Heap Dependence Analysis for Sequential Programs

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Technology

by

Barnali Basak
Y9111009

under the guidance of

Prof. Amey Karkare

and

Prof. Sanjeev K Aggarwal

Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur
May 2011

CERTIFICATE

It is certified that the work contained in this thesis entitled "**Heap Dependence Analysis for Sequential Programs**", by **Barnali Basak** (Roll No. Y9111009), has been carried out under our supervisions and that this work has not been submitted elsewhere for a degree.

Amey Karkare 30/5/11

Prof. Amey Karkare,
Dept. of Computer Science and Engg.,
Indian Institute of Technology,
Kanpur-208016.

Sanjeev Kumar

Prof. Sanjeev K Aggarwal,
Dept. of Computer Science and Engg.,
Indian Institute of Technology,
Kanpur-208016.

Abstract

Identifying dependences present in the body of sequential program is used by parallelizing compilers to automatically extract parallelism from the program. Dependence detection mechanisms for programs with scalar and static variables is well explored and have become a standard part of parallelizing and vectorizing compilers. However, detecting dependences in the presence of dynamic (heap) recursive data structures is highly complex because of the unbound and dynamic nature of the structure. The problem becomes more critical due to the presence of pointer-induced aliasing .

This thesis addresses the aforementioned problem and gives a novel approach for dependence analysis of sequential programs in presence of heap data structure. The novelty of our technique lies in the two-phase mechanism, where the first phase identifies dependences for whole procedure. It computes abstract *heap access paths* for each heap accessing statement in the procedure. The access paths approximate the locations accessed by the statement. For each pair of statements these access paths are checked for interference. The second phase refines the dependence analysis in the context of loops. The main aspect of the second phase is the way we convert the precise access paths, for each statement, into equations that can be solved using traditional tests, e.g. GCD test, and Lamport test. The technique discovers *loop dependences*, i.e. the dependence among two different iterations of the same loop. Further, we extend the intra-procedural analysis to inter-procedural one.

This thesis is dedicated
to my parents and my beloved friend.

Acknowledgment

I take this opportunity to express my solemn gratitude and esteemed regards to my thesis guide **Prof. Amey Karkare**, for his invaluable help and guidance throughout this work, giving me his insights whenever I was stuck, motivating me at times when I needed it at most and above all for being a kind person.

I wish to thank my thesis co-guide **Prof. Sanjeev K. Aggarwal**, for his immense support and invaluable suggestions that I have received from him.

I would also like to thank **Prof. Subhajit Roy** for all the enriching discussions that I had with him.

In the preparation of this report, I had to consult constantly various papers, articles and journals. I, hereby, acknowledge my indebtedness to them all.

Special thanks to my friend and fellow mate Sandeep Dasgupta for his encouragement and enormous help and lots of thanks to my institution without which this thesis would have been a distant reality.

Finally, I wish to extend my thanks to my parents, my sister and my beloved friend Debmalya who have always been a source of inspiration and encouragement. Their love and blessings have always been a driving force in my life to carry all my works with hope and optimism.

Every effort has been made to give credit where it is due for the material contained here in. If inadvertently I have omitted giving credit to anyone, I apologize and express my gratitude for their contribution to this work.

Barnali Basak

May 2011

Indian Institute of Technology, Kanpur

Contents

List of Figures

List of Tables

Chapter 1

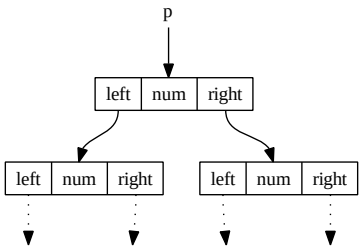
Introduction

In the recent arena of parallel architectures (multi-cores, GPUs, etc.), software side lags behind hardware. This is due to the reason that dealing with parallelism adds a new dimension to the design of programs, therefore makes it complex. One approach for efficient parallelization of programs is to explicitly induce parallelism. It includes designing of concurrent programming languages like Go, X10 etc., libraries, APIs like POSIX, OpenMP, Message Passing Interface(MPI) etc. The main advantage of such approach is that it gives simple and clear directions to the compiler to parallelize the code. But the main drawbacks of such approach is that the degree of parallelism totally depends upon the efficiency of programmer and imposing external parallelism turns the code into legacy one. The other approach is to automatically parallelizing sequential programs by compiler which extract parallelism without violating correctness. This is a key step in increasing the performance and efficiency.

1.1 Motivation

Over the past years, lot of work has been done on automatically parallelizing sequential programs. These approaches have mainly been developed for programs having only static data structures (fixed sized arrays) and written in languages such as FORTRAN [?, ?, ?, ?]. Almost all programming languages today use the heap for dynamic recursive data structures.

Therefore, any parallelization must also take into account the data dependency due to the access of common heap nodes of such structures. Finding parallelism in sequential programs written in languages, with dynamically allocated data structures, such as C, C++, JAVA, LISP etc., has been less successful. One of the reason being the presence of pointer-induced aliasing, which occurs when multiple pointer expressions



(a) Data structure

```
        void treeAdd(tree t) {
           if(t == NULL)
              return;
S1.        tl = t→left;
S2.        treeAdd(tl);
S3.        tr = t→right;
S4.        treeAddd(tr);
           t→num = tl→num + tr→num;
        }
```

(b) Function traversing the data structure.

Figure 1.1: Motivating example: function-call parallelization

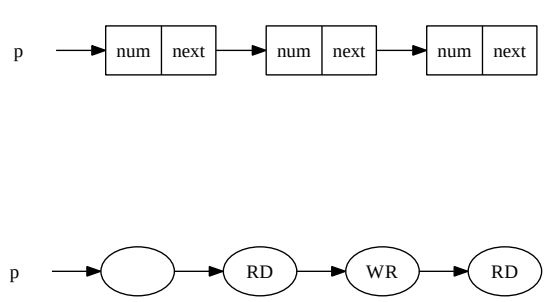
refer to same storage location. Compared to the analysis of static and stack data, analyzing properties of heap data is challenging because the structure of heap is unknown at compile time. It is also potentially unbounded and the lifetime of a heap object is not limited by the scope that creates it. As a consequence, properties of heap (including dependence) are approximated very conservatively. The approximation of the heap data dependence information inhibits the parallelization.

The objective of our analysis is to detect both coarse-grained parallelism in the context of function calls, loops and fine-grained parallelism in the context of statements. We show the following two examples as motivating examples of our work.

Example 1. Consider Figure **??** which gives a motivational example for parallelism in the context of function calls. It shows the tree data structure and the function *treeAdd* traversing on the data structure. In the code fragment the two calls to the function *treeAdd* respectively perform the additions of left and right subtrees recursively. If the analysis can ensure that the two function calls do not access any common region of heap, they can be executed in parallel. □

Example 2. Figure **??** shows an example of loop level parallelism. It shows list data structure and the nodes of the structure being read and written, tagged by *Read* (`RD`) and *Write* (`WR`) access by the code fragment traversing the data structure. Note that, the first node of the structure is a special node which is neither read nor written by the code fragment. The performance of the code can be improved if the loop can be executed in parallel. However, without the knowledge of precise heap dependences, we have to assume worst case scenario, i.e., the location read by the statement `S3` in some iteration could be the same as the location written by the statement `S5` in some other iteration. In that case, it is not possible to parallelize the loop.

Our dependence analysis can show that the locations read by `S3` and those written



(a) Nodes read and written by code

```
      ...
S1.    p = list;
       while (p→next != NULL) {
S2.       q = p→next;
S3.       temp = q→num;
S4.       r = q→next;
S5.       r→num = temp;
S6.       p = r;
       }
       ...
```

(b) Loop traversing the data structure.

Figure 1.2: Motivating example: loop parallelization

by S5 are mutually exclusive. Further, it also shows the absence of any other dependences. This information, along with the information from classical control and data dependence analysis, can be used by a parallelizing compiler to parallelize the loop. □

This report explains our approach for a practical heap data dependence analysis. As it is understood that we are only talking about data dependences, we drop the term data in the rest of the report.

1.2 Contributions of our Work

Our work contributes in the area of heap based dependence analysis. We present a novel approach which identifies dependences and extracts parallelism for a sequential program. In particular, our approach finds out dependences between two statements. This enables us to find out whether two procedure calls access disjoint structures, hence can be executed in parallel. Then we refine this technique to work better in presence of loops. We also extend the work of loop analysis for static and scalar data to support heap intensive loops.

1.3 Organization of the Thesis

The rest of the thesis is organised as follows. We discuss about related work done in the field of heap intensive dependence analysis in Chapter **??**. Chapter **??** specifies the imperative programming model for which our analysis is defined and gives the other background details. Chapter **??** through **??** provide a complete description of our practical dependence analysis applied to dynamically allocated structure. Chapter **??**

gives the detailed explanation of the intra-procedural dependence detection technique which separately works on each procedure of a program. Chapter ?? presents our method to handle loops in a more specific way. We also give the inter-procedural framework for our analysis in Chapter ??. Chapter ?? demonstrates our whole method by extensively analysing few benchmark codes. We conclude the report in Chapter ?? by giving the direction for future research.

Chapter 2

Related Work

Data dependence analysis for sequential programs, working on only static and stack related data structure, such as array, is well explored in literature [?, ?, ?, ?, ?] etc. Our work extends the work to handle heap data structure. Various approaches have been suggested for data flow analysis of programs in the presence of dynamic data structures. Classic work done by Jones and Muchnick [?] have suggested flow analysis approach for lisp-like structures. It can not handle procedures, and was designed to statically distinguish among nodes which can be immediately deallocated, nodes which are garbage collected and nodes which are referred. They have also introduced the notion of k-limited graphs as finite approximation of unlimited length of linked structure. This k-limited graph can only keep paths of at most length k, and summarizes all the nodes beyond length k. This approach is not precise enough to be used in the context of interference analysis and extract parallelism.

