Identifying Interesting Code Patterns in a Given Code Fragment for Compiler Optimization

By

Chitra Ramachandran

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

Master of Technology

in

Information Technology

International Institute of Information Technology, Bangalore

Thesis Committee

Prof. Srinath Srinivasa (Chair)

Prof. K. V. Dinesha

Prof. Arvind Keerthi

June 2005

Thesis Certificate

This is to certify that the thesis titled **Identifying Interesting Code Patterns in a Given Code Fragment for Compiler Optimization**, submitted by **Chitra Ramachandran (Roll No. 2003-026)**, to the **International Institute of Information Technology, Bangalore**, for the award of the degree of **Master of Technology** in **Information Technology**, is a *bona fide* record of the research work done by her under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Bangalore

June 30, 2005 **[Prof. Srinath Srinivasa]**

Other Members of the Thesis Committee

1.

[Prof. K. V. Dinesha] June 30, 2005

2.

[Prof. Arvind Keerthi] June 30, 2005

Identifying Interesting Code Patterns in a Given Code Fragment for Compiler Optimization

by

Chitra Ramachandran

submitted to the

International Institute of Information Technology, Bangalore

on

June 30, 2005

in partial fulfillment of the requirements for the award of the degree of

Master of Technology in **Information Technology**

Abstract

The current compiler technology allows code optimization to great extents, using different mechanisms such as code parallelism, dead code elimination, constant expression evaluation etc. Though these compilers optimize any given implementation they tend to ignore the context of the implementation and the algorithm implemented in the code. An ideal system would be a profiling/compiling tool in which the profiler could make suggestions, and the compiler dynamically does the compiling based on profiler suggestions and user input. A sub-module of the *suggestion making engine* would be to identify code fragments in the input that may be of interest to the user. Such a module would require, some kind of heuristic or explicit knowledge of what the user thinks is interesting. In this thesis we address the problem of identifying code patterns that may be of interest to the user. The approach suggested is based on control flow graphs.

This work is dedicated to ***OSL***.

Acknowledgements

I am highly indebted to **Prof. Srinath Srinivasa** for his invaluable guidance, motivation and support. He has positively influenced me not only academically, but also as an individual. I would like to show my appreciation towards **Prof. S Sadagopan** for having given me the opportunity to pursue research. I would like to express my gratitude to **Prof. K.V. Dinesha** for his questions and suggestions, during the talks I gave, as they have help shape and understand the problem better.

I would like to thank **Ms. Geetha Manjunath** and **Dr. Dibyendu Das** of HP India for their invaluable suggestions during the course of the project. I would also like to thank **HP India** for having sponsored the project.

I would like to show my gratitude to my fellow researchers **Pradeep S**, **Sanket Patil**, **Mandar Mutalikdesai**, **Saikat Mukherjee**, **Sripathi Kamath** and **Ambar Hegde** for their positive criticism and encouragement. Last but not the least I would like to thank my friends at IIITB, for their moral support.

- *Chitra*

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Compilers

Computers understand the language of only bits and bytes. There has to be a mechanism to convert the high level language code to computer understandable language of 0s and 1s. Compilers[1] perform this code translation. A compiler converts a high level language program to system executable machine language code. For example a *C compiler* will take as input a C program and give as output an executable file.

Currently compilers are more robust than they previously were and perform lot more than just code conversion. Compilers throw warning messages for seemingly unintentional mistakes, optimize code, compile code for use with a debugger, dump intermediate object codes etc.

Also there are tools such as *lint, FunctionCheck, gprof* etc., which allow analysis of code. For example *lint* can be used to check if the function return type and the actual return type expected by the calling program are the same. These tools also help analyze the data flow in a code. These tools are called as *profilers.*

[1] [1] was used as reference for Compiler Theory

1.1.1 Profilers

Profilers help analyze the run time features of the code. The functionalities provided by the profiler varies from profiler to profiler. The general features provided by profilers are :

1. Type checking for values returned by called functions and values expected by calling function.
2. Unused variables in the code.
3. Runtime performance of codes, in terms of execution time and memory.

A profiler if integrated with an compiler can make *learned suggestions* to the compiler, based on which the compilation can become more efficient (or become tailor-made for an application, as the need of the application be).

1.2 Optimization in Compilers

Optimization refers to improving the execution speed or the memory requirements of the given code implementation as required. The *optimizer* usually comes built-in with the compiler these days. The current optimization techniques optimize the specific instance of code, making it independent of the programming skills of the programmer to a large extent.

These optimization schemes lack in one respect, i.e. to recognize the context of algorithm and suggest or make changes. The compiler recognizes the inefficiencies of the implementation but fails when it comes to recognizing the whole algorithm (or part of algorithm) as inefficient, to do so the compilers will need to have some heuristics, the knowledge could be in the form of :

1. Set of Rules : The compiler could be given rules on the basis of which it can identify/classify algorithms, for example an algorithm that uses a lot of recursion in its function should be avoided etc.

2. Set of Code Patterns : The compiler could be given a set of codes/code fragments and asked to classify all similar fragments as inefficient. Here again the criterion for similarity has to be defined.

In this thesis we use the second approach (that of educating the compiler using a set of code patterns). Recognizing inefficient patterns (or any pattern of interest) can lead to better understanding of code and hence optimization.

1.3 The Problem

Given a code, a user might like to identify the interesting code patterns present in the code. The interestingness criteria could be any user defined code attribute/feature like *inefficient implementation, error prone implementation, code taking variable amount of time to execute* etc. The compiler/profiling tool is educated about what the user thinks is interesting, by giving it a collection of code patterns[2]. This is done by creating a database of interesting code pattern (i.e. to say that all patterns in the database are such that the user finds them interesting).

Given an input file we would like identify the presence of any of the given interesting code patterns (i.e. the patterns in the database)in it, in a single parse of the input. A trivial approach to the problem could be to scan for each individual pattern in the database successively in the main input code. This solution may become infeasible as the size of database (number of code patterns in the database) increases, also an exact character by character match cannot be done as the variable names could differ, extra comments could be present, *while/for* looping constructs could have been used interchangeably etc. All these above problems entail the need for code fragments to be represented in a form that is largely independent of the variable names, individual coding practices etc. Also an efficient algorithm should be available for comparing the code fragments with the main input code.

[2]code fragments and code patterns are interchangeably used in this thesis report

1.4 The Approach

The intermediate representation of a compiler can be used to represent the code fragments and do comparisons on them as they have the following advantages:

1. All complex expressions and statements are broken down into simple single operation statements. For example $a = b + c * rand();$ will be broken down into three instructions as follows :

   (a) $Temp = rand();$

   (b) $Temp1 = c * Temp;$

   (c) $a = b + Temp1;$

   This would make expression comparison slightly more simple.

2. All looping constructs will be reduced to simple *if-else* constructs.

3. Comments, programmer dependent formatting will be eliminated.

But intermediate code still has the problem of variable names being a part of it. Hence we use a representation that is based on *Control Flow Graphs (CFGs)* and *node labeling.* By doing so the problem reduces to a class of graph problems, wherein we need to find if any of the patterns in the database is a subgraph of the main input graph[3].

