementations, incremental liveness and incremental points-to analysis. These implementations are tested on small program and the analysis is generating the correct results. The implementation fails to give correct and precise results for large size codes such as the benchmark programs. More debugging of code is required in order to find the cases where the implementation fails to give the correct results.

5.6 Summary

Incremental pointer analysis may be very useful in case of large programs because we have seen that with the size of program, number of pointer statements also increases and as a result we have large number of pointer pairs in the points-to set at a program point. Incremental analysis will reduce number of computation as only small incremental set will be propagated. The number of incremental pointer pairs computed for L-FCPA shows that the incremental analysis can improve the L-FCPA algorithm by avoiding redundant computations. Widening operation is required in order to achieve distributivity in incremental points-to analysis.The current implementation of incremental pointer analysis gives correct results with small programs. But for large size codes it fails to give the correct results. So the implementation need to be scaled for large size codes.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We can conclude that L-FCPA performs flow and context sensitive pointer analysis effectively. Also, the emami's algorithm which is considered as most context sensitive, is proved to be context insensitive. The experiments performed on the SPEC benchmark using the current implementation of L-FCPA in GCC, shows that its performance is far better than gcc pta. The number of pointer pairs are decreased significantly because of sparse propagation and lazy computation of pointer information. This implementation uses call string approach for interprocedural analysis which allowed context sensitive flow of information.

It is observed that current implementation fails for the programs having more than 30KLOC. Results show that it is because of use of linked list. An efficient data-structure is required in order to improve the efficiency of code. Data-structure is accessed directly in the current implementation and therefore, we have designed the APIs, such that, access to data-structure is hidden behind these APIs. This will make process of changing data-structure easier.

From the observation, it can be concluded that there is a great scope of performing incremental analysis because there are few changes in data flow set from previous iteration and set obtained in current iteration. This could further enhance the analysis particularly for the programs having large size points-to set at a program point. Incremental pointer analysis of L-FCPA is non-distributive, therefore in order to ensure distributivity widening operation is performed. We have formulated the incremental analysis across different iterations at interprocedural level for L-FCPA, where representation and regeneration is required for callstring approach. Incremental approach can be used across different iterations and across different fixed point computations of liveness and points-to analysis for L-FCPA. We have seen in the examples that in both the cases, use of incremental analysis reduced large number of computations. This is because we propagate only the incremental information. There is third level of incremental analysis, performed across different applications. But this analysis is not preferred as it is non-monotonic.

6.2 Future Work

Future Work includes the following task :

- Replacing the link list data structure implementation used for storing pointer information by more sophisticated data structure such as vector, BDDs or bitmaps.
- Extending the L-FCPA algorithm for heap variables.
- Including the L-FCPA in GCC, such that results of L-FCPA can be used by other optimization passes.
- Modifying APIs to have a precise design.
- Study of incremental analysis across different fixed point computation.

Appendix A

Incremental interprocedural L-FCPA example

This is the detailed working of example in Figure 5.7. First we perform incremental liveness pass on the above supergraph. Dataflow value represented by set (λ, {x}, {y}), where x is incremental value and y is accumulated value. Following order is followed while computing the dataflow values, $\Delta \mathit{Lout}_n, \Delta \mathit{Lin}_n$ and now these incremental values are used to compute Lout_n and Lin_n

