CONTENTS

1. PROJECT SYNOPSIS

1.1 Introduction and aim

Translation from assembly code to high-level language (HLL) code is of importance in the maintenance of legacy code, as well as in the areas of program understanding, porting and recovery of code. It is very useful in making assembly codes platform independent. The translation process aims not to get the original source program but a program, which is identical to the original program with respect to the output generated. The translation process is part of a larger process, which converts machine code to HLL code. A program, which converts machine code to HLL code, is called a ***decompiler.*** We can write a decompiler by first converting the machine code to assembly code and then by making use of the assembly to C (asm2C) translator to get the HLL code.

1.2 Objective of our Project

As a matter of fact, there exists an object code to ASM converter (disassembler) called *objdump* by GNU, but there isn't yet a single fully functional ASM to C translator for Linux Environment. And this is precisely what relipmoC does.

Our Reverse Engineering project, relipmoC[1] generates an equivalent C code, for a given input Assembly language (80386) code.

1.3 Description

In essence, the problem of asm2C translation becomes one of program comprehension; being able to understand existing code generated

[1] Read reverse of Compiler

by assembler and compiler tools, and to apply techniques, which will "undo" what, the assembler/compiler has done.

Consider that the processor, by executing the machine language program, can successfully figure out what to do with any input, to create the correct output. An assembly code program is a set of instructions, plus a set of data bytes. A HLL is more abstract than an assembly code program - details (such as individual instructions, registers, and so on) are abstracted away. In a way, asm2C translation is the *judicious deleting of unneeded information.* A series of transformations can remove the machine specific aspects of the program semantics. Machine code features such as individual registers disappear with these transformations. Finally, the intermediate representation can be structured, replacing jump instructions with loops and conditionals.

1.4 The techniques used

The main types of analyses are:

Data flow analysis to recover HLL expressions and statements (other than control-transfer statements), actual parameters, and function return values, and to remove hardware references from the code, such as registers, condition codes and stack references;

Control flow analysis to recover control flow information, such as loops and conditional statements, as well as their nesting level and;

Type analysis to recover high-level type information for variables, formal and actual parameter types, and function return types.

1.5 Implementation

The project *"relipmoC" ver 0.01*, which is an i386 ASM to C-code translator for Linux environment, is implemented and successfully tested for simple codes. The next version *relipmoC1.1* is under development.

The three phases discussed above are implemented in C++ using the following data structures, using the *Standard Template Library* provided by the GNU g++ compiler:

- *blocktab:* blocktab contains pointers to all the basic blocks. It also contains a list of all intervals (see intervals) used in the original source code.
- *basicblock:* contains the actual assembly instructions, a list of pointers to it predecessors.
- *stringtab:* contains a list of strings that are used in the original source program.
- *symtab:* this stores information regarding the global variables used in the original source code.
- *interval:* this contains the list of all basic blocks that belong to an interval

We have made use of GNU's *LEX and YACC* programming tools for parsing the input assembly program.

2 INTRODUCTION

An assembly to C language translator can be better termed as a **'Decompiler for Assembly Languages.'**

Consider this fact given below:

> In the context of fixing the year 2000 bug, the Gartner Group estimated that many organizations are missing 3% to 5% of their source code portfolios. This means that a medium-sized information systems organization with a software portfolio of 30 to 50 millions lines of code could easily be missing a million lines or more. Further, some large organizations have thousands of lines of code written in assembly code and cannot benefit from state of the art object-oriented techniques unless the assembly code is reengineered into some HLL.

This fact highlights the necessity of a tool to re engineer the source code from an executable or Assembly level file. In addition to recovery of lost source code, an asm2C translator can be used in the area of software reuse, portability of code.

But unfortunately, not much of research has been done in this field and decompilers are not freely available today. As a result of this, we chose to implement such a tool that can be useful as a translator from i386 Assembly level code to C code by implementation, and a Reverse Engineering tool by principle. Why we chose this topic and i386 language? – This question is answered in Appendix D.

relipmoC, read reverse of Compiler is a translator implemented in C++ on Linux platform, using tools like GNU Lex, Yacc and STL®. The input is an i386 Assembly code file generated by gcc 3.2.2 compiler and output generated is a C program, equivalent to the source code in terms of output.

3 LITERATURE SURVEY/PREVIOUS WORK

Decompilers have been considered a useful software tool since they were first used in the 1960s. At that time, decompilers were used to aid in the program conversion process from second to third generation computers. In this way, manpower would not be spent in the time _consuming task of rewriting programs for the third generation machines.

The first decompiler was used as a tool in the translation of software from second to third generation machines in 1960. Maurice Halstead at the Navy Electronic Labs developed this decompiler called D-neliac. This decompiler took machine code for IBM 7094 as input and produced Neliac code for Univac 1108.

In the 1970s and 1980s, decompilers were used to port programs, recreate lost source code, modify and debug binaries. In 1990s, decompilation has become part of a wider area, called reverse engineering. A detailed study and implementation of Decompilers for Windows platform, called *dcc* is done by Cristina Cifuentes [CIFUENTES1993].

In the Linux platform, a disassembler (objdump) is available, but no ASM to C translators are available. Our project, relipmoC is such a conversion tool in Linux.

4 REQUIREMENTS DOCUMENT

4.1 Introduction

Translation from assembly code to high-level language (HLL) code is of importance in the maintenance of legacy code, as well as in the areas of program understanding, porting and recovery of code. It is very useful in making assembly codes platform independent and software reuse. The translation process aims not to get the original source program but a program, which is identical to the original program with respect to the output generated. The translation process is part of a larger process, which converts machine code to HLL code.

4.2 Glossary

Translator$_{[n]}$: Program that takes as input a program written in one programming language and produces as output a program in another language [AHO86].