1.5 Organization of the Thesis

Chapter 2 discuses the general concepts of a *compiler*, giving a bird's eye view of the current approaches in compiler optimization. Chapter 3 introduces the concept of *Control flow graphs*, the reason we chose control flow graphs and its representation for the purpose. Chapter 4 presents our *indexing structure* and modification of CFG

[3]refers to the input program in which we would like to identify the interesting patterns

for the purpose of indexing. Chapter 5 gives an overview of the design and implementation details. Experimental results obtained are shown in Chapter 6. Chapter 7 discusses related literature in both compilers and graph algorithms. Conclusions and future directions are presented in Chapter 8.

Chapter 2

Compilers

2.1 The Compilation Process

The compilation process can be divided into phases, with each phase dealing with one particular aspect of the compiler:

1. Source code Parser : The source code parser, verifies the syntax of the input source code. The parser generally outputs an intermediate representation of the input code. The parser is further subdivided into :

   (a) lexical analyzer

   (b) syntax analyzer

   (c) semantic analyzer

2. Intermediate Representation Generation : The parser outputs an intermediate code, this intermediate representation is internal to the compiler. Code optimization is performed on the intermediate code.

3. Code Generator : The intermediate representation is a platform independent but compiler specific representation. This representation needs to be converted to platform specific representation in terms of the instructions supported by the platform.

2.2 General optimization techniques

Optimization involves improving the runtime performance of the code in terms of memory used or number of clock cycles taken for execution. Optimization is one of the phases of the compilation process. Standard optimization techniques proceed on the lines of :

1. Dead Code Elimination : Code Fragments that will never be executed need not be included in the executable, and are called Dead Code. For example as shown in 2.1.

   ```
   if( a > b) {
   /*perform something*/
   } else if(a <= b)
   {
   /*perform something else*/
   }else
   {
   /*CAN BE ELIMINATED*/
   }
   ```

   Table 2.1: An Example for Dead Code Elimination

   In the above case the code under the else condition never gets executed and hence can be ignored.

2. Constant Expression Evaluation : Expressions that evaluate to constant value need not be computed at runtime they can be evaluated during the compilation and their values can be substituted. For Example :

   $a = 1 * 2 * 3 * 4$; can be written as $a = 24$;

3. Loop Unrolling : Loops are inefficient structures with respect to the execution time. In loop unrolling the number of iterations in a given looping structure is

original code	Optimized Code
$for(i = 0; i < 10; i++)$ $\{$ $\quad notAFunctionOf\_i();$ $\}$	$for(i = 0; i < 5; i+ = 2)$ $\{$ $\quad notAFunctionOf\_i();$ $\quad notAFunctionOf\_i();$ $\}$

Table 2.2: An Example for Loop Unrolling

reduced by either replicating the code inside the loop or some other technique. As shown in 2.2.

4. Common Subexpression Evaluation : The compiler generally splits all complex equations into micro-instructions that just take two arguments and stores the value in a temporary variable. For example, "$x = a + b + c$" becomes "$Temp1 = a+b$" and "$x = Temp1+c$" or "$Temp1 = b+c$" and "$x = Temp1+a$". Consider a scenario where :

   $x = a + b + c;$

   $y = (b + c) * a$

   In this case if the compiler splits the equations as " $Temp1 = b + c$", "$x = Temp1 + a$" and "$y = Temp1 * a$" the implementation would be more efficient as the same temporary variable is used for both the equations. Identifying such expressions and accordingly splitting the equation is called common subexpression evaluation. A lot of research has gone into identifying common subexpressions in a given code fragment.

5. Dead Variable Elimination : Variables which are declared and probably initialized but never used are called dead variables and memory need not be allocated for them.

2.3 The need for better optimization

The current optimization techniques tend to address the specific implementation. For example if a loop is present in the code the optimizer looks only at the inefficiencies in loop (like loop invariant computation inside the loop), it does not actually look at the loop itself as being an inefficient construct. To recognize the latter form, some kind of heuristics about the code, its implementation etc. is required. The compiler should be able to recognize the similarity/context of application and make an intelligent suggestion to the user about an alternative. There may be many coding patterns/structures that may have been recognized to be inefficient, the compiler must be able to recognize these patterns during compilation and make suggestions to the user. However to make these suggestions the compiler will need to have some knowledge and not just the code to be compiled, as is the case with current compiler technology.The knowledge could be in the form of database of code patterns, rules defined on the patterns etc

Chapter 3

Control Flow Graphs as a Representation

Control Flow Graphs(CFG) represent the flow of control in given code fragment (i.e the sequence in which the instructions are executed). As the name suggests the control flow is represented as a graph data structure.

3.1 Graphs and Graph Terminologies

A graph can be defined as a set of objects called nodes, linked together by another set of objects called edges. For example the Figure 3.1 shows a graph with 4 nodes and 5 edges. Graphs can be expressed as $G = (V, E)$ where V is the set of all nodes present in the graph and E is the set of all edges in the graph.

Graphs can either be *directed* or *undirected*. In an undirected graph an edge from node A to node B would imply an edge from node B to node A i.e. $A - -B \implies B - -A$. For a directed graph an edge from node A to node B does not imply and edge from node B to node A i.e. $A- > B \not\Longrightarrow B- > A$.

A *rooted* graph is graph in which a single node is represented in a special way. It is expressed as $G = (V, E, v')$ where V and E are the set of nodes and edges respectively and v' is the root node such that $v' \in V$. An example of a rooted graph is shown in

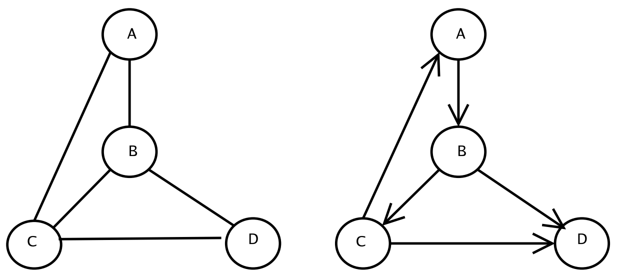


Figure 3.1: Undirected and Directed Graph

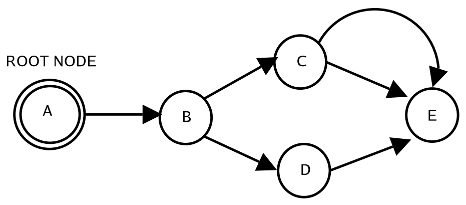


Figure 3.2: Rooted Graph

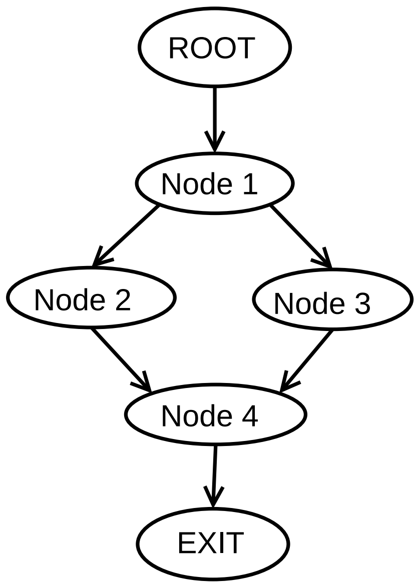
Figure 3.2.

3.2 CFGs as graphs

A CFG can be represented as a *rooted directed* graph. The nodes of the graph would represent instructions that have to be sequentially executed and an edge would represent a change of control from node to other. The root node would be the entry point for the code fragment. The CFG can be expressed as $G = (V, E, v')$ where V is the set of all nodes, E is the set of all edges, v' is the root node for the graph.