First pass of incremental liveness

S. No	Node	New value	Accumulated value at Entry	Resulting Worklist (list of Node: [value])	Remarks
1	E_m	(λ, $\emptyset$, $\emptyset$)	(λ, $\emptyset$, $\emptyset$)	n_5 : [(λ, $\emptyset$, $\emptyset$)]	
2	n_5	(λ,{y},{y})	(λ,{y},{y})	R_1 : [(λ,{y}, $\emptyset$)]	
3	R_1	(c_1,{y},{y})	(c_1,{y},{y})	E_p : [(c_1,{y}, $\emptyset$)]	
4	E_p	(c_1,{y},{y})	(c_1,{y},{y})	n_4 : [(c_1,{y}, $\emptyset$)]	
5	n_4	(c_1,{x},{x})	(c_1,{x},{x})	R_2 : [(c_1,{x}, $\emptyset$)] n_2 : [(c_1,{x}, $\emptyset$)]	
6	R_2	(c_1c_2,{x},{x})	(c_1c_2,{x},{x})	E_p : [(c_1c_2,{x}, $\emptyset$)] n_2 : [(c_1,{x}, $\emptyset$)]	
7	E_p	(c_1c_2,{x},{x})	(c_1,{y},{y}) (c_1c_2,{x},{x})	n_4 : [(c_1c_2,{x}, $\emptyset$)] n_2 : [(c_1,{x}, $\emptyset$)]	

S. No	Node	New value	Accumulated value at Entry	Resulting Worklist (list of Node: [value])	Remarks
8	n_4	(c_1c_2,{x},{x})	(c_1,{x},{x}) (c_1c_2,{x},{x})	R_2 : [(c_1c_2,{x}, ∅)] n_2 :[(c_1,{x},∅),(c_1c_2,{x},∅)]	
9	R_2	($c_1c_2c_2$,{x},{x})	(c_1,{x},{x}) (c_1c_2,{x},{x}) ($c_1c_2c_2$,{x},{x})	E_p : [($c_1c_2c_2$,{x}, ∅)] n_2 :[(c_1,{x},∅),(c_1c_2,{x},∅)]	
10	E_p	No Change	(c_1,{y},{y}) (c_1c_2,{x},{x})	n_2 :[(c_1,{x},∅),(c_1c_2,{x},∅)]	$c_1c_2c_2$ is represented by c_1c_2
11	n_2	(c_1,{x},{x}) (c_1c_2,{x},{x})	(c_1,{x},{x}) (c_1c_2,{x},{x})	S_p :[(c_1,{x},∅),(c_1c_2,{x},∅)]	
12	S_p	(c_1,{x},{x}) (c_1c_2,{x},{x}) ($c_1c_2c_2$,{x},{x})	(c_1,{x},{x}) (c_1c_2,{x},{x}) ($c_1c_2c_2$,{x},{x})	c_2 :[(c_1c_2,{x},∅), ($c_1c_2c_2$,{x},∅)] c_1 :[(c_1,{x},∅)	Regenerate $c_1c_2c_2$ from c_1c_2
13	c_2	(c_1,{x},{x}) (c_1c_2,{x},{x})	(c_1,{x},{x}) (c_1c_2,{x},{x})	n_3 :[(c_1,{x},∅), (c_1c_2,{x},∅)] c_1 :[(c_1,{x},∅)	
14	n_3	(c_1,{x},{x}) (c_1c_2,{x},{x})	(c_1,{x},{x}) (c_1c_2,{x},{x})	n_2 :[(c_1,{x},∅), (c_1c_2,{x},∅)] c_1 :[(c_1,{x},∅)]	
15	n_2	(c_1,∅,{x}) (c_1c_2,∅,{x})	(c_1,∅,{x}) (c_1c_2,∅,{x})	c_1 :[(c_1,{x},∅)]	Since ΔLin_n is ∅, we will not add pred of n_2.
16	c_1	(λ,{x},{x})	(λ,{x},{x})	n_1 :[(λ,{x},∅)]	
17	n_1	(λ,∅,∅)	(λ,∅,∅)	S_m :[(λ,{x},∅)]	

.

It can be seen from above table that, only incremental values are propagated to the pred nodes and as soon as incremental value becomes ∅, we stop adding nodes in the worklist. As incremental value ∅ means that there is no further change in the dataflow values.