Compiler$_{[n]}$: A translator, which has source language as a high level language like C and target language is a low – level language.

4.3 Functional requirements definition

relipmoC serves as a tool for translation of intel 80386 assembly language to C code. The input to this software is the i386 code file and output is the corresponding C program.

4.4 Non – functional requirements definition

Software requirements: gcc 3.2.2 optimizing compiler.

OS requirements: Linux

Input constraints: The input file should be in i386 language and generated by the gcc 3.2.2 compiler with 0 optimization level. It is assumed to contain no syntax errors. The C source program should not contain bit – fields, structs, arrays and pointers.

Output constraints: The output generated is a program, equivalent to the source program, in terms of its output; and not the same program.

4.5 System Evolution

Fundamental Assumptions:

- The assembly language program used for translation was obtained by compiling a C program,
- The instruction set used (in the assembly language program) is that of Intel 80386 processor,
- The compiler used for compiling is gcc 3.2.2

Fundamental Limitations:

- User-defined data types like **struct**s, **typedef**s, **union**s and **bit-fields** add more to the confusion of the asm2C translator. Though it is easy to use structures while writing the code, it is almost impossible to figure out if a variable is a part of a structure or is it a basic type on its own, by looking at the compiled output.
- Use of processor-specific instructions/optimizations.
- The C code generated might need certain minor changes before execution.

Future improvements:

- relipmoC could be extended to make it a generic decompiler, by accepting input from the output of *objdump*, the disassembler by GNU.
- Make relipmoC fully automated by use of Artificial Intelligence techniques.
- Make relipmoC work for `arrays`, `pointers` and `structures`.

5 SYSTEM DESIGN

5.1 Block Diagram of relipmoC

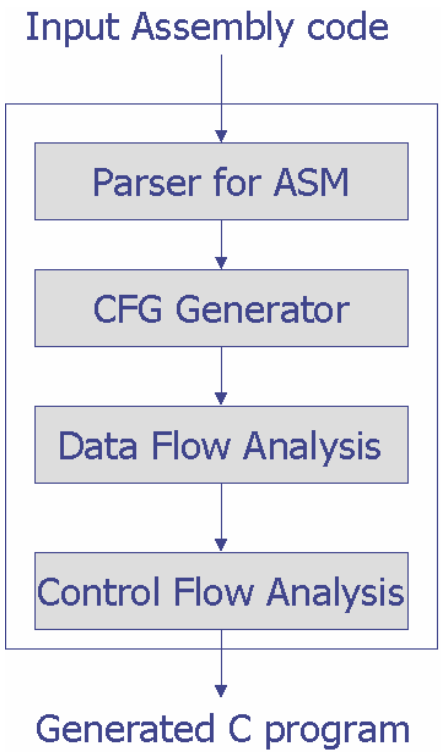

Fig 5.1: Block Diagram of relipmoC

5.2 Data Flow Diagrams

5.2.1 Data Flow Diagram - 0 level (Context Diagram) for relipmoC

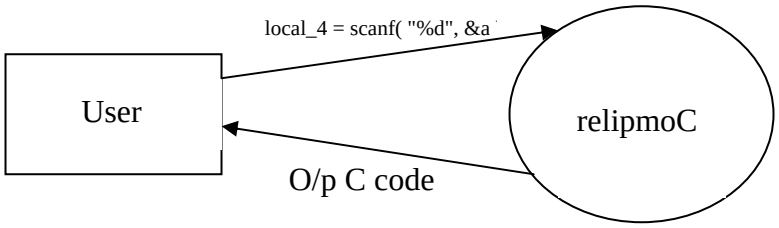

Fig 5.2: Context diagram of relipmoC

5.2.2 Data Flow Diagram for relipmoC

Generation of C code from input Assembly file:

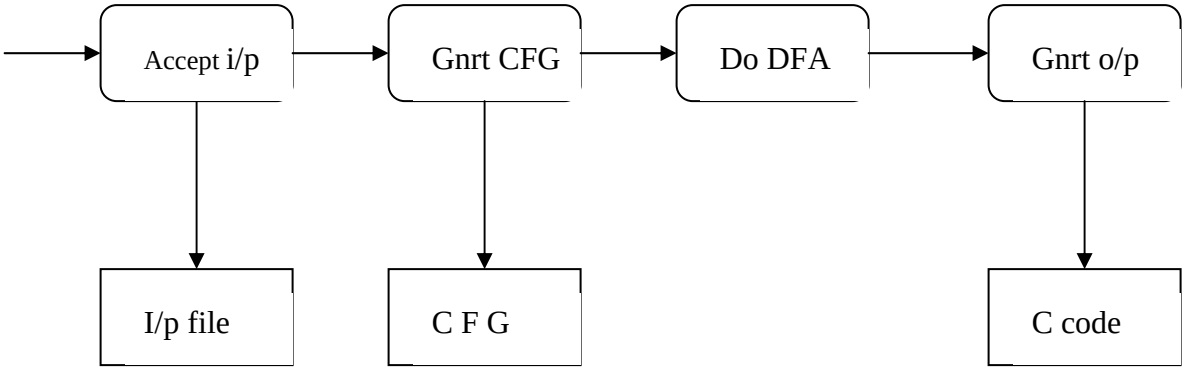

Fig 5.3: DFD (Level – 1) for relipmoC

5.3 ER Diagram for relipmoC

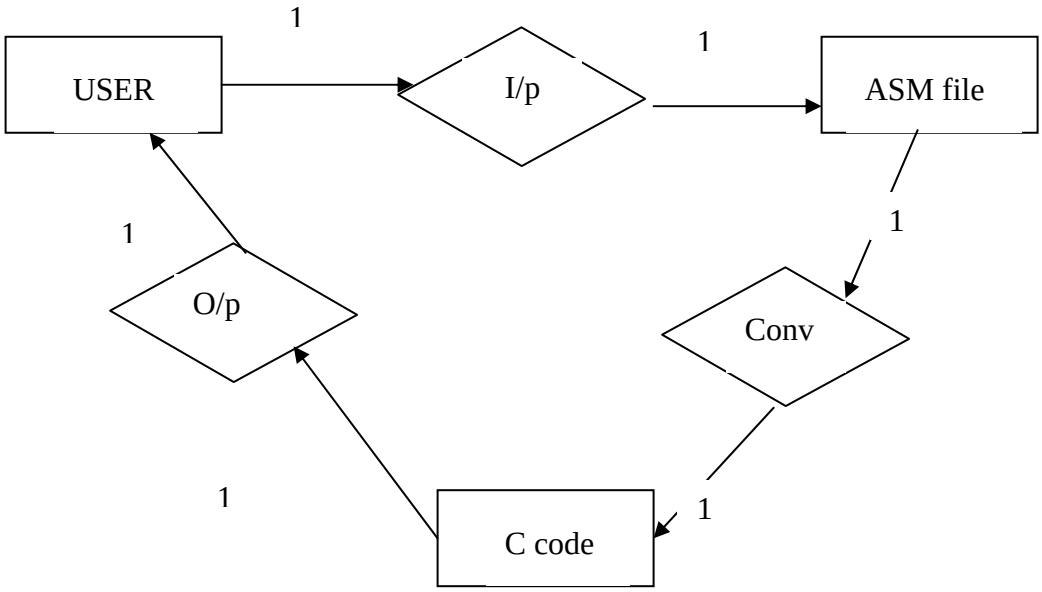

Fig 5.4: ER diagram for relipmoC

6 DETAILED DESIGN

The decompilation process starts with the input assembly language program. It then parses the program, and produces the control flow graph. The different phases, algorithms, data structures and functions used are discussed here.

6.1 CONTROL FLOW GRAPH GENERATION

The control flow graph generation phase constructs a control flow graph of basic blocks for each subroutine of the program. These graphs are used to analyse the program to find the control structures used in the source program.

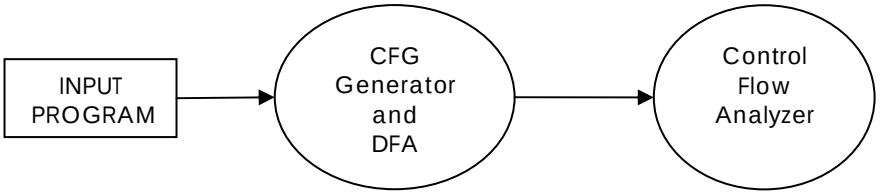

Fig 6.1: Context of CGF in decompilation process

6.1.1 Basic Blocks

In this section we formalize the definition of basic blocks. In order to characterize we need some definitions of program structure.

Definition: Let

- P be a program
- $I = \{ i_1, i_2, \ldots i_n \}$ be the instructions of P
- $D=\{ d_1,d_2,\ldots d_n \}$ be the data of P

Then, P = I U D

Definition:

Let

- P be a program
- $I = \{ i_1, i_2, \ldots i_n \}$ be the instructions of P

Then S is any instruction sequence if and only if

$S = [i_j, i_{j+1}, i_{j+2} \ldots i_{j+k}]$ $1 <= j < j + k <= n$ & i_{j+1} is in a consecutive memory location to i_j, for all $1 <= j <= k - 1$

Instructions are classified in two sets for the purposes of control flow graph generation:

Transfer Instructions (TI): The set of instructions that transfer flow of control to an address in memory different from the address of the next instruction. These instructions are:

- Unconditional jumps: the flow of control is transferred to the target jump address
- Conditional jumps: the flow of control is transferred to the target jump address if the condition is true, otherwise the control is transferred to the next instruction in the sequence.
- Subroutine call: the flow of control is transferred to the invoked subroutine.
- Subroutine return: the flow of control is transferred to the subroutine that invoked the subroutine with the return instruction.
- End of program: the program ends.

Non-Transfer Instructions: The set of instructions that transfer control to the next instruction in the sequence, i.e., all instructions that do not belong to the TI set.

Definition of Basic Block b = [i_1, i_2, …, i_{n-1}, i_n], n >= 1 is an instruction sequence that satisfies the following conditions:

1. [i_1, i_2, i_3 … i_{n-1}] belongs to NTI
2. i_n belongs to TI

or

1. [i_1, i_2, …i_{n-1}, i_n] belongs to NTI
2. i_{n+1} is the first instruction of another basic block.

A basic block has a sequence of instructions that has one entry point and one exit point. If one instruction of the basic block is executed, all other instructions executed as well.
The set of instructions of a program can be uniquely partitioned into a set of non-overlapping basic blocks, starting from the program's entry point.

6.1.2 Algorithm For Partitioning Into Basic blocks

Given a sequence of assembly statements we partition them into a disjoint set of basic blocks.

1. We first determine the set of *leaders,* the first statements of basic blocks. The rules we use are the following.
   i) The first statement is a leader.
   ii) Any statement that is the target of a conditional or unconditional jump is a leader.
   Iii) Any statement that immediately follows a conditional or unconditional jump is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

6.1.3 Control flow graphs:

A control flow graph is a directed graph that represents the flow of control of a program, thus, it only represents the instructions of such a program. The nodes of this graph represent the basic blocks of the program, and the edges represent the flow of control between nodes.

For the purpose of control flow graph (cfg) generation, basic blocks are classified into different types, according to the last instruction of the basic block. The available types of basic blocks are:

- 1-way basic block: the last instruction of the basic block is an unconditional jump. The block has one out-edge.
- 2-way basic block: the last instruction is a conditional jump, thus the block has two out-edges.
- Fall basic block: the next instruction is the target address of a branching instruction (i.e., the next instruction has a label). This node is seen as a node that falls through the next one, thus, there is only one out-edge.
- Return basic block: the last instruction is a procedure return or an end of program. There are no out-edges from this basic block.