3.3 Graph file format : Dotty

Graphs quiet often need to be expressed as files(text files), to do this there are a variety of graph file formats, Example: GML, VCG, dotty. We use the *dotty* file format for representing the graphs. An example of a graph and its corresponding dotty file representation is given in 3.3.



```
digraph G {
0 [label="ROOT"];
1 [label="Node 1"];
0 -> 1;
2 [label="Node 2"];
1 -> 2;
3 [label="Node 3"];
1 -> 3;
4 [label="Node 4"];
2 -> 4;
3 -> 4;
5 [label="Exit"];
4 -> 5;
}
```

Figure 3.3: A Graph and its Dotty Representation

Each line in a dotty file ends with a semi-colon. The first line indicates if the graph is directed or undirected, on the basis the keyword *'graph' or 'digraph'* used. The dotty format supports $<$*attribute,value*$>$ pairs for both edge and node objects.

3.4 Control flow graph as a dotty graph

In this research we represent the CFG in *dotty* format. Each basic block[1] in the compiler corresponds to a node in the CFG, and a jump from one block to another corresponds to an edge of the CFG. Since the change of control from one basic block to another does not imply the reverse the edges of the CFG are directed.

3.4.1 Node Labeling

In a control flow graph, each node represents a set of sequential instructions. Each possible instruction is assigned a *code*, for example the instruction $a = b + c$; represents an addition and hence can be coded as **Add**. The label for the entire node can be obtained by concatenating the *codes* of all the instructions in the node. By doing so the representation becomes *variable name independent.* As an example consider :

0 [**commands** = " $a = b + c$; $e = d * a$; "];

$a = b + c$	**Add**
$e = d * a$	**Mul**

Table 3.1: Node Labeling Example

[1]Basic block is a set of sequential statements in the program, that belong to the same node in the CFG

3.4.2 The complete CFG representation

As an illustration consider the *C* program shown in figure 3.4, it would result in the CFG shown in figure 3.5 and the corresponding dotty file will be as shown in figure 3.6:

```
The C Program Example.C
#include<stdio.h>

int main()
{
        int i,j,k;
        i = rand();
        j = rand();

        if(i > j)
        {
                k = i;
                printf("I is Greater than J");
        }
        else
        {
                k = j;
                printf("J is Greater than I");
        }

        return 0;
}
```

Figure 3.4: An Example C Program

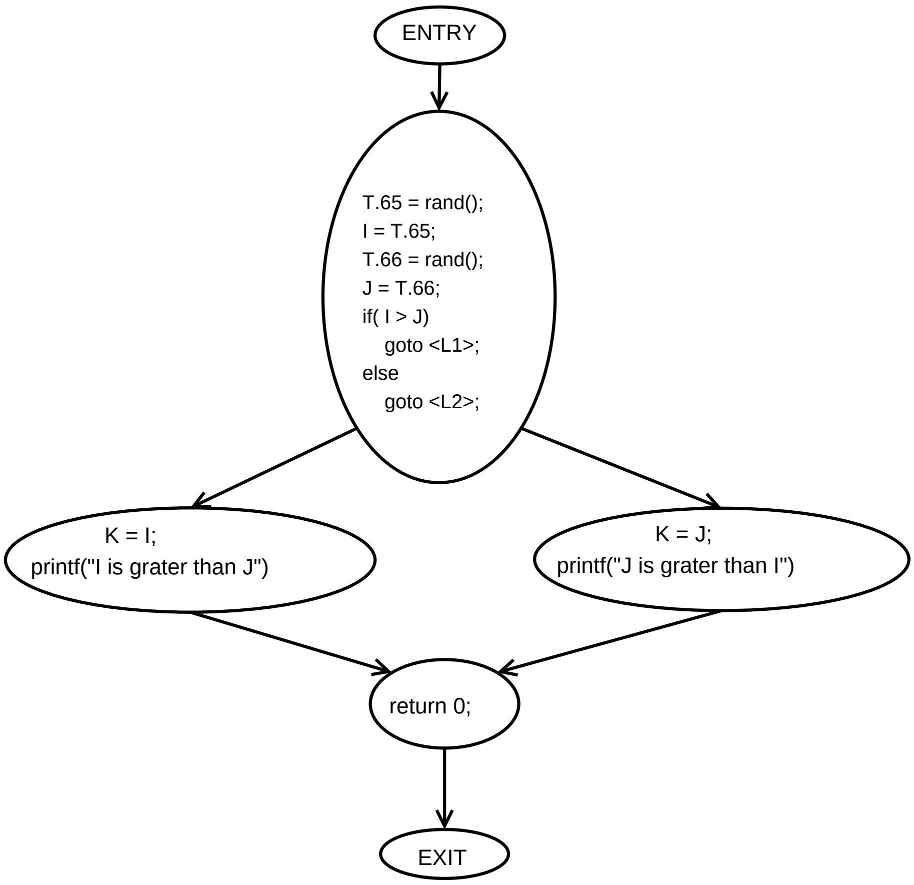


Figure 3.5: An Example of The CFG

3.5 Generating CFGs using GCC

GCC outputs various graph files during the compilation process, these graphs are in VCG format. To dump the CFG in dotty format the GCC source was modified. The details of the modification and other changes made to the GCC code is explained in *Appendix A*.

The Dotty File Example.dot

```
digraph Example {
ENTRY − > 0;
0 [label = “ T.65 = rand(); i = T.65; T.66 = rand(); j = T.66
        if (i > j) goto < L0 >; else goto < L1 >; ”];
0 − > 1;
1 [label = “< L0 >:; k = i; printf(“I is greater than J”);”];
0 − > 2;
2 [label = “< L1 >:; k = j; printf(“J is greater than I”);”];
1 − > 3;
2 − > 3;
3 [label = “< L2 >:; return;”];
3 − > EXIT;
}
```

Figure 3.6: An Example of CFG in dotty format

Chapter 4

Index for Code Patterns

Indexing is technique of creating references to objects such that access/retrieval/search on them becomes optimized. For example, consider a medium sized database, where there will typically be more than a thousand records in the table. Storing them in sequential manner will be highly inefficient if you want to retrieve some record from the end, a slightly better approach would be to keep a table which specifies the location of every 1000^{th} element, in this case if you want to search for record 1002^{nd} position you need not traverse all the 1001 records, just find the entry for the 1000^{th} record and traverse two records.

4.1 The need for indexing

In our case we have multiple control flow graphs (this number can vary and in a full fledged application can be very large) and we have to find all of their instances if any in the main input graph. Searching the input sequentially for all the interesting patterns that are there in the database may prove computationally expensive. Hence we need an index over the interesting pattern database.

A very simple depiction of the situation is shown in figures 4.3, 4.1, 4.2. For the scenario shown in the figures, the system should identify the presence of both *CODE1* and *CODE2* in the main input code fragment.