First pass of incremental points-to analysis

S. No	Node	New value	Accumulated value at Exit	Resulting Worklist (list of Node: [value])	Remarks
1	S_m	$(\lambda, \emptyset, \emptyset)$	$(\lambda, \emptyset, \emptyset)$	n_1 : $[(\lambda, \emptyset, \emptyset)]$	
2	n_1	$(\lambda$,(x,y),(x,y))	$(\lambda$,(x,y),(x,y))	c_1 : $[(\lambda$,(x,y),$\emptyset$)]	
3	c_1	(c_1,(x,y),(x,y))	(c_1,(x,y),(x,y))	S_p : [(c_1,(x,y),$\emptyset$)]	
4	S_p	(c_1,(x,y),(x,y))	(c_1,(x,y),(x,y))	n_2 : [(c_1,(x,y),$\emptyset$)]	
5	n_2	(c_1,(x,y),(x,y))	(c_1,(x,y),(x,y))	n_3 : [(c_1,(x,y),$\emptyset$)] n_4 : [(c_1,(x,y),$\emptyset$)]	
6	n_3	(c_1,(x,y),(x,y))	(c_1,(x,y),(x,y))	c_2 : [(c_1,(x,y),$\emptyset$)] n_4 : [(c_1,(x,y),$\emptyset$)]	
7	c_2	(c_1c_2,(x,y),(x,y))	(c_1c_2,(x,y),(x,y))	S_p : [(c_1c_2,(x,y),$\emptyset$)] n_4 : [(c_1,(x,y),$\emptyset$)]	
8	S_p	No change	(c_1,(x,y),(x,y))	n_4 : [(c_1,(x,y),$\emptyset$)] E_p : []	c_1c_2 is represented by c_1
9	n_4	(c_1,$\emptyset$,$\emptyset$)	(c_1,$\emptyset$,$\emptyset$)	E_p : []	Since $\Delta Aout_n$ is $\emptyset$, no succ is added
10	E_p	(c_1,$\emptyset$,$\emptyset$) (c_1c_2,$\emptyset$,$\emptyset$)	(c_1,$\emptyset$,$\emptyset$) (c_1c_2,$\emptyset$,$\emptyset$)		

.

The dataflow value {(x,y)} in above table, represents pointer pair x $\rightarrow$ y.

Second pass of incremental liveness

S. No	Node	New value	Accumulated value at Entry	Resulting Worklist (list of Node: [value])	Remarks
1	E_m	$(\lambda, \emptyset, \emptyset)$	$(\lambda, \emptyset, \emptyset)$	n_5 : $[(\lambda, \emptyset, \emptyset)]$	
2	n_5	$(\lambda$,{y},{y})	$(\lambda$,{y},{y})	R_1 : $[(\lambda$,{y}, $\emptyset$)]	
3	R_1	(c_1,{y},{y})	(c_1,{y},{y})	E_p : [(c_1,{y},$\emptyset$)]	
3	E_p	(c_1,{y},{y})	(c_1,{y},{y})	n_4 : [(c_1,{y}, $\emptyset$)]	
4	n_4	(c_1,{x,t},{x,t})	(c_1,{x,t},{x,t})	R_2 : [(c_1,{x,t}, $\emptyset$)] n_2 : [(c_1,{x,t}, $\emptyset$)]	