The different types of basic blocks are shown below:

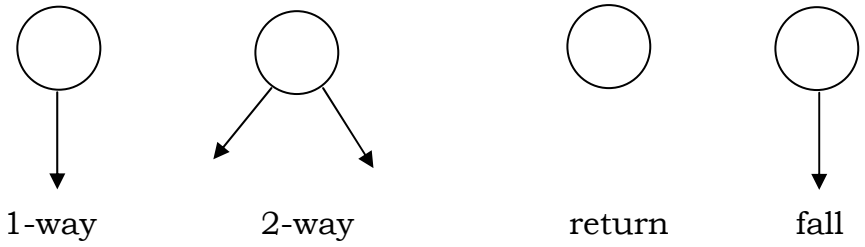

Fig 6.2: Node representation of different types of basic blocks

Consider the following fragment of code:

Example 6.1:

```
1           movl  $0, %eax
2           subl  %eax, %esp

3     .L2:  cmpl  $0, a
4           jne   .L4
5           jmp   .L3

6     .L4:  cmpl  $0, a
7           je    .L5
8           incl  a
9           subl  $8, %esp
10          pushl aa
11          movl  aa, %eax
12          addl  a, %eax
13          pushl %eax
14          call  function
15          addl  $16, %esp
16          movl  %eax, a
17          cmpl  $-1, aa
18          je    .L2

19    .L7:  cmpl  $0, ss
20          jle   .L9
21          jmp   .L2

22    .L9:  addl  $1000, aa
23          jmp   .L7
24    .L5:  movl  $34, aa

25    .L11: cmpl  $12, aa
26          jne   .L13
27          jmp   .L12

28    .L13: movl  aa, %eax
29          movl  %eax, ss
```

```
30          decl  aa
31          jmp   .L11

32    .L12: movl  ss, %eax
33          addl  b, %eax
34          xorl  a, %eax
35          movl  %eax, a
36          jmp   .L2
37    .L3:  leave
38          ret
```

The code has the following basic blocks:

Basic Block Type	**Instruction Extent**
fall	1-2
2-way	3-4
1-way	5
2-way	6-7
2-way	8-18
2-way	19-20
1-way	21
1-way	22-23
fall	24
2-way	25-26
1-way	27
1-way	28-31
1-way	32-36
ret	37-38

CONTROL FLOW GRAPH FOR EX. 6.1

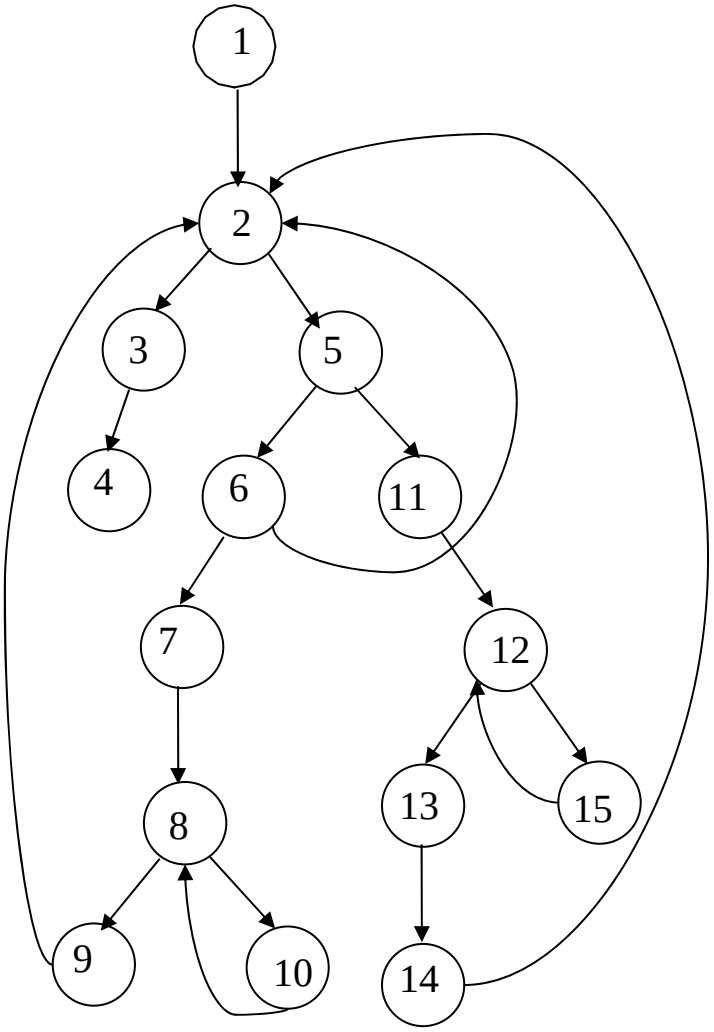

Fig 6.3: Control flow graph for example 6.1

6.1.4 Implementation

Control flow graphs have on average 2 out-edges per node, thus, a matrix representation (e.g incidence and adjacent matrices) is very sparse and memory inefficient (i.e., most of the matrix is zero).

Since the size of the graph is unknown (i.e., the number of nodes is not fixed), it is possible to construct the graph dynamically as a pointer to a basic block, which has list of predecessor and a list of successors attached to it. The predecessors and successors are pointers to basic blocks nodes as well; in this way a basic block is represented only once. This representation is plausible in any high-level language that allows dynamic allocation of memory. Consider the C++ definition of the basic block in the figure below. The basic block structure defines a basic block node,

```
class BasicBlock {

     typedef vector< string > InstContainer;
     InstContainer instructions;
     BasicBlock *taken;
     int takenNo;
     BasicBlock *ntaken;
     int ntakenNo;
     set< BasicBlock * > predecessors;
};
```

`instructions` is a `vector` of `string`s. Each `string` holds an instruction. The pointer `taken` is a pointer to a `BasicBlock` that is reached from this block if the condition represented by this block is satisfied, whereas the pointer `ntaken` is a `BasicBlock` that is reached from this block if the condition is not satisfied.