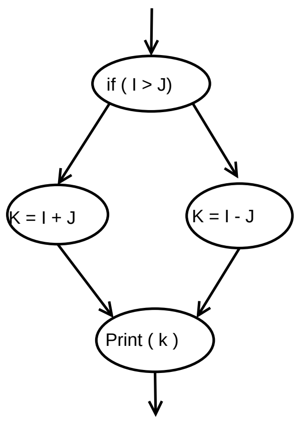


Figure 4.1: An Example Code Pattern : CODE1

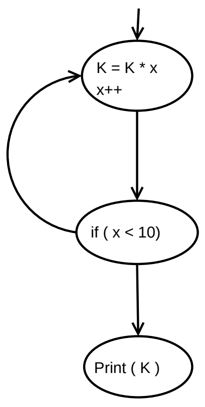


Figure 4.2: An Example Code Pattern : CODE2

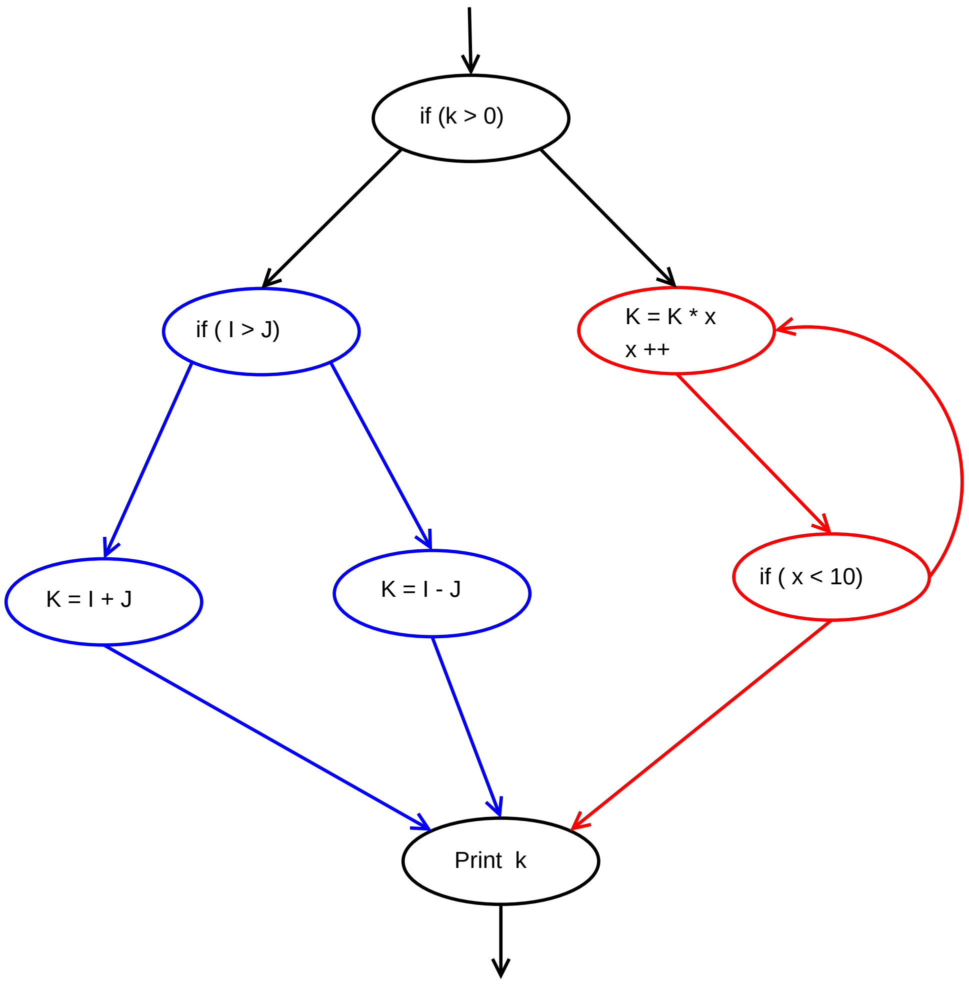


Figure 4.3: The CFG of the Input Code

4.2 The modified CFG

The code patterns are not directly indexed, but their CFGs are modified and then the index is generated. The CFG is modified to capture the *if-else* structure of the code. All nodes where the control flow does not branch are deleted and their node labels are pushed to the edges.

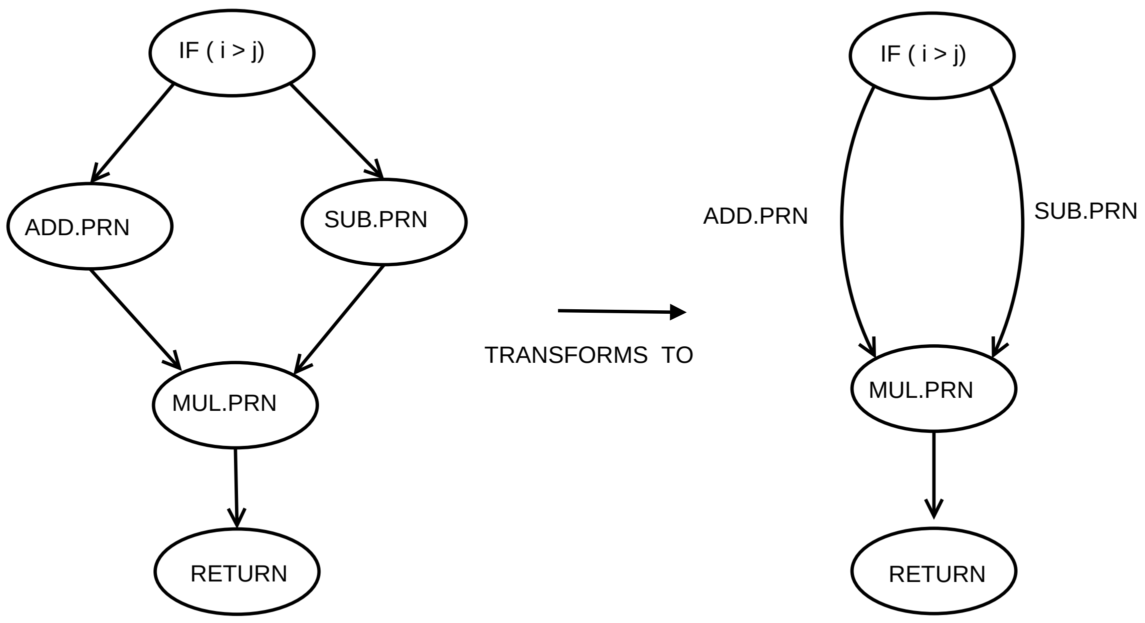


Figure 4.4: The Modified CFG

Doing so captures only the control flow structures, decision making constructs as nodes. Also all the instructions except the branching instructions in the branching node (i.e. the *if* instruction) are pushed to the edge and only the branching instruction is kept in the node as shown in 4.5.

4.3 The indexing structure

The indexing structure is built on the modified CFG. The index is a tree data structure with the root node of the tree representing the entry node in the modified control flow graphs of all the patterns in the database. The root node has only one child node.