Jones and Muchnick in [?] have proposed a general purpose framework for data flow analysis of programs with recursive data structure. It depends on tokens to designate the points in a program where the dynamic recursive data structure is either created or modified and approximates the values of these tokens. Retrieval function is used to represent the inter-relationship among tokens and their corresponding values. By efficient choice of token sets and lattice sets, which approximate data values, high degree of exact data flow information can be obtained. Although flexible the method is mostly of theoretical interest and is potentially expensive in both time and space. Larus and Hilfinger [?] describe a dataflow computation using alias graph that records aliases between variables, structures, and pointers of the underlying data structure. This information is further used to detect conflicts between the locations accessed by the program statements.

The work by Hendren et al. in [?, ?] considers shape information and approximates

the relationships between accessible nodes in larger aggregate data structures. These relationships are represented by path expressions, restricted form of regular expressions, and are encoded in path matrices. Such matrices are used to deduce the interference information between any two heap nodes. These informations are further used to extract parallelism. Their method focuses on three levels of parallelization; (a)if two statements can be executed in parallel, (b)identifies procedure-call parallelism, and (c)whether two sequences of statements can be parallelized. They have further extended this work in [?], where they provide the programmer with a descriptor mechanism such as *Abstract Description of Data Structures*. The properties of data structure, expressed by such descriptor, are used to increase the accuracy and effectiveness of alias analysis. This efficient analysis is used for transformation of programs with recursive pointer structure.

The idea behind the work done by Hummel et. al. [?] is to detect precise dependences between two statements by collecting access paths with respect to handle node and deducing the interference of these paths by proving theorems with the help of aliasing axioms. The axioms describe uniform properties of underlying data structures which precisely works for even complex cyclic structures. Although this approach precisely identifies dependences between statements in sequence, iterations of loop and block of statements, this technique is mainly of theoretical interest.

Ghiya et. al. [?] uses coarse characterization of the underlying data structure as Tree, DAG or Cycle. The work done by Ghiya computes complete access paths for each statement in terms of *anchor* pointer, which points to a fixed heap node in the data structure within the whole body of the program. The test for aliasing of the access paths, relies on connection and shape information that is automatically computed. They have also extended their work to identify loop carried dependences for loop level parallelism. Hwang and Saltz [?] present a technique to identify parallelism in programs with cyclic graphs. The method identifies the patterns of the traversal of program code over the underlying data structure. In the next step the shape of the traversal pattern is detected. If the traversal pattern is acyclic, dependence analysis is performed to extract parallelism from the program.

Navarro et. al. in [?, ?] propose a intra-procedural dependence test which intermixes shape analysis and dependence analysis together. During the analysis, the abstract structure of the dynamically allocated data is computed and is also tagged with read/write tags to find out dependencies. The resulting analysis is very precise, but it is costly. Further their shape analysis component is tightly integrated within the dependence analysis, while in our approach we keep the two separate as it gives

us modularity and the scope to improve the precision of our dependence analysis by using a more precise shape analysis, if available. They have extended their dependence related work in [?]. In this paper they have implemented a context-sensitive interprocedural framework which successfully detects dependences for both non-recursive and recursive functions.

Work done by Marron et. al. in [?] tracks a two program location, one read and one write location, for each heap object field. The technique uses an explicit store heap model which captures the tag information of objects for each program statement. The read and write information are used to detect dependences. This space effective and time efficient technique analyses bigger benchmarks in shorter time. But the effectiveness of this approach lies in the use of predefined semantics for library functions [?], which recognizes a traversal over a generic structure.

Our approach is closest to the technique proposed by Horwitz et. al. [?]. They also associate read and write sets with each program statement to detect heap dependencies. They have also proposed technique to compute dependence distances for loop constructs. However, there technique requires iterating over a loop till a fixed point is reached, which is different from our method of computing loop dependences as a set of equations in a single pass, and then solving these equations using classical tests.

Another recent approach for dependence detection and parallelization is using separation logic. It has also been used in the area of shape analysis and program verification. This technique can not directly fit into parallelization as it only expresses separation of memory at a single program point which is not sufficient to determine independences between statements. Raza et. al. [?] has presented a technique to extract parallelism from heap intensive sequential programs. The objective behind the approach is to record how parts of the heap are disjointly accessed by different statements of the program. They have extended the separation logic with *labels*, that keep track of memory regions throughout an execution of the program. They have also proved the soundness of the approach for simple list and tree structure.

Chapter 3

Background

In this chapter we present the background details required for our heap intensive data dependence analysis.

3.1 Programming Model

Each and every node of a heap recursive data structure can be accessed by access path, which is defined as pointer variable or variable followed by link fields, similar to languages like Fortran 90, C. As an example, a node N can be accessed by either pointer variable ptr or pointer variable followed by link fields like $ptr \rightarrow f_1 \rightarrow f_2 \cdots f_k$, where f_1, f_2, $\cdots$, f_k are pointer fields of heap structure. All statements in the program are pre-processed to provide normalized binary access paths, defined as pointer variable followed by single pointer field reference like $ptr \rightarrow f$. The model of the programming language to be analysed closely resembles the model of imperative language like C. We are interested in analysing only heap related statements. Here we enumerate with details the basic statements operating on heap. Note that, arithmetic operations of pointers, as in C, are not allowed.

- Heap allocating statements :
  - `p = malloc()` : A new heap object is allocated, which is pointed to by pointer `p`. Hence, this statement is called as memory allocation statement.
- Pointers assigning statements :
  - `p = q` : Pointer `p` points to the same heap location as pointed to by `q`. It inherits all the relations and properties of `q`. Hence this statement results in `p` and `q` to be alias.

  - `p = q→f` : This statement makes pointer `p` to access the heap object which is accessible by pointer `q` through the pointer field `f`. This type of statement is mainly used to traverse the links of recursive dynamic data structure.
  - `p = NULL` : This statement assigns pointer variable `p` to null, such that `p` does not point to any heap location.

- Link defining/ structure updating statements :
  - `p→f = NULL` : This statement breaks the link `f` emanating from the heap node pointed to by pointer variable `p`. After the execution of the statement `p` can not reach any other heap node through link field `f`.
  - `p→f = q` : This statement first breaks the link `f` of the heap node pointed to by `p` and then resets `f` such that `p` through link field `f` access the same heap node pointed to be pointer variable `q`.

- Heap reading/ writing statements :
  - `··· = p→data` : Data field of the heap node pointed to by pointer variable `p` is accessed. Hence this statement is used to read the data value of heap node.
  - `p→data = ···`: Data field of heap node, pointed to by heap directed pointer variable `p`, is written by this statement. Hence this statement clearly write into heap nodes.

Our analysis mainly works on those statements which do not update or modify the structure of the underlying dynamic data structure. The statements which only traverse the heap structure, reading or writing the heap data, are main candidate statements of our dependence analysis. The effects of the statements, which update the structure, are captured by shape analysis explained in later section. Here we list the statements which are handled by our data dependence analysis.

- `p = q` : aliasing statement.
- `p = q→f` : link traversing statement.
- `··· = p→data` : statement reading heap data.
- `p→data = ···` : statement writing into heap data.

Note that, only single-level of pointer dereferencing is allowed. Other than basic heap related statements, heap intensive procedure calls are also taken into account. Hence procedure calls, whose parameters point to heap nodes, are also analysed by our analysis.

3.2 Dependence Analysis

Dependence analysis produces the execution order constraints between any two statements in a program as described in literature [?, ?, ?] etc. Two classes of dependences are present; (a)Control dependence, where the execution of a statement depends on the control flow constructs, (b)Data Dependence, which arise between two statements `S` and `T` if there exists an execution path between these two statements and they access or modify same data resource.

3.2.1 Control Dependence

Use of control constructs in the program body imposes control dependences. Statement `T` is said to be control dependent on a statement `S` if (a)there exists an execution path from `S` to `T` and (b)the execution of `T` depends upon the outcome of statement `S`. A typical example of such dependence is the use of *if-then-else* construct. In this case the statements present in then or else body can not be executed before the execution of if statement. The other examples of such dependence occurs due to control flow construct like *while*, *do-while* etc.

3.2.2 Data Dependence

The other type of dependence is data related dependences [?, ?, ?], which can be generally classified into following four categories.

1. Flow (True) Dependence(Read after Write): Statement `T` is flow dependent on statement `S` if and only if an execution path exists from `S` to `T` and `T` reads a data which is already modified by `S`.

2. Anti Dependence(Write after Read): Statement `T` is antidependent on statement `S` if and only if statement `T` modifies a data which is already read by `S` and `S` precedes `T` in execution.

3. Output Dependence(Write after Write): Statement `T` is output dependent on statement `S` if and only if both `S` and `T` modify the same data and `S` precedes `T`

in execution.

4. Input Dependence(Read after Read): A statement `T` is input dependent on statement `S` if and only if `S` and `T` read the same data resource and `S` precedes `T` in execution.

Anti and output dependences are false dependence because they can be easily removed by some techniques like variable renaming etc. Input dependence does not impose any dependence as it does not prohibit reordering of instructions. This data dependence analysis is extended to tackle dependencies within loops. The next section gives an overview of loop dependence analysis.

3.2.2.1 Loop Dependence

Loop dependence analysis is a task of determining whether statements present in the loop body form dependency within same iteration or across iterations. These dependences can be categorized into the following classes:

1. Loop-carried Dependence : If statement `S` in one iteration depends on statement `T` executed in other iteration.

2. Loop-independent Dependence : If two statements `S` and `T` depend on each other in the same iteration of loop.

Different iterations of loop can be effectively executed in parallel if the execution of iterations do not depend on each other, i,e., no loop-carried dependence is present. To classify dependence, compiler uses two parameters: (a)Distance Vector, which indicates distance between two iterations dependent on each other, (b)Direction vector, indicates the sign of the distance. Based on the direction vector different classes of dependence can be identified. There exist several techniques which are used to tackle loop dependence problem. For detect whether a dependence exists, *GCD*, *Lamport*, and *Banerjee* tests are most general tests in use. Here we give brief details of *GCD* and *Lamport* tests.

GCD Test

A simple and sufficient test for the absence of loop carried dependences is the GCD test [**?**]. A loop carried dependency can occur between any two accesses of the same array `X` such as `X[a*i+b]` and `X[c*i+d]`, if greatest common divisor of `a` and `c` divides `(d-b)`. GCD test has some limitations such that it does not consider loop bounds,

and does not provide distance and direction vectors. Beside these GCD ends up by producing very conservative result as GCD of any two integers is often one.