The construction of a control flow graph from the set of basic blocks is trivial and is not considered here.

6.2 DATA FLOW ANALYSIS

The assembly code that is input to the decompiler makes use of registers and condition codes. This code can be transformed into a higher-level representation that does not make use of such low-level concepts, and that regenerates the high-level concept of an expression. The transformation of low-level to high-level intermediate code is done by means of program transformations, traditionally referred to as optimizations. These transformations are applied to the assembly code, to transform it into the high-level code.

The types of transformations that are required by the data flow analysis phase include, the elimination of useless instructions, the elimination of condition codes, the determination of register arguments and function return register, the elimination of registers and intermediate instructions by the regeneration of expressions, the determination of actual parameters, and the propagation of data types across subroutine calls. Most of these transformations are used to reconstruct the information that is lost during the compilation process. In the case of elimination of useless instructions, this step is required even for optimizing compilers when there exist machine instructions that perform more than one function at a time.

Conventional data flow analysis is the process of collecting information about the way variables are used in a program, and summarizing it in the form of sets. This information is used by the decompiler to transform the assembly code. Several properties are required by code-improving transformations, including:

1)A transformation must preserve the meaning of programs.
2)A transformation must be worth the effort.

Techniques for decompilation are discussed in the following sections.

6.2.1 Previous Work

Not much work has been done in the area of data flow of analysis of a decompiler, mainly due to the limitations placed on many of the decompilers available in the literature. Data flow analysis is essential when decompiling pure binary files, as there is no extra information on the way data is used, and the type of it.

6.2.2 Condition Code propagation

Condition codes are flags used by the machine to signal the occurrence of a condition. In general, several machine instructions set these flags, ranging from 1 to 3 different flags being set by the one instruction, and fewer instructions make use of those flags, only using 1 or 2 flags. Consider the following from basic block 2 in Ex 6.1.

```
3          cmpl $0, a
4          jne 5
```

Instruction 3 defines the sign flag by comparing two operands, and instruction 4 uses this flag to determine whether the two operands of the previous instruction are equal or not. These two conditions are functionally equivalent to a high-level conditional jump instruction that checks if its two operands are equal or not. Thus whenever we see an assembly comparison instruction we store the left and the right operands of that instruction in the basic block that contained that instruction. Since every block has at most one comparison instruction, this works fine. Even if by some chance the block has two comparison instructions, the second instruction overwrites the flags set by the first one. After this, whenever a conditional jump instruction is seen store the corresponding opcode in its high-level representation in the basic block. In the above example, the left and right

operands are **0** and the variable **a**. The high-level representation of the opcode **jne** is **!=** and this stored in the block. Thus we eliminate instructions 3 and 4 and all future references to the condition codes.

6.2.3 Function Return Register

Subroutines that return a value are called functions. Functions usually return values in registers, and these registers are then used by the caller subroutine. Consider the following code from *basic block 6* from example 6.1.

```
12        addl a, %eax
13        pushl %eax
14        call function
15        addl $16, %esp
16        movl %eax, a
```

Instruction 14 invokes the subroutine function. After subroutine return, instruction 16 uses the register eax. Inside the subroutine eax will be defined to hold the return value from this subroutine.

6.2.4 Register Copy Propagation

An instruction is intermediate if it defines a register value that is used by a unique subsequent instruction. In machine language, intermediate instructions are used to move the contents of operands into registers, move the operands of an instruction into the registers that are used by a particular instruction, and to store the computed result in registers into a local or global variable. Consider the following instructions from example 5.1.

```
32   .L12:     movl ss, %eax
33             addl b, %eax
34             xorl a, %eax
35             movl %eax, a
```

Instruction 32 defines the register eax by copying the contents of the variable ss into it. This register is then used in the instruction 33 and the result is then placed in the same register. This is again used and defined in the instruction 34 and finally its contents are copied into the variable a. As seen, instruction 32 defines the temporary register eax to be used in instruction 33 and 34, and these again define eax to be used in instruction 35. These intermediate registers can be replaced with the local variables that were used to define them. Thus in instruction 33 the eax register is replaced with ss which defined eax in the previous instruction:

```
33         eax = b + ss
```

and instruction 32 is removed. In a similar way the resultant register from instruction 33 is replaced in instruction 34, leading to the following code:

```
34         eax = (b + ss) ^ a
```

Finally, a move to a memory location is made from eax, and hence we get the following code

```
35         a = (b + ss) ^ a
```

The final instruction 35 is a reconstruction of the original high-level expression.

6.2.5 Actual Parameters

Actual parameters to a subroutine call are placed on the stack before the subroutine is invoked. These arguments can be mapped against the formal argument list of the subroutine, and placed in the actual parameter list of the call instruction. Consider the following code from basic block 6 from example 6.1 after register copy propagation:

```
10   pushl aa
13   pushl a + aa
14   call function
```

Here the suffix 'l' in pushl indicates that a variable of size 4 bytes is being pushed onto the stack. Thus, the subroutine function takes two arguments: aa and a + aa. These identifiers can be replaced on the actual argument list of function at instruction 14, in reverse order due to the C calling convention(i.e., the last instruction pushed is the first one in the argument list). The modifications lead to the following code:

```
14   call function (a + aa, a)
```

and instructions 10 and 13 are eliminated.