All others nodes can have upto three children. The three children are represented

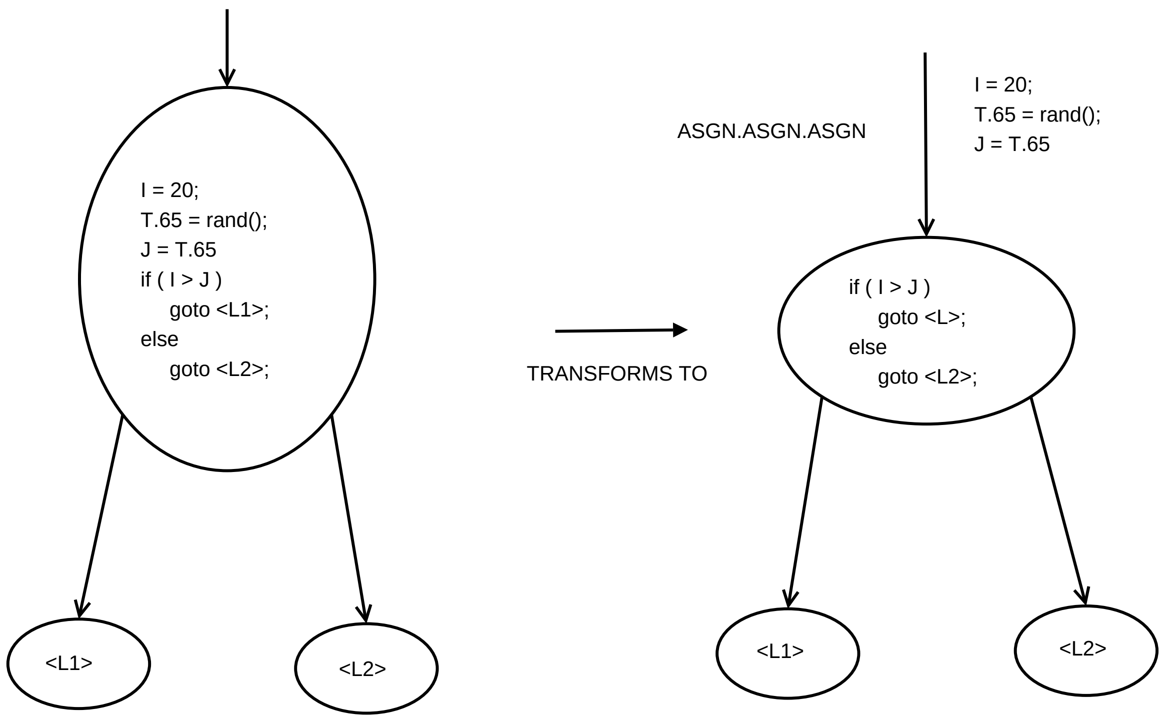


Figure 4.5: The Modified CFG

by *if-child*, *else-child* and *join-child*. The *if-child* node represents the execution path from the current node when the corresponding *if* condition is true, similarly the *else-child* represents the execution path when the *else* condition is true. The *join-child* represents the point where the *if* and *else* controls merge(i.e. where the entire *if-else* structure ends). The *index* looks as shown in figure 4.6.

The table is of the form as shown in 4.1. The *table* has two columns, the *Weight/Index* column stores the index *key*. The *key* could be any of the edge attributes. For example, the key could be :

1. The number of instructions represented by the edge

2. The *edge label* of the modified CFG

The corresponding row in the *Graphs* column contains the name of all programs that have the corresponding *Weight/Index*. A Graph can occur only once in a table, because two entries for the same graph would indicate simultaneous execution paths (i.e. parallel execution paths) which is not possible.

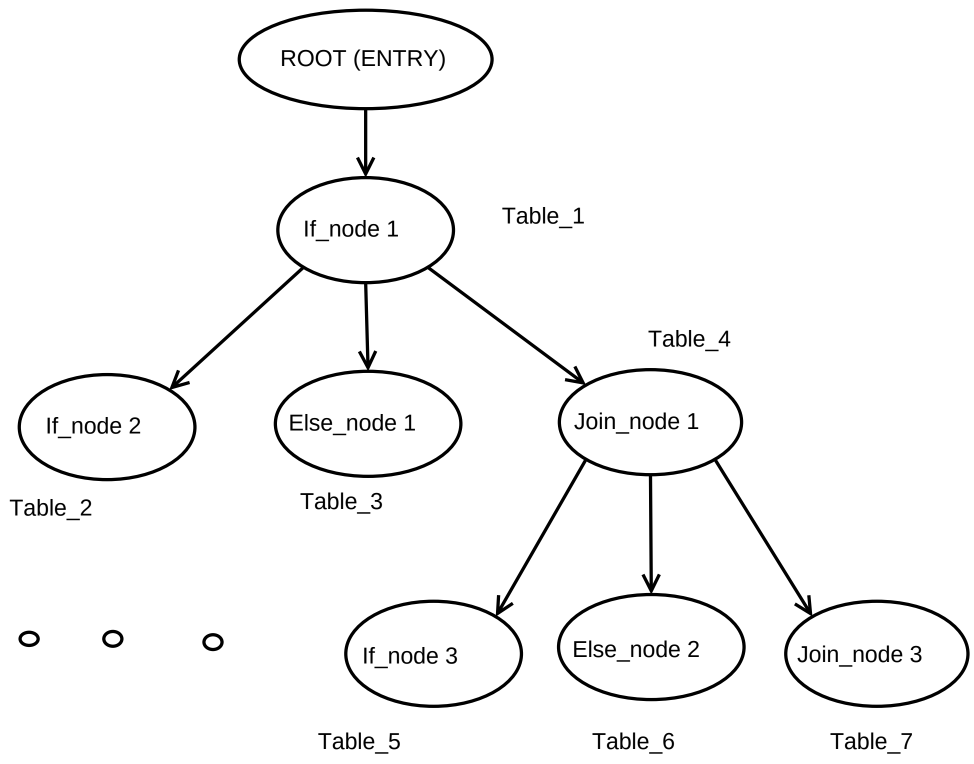


Figure 4.6: The Indexing Structure

Weight / Index	**Graphs**
Add-Prn	g1,g2
Prn-Prn	g3,g4

Table 4.1: An Example of the Table_x

4.4 An Example

The input *C* code is a simple implementation to find the maximum of three numbers. The code is as shown in figure 4.7. Figures 4.8 and 4.9 represent the CFG and the modified CFG of the program respectively. Figure 4.10 shows the corresponding index for the code. In figure 4.10 the values in the parentheses represents the content of the table, since the index was generated for only one program all tables have only one entry.

```
/* A C program to find Maximum of three numbers */

#include<stdio.h>

int main(void)
{
        int x=10,y=20,z=30,max;
        if(x > y)
        {
                if(x > z)
                {
                        max = x;
                }else
                {
                        max = z;
                }
        }else
        {
                if(x > z)
                {
                        max = y;
                }else
                {
                        max = z;
                }
        }
        return max; }
```