Lamport Test

Lamport test [?] is a simple test for index expressions involving a single index variable and with a constraint that the coefficients of the index variable must be same. In this given scenario, Lamport test can detect both loop-carried and loop- independent dependencies with both distance and direction vector. Let us consider an example where two accesses of same array such as `X[a*i + b]` and `X[a*i + c]` form equation

$$\mathtt{a} * \mathtt{i}_1 + \mathtt{b} = \mathtt{a} * \mathtt{i}_2 + \mathtt{c} \equiv \mathtt{i}_2 - \mathtt{i}_1 = (\mathtt{b} - \mathtt{c})/\mathtt{a}$$

If the above equation returns an integer solution then any potential dependence is reported. Here dependence distance σ is `(b-c)/a`, if σ exists between lower and upper bound of the loop. It reports true dependence if σ>0, anti dependence when σ<0, and loop-independent dependence if σ=0.

3.3 Data Flow Analysis

Data flow analysis [?, ?, ?] is a technique for gathering particular information at each program point of a program. It is inherently flow sensitive, i,e., depends on the order of statements of the program. The flow of analysis mainly fits into one of the following three categories: (a)Forward flow analysis, where the flow of analysis propagate in the forward direction, and exit or out state of a basic block is the function of the entry or in state of it, (b)Backward flow analysis, if the analysis move in the backward direction, and the transfer function is applied to the exit state yielding the entry state, (c)Bidirectional analysis, if the flow of analysis move in both direction.

Transfer function is mainly the composition of the effects of the statements in the basic block. Hence, for each block `b` transfer function $\mathtt{trans}_\mathtt{b}$:

$$\mathtt{out}_\mathtt{b} = \mathtt{trans}_\mathtt{b}(\mathtt{in}_\mathtt{b})$$

$$\mathtt{in}_\mathtt{b} = \mathtt{join}_{\mathtt{p} \in \mathtt{pred}_\mathtt{b}}(\mathtt{out}_\mathtt{p})$$

The join operation combines the out or exit states of the predecessors $\mathtt{p} \in \mathtt{pred}_\mathtt{b}$ of `b`, returning the entry state of `b`. By solving this set of equations, the entry and/ or exit states are used to derive properties at each block boundary. Properties for each

```
initialize sets of Entry node
for i ← 1 to N of basic blocks
   initialize the sets of block i
change ← true
while (change)
   change ← false
   for i ← 1 to N of basic blocks
      In[i] = ⋃_{j∈pred[i]}(Out[j])
      Temp = trans_i(i)
      if Out[i] ≠ Temp then
         change ← true
         Out[i] ← Temp
```

Figure 3.1: Round-robin iterative analysis.

statement inside a block can also be derived separately by applying proper transfer function.

Iterative analysis is one of the most widely used techniques for data flow analysis. In case of forward flow analysis, it starts with an approximation of the in state for each basic block. The transfer function computes the out state for each block from its in state. Again the in states are updates by applying the join operations on the out states of its predecessors. These steps, excluding initialization, are repeated until the the system is stabilized, i,e., reaches fixed-point. The basic *round-robin iterative* algorithm for forward flow analysis is given in Figure **??**. This classic round-robin iterative algorithm completely sweeps over the graph such that it visits every node in a fixed-order. The time bound for this algorithm is high due to the fact that the algorithm evaluates some unnecessary computations. The *worklist iterative analysis* approach improves on the round-robin iterative algorithm, in terms of time, by computing on regions in the graph where information is changing. Figure **??** outlines the worklist iterative algorithm. The algorithm initializes all the nodes accordingly and construct an initial worklist. It then continues by removing a node from the worklist and updating its data flow information. If the value of the node changes, then all the nodes that depend on the changed information are added to the worklist. These algorithms can also be improved by bit vector technique, where data sets are are represented efficiently as *bit vectors*, in which bit represents set membership of one particular element.

```
Worklist ← φ
for i ← 1 to N of basic blocks
   initialize the sets of block i
   add i to the Worklist
while (Worklist ≠ φ)
   remove a node i from the Worklist
   recompute set at node i
   if new set ≠ old set for i then
      add each successor of i to Worklist, uniquely
```

Figure 3.2: Worklist iterative analysis.

3.4 Shape analysis

Shape analysis, as described in literature [?, ?, ?, ?, ?] etc., is used to statically analyse a program to determine various informations regarding dynamically allocated recursive data structures. It detects various features of the heap structure like interfering and sharing of nodes, reachability of heap node, disjointness of structures etc. Shape analysis also gives a coarse classification of the shape of underlying recursive heap structure. The shape is classified into one of the following three categories: (a)*Tree*, in which each node has at most one parent and no two paths can lead to same heap node, (b)*DAG*, in which some node has more than one parent and two paths can access same node, but it does not contain any cycle, (c)*Cycle*, structures having graph theoretic cycle, and a node can be potentially accessed by infinite number of paths.

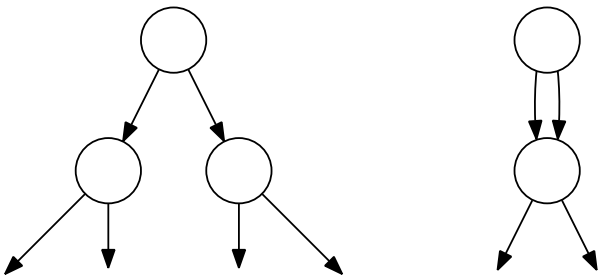


Figure 3.3: Structure of Tree and DAG

The potential for parallelism in programs that use recursive structures arises from the following observation. If the underlying data structure is of type tree, then unrelated sub-trees, $\mathtt{T_i}$ and $\mathtt{T_j}$, of tree T are guaranteed to share no common storage, hence

computation on T_i will not interfere with computation on T_j or any sub-tree of it. For the DAG structure, sub-tree T_i can potentially interfere with sub-tree T_j. Hence parallelism can be extracted if and only if it is ensured that the body of code do not access any shared node. Parallelism from cyclic structure can not be easily extracted due to the presence of cyclic nature, hence conservative decision is made.

Chapter 4

Intra-Procedural Dependence Analysis

Recall from Section ??.2 that two statements `S` and `T` are said to be dependent on each other if (a)there exists an execution path from `S` to `T` and (b)both statements access same data. As input dependence does not put any constraint on parallelization, our analysis takes into account the other types of dependences like flow, anti and output dependences. Here we redefine the general definition of data dependence in the context of heap.

Definition 4.1. *Two statements* `S` *and* `T` *are said to heap dependent on each other if (a) there exists an execution path from* `S` *to* `T` *and (b)both statements access same heap location and (c)at least one of the statements writes to that location.*

We have developed a novel technique which finds out heap induced dependencies, between any two statements in the program. The novelty of our approach lies in the separation of shape analysis phase from the dependence detection phase and the workflow of our dependence detection technique. Our intend is to identify dependences present in the program following the algorithms outlined in this chapter. Note that, our algorithms only deal with normalized statements (refer to Section ??.1), having single level of pointer dereference.

In brief, our analysis works as follows: for each heap accessing statement in the program, our approach computes set of states i,e., set of symbolic locations potentially accessed by the variables in the statement and then it computes sets of locations which is read or written by each statement. These sets are then tested to identify dependences. This dependence analysis technique identifies memory locations in terms of abstract access paths. The abstraction scheme has been designed to reach the fixed-point for our algorithm. The details of such abstraction scheme are given next. Section ?? gives overall algorithms of our analysis with details and presents the working of our algorithms with an example.

```
fun analyze(f, k) {
  InitSet = initialize(); /⋆ Initialize parameters and globals  ⋆/
  ∀ stmt S_i ∈ HeapStmt[f]
     tagStmt(S_i, tagDir(UsePtrSet, DefPtrSet, AccType, Accfield));
  HeapState = stateAnalysis(CFG[f]:(N, E, Entry, Exit), k);
  ReadWriteSet = computeReadWrite(HeapState);
  detectDependence(ReadWriteSet);
}
```

Algorithm to analyze a function `f` for dependence detection. Parameter `k` is used for limiting the length of access paths, to keep the analysis bounded.

Figure 4.1: Intra-procedural dep. detection for a function

4.1 Access Path Abstraction

An access path is either a symbolic location l_0 or location followed by a sequence of one or more pointer field names like $l_0 \rightarrow f_1 \rightarrow f_2 \rightarrow ... \rightarrow f_k$. Since an access path represents a path in a memory graph, it can be used to identify a heap node. It is hard to handle full length access path and the termination of the analysis becomes impossible. Hence we limit the length of access path to length k i,e., maximum k level of indirection is allowed for dereferencing. A special summary field '*' is used to limit the access paths, which stands for any field dereferenced beyond length k.

Example 3. For k = 1, all the access paths in the set $\{l_0 \rightarrow \texttt{next} \rightarrow \texttt{next}, l_0 \rightarrow \texttt{next} \rightarrow \texttt{next} \rightarrow \texttt{next}, l_0 \rightarrow \texttt{next} \rightarrow \texttt{next} \rightarrow \texttt{next} \cdots \rightarrow \texttt{next}\}$ can be abstracted as single summarized path $l_0 \rightarrow \texttt{next} \rightarrow *$. Similarly assuming a data structure has two reference fields `left` and `right`, the summarized path $l_0 \rightarrow \texttt{left} \rightarrow \texttt{right} \rightarrow *$ could stand for any of the access paths $l_0 \rightarrow \texttt{left} \rightarrow \texttt{right} \rightarrow \texttt{left}$, $l_0 \rightarrow \texttt{left} \rightarrow \texttt{right} \rightarrow \texttt{right}$, $l_0 \rightarrow \texttt{left} \rightarrow \texttt{right} \rightarrow \texttt{left} \rightarrow \texttt{left}$, $l_0 \rightarrow \texttt{left} \rightarrow \texttt{right} \rightarrow \texttt{left} \rightarrow \texttt{right}$ and more such paths. □

4.2 Dependence Detection Framework

Our method investigates if there is any heap dependency between any two statements in the program and the type of the dependencies following the algorithm `analyze` that we have outlined in Figure **??**. The algorithm works on each function separately resulting in intra-procedural analysis. It takes as parameter the function to be analysed and the maximum length of access path, which has been set at prior. Summarizing, the

Statement	Annotation directive
p = q	tagDir({q}, p, AliasStmt, null)
p = q→next	tagDir({q}, p, LinkTraverseStmt, next)
··· = p→data	tagDir({p}, null, ReadStmt, null)
p→data = ···	tagDir({p}, null, WriteStmt, null)

Table 4.1: Example showing different tagging directives

algorithm can be divided into the following steps:

All global variables and parameters of the function under analysis are initialized with proper values such that the correctness of the function is not violated. As our technique looks at only heap related statements, initialization of only global variables and parameters accessing heap is sufficient. The function `initialize` returns `InitSet`, a set of `<heap directed pointer variable, symbolic location>` pairs after initialization. The symbolic locations are identified in terms of access paths as described earlier. For reasons of efficiency, length of access paths are limited to 1.