6.2.6 Data Type Propagation Across Procedure Calls

The types of the actual parameters of a subroutine are to be the same as the types of the formal arguments. Consider the following call from basic block 6 from example 5.1

```
14   call function (a + aa, a)
```

We know from previous instructions that the types of a + aa and a are both LONG and hence the types of the corresponding formal arguments must also be LONG.

6.3 CONTROL FLOW ANALYSIS

The control flow graph that was constructed has no information on high-level language control structures, such as if..then..elses and while( ) loops. Such a graph can be converted into a high-level language graph by means of a structuring algorithm. High-level control structures are detected in the graph, and subgraphs of control structures are tagged in the graph. The relation of this phase with the data flow analysis and code generation phase is shown below:

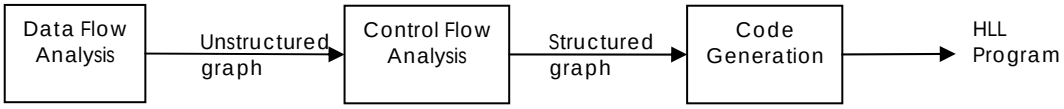

Fig 6.4: Context of the control flow analysis phase

A generic set of high-level control structures is used to structure the graph. Such structures should include different types of loops and conditionals. Since the underlying structure of the graph is not modified, functional and semantical equivalence is preserved by this method.

6.3.1 Graph Structuring

The structuring of a sample control flow graph is presented in an informal way. The following figure is a control flow graph of a sample program containing a few control structures. The data flow analysis phase has been completed and all variables have been given names.

The aim of a structuring algorithm for decompilation is to determine all underlying control structures of a control flow graph, based upon a predetermined set of high-level control structures.

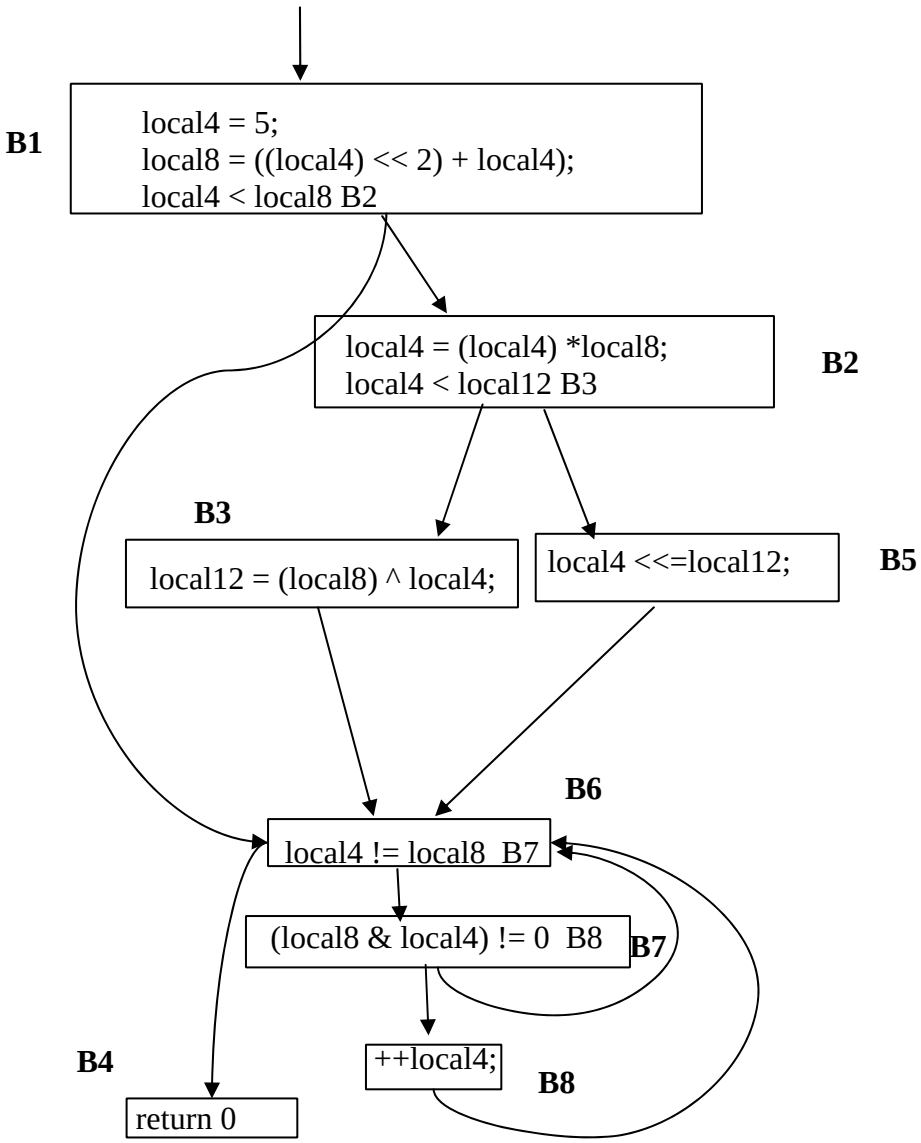

Fig 6.5: Graph Structuring

6.3.2 Structuring Loops

In graphs, loops are detected by the presence of a back-edge; that is, an edge from a "lower" node to a "higher" node. The notion of lower and higher can be thought of as the nodes that are lower and higher up in the diagram (for a graph that is drawn starting at the top). In the above figure there is only back-edge (B7 – B6). This back-edge represents a loop.

