1 Introduction

1.1 Brief Introduction

Shape analysis,the aim of this static analysis technique is to infer some useful properties about the the programs changing heap structure which can be used in many areas like garbage collection,parallelization, compiler time optimization,instruction scheduling etc.

In this report we discuss the effectiveness of Sandeep's [1] work which is about field sensitive shape analysis in comparison to that of Ghiya's [2].Some of the implementation details about Sandeep's [1] analysis is also provided .When considering the interprocedural anlysis should we go for context sensitive or a context insensitive analysis; feasibility of each of it is discussed along with the supporting arguments and data. In the context insensitive case a new method of merging contexts at function calls is looked at.

Assume a particular function in a program which uses not all field pointers ,and we were asked to find the shape of a heap pointer in that function.If we go by Sandeep's[1] ,it doesn't take into consideration the field pointers absent in that function because of which the shape may have changed giving less precise shape i.e instead of considering just a subset of all the fields,everything was considered .We create a subset for each function and use the field sensitive information obtained by Sandeep's [1] work, to find the shape of the heap pointer inside that function.This subset-based field sensitive analysis can give even more precise results.

1.2 Organization Of Thesis

Some of the earlier related work is discussed in Chapter 2. Chapter 3 tells about the details of implementation of field sensitive analysis, optimization's suggested and results run on benchmarks.Details about the subset-based analysis is discussed in chapter 4 with the help of examples.The advantages and disadvantages of some interprocedural analysis methods are given in Chapter 5 along with supporting numbers.Finally details about future work in chapter 6.

2 Related Work

???????????? —about history of it ????required
??????????——-should I write about the definitions and notations of Dir,Int matrices???——-

This work actually has its roots from Ghiya's [2] analysis.In this data flow analysis the data flow values used are direction matrices,interference matrices which has entries for every heap pointer pair.This analysis doesn't take into consideration any field information so its fast but results are not precise.
In Sandeep's [1] analysis limited field sensitivity is used to infer the shape of the heap i.e a

TREE,DAG or a CYCLE. For identifying the shape they capture field sensitivity information in two ways;first by using Boolean variables to remember the direct connection between two pointer variables ,second by the use of field sensitive matrices that store the approximate path information. And at every statement Boolean equations are formed which will be able to infer the shape. The information coming out of this field sensitive analysis is a set of data flow values named Direction Matrix, Interference Matrix , Boolean Equations and field based Boolean variables for every statement(those which access the heap pointer variables) in the program across functions.It's pretty clear that the data flow values of [2] and [1] are somewhat similar. This information only is refined and used as per requirement in subset based analysis.Very little details are given about the interprocedural analysis of the same,but our report discusses it in detail.

3 Implementation Details

3.1 DFA framework

This Field Sensitive analysis is actually a data flow analysis with data flow values being Direction Matrix,Interference Matrix Boolean Equations,Boolean variables.The data flow values changes whenever any statement belonging to the 6 class of statement's of Fig. 1 is encountered.We have implemented this analysis as a GCC dynamic plugin.This next section gives us a small overview of GCC plugins.

- p = malloc();
- p=NULL;
- p=q; p=&(q→f); p=q op n;
- p→f=NULL;
- p→f=q;
- p=q→f;

$\forall$ p,q $\in$ H;

Figure 1:

3.1.1 gcc plugins

GCC architecture consists of many passes each being either a GIMPLE,IPA(Interprocedural Analysis) or RTL(Register Transfer Language) pass.GIMPLE is a three-address intermediate representation.Our analysis works by inserting a GIMPLE pass in the GCC architecture which is done by plugins.
Plugin's make the developer add new features to the compiler without modifying the compiler itself. It is a way by which we can add,remove and maintain modules independently. GCC has two types of plugins ,static and dynamic plugin's. Static plugins involve minor changes in the source code and static linking,while dynamic plugin requires no change to the source code and require dynamic linking [3] .For our case we have used dynamic plugin because we need not build GCC every time we change the plugin code.

What our plugin does is ,first it finds out what are the heap pointers present in the program and creates a table for them.That table will contain details like which structure the pointer points to,what are the fields present in it,which of those fields are also pointers etc.Similarly another table is created for the field pointers present in the program.In this process Id's are given to each field and heap pointer. After this we parse along all the gimple statements one by one and check it belongs to any of the statements of Fig. 1. If yes then the GEN and KILL data flow values are generated for that statement,OUT is calculated by below equation.All the equations for GEN and KILL of each statement are mentioned in [1].

OUT = GEN $\cup$ (IN - KILL)

This analysis is run until the data flow values reach a fixed point.When considering the fixed point we only check for the Direction Matrices ,Interference Matrices and Boolean variables,the Boolean equations are left aside because as mentioned in Sandeep's report [4] the Boolean equations reach a fixed point at the end of third iteration itself.

3.2 Optimization's suggested

3.3 Results for Benchmarks

4 Subset Based Analysis

4.1 Motivating Example

EXAMPLE 1: Consider the code segment in the Fig: 4.1(a) which has the code for searching data in a binary tree and Fig: 4.1(b) ,a small code snippet of the insert function.The datstructure of they binary tree node is also shown.The code snippet of insert is just a small set of statements that are of inserting a node into a binary tree. In the table the value of boolean varaibles $LEFT_{p,t}$,$PARENT_{t,p}$ are shown after each statement.(The statements S0,S6 were not considered at all as they donot effect the shape of pointer in any way).The value of $LEFT_{p,t}$ has become TRUE after statement S4 and value of $PARENT_{t,p}$ after S5. The direction and interference matrices after each statement is also shown in Fig: 4.1(c).