Each statement in the function is annotated with a tagging directive `tagDir`. It consists of four attributes which give information regarding the heap accessing statement S_i : (a)Used pointer set `UsePtrSet` is the set of heap directed pointer variables which are used in the statement S_i; (b)Defined pointer set `DefPtrSet` is the set of pointer variables defined by the statement S_i; (c)Access type `AccType` identifies the pattern of heap access by statement S_i which can be categorized into following six classes (refer to Section ??.1).

- `AliasStmt` : aliasing statement.
- `LinkTraverseStmt` : link traversing statement.
- `ReadStmt` : statement reading heap.
- `WriteStmt` : statement writing into heap.
- `FunCallStmt` : function call statement.
- `OtherStmt` : any other statement.

(d) The access field `AccField` is the pointer field accessed by the statement S_i. Table ?? shows example statements and corresponding tagging directives.

4.2.1 State Analysis

This subsection introduces state analysis for heap directed variable. State analysis for heap variable involves computing a safe approximation of the binding of pointer

variable to a set of symbolic memory locations that can be potentially accessed by the variable at a particular program point. This binding is referred as state of the variable and is represented as $<$*variable*, $\{$*set of symbolic locations*$\}>$. This analysis is essentially used by dependence analysis in future.

Definition 4.2. *The state of variable* `x` *at a program point* `u` *is the set of symbolic memory locations such that some paths from the* `Entry` *point to* `u` *result in the access of symbolic locations by the variable* `x`.

The analysis works on multiple symbolic execution of the function. It follows general data flow analysis algorithm described in literature [?, ?, ?, ?] etc. It involves the following steps: (a) computation of local information, set of states after symbolic execution of each statement. At each program point it produces a set of states such that it contains all local and global variables and conservative approximation of symbolic locations accessed by each variable in the set. (b) computation of global information, set of possible states just before and after the execution of a block of statements. The control of the analysis flows in forward direction. Equations for state analysis follow the traditional data flow form:

$$\mathtt{In}[\mathtt{B}] = \bigcup_{\mathtt{P} \in \mathtt{pred}[\mathtt{B}]} \mathtt{Out}[\mathtt{P}] \tag{4.1}$$

$$\mathtt{Out}[\mathtt{B}] = \mathtt{f_B}(\mathtt{In}[\mathtt{B}]),\ \mathtt{f_B} \text{ is } \textbf{block level} \text{ transfer function of } \mathtt{B} \tag{4.2}$$

$$\mathtt{f_B}(\mathtt{In}[\mathtt{B}]) = \mathtt{g_{S_k}}(\cdots(\mathtt{g_{S_1}}(\mathtt{In}[\mathtt{B}]))),\ \mathtt{g_{S_1}} \cdots \mathtt{g_{S_k}} \text{ are } \textbf{statement level} \text{ transfer functions} \tag{4.3}$$

Transfer function $\mathtt{f_B}$ is a composition of series of transfer function $\mathtt{g_S}$ applied to each statement present in the block. The first statement `S` of block `B` has `In` set same as the `In` set to the block. Each statement locally generates `Kill` and `Gen` sets which are used by function $\mathtt{g_S}$ to produce `Out` set for each statement. Table ?? shows the local effects of the statements handled by our analysis. Hence the statement level equations for state analysis are:

$$\mathtt{Out}[\mathtt{S_i}] = \mathtt{g_{S_i}}(\mathtt{In}[\mathtt{S_i}]),\ \mathtt{In} \text{ and } \mathtt{Out} \text{ sets for statement } \mathtt{S_i} \tag{4.4}$$

$$\mathtt{g_{S_i}}(\mathtt{In}[\mathtt{S_i}]) = (\mathtt{In}[\mathtt{S_i}] - \mathtt{Kill}[\mathtt{S_i}]) \cup \mathtt{Gen}[\mathtt{S_i}],\ \mathtt{Gen} \text{ and } \mathtt{Kill} \text{ are local to } \mathtt{S_i} \tag{4.5}$$

Figure ?? demonstrates both block level and statement level transfer functions. As statement `S` is the first statement in the basic block `B` `In[S]` has been set to `In[B]`. Again `Out[B]` will be `Out[T]` as `T` is the last statement in the block `B`.

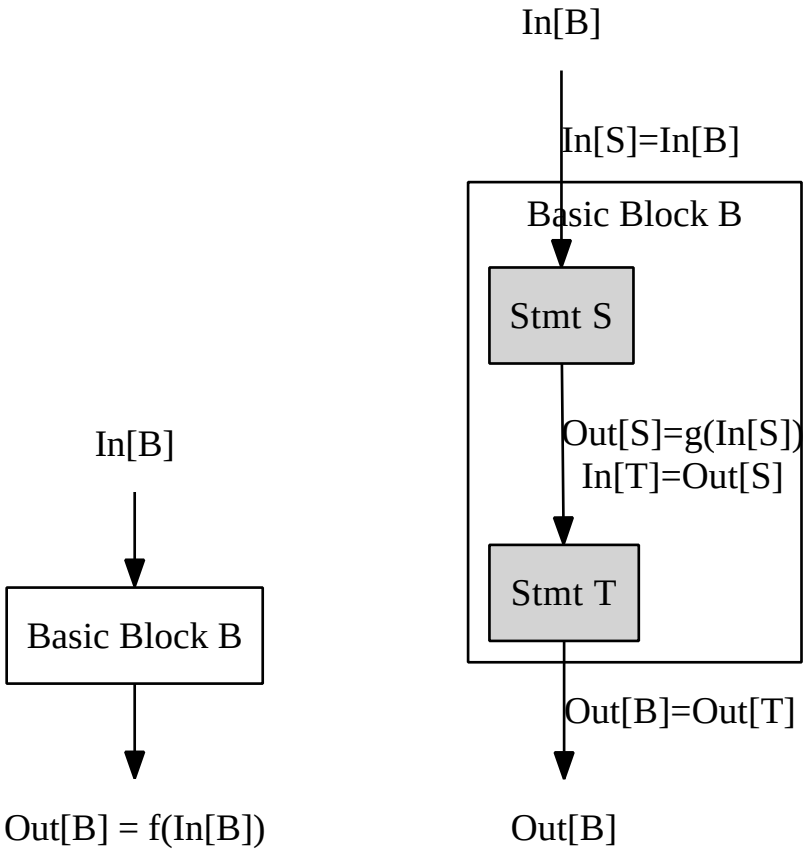


Figure 4.2: Transfer function for basic block

Statement	Gen set	Kill set
1. `p = q`	{<p,m> | <q,m>∈In[S]}	{<p,l> | <p,l>∈In[S]}
2. `p = q→next`	{<p,m→next> | <q,m>∈In[S]}	{<p,l> | <p,l>∈In[S]}
3. `··· = p→data`	ϕ	ϕ
4. `p→data = ···`	ϕ	ϕ
5. `fun(p, q)`	ϕ	ϕ

Table 4.2: *Gen* and *Kill* set for each statement

```
fun stateAnalysis(CFG[f]:(N, E, Entry, Exit), k) {
  Out[Entry] = InitSet;       /* Boundary condition  */
  for each basic block B other than Entry
          /* Initialization for iterative algorithm  */
     Out[B] = φ;
  while (changes to any Out[ ] occur) {  /* Iterate  */
     for each basic block B other than Entry {
        In[B] = ⋃(Out[P]), for all predecessors P of B;
        Out[B] = stateTrans(B, In[B], k);
     }
  }
}
```

Figure 4.3: Algorithm defining state analysis.

The overall algorithm for state analysis is outlined in Figure ??. `Out[Entry]` is initialized to `InitSet` to set the boundary condition. The *while* loop in the algorithm iterates until it reaches the fixed-point. `stateTrans` gives the algorithm for transfer function which works on each block. Though `Gen` and `Kill` informations are local to each statement they are computed in each iteration of the analysis, conflicting general data flow analysis algorithm.

Example 4. We illustrate via an example the way our algorithm works. Let's consider the code fragment shown in Figure ??.2(b). Global variables and parameters of the code are initialized to `InitSet` consisting of {<list, l_0>}. Table ?? shows the set of states produced by each statement in the code and demonstrates how the analysis reaches fixed-point. Note that the length of access path is limited to 1. In this example `Out` set for each basic block in iteration number 2 is same as `Out` set produced by each block in iteration number 3. Hence in third iteration the algorithm reaches fixed-point.