S. No	Node	New value	Accumulated value at Entry	Resulting Worklist (list of Node: [value])	Remarks
5	R_2	(c_1c_2,{x,t},{x,t})	(c_1c_2,{x,t},{x,t})	E_p : [(c_1c_2,{x,t}, $\emptyset$)] n_2 : [(c_1,{x,t}, $\emptyset$)]	
6	E_p	(c_1c_2,{x,t},{x,t})	(c_1,{y},{y}) (c_1c_2,{x,t},{x,t})	n_4 : [(c_1c_2,{x,t},$\emptyset$)] n_2 :[(c_1,{x,t},$\emptyset$)]	
7	n_4	(c_1c_2,$\{x,t\}$,$\{x,t\}$)	(c_1,{x,t},{x,t}) (c_1c_2,$\{x,t\}$,$\{x,t\}$)	R_2 :(c_1c_2,$\{x,t\}$,$\{x,t\}$) n_2 :[(c_1,{x,t},$\emptyset$) (c_1c_2,$\{x,t\}$,$\emptyset$)]	
8	R_2	($c_1c_2c_2$,{x,t},{x,t})	($c_1c_2c_2$,{x,t},{x,t})	E_p :[($c_1c_2c_2$,{x,t},$\emptyset$)] n_2 :[(c_1,{x,t},$\emptyset$) (c_1c_2,$\{x,t\}$,$\emptyset$)]	
9	E_p	No change	(c_1,{y},{y}) (c_1c_2,{x,t},{x,t})	n_2 :[(c_1,{x,t},$\emptyset$) (c_1c_2,$\{x,t\}$,$\emptyset$)] S_p []	Representation
10	n_2	(c_1,{x,t},{x,t}) (c_1c_2,$\{x,t\}$,{x,t})	(c_1,{x,t},{x,t}) (c_1c_2,$\{x,t\}$,{x,t})	S_p :[(c_1c_2,{x,t},$\emptyset$), (c_1,{x,t},$\emptyset$)]	
11	S_p	(c_1,{x,t},{x,t}) (c_1c_2,$\{x,t\}$,{x,t}) ($c_1c_2c_2$,$\{x,t\}$,{x,t})	(c_1,{x,t},{x,t}) (c_1c_2,$\{x,t\}$,{x,t}) ($c_1c_2c_2$,$\{x,t\}$,{x,t})	c_2 :[(c_1c_2,$\{x,t\}$,$\emptyset$), (c_1,{x,t},$\emptyset$)] ($c_1c_2c_2$,$\{x,t\}$,$\emptyset$)] c_1 :[(c_1,{x,t},$\emptyset$)]	Regeneration
12	c_2	(c_1,{x,t},{x,t}) (c_1c_2,$\{x,t\}$,{x,t})	(c_1,{x,t},{x,t}) (c_1c_2,$\{x,t\}$,{x,t})	n_3 :[(c_1c_2,$\{x,t\}$,$\emptyset$) c_1 :[(c_1,{x,t},$\emptyset$)]	
13	n_3	(c_1,{x,a},{x,a}) (c_1c_2,$\{x,a\}$,{x,a})	(c_1,{x,a},{x,a}) (c_1c_2,$\{x,a\}$,{x,a})	n_2 :[(c_1c_2,$\{x,a\}$,$\emptyset$)] c_1 :[(c_1,{x,t},$\emptyset$)]	
14	n_2	(c_1,{a},{x,a,t}) (c_1c_2,$\{a\}$,{x,a,t})	(c_1,{a},{x,a,t}) (c_1c_2,$\{a\}$,{x,a,t})	S_p :[(c_1,{a},$\emptyset$) (c_1c_2,$\{a\}$,$\emptyset$)] c_1 :[(c_1,{x,t},$\emptyset$)]	
15	S_p	(c_1,{a},{a,x,t}) (c_1c_2,$\{a\}$,{a,x,t}) ($c_1c_2c_2$,$\{a\}$,{a,x,t})	(c_1,{a},{a,x,t}) (c_1c_2,$\{a\}$,{a,x,t}) ($c_1c_2c_2$,$\{a\}$,{a,x,t})	c_2 :[(c_1c_2,$\{a\}$,$\emptyset$) ($c_1c_2c_2$,$\{a\}$,$\emptyset$)] c_1 :[(c_1,{a,x,t},$\emptyset$)]	Regeneration

S. No	Node	New value	Accumulated value at Entry	Resulting Worklist (list of Node: [value])	Remarks
16	c_2	(c_1,{a},{a,x,t}) (c_1c_2,{a},{a,x,t})	(c_1,{a},{a,x,t}) (c_1c_2,{a},{a,x,t})	n_3 :[(c_1,{a},∅) (c_1c_2,{a},∅)] c_1 :[(c_1,{a,x,t},∅)]	
17	n_3	(c_1,∅,{a,x}) (c_1c_2,∅,{a,x})	(c_1,∅,{a,x}) (c_1c_2,∅,{a,x})	c_1 :[(c_1,{a,x,t},∅)	
18	c_1	(λ,{a,x,t},{a,x,t})	(λ,{a,x,t},{a,x,t})	n_1 :[(c_1,{a,x,t},∅)]	
19	n_1	(λ,{t},{t})	(λ,{t},{t})	S_m :[(c_1,{t},∅)]	
20	S_m	(λ,{t},{t})	(λ,{t},{t})		

.