In that the pointer parent is used mainly for debugging.As you can notice the parent pointer is not at all used in the function search.During the build of the binary tree each child's parent pointer is made to point to its parent making shape of every node in the tree a cycle.But when you see the function search, parent pointer is not used at all.Though the search($s \rightarrow LEFT$,key) and search($s \rightarrow RIGHT$,key) functions can be executed in parallel ,its inhibited because the shape at pointer s is identified as a cycle. If we could identify the field pointers not used in the function and try to find the shape excluding that field pointer we can get precise shapes. Fig: 4.1(d) gives the values of boolean equations at the end of statement S5.To each entry we find S4 or S5 appended to the curly brackets just to tell that those values are to be taken from that equations data flow output values.

One way to find the shape without considering PARENT field for the function search is to do

the following, make $\forall x, y PARENT_{x,y} = 0$ and then substitute the datflow values in the boolean equation.

```
Struct Node {
Struct Node *LEFT,*RIGHT,*PARENT;
int key;
}

bool search(node *s,int key) {
if(s)
return (key==s->data)||search(s->LEFT,key)||search(s->RIGHT,key);
return 0;
}
```

```
insert(node *s,int key) {
.
.
S0 Node *p;
S1.  p=malloc(sizeof(struct Node));
S2.  p->LEFT=NULL;
S3.  p->RIGHT=NULL;
S4.  s->LEFT=p;
S5.  p->PARENT = s;
S6.  p->data=key;
.
.
}
```

(a) A code fragment

After	$LEFT_{p,t}$	$PARENT_{t,p}$
S1	false	false
S2	false	false
S3	false	false
S4	true	false
S5	true	true

(b) Boolean Variables

After Stmt	D_F			I_F		
S1		p	s		p	s
	p	ϵ	$\emptyset$	p	$\{(\epsilon,\epsilon)\}$	$\emptyset$
	s	$\emptyset$	$\emptyset$	s	$\emptyset$	$\emptyset$
S2		p	s		p	s
	p	ϵ	$\emptyset$	p	$\{(\epsilon,\epsilon)\}$	$\emptyset$
	s	$\emptyset$	$\emptyset$	s	$\emptyset$	$\emptyset$
S3		p	s		p	s
	p	ϵ	$\emptyset$	p	$\{(\epsilon,\epsilon)\}$	$\emptyset$
	s	$\emptyset$	$\emptyset$	s	$\emptyset$	$\emptyset$
S4		p	s		p	s
	p	ϵ	$\emptyset$	p	$\{(\epsilon,\epsilon)\}$	$\{(\epsilon, left^D)\}$
	s	$\{left^D\}$	$\emptyset$	s	$\{(left^D,\epsilon)\}$	$\emptyset$
S5		p	s		p	s
	p	ϵ	$\{parent^D\}$	p	$\{(\epsilon,\epsilon)\}$	$\{(\epsilon, left^D), (parent^D,\epsilon)\}$
	s	$\{left^D\}$	$\emptyset$	s	$\{(left^D,\epsilon), (\epsilon, left^D)\}$	$\emptyset$

(c) Direction (D_F) and Interference (I_F) matrices

Heap Pointer	Shape	BooleanEquation
p	dag	$\{ (PARENT_{p,s} \wedge s^{in}_{cycle}) \vee (PARENT_{p,s} \wedge (\|D[s,p]\| >= 1)) \}$S5 $\vee$ $\{ LEFT_{s,p} \vee (\|D[p,s]\| >= 1) \}$S4
	cycle	$\{ (PARENT_{p,s} \wedge (\|I[p,s]\| > 1)) \}$S5
s	dag	$\{ PARENT_{p,s} \wedge (\|D[s,p]\| >= 1) \}$S5 $\vee$ $\{ (LEFT_{s,p} \wedge p^{in}_{cycle}) \vee (LEFT_{s,p} \wedge (\|D[p,s]\| >= 1)) \}$S4
	cycle	$\{ (LEFT_{s,p} \wedge (\|I[s,p]\| > 1)) \}$S4

(d) Boolean Equations at the end

5 Interprocedural analysis

In this Chapter we discuss what interprocedural analysis method should be considered for implementing the field sensitive analysis of [1]. We have implemented this analysis in two methods, one is Callstring based and other a context insensitive analysis which intelligently merges the contexts at function calls.

5.1 Callstring Based

Before we understand what is this Callstring method we need to know the diffeence between context sensitive and context insensitive .

————Draw a diagram expaliing the difference of both————

————What is Callstring based —-(very few details abt implementation)————

————Problem ,i.e memory fault————-

————-System Config,(table showing for which benchmarks it was working and for it was notations

(benchmark name(C++ programs) ,(tree,dag,cycle),memory???,time

Why is this happening

References

[1] Sandeep Dasgupta and Dr.Amey Karkare. Precise shape analysis using field sensitivity.

[2] Rakesh Ghiya and Laurie J. Hendren. Is it a Tree, a DAG, or a Cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL '96*.

[3] `http://gcc.gnu.org/wiki/plugins`.

[4] Sandeep Dasgupta. Precise shape analysis using field sensitivity. Technical report, Master's thesis, IIT Kanpur, 2011. `http://www.cse.iitk.ac.in/users/karkare/MTP/2010-11/sandeep2010precise.pdf`.