The type of the loop is detected by checking the header and the last node of the loop. The loop (B7 - B6) has a conditional check in its header node; thus it is a pre-tested loop such as `for( )` and `while( )`. The subgraph that represents this loop can be logically transformed into the subgraph shown below:

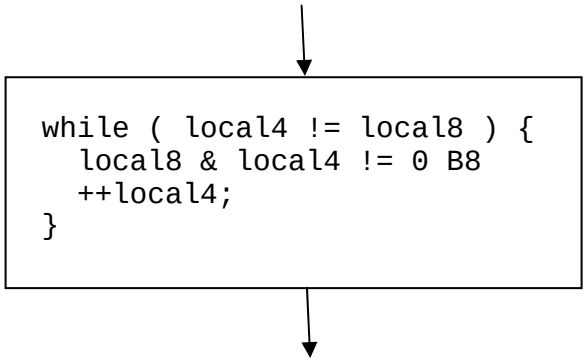

Fig 6.6: Pre-Tested Loop

The loop sub graph was replaced by one node that holds all the intermediate code instructions, as well as information on the type of the loop. Post-tested loops can be deduced in a similar manner. The difference is that in a post-tested loop condition checking takes place in the node with a back-edge to the header node.

6.3.3 Structuring Conditionals

The 2-way conditional node B2 branches control to node B5 if the condition local4 < local12 is true; otherwise it branches to node B3. Both these nodes are followed by node B6, in other words, the conditional branch that started at node B2 is finished at node B6. This graph is clearly an if...then...else structure, and can by logically transformed into the sub graph shown in the below figure, where the node represents basic blocks B2, B3 and B5. Note that all instructions before the conditional jump that belong to the same basic block are not modified.

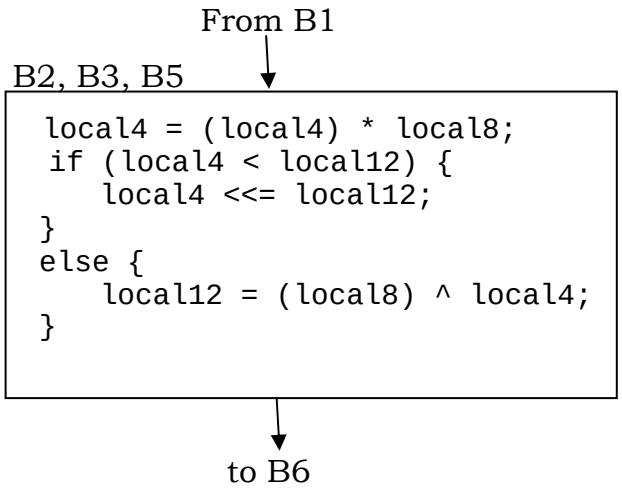

Fig 6.7: structuring if...then...else

The 2-way conditional node B1 transfers control to node B6 if the condition local4 < local8 is false, otherwise it transfers control to node B2. From our previous example, node B2 has been merged with nodes B3 and B5, and transformed into an equivalent node with an out-edge to node B6; thus there is path from node B2 -> B6. Since B6 is one of the target branch nodes of the conditional at node B1, and it is reached by the other branch of the conditional, this 2-way node represents a single branch conditional

(i.e., an `if…then`). This sub graph can be transformed into the node of the below figure.

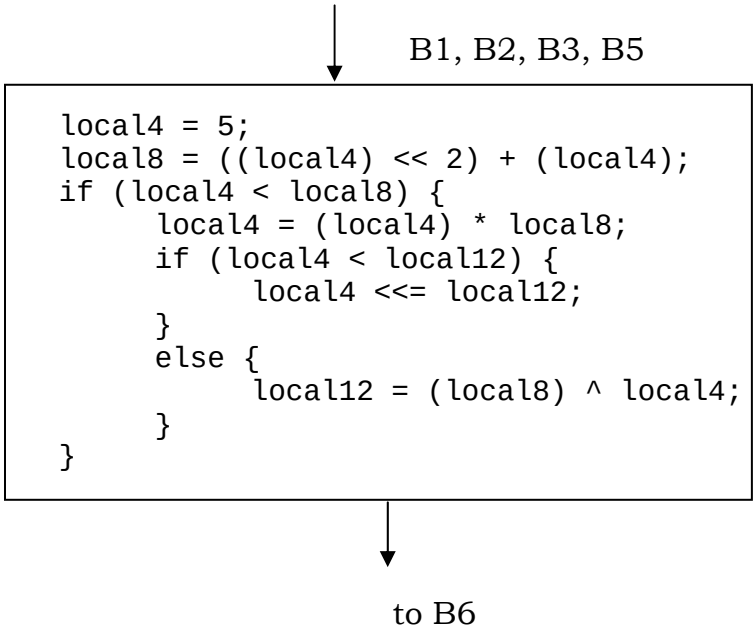

Fig 6.8: structuring if ( )

6.3.4 Interval Theory

An interval is a graph theoretic construct defined by J.Cocke in [COC70], and widely used by F.Allen for control flow analysis and data flow analysis. The following sections summarize interval theory concepts.

Definition:
Given a flow graph G with initial node n_0, and a node n of G, the interval with header n, denoted I(n), is defined as follows.
1) n is in I(n).
2) If all the predecessors of some node m != n_0 are I(n), then m is in I(n).
3) Nothing else is in I(n).

We therefore may build I(n) by starting with n, and adding nodes m by rule (2). It does not matter in which order we add two candidates m because once a node's predecessors are all in I(n), they remain in I(n), and each candidate will eventually be added by rule (2). Eventually, no more nodes can be added to I(n), and the resulting set of nodes is the interval with header n.
By selecting the correct set of header nodes, the graph G can be partitioned into a unique set of disjoint intervals I = { $I(h_1)$, $I(h_2)$, $I(h_3)$,...$I(h_n)$ }, for some n >= 1.