□

4.2.2 Read/Write State Computation

For each statement we intend to compute two sets of heap access paths: (a)*Read set*: the set of paths which are accessed to read a heap location and (b)*Write set*: the set of paths which are accessed to write to a heap location. These sets are obtained from the set of states, generated by the state analysis, in a single pass over the function. The read and write sets are used later to identify dependences.

```
fun stateTrans(Basic Block:BB, In set of BB:In[BB], k) {
  TempState = In[BB];
  for each Stmt S_i ∈ HeapStmt[BB] { /* HeapStmt[BB] contains  */
                          /* heap intensive statements of BB  */
     In[S_i] = TempState;
     Kill[S_i] = findStateDefVar(TempState, S_i);
     Gen[S_i] = computeState(In[S_i], S_i);
     Out[S_i] = (In[S_i] - Kill[S_i]) ∪ Gen[S_i];
     tempState = Out[S_i];
  }
  return tempState;
}
```

Figure 4.4: Algorithm for block level transfer function.

```
fun findStateDefVar(Set of States:TempState, Statement:S_i) {
  Set of States :  CurrState, LocalState = φ;
  for each variable V_i ∈ DefPtrSet {
     Find the state CurrState of V_i from TempState;
     LocalState = LocalState ∪ CurrState;
  }
  return LocalState;
}
```

Figure 4.5: Computing *Kill* set.

```
fun computeState(In set of stmt:In[Stmt], Statement:Stmt) {
  UseVarSet = findStateUseVar(In[Stmt], Stmt);
  if Stmt ≡ p = q
     Gen[Stmt] = { <p,l0> | <q,l0> ∈ UseVarSet }
           ∨ { <p,l0 →sel> | <q,l0 →sel> ∈ UseVarSet }
           ∨ { <p,l0 →sel→*> | <q,l0 →sel→*> ∈ UseVarSet };
  else if Stmt ≡ p = q→next
     Gen[Stmt] = { <p,l0 →next> | <q,l0> ∈ UseVarSet }
           ∨ { <p,l0 →sel→*> | <q,l0 →sel> ∈ UseVarSet };
  else if Stmt ≡ ··· = p→data
     Gen[Stmt] = In[Stmt];
  else if Stmt ≡ p→data = ···
     Gen[Stmt] = In[Stmt];
  else if Stmt ≡ f(p,q)
     Gen[Stmt] = In[Stmt];
  else Gen[Stmt] = φ;
}
```

Figure 4.6: Computing *Gen* set.

```
fun findStateUseVar(Set of States:TempState, Statement:Si) {
  Set of States :  CurrState, LocalState = φ;
  for each variable Vi ∈ UsePtrSet {
     Find the state CurrState of Vi from TempState;
     LocalState = LocalState ∪ CurrState;
  }
  return LocalState;
}
```

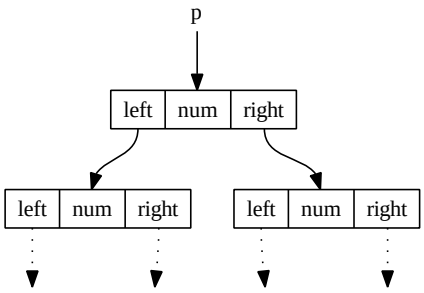
Figure 4.7: Computing states of used variables.

Statement	Iteration 1	Iteration 2
S1: p = list	In = {<list,l_0>} Gen = {<p,l_0>} Kill = {ϕ} Out = {<list,l_0>,<p,l_0>}	In = {<list,l_0>} Gen = {<p,l_0>} Kill = {ϕ} Out = {<list,l_0>,<p,l_0>}
while() {	In = {<list,l_0>,<p,l_0>} Out = {<list,l_0>,<p,l_0>}	In = {<list,l_0>,<p,l_0>, <p,l_0 →next→*>, <q,l_0 →next>, <r,l_0 →next→*>} = C(say) Out = C
S2: q = p→next	In = {<list,l_0>,<p,l_0>} Gen = {<q,l_0 →next>} Kill = ϕ Out = {<list,l_0>,<p,l_0>, <q,l_0 →next>} = A(say)	In = C Gen = {<q,l_0 →next>, <q,l_0 →next→*>} Kill = {<q,l_0 →next>} Out = {C,<q,l_0 →next→*} = D(say)
S3: temp = q→num	In = A Out = A	In = D Out = D
S4: r = q→next	In = A Gen = {<r,l_0 →next→*>} Kill = ϕ Out = {A,<r,l_0 →next→*>} = B(say)	In = D Gen = {<r,l_0 →next→*>} Kill = {<r,l_0 →next→*>} Out = D
S5: r→num = temp	In = B Out = B	In = D Out = D
S6: p = r	In = B Gen = {<p,l_0 →next→*>} Kill = {<p,l_0>} Out = {<list,l_0>, <p,l_0 →next→*>, <q,l_0 →next>, <r,l_0 →next→*>}	In = D Gen = {<p,l_0 →next→*>} Kill = {<p,l_0>, <p,l_0 →next→*>} Out = {<list,l_0>, <p,l_0 →next→*>, <q,l_0 →next>, <q,l_0 →next→*>, <r,l_0 →next→*>}

Table 4.3: Set of states for each statement of an example code

```
fun computeReadWrite(Set of States:HeapStates) {
  Set of States :  CurrState, LocalState = φ;
  for each stmt S_i {
     if (S_i ≡ ··· = p→num) {            /⋆ statement reading heap  ⋆/
        ReadSet = findStateUseVar (HeapStates[S_i], S_i);
        WriteSet = φ;
     }
     if else (S_i ≡ p→num = ···) { /⋆ statement writing into heap  ⋆/
        ReadSet = φ;
        WriteSet = findStateUseVar (HeapStates[S_i], S_i);
     }
     if else (S_i ≡ f(p,q)) {           /⋆ function call statement  ⋆/
        ReadSet = findStateUseVar (HeapStates[S_i], S_i);
        WriteSet = findStateUseVar (HeapStates[S_i], S_i);
     }
     else {
        ReadSet = φ;
        WriteSet = φ;
     }
  }
}
```

Figure 4.8: computing Read and Write sets.



(a) Tree data structure.

```
        ...
S11.    p = tree;
        while (p→left != NULL) {
S12.       newFunc (p);
S13.       p = p→left;
        }
        ...
```

(b) Code fragment traversing the data structure.

Figure 4.9: Program with function call.

Function `computeReadWrite` referred in Figure ?? computes such sets in a single symbolic execution of the function. Function `findStateUseVar` is used to generate `Read` and `Write` sets for statements reading/writing heap. Function calls are handled by conservative read/write sets that over approximate the heap locations that could potentially be read or written inside the called function. Read and write sets for other statements are set to ϕ.

Example 5. Consider the code fragment shown in Figure ??(b) which traverses the Tree data structure shown in Figure ??(a). Our analysis conservatively approximates read and write sets for statement `S12` which is a function call statement. The analysis generates the read and write sets as {<p, l_0>, <p, $l_0 \rightarrow *$>} which is the worst case approximations of such sets. □

Our approach is conservative in the sense that the read set and write set we compute for a statement are over approximations of the actual locations that are read or written by the statement. Therefore it is possible that our analysis reports two statement to be dependent when they are not really dependent on each other. However, this can inhibit some parallelizing optimization but can not result in an incorrect parallelization.

4.2.3 Dependence Detection

Our approach identify memory locations in terms of access paths as mentioned earlier. Multiple access paths can exist at a time leading to same location. Hence we need some method to detect whether two paths access same location or not. Our analysis

Statement	Read set	Write set
S1: `p=list`	ϕ	ϕ
S2: `q=p→next`	ϕ	ϕ
S3: `temp=q→num`	{<q,l_0 →next>, <q,l_0 →next→*>}	ϕ
S4: `r=q→next`	ϕ	ϕ
S5: `r→num=temp`	ϕ	{<r,l_0 →next→*>}
S6: `p=r`	ϕ	ϕ

Table 4.4: Read and write sets accessed by each statement

needs to know if any two access paths potentially share a common heap object. Shape analysis as referred in [?] produces an interface `isInterfering(p, α, q, β)` to detect such interference. For heap pointers `p`, `q` and field sequences $\alpha, \beta, \alpha', \beta'$, this function returns true if the paths $\texttt{p}.\alpha'$ and $\texttt{q}.\beta'$ interfere (potentially reach the same heap node at run-time), and α and β are prefixes of α' and β' respectively.

Read and write sets thus generated are tested to detect various dependences (flow, anti or output). Let `S` and `T` be statements in the program such that there exists an execution path from `S` to `T`. Then the dependence of `T` on `S` can be defined as follows:

$$\begin{aligned}
\mathsf{interfere}(\text{set}_1,\ \text{set}_2) &\equiv \text{isInterfering}(p,\alpha,q,\beta) \text{ where } p.\alpha \in \text{set}_1 \wedge q.\beta \in \text{set}_2 \\
\text{flow-dep}(S,T) &\equiv \mathsf{interfere}(\text{write}(S),\ \text{read}(T)) \\
\text{anti-dep}(S,T) &\equiv \mathsf{interfere}(\text{read}(S),\ \text{write}(T)) \\
\text{output-dep}(S,T) &\equiv \mathsf{interfere}(\text{write}(S),\ \text{write}(T))
\end{aligned}$$

where isInterfering is the function provided by shape analysis.

Example 6. Table **??** shows the read and write sets for each statement in the example code of Figure **??**.2(b). From the table, we can infer all the dependences of which few are listed here:

1. loop independent anti-dependence from statement S3 to statement S5
2. loop carried flow-dependence from statement S5 to statement S3
3. loop independent output-dependence from statement S5 to statement S6
4. loop carried anti-dependence from statement S6 to statement S5

Due to presence of loop carried dependences different iterations of the loop can not be executed in parallel. Next we explain how we can further refine our dependence analysis to filter out some spurious dependences. □

Chapter 5

Loop Sensitive Dependence Analysis

This chapter focuses on detecting the presence of dependences on loops which traverse recursive heap data structure. Two statements `S` and `T` may induce (a) `Loop Independent Dependence` (LID), where statements `S` and `T` access same memory location in a single iteration of the loop, (b) `Loop Carried Dependence` (LCD), if the memory location accessed by statement `S` in a given iteration, is accessed by statement `T` in other iteration. In either case at least one of the accesses must be write access.

Our approach for dependence analysis, as explained earlier, does not work well for loops. This is because it combines the paths accessed in different iterations of a loop. To get better result in presence of loops we need to keep the accesses made by different iterations of a loop separate. To do so, we have devised another novel approach, which works as follows: Given a loop, we first identify the navigators for the loop, then by a single symbolic traversal over the loop, we compute the read and write accesses made by each statement in terms of the values of the navigators. The read and write sets thus obtained are generalized to represent arbitrary iteration of the loop, using the iteration number as a parameter. These generalized sets, in terms of equations, are tested by *GCD* or *Lamport* test to find out any integer solution of those equations. Presence of loop dependences indicates that the iterations are not independent, hence can not be executed in parallel. The top level algorithm of loop analysis is outlined in Figure ??.