In second pass of liveness, at node n_2, we have dataflow value (c_1,{x,t},{x,t}) (c_1c_2,{x,t},{x,t}) during first visit to n_2. During second iteration, {a} also becomes live at exit of n_2. When we compute the dataflow value at exit of n_2, we merge the dataflow value coming from the two paths, so we get {a,x,t} as new value. But {x,t} are already present at the out of n_2. So we get,{a} as incremental value. Now,we propagate only {a}.

Second pass of incremental points-to analysis

S. No	Node	New value	Accumulated value at Exit	Resulting Worklist (list of Node: [value])	Remarks
1	S_m	(λ,{(t,?)},{(t,?)})	(λ,{(t,?)},{(t,?)})	n_1 : [(λ, {(t,?)}, ∅)]	
2	n_1	(λ, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	(λ, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	c_1 : [(λ,{(x,y),(a,b),(t,?)},∅)]	
3	c_1	(c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	(c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	S_p : [(c_1,{(x,y),(a,b),(t,?)},∅)]	
4	S_p	(c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	(c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	n_2 : [(c_1,{(x,y),(a,b),(t,?)},∅)]	

S. No	Node	New value	Accumulated value at Exit	Resulting Worklist (list of Node: [value])	Remarks
5	n_2	(c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	(c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	n_3 : [(c_1,{(x,y),(a,b)},∅)] n_4 : [(c_1,{(x,y),(a,b),(t,?)},∅)]	
6	n_3	(c_1, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)})	(c_1, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)})	c_2 : [(c_1,{(x,y),(a,b),(t,b)},∅)] n_4 : [(c_1,{(x,y),(a,b),(t,?)},∅)]	
7	c_2	(c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)})	(c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)})	S_p : [(c_1c_2,{(x,y),(a,b),(t,b)}, ∅)] n_4 : [(c_1,{(x,y),(a,b),(t,?)},∅)]	
8	S_p	(c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)}) (c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	(c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)}) (c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	n_2 : [(c_1c_2,{(x,y),(a,b),(t,b)}, ∅)] n_4 : [(c_1,{(x,y),(a,b),(t,?)},∅)]	
9	n_2	(c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)})	(c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)}) (c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	n_3 : [(c_1c_2,{(x,y),(a,b),(t,b)}, ∅)] n_4 : [(c_1,{(x,y),(a,b),(t,?)},∅) (c_1c_2, {(x,y),(a,b),(t,b)},∅)]	

S. No	Node	New value	Accumulated value at Exit	Resulting Worklist (list of Node: [value])	Remarks
10	n_3	(c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)})	(c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)}) (c_1, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)})	c_2 : [(c_1c_2,{(x,y),(a,b),(t,b)}, ∅)] n_4 : [(c_1,{(x,y),(a,b),(t,?)},∅) (c_1c_2, {(x,y),(a,b),(t,b)},∅)]	
11	c_2	($c_1c_2c_2$, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)})	($c_1c_2c_2$, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)}) (c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)}) (c_1, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)})	S_p : [($c_1c_2c_2$,{(x,y),(a,b),(t,b)}, ∅)] n_4 : [(c_1,{(x,y),(a,b),(t,?)},∅) (c_1c_2, {(x,y),(a,b),(t,b)},∅)]	
12	S_p	No Change	(c_1c_2, {(x,y),(a,b),(t,b)}, {(x,y),(a,b),(t,b)}) (c_1, {(x,y),(a,b),(t,?)}, {(x,y),(a,b),(t,?)})	n_4 : [(c_1,{(x,y),(a,b),(t,?)},∅) (c_1c_2, {(x,y),(a,b),(t,b)},∅)] E_p[]	Representa--tion of $c_1c_2c_2$ by c_1c_2
13	n_4	(c_1c_2, {(x,y),(t,b)}, {(x,y),(t,b)}) (c_1,{(y,?)},{(y,?)})	(c_1c_2, {(x,y),(t,b)}, {(x,y),(t,b)}) (c_1,{(y,?)},{(y,?)})	E_p : [(c_1c_2,{(x,y),(t,b),(y,b)}, ∅) (c_1,{(y,?)},∅)]	