Figure 4.7: A Program to find Maximum of three numbers

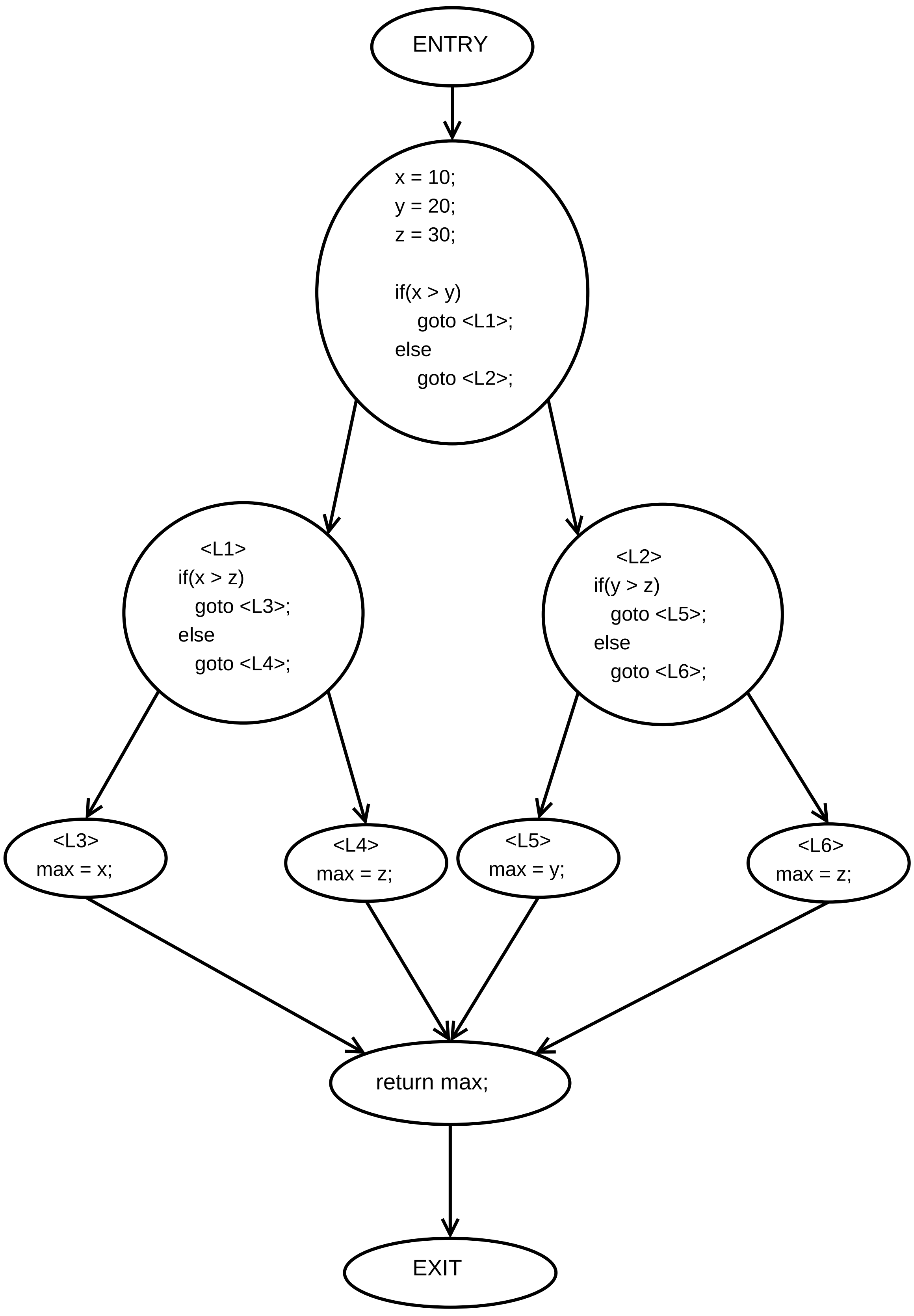


Figure 4.8: The CFG for a program to find Maximum of three Numbers

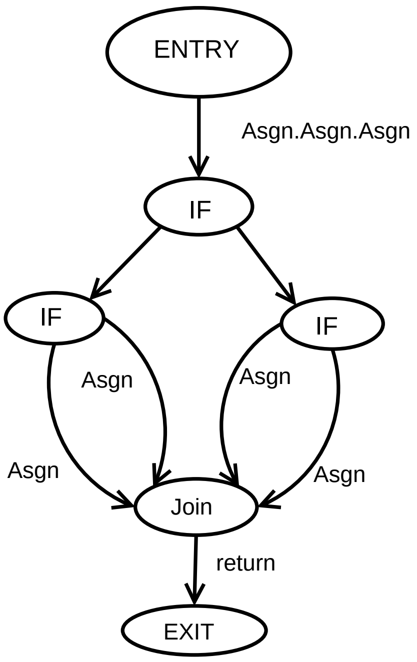


Figure 4.9: The Modified CFG for a program to find Maximum of three Numbers

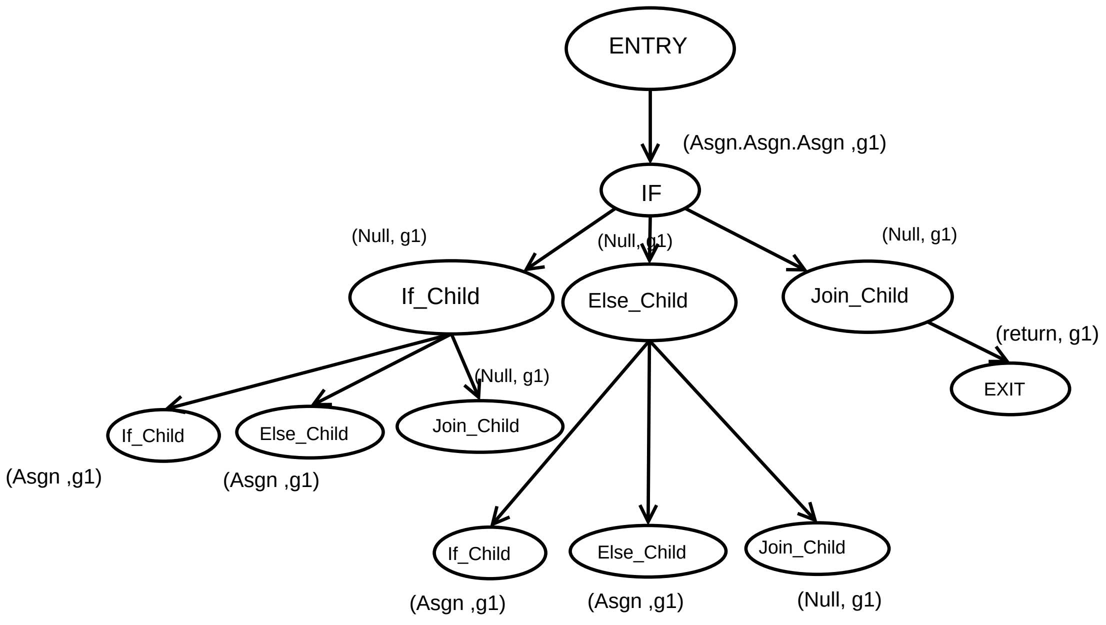


Figure 4.10: The index tree for the Program

Chapter 5

An Overview of Design and Implementation

The initial prototype was implemented in *object oriented perl.* Perl was chosen, as it allows for fast prototype building and also for its powerful regular expression (required for parsing) support.

5.1 The Classes

Each data structure (graphs for CFG and trees for the indexes) was implemented as a collection of perl objects. The basic perl object was a *node* object. There were two types of node object,

1. *graphNode*: This class represents the node in a CFG (and modified CFG).
2. *indexNode* : This class represents a node in the index tree.

5.1.1 The CFG Data Structure

The CFG was implemented as collection of nodes, each having pointers to other nodes that it has an edge to. Since every outgoing edge has a corresponding *edge label*, there were two dynamic arrays one for the edges and the other for the *edge labels.*

5.1.2 The ***Index*** Data Structure

The *indexnode* has three pointers each pointing to its child nodes i.e. the *ifChild*, *elseChild* and *joinChild* respectively. The *indexNode* also stores a hash table to implement the *Table_x* structure shown in table 4.1. The *key* for the hash is the *weight/label* parameter shown in table 4.1, and the value was an array storing the graph names.