We assume that the loop under analysis is heap intensive i,e., reads/writes heap and the execution of the loop does not stop prematurely using irregular control flow constructs such as *return*, *continue*, *break* statements or function calls like *exit*, *abort*. Hence testing loop condition is the only way to exit control from the loop.

The rest of the chapter is organised as follows: Section ?? gives a brief description of finding navigator of the loop. Section ?? explains about how to compute read and

```
fun loopAnalyse(Loop) {
  <NavigatorVar, NavigatorExpr> = identifyNavigator(Loop);
  ReadWriteSet = generateReadWrite(NavigatorVar);
  identifyDep(ReadWriteSet);
}
```

Figure 5.1: Dep. detection for loop.

wriets sets of access paths and how our approach identifies both loop independent and loop carried dependences.

5.1 Identifying Navigator

Dependence analysis for loops relies on the computation of read and write sets, in terms of access path, for each statement in a single symbolic iteration of the loop. Access paths are computed with respect to *navigator* as mentioned in [?]. A navigator consists of (a)*navigator variable* `NavigatorVar`, pointer variable used to traverse the loop and (b)*navigator expression* `NavigatorExpr`, ordered set of pointer field references. Navigator variable in association with the navigator expression, iterates the loop traversing the data structure.

Navigator variable is closely related to the variable `TestVar` used to test the stopping criteria for the loop in the program. The algorithm generates the definition chain `DefChain` of `TestVar` using statements inside the loop. If the definition chain of `TestVar` encounters a loop resident statement twice, recurrence is reported. Otherwise the creation of definition chain returns null if it fails to find a loop resident statement for `DefChain`. Hence `DefChain`, thus generated for the later case returns an access path consisting of a pointer variable followed by an ordered sequence of pointer field references. The base pointer variable obtained from the access path is potential candidate to be navigator variable and the sequence of field references results in navigator expression. The details for identifying navigator can be found in [?].

Example 7. Consider the code shown in Figure ??.2(b). We identify `p` as loop condition test variable. Definition chain for `p` comes from the sequence of following loop-resident statements, `S7: p = r`, `S4: r = q→next`, `S2: q = p→next` and `S7: p = r`. Note that statement `S7` is encountered twice, leading to recurrence. Hence the statements `S7`, `S4`, `S2` are added to `DefChain` which returns access path as `p→next→next`. Hence the resulting navigator consists of navigator variable `p` and

```
fun generateReadWrite(NavigatorVar) {
  for each statement Si in Loop {
     UseVar = UseVarSet[Si];
     DefChain[Si] = findDefChain(UseVar, NavigatorVar);
     AccPath[Si] = findAccPath(DefChain[Si]);
     if (Tag[Si] == ReadStmt) {
        ReadSet[Si] = AccPath[Si];
        WriteSet[Si] = φ;
     }
     else if (Tag[Si] == WriteStmt) {
        WriteSet[Si] = AccPath[Si];
        ReadSet[Si] = φ;
     }
     else {
        ReadSet[Si] = φ;
        WriteSet[Si] = φ;
     }
  }
}
```

Figure 5.2: Generating Read and Write sets.

navigator expression `next`→`next` □

5.2 Computing Read/Write Sets

As mentioned earlier, our analysis computes read and write sets for each statement residing in the loop in a single symbolic execution of the loop. Read and write sets consist of paths that access heap locations for reading or writing. Unlike previous analysis, full length access paths are used by loop dependence analysis. For each loop-residing statement full length access paths, referred as `AccPath`, are computed in terms of navigator variable. Access paths `AccPath` are computed from definition chains, that are evaluated by recursively traversing all the reaching definitions of the pointer variable used by the statement until the navigator variable is encountered.

These access paths, thus constructed, return read/write sets based on statements reading or writing heap data. Function `generateReadWrite` showed in Figure ?? computes such sets of access paths with respect to navigator variable `NavigaotrVar`. Definition chain, `DefChain`, produced by function `findDefChain`, is processed by function `findAccPath` to compute access path. `ReadSet`/`WriteSet` sets, for each statement, are then computed from `AccPath`. The access paths, thus obtained, are generalized

Statement	Read Set	Write Set
S2	ϕ	ϕ
S3	p→next	ϕ
S4	ϕ	ϕ
S5	ϕ	p→next→next
S6	ϕ	ϕ

Table 5.1: Read and write sets for each loop residing statement

by arbitrary iteration of the loop, using iteration number as parameter, for further processing.

Example 8. Let us again consider the example given in Figure ??.2(b). Navigator variable and navigator expression for the loop are `p` and `next→next` respectively. Table ?? shows the read and write sets of full access paths constructed for each loop residing statement. Note that, the access paths are not abstracted and are constructed in terms of `p`. □

5.3 Loop Dependence Detection

Let `S` and `T` be two statements inside a loop. Further, let `write(S,i)` denote the set of access paths written by statement `S` in the iteration number `i`, and let `read(T,j)` denote the set of access paths read by statement `T` in the iteration number `j`. Predicate `sharing`($\texttt{Set}_1$, $\texttt{Set}_2$) returns true if two access paths $\texttt{AccPath}_1 \in \texttt{Set}_1$ and $\texttt{AccPath}_2 \in \texttt{Set}_2$ share a common heap node. Then

- `T` is loop independent flow dependent on `S` if there is an execution path from `S` to `T` that does not cross the loop boundary and there exist `i` within loop bounds such that `sharing(write(S,i), read(T,i))` is true.

- `T` is loop carried flow dependent on `S` if there exist `i` and `j` within loop bounds such that `j`>`i`, and `sharing(write(S,i), read(T,j))` is true.

Note that, in case loop bounds can not be computed at compile time, we can assume them to be (-∞,∞). We can similarly define loop independent and loop carried anti-dependence and output-dependence. Read and write sets of access paths, thus obtained for each statement inside a loop are tested for both loop independent and loop carried dependences.

5.3.1 Identifying Loop Independent Dependence

Loop independent dependences can be detected for any two statements by checking for any sharing of node by their respective read/write sets. Sharing of a node, in this level, occurs due to the shape of the underlying data structure. Shape analysis gives the probable shape attribute of the navigator variable traversing dynamic data structure. Based on the shape we can figure out whether there exists any dependence due to sharing within the underlying data structure.

Observation 1: If the shape attribute of navigator variable is Tree, then there exist no sharing of nodes by different access paths rooted at the navigator variable. Two access paths can only visit a common node if the paths are equivalent. Let `lp` be the navigator variable. Hence `lp→f` and `lp→f` are equivalent paths leading to a common node, whereas `lp→f` and `lp→g` lead to different nodes.

Observation 2: If the shape attribute is DAG, the navigator expression will lead navigator variable to a distinct node in each iteration of the loop. If an access path is a proper subpath of another access path then they surely visit distinct nodes. However, paths being either equivalent or distinct, having different pointer field references may access a common node. For example, `lp→f` is a proper subpath of `lp→f→g` whereas, `lp→f` and `lp→g` are not. Hence in the former case they do not share a common node, whereas in later case they might result in sharing of node.

Observation 3: If shape attribute of navigator variable is Cycle, we make conservative decision such that the loop can not be executed in parallel.

To detect various types of LIDs, read and write sets of different statements are tested accordingly for detecting sharing of node. Let two statements `S` and `T` access paths `PathS` and `PathT` respectively and `read(S) = {PathS}` and `write(T) = {PathT}`. Hence there is loop independent flow dependence from `S` to `T` if `sharing(PathS, PathT)` returns true. We check for the shape of the underlying data structure and test the paths as follows:

1. If the paths `PathS` and `PathT` are equivalent and the data structure is either Tree or DAG, the paths will access same node. Hence,

$$\texttt{sharing}(\texttt{PathS}, \texttt{PathT}) = True$$

   for both Tree and DAG structure.

2. If `PathS` is subpath of `PathT` then these paths do not lead to any common node for both Tree and DAG data structure. Hence,

$$\texttt{sharing}(\texttt{PathS}, \texttt{PathT}) = False$$

   for both Tree and DAG structure.

3. If `PathS` and `PathT` are not equivalent and one is not subpath of other, then these two paths share a common node only if shape of the underlying structure is DAG. For Tree structure they lead to disjoint nodes.

$$\texttt{sharing}(\texttt{PathS}, \texttt{PathT}) = False$$

   if shape attribute is TREE.

$$\texttt{sharing}(\texttt{PathS}, \texttt{PathT}) = True$$

   if shape attribute is DAG.

Example 9. Consider the loop shown in Figure ??.2(b) and the corresponding read and write sets for each statement in Table ??. Read set of statement `S3` and write set of statement `S5` are checked for sharing of any node. As the shape attribute of the navigator variable `p` is Tree, and the paths `p`→`next` and `p`→`next`→`next` are not equivalent the following will result.

$$\texttt{sharing}(\texttt{p} \rightarrow \texttt{next}, \texttt{p} \rightarrow \texttt{next} \rightarrow \texttt{next}) = False$$

Hence no loop independent dependence is detected. □

5.3.2 Identifying Loop Carried Dependence

Loop carried dependence is incurred in the loop when two statements from different iterations access same memory location. LCDs can be introduced when the statements in a single iteration of loop access both current node and neighbour heap nodes. Current node means the node which is being currently accessed by the navigator variable, whereas, neighbour nodes mean nodes other than the one being currently accessed.