S. No	Node	New value	Accumulated value at Exit	Resulting Worklist (list of Node: [value])	Remarks
14	E_p	($c_1c_2c_2$, {(x,y),(t,b)}, {(x,y),(t,b)}) (c_1c_2, {(x,y),(t,b)}, {(x,y),(t,b)}) (c_1,{(y,?)},{(y,?)})	($c_1c_2c_2$, {(x,y),(t,b)}, {(x,y),(t,b)}) (c_1c_2,{(x,y),(t,b)}, {(x,y),(t,b)}) (c_1,{(y,?)},{(y,?)})	R_2 : [($c_1c_2c_2$, {(x,y),(t,b)},∅) (c_1c_2, {(x,y),(t,b)},∅)] R_1 : (c_1,{(y,?)},∅)	Regeneration
15	R_2	(c_1c_2, {(x,y),(t,b)}, {(x,y),(t,b)}) (c_1, {(x,y),(t,b)}, {(x,y),(t,b)})	(c_1c_2, {(x,y),(t,b)}, {(x,y),(t,b)}) (c_1, {(x,y),(t,b)}, {(x,y),(t,b)})	n_4 : [(c_1c_2,{(x,y),(t,b)},∅), (c_1,{(x,y),(t,b)},∅] R_1 : (c_1,{(y,?)},∅)	
16	n_4	(c_1c_2,∅, {(x,y),(t,b)}) (c_1, {(y,b)}, {(y,?),(y,b)})	(c_1c_2,∅, {(x,y),(t,b)}) (c_1, {(y,b)}, {(y,?),(y,b)})	E_p : (c_1c_2,∅,∅) (c_1,{(y,b)},∅) R_1 : (c_1,{(y,?)},∅)	
17	E_p	(c_1, {(y,b)}, {(y,?),(y,b)})	(c_1, {(y,b)}, {(y,?),(y,b)})	R_1 : (c_1,{(y,?),(y,b)},∅)	
18	R_1	(λ,{(y,?),(y,b)}, {(y,?),(y,b)})	(λ,{(y,?),(y,b)}, {(y,?),(y,b)})	n_5 : (λ,{(y,?),(y,b)},∅)	
18	n_5	(λ,∅,∅)	(λ,∅,∅)		

.

In second pass of pointsto-analysis, in subsequent iteration, at node n_4, we have dataflow value {(c_1,{(x,y),(t,?)},∅) (c_1c_2,{(x,y),(t,b)},∅)} and {(c_1,{(x,y),(t,b)},∅) (c_1c_2,{(x,y),(t,b)},∅)} reaching at the entry. When we merge the information, we get {(c_1,{(t,b)},{(x,y),(t,?),(t,b)}) (c_1c_2,∅,∅) } as new incremental set. Now for c_1 only {(t,b)} will be propagated and there is no need to propagate dataflow value for callstring c_1c_2.

References

[1] Standard performance evaluation corporation. `http://www.spec.org/cpu2006/`(Last accessed in Dec,2011).

[2] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN*, 1994.

[3] Uday Khedkar, Prashant Singh Rawat, and Alan Mycroft. Liveness based pointer analysis. *Static Analysis Symposium*, 2012(To Appear).

[4] Uday Khedkar, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice,*pages 119-132, 293-328. CRC Press, Inc., Boca Raton, FL, USA, 2009.