5.2 Code Organization

The code was organized into three modules. They are:

1. *modifyGraph* : The *modifyGraph* module takes the output of GCC i.e. the CFG in dotty format and generates the modified CFG with all *node labels* pushed to the edge. It assumes all files in the current directory with extension ".cfg" to be CFG files and outputs ".mg"[1] files.

2. *generateIndex* : The *generateIndex* script accepts all modified graph files in the current directory, generates the index and writes the index into a file.

3. *searchPattern* : Given the index file name and an input file, this module searches for any of the indexed files in the input file.

[1]mg stands for modified graph

Chapter 6

Performance Results

The algorithm was implemented using *perl scripts*. To test the algorithm, a database of programs (having 7 programs) was created. Some of the programs in the database were such that, they were sub-parts of other bigger programs in the database. Ideally the system should identify the smaller programs as being part of other bigger programs.

6.1 An Example of the Test Scenario

The code database was populated with the codes shown in figures 6.1 and 6.2, along with a few other unrelated codes. The system when given *Code Sample2* as input, recognized the presence of *Code Sample1* in *Code Sample2*, since *Code Sample2* was itself there in the database the system also recognized *Code sample2* being a part of itself. Similar experiments were carried out with different code patterns, with one subsuming the other, one subsuming two others etc. the results obtained were consistent with the expected results.

```
#include<stdio.h>
int main() {
        int i,j,k;
        i = 20;
        j = 30;
        if(i>j)
        {
                k=i;
                printf("%d",k);
        }
        else
        {
                k=j;
                printf("%d",k);
        }
}
```

Figure 6.1: Code Sample1

```
#include<stdio.h>
int main() {
        int i=20,j=30,l=40,k;
        if(l > i)
        {
                i=20;
                j=30;
                if(i>j)
                {
                        k=i;
                        printf(“%d”,k);
                }
                else
                {
                        k=j;
                        printf(”%d”,k);
                }
        }
        else
        {
                printf(“Sample Program2 : It contains Program 1”);
        }
        return 0;
}
```

Figure 6.2: Code Sample2 : Containing Code Sample1

Chapter 7

Related Literature

A lot of research effort has been focused on determining code equivalence, i.e. to see if two given code fragments essentially perform the same. Though the final end goal would be an ideal system that could identify code equivalence, this thesis addresses just a very small issue.

Since our approach is based on graph matching (*control flow graphs*), graph algorithms for finding subgraph isomorphism, graph matching are also discussed. The above mentioned problem of subgraph isomorphism (determining if a given graph is a subgraph of another given graph) is a well addressed research topic [9, 14, 12]. We are addressing a scenario where we have a database of interesting graphs and we have to parse a given input graph to identify all the interesting sub-graphs present in it. This problem is more complex than the previously presented subgraph problem as we do not know the exact sub-graph we are searching for, it could be any of the ones in the database.

7.1 Code Pattern Matching

Paul and Prakash describe a framework for searching similar code patterns and code re-engineering in [10]. Given an input search pattern, the algorithm checks for similar patterns in code fragments. The search pattern is specified in *scruple*, a regular expression based language described in the paper. In *scruple* each programming construct has an equivalent *scruple* representation. For example the declaration of variable x (of any data type) is represented as "\$$t$ x", there are similar representations for other programming constructs. The difference between our approach and [10] is that in *scruple* a search is made for only a single code pattern, not multiple as we intend.

Baxter et al. in [3] suggest an algorithm based on abstract syntax trees for determining code *clones*. Here *clones* refers to a copy of a code fragment. The copy could be modified slightly or kept unchanged. Hashes are used to reduce the computational complexity of the algorithm. The algorithm differs from ours in that it uses abstract syntax trees instead of CFGs and also it does not recognize a give code pattern, but recognizes replication of any pattern.

In [5] Chang et al. describe a two-phase optimizing compiler, the first phase is a profiling tool which collects various statistics about the code, and the second phase is a profiler-based compiler that takes into account the profiling statistics collected in the first phase, while compiling. This approach is similar to our idea of a profiler aiding the compilation process, except that in our case the profiling tools uses other code pattern examples (the code patterns in the database) to come up with suggestions for the compiler.

Ferrante et al. in [4] describe an approach based on an intermediate code representation, *Program Dependence Graphs*(PDG). In PDG representation the data flow and control flow dependencies are explicitly captured.

7.2 Graph Algorithms

Srinath Srinivasa et al. in [12], suggest an indexing structure called LWI (Labeled Walk Index) to address the sub-graph isomorphism problem. The LWI consists of a prefix-tree constructed by enumerating all walks in given graph of length upto *'n'*, bigger the value of *'n'* the bigger the index. Our approach is based largely on the LWI index.

In [14] Yan and Han suggest a solution the to sub-graph isomorphism problem, the algorithm proposed builds a lexicographic order of graphs, on the basis of a DFS search of the graph.

In [9] Giugno and Shasha describe a graph database approach, to find all occurrences of a subgraph in database of graphs. We address the scenario wherein we need to verify if all the graphs in the database are sub-graphs of the input graph. The algorithm suggests an hash-based fingerprinting to represent graphs.

Chapter 8

Conclusion and Future Directions

8.1 Conclusion

The thesis addresses the problem of determining if two given code fragments or *part code fragments* are equivalent. The approach suggested assumes two code fragments to be equivalent if the instruction ordering in them is same. This assumption is not necessarily true. The effect of the operands on the instruction, is ignored. The suggested approach can be used as a *first level filter* to give the most likely candidates for equivalence, a further *second level filter* might be needed to determine exact code equivalence. The *second level filter* can afford to have a higher computational complexity, since its input set is going to be very small as compared to the *first filter*.

The suggested approach can also be used to find the structural similarity of code fragments, this could find application in code restructuring, code optimization etc.

8.2 Enhancements

The system is currently implemented as a stand-alone entity. It would be more beneficial if the system is incorporated as part of a full fledged compiling tool. The suggested algorithm can be used to identify patterns of interest in the input code. The identified code fragments can then be treated differently as the need of the application may be, for example:

1. The code fragment can be highly un-optimized, hence perform rigorous optimization on it.

2. The code fragment can be replaced with another code fragment, which suits the application needs better.

8.3 Future Directions

Program Understanding and *Program Equivalence* are highly researched topics [11] [2] [13]. Though the final aim of the system is to achieve code understanding, it is currently far from it. To begin with the *second level filter* described in 8.1 needs to be developed to determine exact code equivalence.

A larger goal would be to achieve *Program Understanding*, for example given two algorithms for sorting like bubble sort and quick sort the system should recognize that the two algorithms essentially perform the same operation of sorting.

Appendix A

GCC - GNU Compiler Collection

A.1 An Overview of GCC

GCC stands for GNU compiler collection, initially GCC supported only the *C* language and was then known as GNU C Compiler, but later it added support for various different languages. GCC currently supports most programming languages including C++, Java, Ada, Pascal, Fortran etc. GCC also has a very powerful optimization engine.

A.2 GCC Architecture

The GCC has a layered architecture, such an architecture makes it easy to add support for new language in GCC. The different layers of GCC being :

1. **Front End** - The frontend converts the code in the high level language to a tree based intermediate representation. GCC supports two different tree based intermediate languages *Generic* and *Gimple*. The code is first converted to *Generic* and then to *Gimple*.