Example 10. Let the shape attribute of the navigator variable `lp` be Tree and a loop is traversing a list using navigator variable `lp` and navigator expression `next`. Statement `lp`→`num++` will not incur any loop carried dependence as the location pointed to by

```
        ...
        p = list;
        while (p→next != NULL) {
S11.       ...= p→num;
S12.       p→next→num = ...;
S13.       p = p→next→next;
        }
        ...
```

(a) *List-1*: Loop without Dependence

```
        ...
        p = list;
        while (p→next != NULL) {
S21.       ...= p→num;
S22.       p→next→num = ...;
S23.       p = p→next;
        }
        ...
```

(b) *List-2*: Loop with Dependence

Figure 5.3: Identifying Loop Dependences

`lp` can't be accessed in consecutive iterations. However, statement like `lp→num = lp→next→num` will still introduce an LCD because both current and neighbour nodes are accessed in the same iteration and neighbour node is visited using pointer field `next` which is also a navigator expression. □

As mentioned earlier, LCDs are only introduced by different iterations of loop, provided there is no sharing of nodes hidden in the data structure. However, DAG has sharing within the structure, it can be traversed by a loop in a manner such that shared nodes are not accessed by loop and access pattern of the structure is Tree. Hence, loops having this type of access pattern can also be tested for loop carried dependencies.

The read-write sets computed by each loop resident statements are generalized for arbitrary iteration number, resulting in a set of equations. Generalization is done using random times of navigator expression, that is used by the navigator variable to traverse over the underlying data structure. These equations are then tested by `GCD` [?] or `Lamport` [?] test, as explained before. If the equations have any integer solution, dependence is reported. Here we demonstrate two examples to show the novelty lying in our approach.

Example 11. Let us consider Figure ??. The code snippet in Figure ??(a) is the reformulation of code given in Figure ??.2(b). The navigator variable for both the loops is `p`. For the code in Figure ??(a), the navigator expression is `next→next`. Using `i` to represent the iteration number, the generalized access path read by S11 is $\texttt{p}\rightarrow\texttt{next}^{2\texttt{i}}$ and the generalized access path written by S12 is $\texttt{p}\rightarrow\texttt{next}^{2\texttt{j}+1}$. Clearly there is no loop independent dependence as the shape of the data structure is Tree and statements do not generate equivalent access paths. To find out loop carried dependence, we have to find out whether for iterations `i` and `j` and $\texttt{i}\neq\texttt{j}$, the two paths point to the same heap location. This reduces to finding out if there is a possible solution to the following equation:

$$\texttt{p} \rightarrow \texttt{next}^{2\texttt{i}} = \texttt{p} \rightarrow \texttt{next}^{2\texttt{j}+1}$$

In other words, we have to find out if integer solutions to the following equation are possible :

$$2 * \mathtt{i} = 2 * \mathtt{j} + 1$$

GCD or Lamport test tell us[1] that this equation can not have integer solutions. Thus, there is no dependence among the statements.

For the code in Figure **??**(b), the navigator expression is `next`. In this case the equation to find out the loop carried dependences among statements S21 and S22 reduces to :

$$\mathtt{i} = \mathtt{j} + 1$$

which has integer solutions. So we have to conservatively report dependence between the two statements □

[1] In this example, visual inspection tells us that for any `i` and `j` LHS is even number while RHS is an odd number. Hence the equation can not have a solution. In general, the equations are more complicated and we need to use standard tests as mentioned.

Chapter 6

Inter-Procedural Dependence Analysis

In the previous chapter we have given detailed explanation of heap dependence analysis specified within each procedure. This intra-procedural analysis does not cross procedure boundary. When the symbolic execution within a procedure reaches a procedure call, the analysis approximates the worst case summary for the called procedure which conservatively approximates the read and write sets of states for the procedure being called. In this chapter we explain how the intra-procedural dependence analysis can be extended into a flow sensitive inter-procedural analysis.

Our inter-procedural approach is based on computing abstract summary for each procedure at prior. Before analysing the whole program, the call graph of the program is preprocessed such that it does not contain any recursion. The pseudo code outlined in Figure **??** gives a top level algorithm of inter-procedural analysis. We discuss each step of such analysis in the following sections. Section **??** detects any recursion present in the program and processes the call graph accordingly. Section **??** gives the details of the analysis which includes preparing abstract summary for each procedure and Section **??** presents the technique to effectively use abstract summary of each procedure for inter-procedural analysis.

6.1 Processing and Ordering of Call Graph

Our inter-procedural analysis works on call graph of the program under analysis. Call graph $G(V, E)$ is a directed graph that represents calling relationships between caller and callee procedures. Specifically, each node $v_i \in V$ represents a procedure and each directed edge $(f, g) \in E$ indicates that procedure f calls procedure g. Thus, a cycle in the graph indicates recursive procedure calls.

Our approach needs to transform the cyclic call graph into directed acyclic graph

```
fun interProcAnal(P) {
  G(V, E) = callGraph(P);       /* G is call graph of program P  */
  if (cyclic(G) = TRUE)                    /* Check for recursion  */
    G'(V, E') = findDAG(G); /* Transform cyclic graph into DAG  */
  else
    G' = G;
  G'' = topologicalSort(G');    /* Find topological order of DAG  */
  for each v_i ∈V following reverse topological order of G'' {
    Summary[v_i] = findSummary(v_i);
                   /* Find abstract summary for each procedure  */
  }
  progAnal(P, Summary);
                /* Use summaries for inter procedural analysis  */
}
```

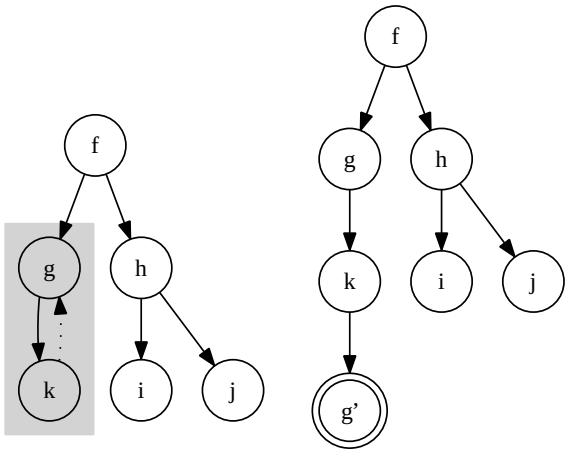
Figure 6.1: Top level algorithm for inter-procedural analysis

```
S1.   procedure f()        S10.   procedure k()
S2.   begin                S11.   begin
S3.        call g();       S12.        call g();
S4.        call h();       S13.   end
S5.   end                  S14.   procedure h()
S6.   procedure g()        S15.   begin
S7.   begin                S16.        call i();
S8.        call k();       S17.        call j();
S9.   end                  S18.   end
```

Figure 6.2: Skeleton of a program with procedure calls

(DAG) for efficient computation of abstract summary. The call graph without any recursion is always DAG. To transform cyclic directed call graph into DAG, the back edges from the cyclic graph are removed and a summary node is introduced. This summary node approximates all the remaining levels of recursion and abstracts the worst case summary of the functions present in the recursion. The nodes of directed acyclic call graph, thus obtained, are ordered using topological sort. Each node in the acyclic call graph will be assigned a sequence order following which the procedures are processed.

Example 12. Consider the program shown in Figure **??**. In this example procedure `f` calls procedure `g` and procedure `h`, whereas, procedure `h` again calls procedures `i` and `j`. Procedure `g` calls procedure `k` which in turn calls procedure `g`, resulting in indirect recursive procedure calls. Figure **??**(a) shows the corresponding directed call graph.



(a) Cyclic call graph (b) Corresponding DAG

Figure 6.3: Example showing call graph

The shadowed cyclic region in the graph indicates recursion in the program. The cyclic call graph is then transformed into acyclic graph by removing the back edge showed as dotted edge. Figure **??**(b) presents corresponding acyclic graph with topological order.

□

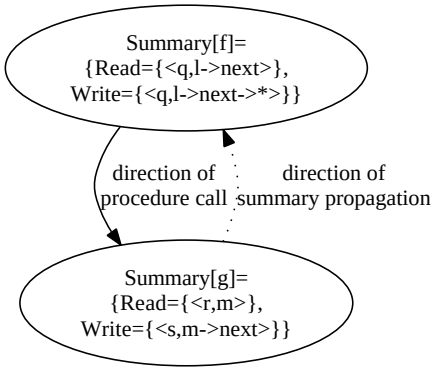
6.2 Computing Abstract Summary

For each callee procedure, we need to obtain an abstract summary that summarizes the procedure. By abstract summary, we mean the *Read* and *Write* sets of symbolic heap locations which are accessed for reading and writing operations respectively inside the procedure. The motivation behind the technique is to summarize effect of called procedure for callers, which in turn is used by the callers to summarize effect for called procedure. Summaries, thus obtained for each procedure are stored in a table for later use. In this analysis propagation of summaries follows the bottom-up approach. Hence, abstract summary for all the procedures are computed following the reverse topological order of nodes in the call graph.

All heap directed global pointer variables throughout the program are initialized once with proper symbolic memory locations. During computation of abstract summary for each procedure, the heap directed pointer variables passed as formal parameters to the called procedure are initialized with random symbolic memory location. As summary, our analysis computes read and write sets of access paths with respect to the corresponding symbolic memory locations. This process of computation follows the intra-procedural analysis as explained before. The same procedure is followed by

```
S1.    procedure f(p)
S2.    begin
S3.       q = p→next;
S4.       g(q);
S5.    end
S6.    procedure g(r)
S7.    begin
S8.       ··· = r→num;
S9.       s = r→next;
S10.      s→num=···;
S11.   end
```

(a) Example program with procedure call



(b) Call graph showing summary of procedures

Figure 6.4: Example showing computation of abstract summary

the caller procedure to summarize its effect.

6.3 Evaluating Procedure Call

Abstract summary for each callee procedure should be used by the caller procedure in the current calling context. Summary of each called procedure, thus obtained by intra-procedural analysis, contains information in the local context of the procedure. When the symbolic execution inside the caller function reaches a procedure call, the context of the actual parameters is mapped to the respective formal parameters of the called procedure. The summary of the callee is translated accordingly to be used in the context of caller procedure.

Summary of a procedure, as mentioned earlier, returns read and write sets of paths accessed by corresponding pointer variables. Access paths are computed with respect to symbolic memory locations local to the callee under analysis. The local symbolic memory locations used in the summary of callee are mapped into symbolic locations accessed by the caller procedure. Such modified summary of caller is used by the callee to summarize its effect. Evaluating summary for recursive function does not differ significantly than evaluating non-recursive function. If a recursive function is encountered, the analysis will go deep inside the function upto the depth of recursion depending upon the precision of analysis.

Example 13. Let us consider the example program and the corresponding call graph shown in Figure **??**. In the example program procedure `f` calls procedure `g`. The solid line in the call graph gives the direction for calling subroutines, whereas, the dotted line shows the direction for propagation of summary information. Summary information of

procedure `f` and `g` are also shown in the call graph. At first, summary of procedure `g` is computed in the local context of itself. The summary consists of read and write sets as `{<r,m>}` and `{<s,m→next>}` respectively, where pointer variable `r` is initialized to symbolic memory location `m`. At the call site of procedure `g`, the actual parameter `q` takes value `l→next`, if formal parameter `p` of procedure `f` points to memory location `l`. The value of actual parameter `q` modifies the summary of procedure `g`. This modified summary is further used to summarize the information of procedure `f`.

□

Chapter 7

Experimental Results

In this chapter we discuss about some experimental results of our analysis. We have developed a prototype model of heap dependence analysis at two levels: One at the intra-procedural level, which works for each procedure separately, and the other at the loop level, where the intra-procedural analysis is refined in the context of loop. The model is implemented in the Static Single Assignment (SSA) intermediate level of GCC (version 4.3.0) [1] framework. The guideline for building and installing GCC from source, with newly added pass is given in Appendix A. We have conducted experiments over two example programs and two benchmark programs. The example programs are used to show how our loop-based approach successfully detects loop-carried dependences. These simple programs are based on single linked-list data structure presented in Figure **??**.3. Other benchmark programs *TreeAdd* and *Bisort* are drawn from Olden Suite. The prototype model, with manual intervention, successfully identifies the dependences present in benchmark programs

We have manually pre-processed the programs to be analysed, such that the heap accessing statements are normalized into binary statements. In the next step, based on type of heap access, the normalized statements are tagged accordingly. In the same step informations regarding pointers, used and defined by the statement and the pointer field used to access the heap object are attached to each statement along with the access type. In the following sections we explain the experimental results for intra-procedural analysis and loop sensitive analysis.

[1] Available from http://gcc.gnu.org/

```
int treeAdd(tree *t) {
   if (t == NULL)
      return 0;
   else {
      tleft = t→left;
      leftval = treeAdd(tleft);
      tright = t→right;
      rightval = treeAdd(tright);
      value = t→val;
      return leftval+rightval+value;
   }
}
```

(a) Function *treeAdd*

```
 int bisort(root,sprval,dir)
    HANDLE *root;
    int sprval, dir;
{
    HANDLE *l;
    HANDLE *r;
    l = root→left;
    r = root→right;
    val = root→value;
    root→value=bisort(l,val,dir);
    ndir = !dir;
    sprval=bisort(r,sprval,ndir);
    sprval=bimerge(root,sprval,dir);
    return sprval;
 }
```

(b) Function *bisort*

Figure 7.1: Functions of benchmark programs

7.1 Benchmarks

Benchmark program *TreeAdd* operates on binary tree and recursively adds the values of tree. The *treeAdd* function from benchmark program *TreeAdd* is analysed. This function recursively calls itself to compute values of its left and right subtrees. Left and right subtree values, thus computed are added to the value of the root to compute the value of whole tree. Figure **??**(a) shows the function *treeAdd*. The main objective behind the analysis is to find out whether the recursive function calls on left and write subtrees can be executed in parallel. The function *bisort* from the benchmark program *Bisort* is also analysed to extract function call level parallelism. Function *bisort* performs bitonic sort over binary tree by recursively calling itself on left and right subtrees of the root. The analysis is fine tuned in the context of loops. The experiments for loop sensitive dependence analysis are done on example programs *List-1* and *List-2* shown in Figure 5.3. Both the programs traverse single-linked list and in each iteration the functions read data from the current heap node and write the same data into the next node. But the function *List-1* navigates iterations using two occurrences of pointer field *next*, whereas, function *List-2* navigates using a single occurrence of *next* pointer field. Hence by manual inspection it is clear that function *List-1* does not have any loop dependence, but function *List-2* poses loop dependences.

Program	Traversal pattern	Potential Dependence	Type of dependence
TreeAdd	2 nested rec. function	No	No dep.
Bisort	2 nested rec. function	No	No dep.
List-1	single loop	Yes	Anti dep.
List-2	single loop	Yes	Anti dep.

Table 7.1: Result of programs tested by intra-procedural analysis

7.2 Results of Intra-Procedural Analysis

The prototype model for intra-procedural dependence detection performs the state analysis which computes the set of pairs, consisting of pointer variable and path accessing corresponding symbolic heap location. In the next pass over the program, the model successfully computes the read and write sets of access paths for each heap accessing statement in the program. Next we manually check for each pair of heap accessing statements to find out any conflict in terms of data access. The interference predicate `isInterfering`, produced by shape analysis returns true if the access paths under objection lead to common heap location. In the Table **??** we have given results of intra-procedural analysis. It reports no dependence at function call level for *treeAdd* and *bisort*. However, the analysis reports dependencies of type anti dependence for both example programs *List-1* and *List-2*.

7.3 Results of Loop-Sensitive Analysis

The basic prototype model identifies navigator variable and navigator expression, used by the loop to traverse over the data structure. It then computes the full length access paths being read or written by each statement within the body of the loop. The programs *List-1* and *List-2* are further analysed to refine dependence analysis. The previous analysis detects spurious dependencies in case of loop. Both example programs work on single linked list, which is of type Tree. As there exists no sharing within the underlying data structure, the generalized equations, formed by the access paths, are tested for loop-carried dependencies. Lamport test successfully reports no dependence for function *List-1* and dependence of type anti dependence for function *List-2*.

Chapter 8

Conclusion and Future Work

In this report we have presented our work on heap induced dependence analysis that can be utilized by a parallelizing compiler to extract both fine-grained and coarse-grained parallelism from sequential programs. Our method gives an easy to implement technique for the same. It is divided into two phases: the intra-procedural dependence analysis phase and loop based analysis phase, with carefully chosen interfaces between phases to combine work done by individual phases. The first phase is helpful to find dependences in both statement level and function call level, whereas, the second phase refines the analysis in the context of loop. This modularity gives us flexibility to work on testing and improving each phase independently.

Our intra-procedural analysis abstracts each actual heap location by symbolic location, which is defined by set of access paths leading to same heap node. It successfully computes the read and write sets of heap access paths at each program point and identifies dependences based on the aliasing information produced by the specific shape analysis framework [**?**]. Our loop dependence analysis abstracts the dependence information in forms of linear equations, that can be solved using traditional dependence analysis tests like GCD, Lamport tests that already exist for finding array dependences. Our intra-procedural analysis use conservative approximation of function calls assuming worst case scenario. We give a direction to extend the intra-procedural analysis to inter-procedural one, which is able to precise function calls more precisely.

Our analysis is too conservative for complex cyclic structures and can not extract any parallelism between any two statements or function calls or different iterations of loop body . We have to further develop our shape analysis technique to handle more frequently occurring complex and cyclic structures and programming patterns to find precise dependences. In this work the analysis only keeps information about the first link field of the access paths and blindly summarizes the rest of the path. Hence

it losses good amount of information for interference analysis. We want to improve the summarization technique for better abstraction of access paths. In case of loop sensitive dependence analysis, our technique assumes the loops without any irregular control flow constructs. We can further extend our technique to automatically detect good loops [?] which do not contain any irregular control flow constructs.

We have implemented a basic prototype model for intra-procedural and loop based dependence analysis. This prototype model with manual intervention detects dependences from heap intensive sequential programs. We have tested the model with a very few number of heap intensive C benchmark programs. It can be further developed to implement inter-procedural dependence analysis and to show its effectiveness on large benchmarks.

Appendix A

Guideline for Adding New Pass in GCC

Pre-requisities for Configuring GCC-4.3.0

1. GMP-4.3.2 and MPFR-2.4.1 should be installed.
2. Build GMP and MPFR in $GMPBUILD and $MPFRBUILD directory respectively.

How to Configure and Build GCC

1. Let $SOURCEDIR be the source directory for GCC and $HOME be the home directory where $SOURCEDIR is present.
2. Create another directory $BUILDDIR in $HOME and follow the following steps.
3. cd $BUILDDIR
4. ../$SOURCEDIR/configure –enable-languages=c –with-gmp=/home/user/$GMPBUILD –with-mpfr=/home/user/$MPFRBUILD
5. make
6. make install
7. make all-gcc TARGET-gcc=/home/$BUILDDIR/gcc/cc1

How to Register Pass in GIMPLE SSA level in GCC

1. Place the new file named *tree-loop-distribution.c* in $SOURCEDIR/gcc/.

2. Adding the pass in pass hierarchy : Add NEXT_PASS (pass_loop_distribution); after NEXT_PASS(pass_linear_transform); in file passes.c

3. Add extern struct tree_opt_pass pass_loop_distribution; in file named tree-pass.h

4. Add the following lines in file named common.opt
   ftree-loop-distribution
   Common Report Var(flag_tree_loop_distribution)
   Enable loop distribution on trees

5. Add DEFTIMEVAR (TV_TREE_LOOP_DISTRIBUTION, "tree loop distributio") after DEFTIMEVAR (TV_TREE_LINEAR_TRANSFORM, "tree loop linear") in file timevar.def

6. Edit Makefile.in : Add following rules tree-loop-distribution.o
   tree-loop-distribution.o : tree-loop-distribution.c $(CONFIG_H)
   $(SYSTEM_H) coretypes.h $(TM_H) $(GCC_H) $(OPTABS_H)
   $(TREE_H) $(RTL_H) $(BASIC_BLOCK_H) $(DIAGONSTIC_H)
   $(TREE_FLOW_H) $(TREE_DUMP_H) $(TIMEVAR_H) $(CFGLOOP_H)

   tree-pass.h $(TREE_DATA_REF_H) $(SCEV_H) $(EXPR_H) $(TARGET_H)

   tree-chrec.h

7. make

8. make install

Test

1. gcc -O -ftree-loop-distribution -fdump-tree-ldist test.c

2. Results dump file named test.c.103t.ldist