2. **Middle End** - The middle end converts the *Gimple* trees to RTL(Register transfer Logic) represntation. RTL can be assumed to be the assembly language for a system with infinite registers. It also performs certain optimization(system

independent optimization) on the *Gimple* representation before converting it into the RTL representation.

3. **Back End** - RTL representation being more close to the underlying target system, is ideal for target specific optimization such as those based on register allocation etc are performed by the back end. The back end generates the compiled output file.

A.3 Intermediate Representations

The intermediate representations in GCC are based on the traditional tree data structure. GCC currently has three intermediate representations.

A.3.1 Generic Representation

Generic is a language independent tree representation. The parse trees of language front ends are converted to *Generic* trees, *Generic* trees being language independent the semantics of the input language need to be explicitly described. *Generic* trees though language independent, they are structurally complex, as in they support nested function calls, multiple operations in a single statement.

A.3.2 Gimple Representation

Gimple trees are structurally simple. They support only single(micro instructions) in every statement. An example for the comparison of *Generic* and *Gimple* representation is shown in table A.1:

Generic	**Gimple**
printf("Value of k is %d",k=i+j);	k = i + j;
	printf("The Value of k is %d",k);
if(foo(a,b))	Tmp_1 = foo(a,b);
{}	if(Tmp_1)
else	{}
{}	else
	{}

Table A.1: Comparison of Gimple and Generic Representations

A.3.3 Register Transfer Language

The Register Transfer Language assumes the system to consist of an infinite number of registers and represents the variables on that basis. All processor specific optimization is done in this representation.

A.4 Optimization in GCC

GCC provides different levels of optimization which can be specified as a command line parameter. Optimization can be done for execution time / memory usage or both.

The various passes of the GCC optimizer is as follows :

1. Remove redundant code : In this pass the entire code is swept to remove any dead code, for example *if* statements which evaluate to constant value, redundant code added for error handling etc.

2. Lower expressions and Build control flow graph : Complex if constructs are lowered to simpler statements and the control flow graph for the entire code fragment is built.

GIMPLE	SSA
$a = 3;$	$a\_1 = 3;$
$b = a + 2;$	$b\_2 = a\_1 + 2;$
$c = a * b;$	$c\_3 = a\_1 * b\_2;$
$a = a * 2;$	$a\_4 = a\_1 * 2;$

Table A.2: Example of GIMPLE with SSA

3. Find all referenced variables : This pass finds all the variables referenced in a function and generates and index on them. Any further reference to the variables are made on the basis of the index.

4. Generate SSA statements : SSA (Static Single Assignment) forms the basis of tree based optimization used in the current GCC versions. In SSA based representation the flow of data is explicitly represented. In SSA every time a variable is assigned a value (i.e. the variable is on the left hand side of the assignment operator) a new version for the variable is created and the latest version of the variable is used henceforth, as shown in A.2 Representing variables along with their version numbers helps keep track of the variables and also code restructuring if any by the compiler.

5. Dead code elimination : In pass all statements whose results are unused are removed. For example if there is a statement $a = (b * c) - 4 * d$ and the value of a is never used anywhere else in the code, the statement $a = (b * c) - 4 * d$ is then redundant and can be ignored.

6. Forward propagation of single use variables : If a variable that has been used only once in the code and it has been initialized, then its value is substituted directly in the code. As shown in A.3 :

7. Dead Store Elimination : This pass eliminates all writes done to memory which

Actual Code	Optimized Code
#include< *stdio.h* >	#include< *stdio.h* >
main()	main()
{	{
int $single\_Use\_Variable = 10$;	...
...	...
/* Variable used once */	/* Variable substituted */
$xyz = myexpression(single\_Use\_Variable)$;	$xyz = myexpression(10)$;
}	}

Table A.3: Forward propagation of single use variables

are over written before a read on them happens.

8. Partial redundancy elimination : This pass is similar to the *common subexpression elimination* explained earlier on. Any sub-expression that is common to 2 or more expressions is evaluated only once and not repetitively for all expressions

9. Loop Optimization : Loops are inefficient programming constructs and optimizing them can lead to better performance. Loop invariant variables/equations (i.e. variables whose value does not change in successive iterations of the loop) can be moved out of the loop.

GCC optimization module performs 34 iterations in all to achieve the highest optimization level. The above mentioned passes are just a few of the 34 passes.

A.5 Modification to GCC source

The GCC had to be modified to dump the CFG of code fragments. GCC official documents describe the GCC internals in good detail and were used as reference

[6, 7, 8]. Two command line options were added to GCC to do the same. The option :

1. "- Z" : As mentioned previously GCC has 34 optimization passes, this option takes an integer as its input and dumps the control flow graph only for that pass. For example giving the option "-Z 1" will dump the CFG of the input code as it is, whereas giving the option "-Z 34" will dump the CFG after GCC has performed all its optimization passes i.e. the CFG of the optimized code.

2. "- F" : This option takes a file name as input, the CFG is then dumped into the file referred to by the file name. If this option is not mentioned the then the CFG is by default dumped to the standard output terminal.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison Wesley, 1986.

[2] Francoise Balmas, Harald Wertz, Janice Singer. *Understanding Program Understanding*, iwpc, vol. 00, no. , p. 256, 8th 2000.

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, Loraine Bier, *Clone Detection using Abstract Syntax Trees*, In Proceedings of ICSM98, November 16-19, 1998, Bethesda, Mayland.

[4] Jeanne Ferrante, Karl J. Ottensein, Joe D. Warren, *The Program Dependence Graph and its Use in Optimization*, ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3, July 1987, Pages 319-349.

[5] Pohua P. Chang, Scott A. Mahlke, Wen-Mei W. Hwu, *Using Profile Information to assist Classic Code Optimizations*, Software-Practice and Expierience, 21(12):1301-1321, December 1991.

[6] *GCC Internals Documentation*, http://gcc.gnu.org/onlinedocs/gccint.ps.gz.

[7] *GCC Man Pages*, http://gcc.gnu.org/onlinedocs/gcc.ps.gz.

[8] *GCC - Tree SSA Architecture*, http://people.redhat.com/dnovillo/pub/tree-ssa/doc/html/index.html.

[9] Rosalba Giugno and Dennis Shasha, *GraphGrep: A Fast and Universal Method for Querying Graphs*, Proceedings of International Conference on Pattern Recognition, Quebec, Canada, August 2002.

[10] Santanu Paul, Atul Prakash, *A Framework for Source Code Search using Program Patterns*, IEEE Transactions on Software Engineering, 20(6), pages 463–475, 1994.

[11] Gregor Snelting, *Concept analysis a new framework for program understanding*, In proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 1-10, Canada, 1998.

[12] Srinath Srinivasa, Martin Maier, Mandar R. Mutalikdesai, Gowrishankar K. A., Gopinath P. S., *LWI and Safari: A New Index Structure and Query Model for Graph Databases*. In Proc. International Conference on Management of Data, pages 138–147, Goa, India, 2005.

[13] Haruki Ueno, *A Generalized Knowledge-Based Approach to Comprehend Pascal and C Programs*, In proceedings JCKBSE 98, pp. 132-139.

[14] X. Yan and J. Han. *gspan: Graph-based substructure pattern mining*. In Proceedings of ICDM 2002, 2002.