6.3.5 Intervals Partitions

Given a flow graph G, we can partition G into disjoint intervals as follows. For any node n, we compute I(n) by the method sketched below:

I(n) = { n }

while there exists a node m != n0, all of whose predecessors are in I(n) **do**
    I(n) = I(n) U { m }

The particular nodes that are headers of intervals in the partition are chosen as follows. Initially, no nodes are “selected”,

construct $I(n_0)$ and “select” all nodes in that interval;
    while there is a node m,
        not yet “selected”,
            but with a selected predecessor **do**
    construct I(m) and “select” all nodes in that interval

Once a candidate *m* has a predecessor *p* selected, *m* can never be added to some interval not containing *p*. Thus, candidate *m*'s remain candidates

until they are selected to head their own interval. Thus, the order in which interval headers m are picked in the above algorithm does not affect the final partition into intervals. Also, as long as all nodes are reachable from n_0, it can be shown by induction on the length of a path from n_0 to n that node n will eventually either be put in some other node's interval, or will become a header of its own interval, but not both. Thus, the set of intervals constructed in the above algorithm truly partition G.

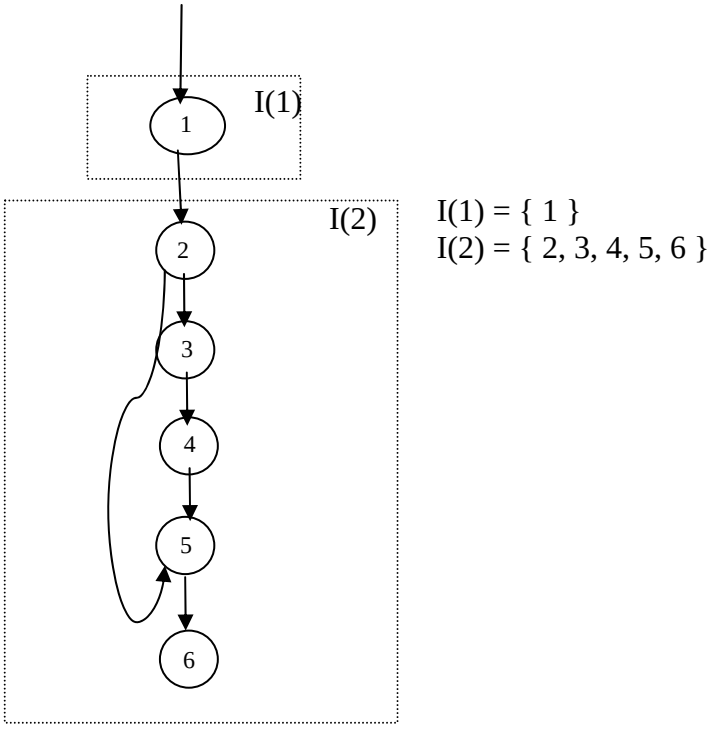

Fig 6.9: Intervals of a graph

6.3.6 Interval graphs (Derived Sequence of Graphs)

From the intervals of one flow graph G, we can construct a new flow graph I(G) by the following rules.

1.The nodes of I(G) correspond to the intervals in the interval partition of G.

2.The initial node of I(G) is the interval of G that contains the initial node of G.

3.There is an edge from interval *I* to a different interval *J* if and only if in G there is an edge from some node in *I* to the header of *J*. Note that there could not be and edge entering some node *n* of *J* other than the header, from outside *J*, because then there would be no way *n* could have been added to *J* in the interval partitioning algorithm.

We may apply the interval partitioning algorithm described above and the interval graph construction alternately, producing the sequence of graphs G, I(G), I(I(G)),… Eventually, we shall come to a graph each of whose nodes is an interval all by itself. This flow graph is called the *limit flow graph* of G.

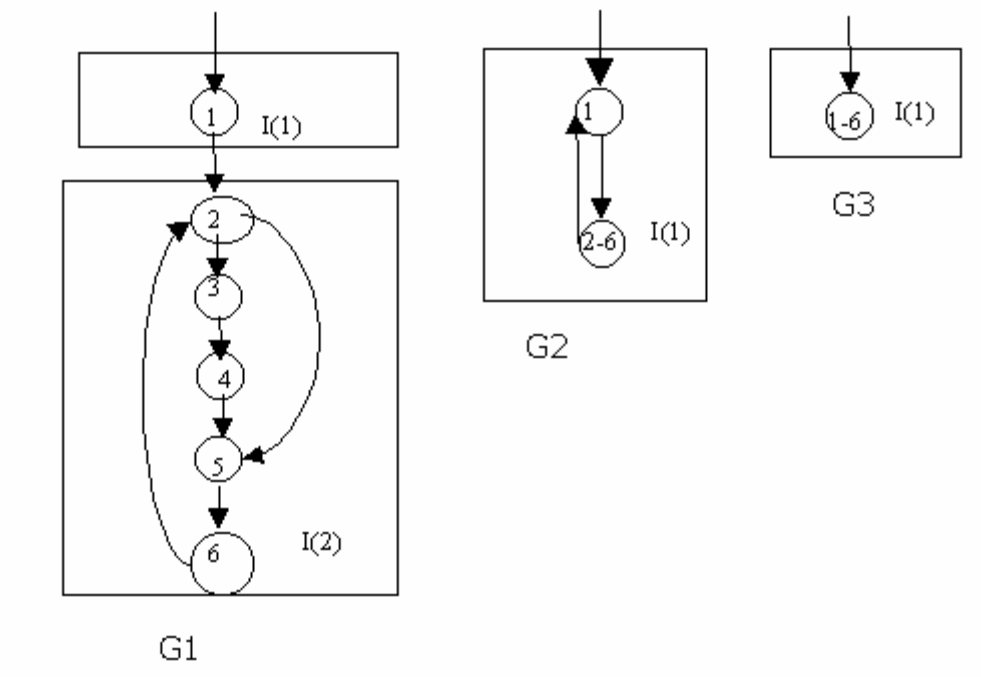

Fig 6.10: Reducing intervals; process of Control Flow Analysis

6.3.7 High-Level Language Control Structures

The different control structures that relipmoC handles are :

1. *Action:* a single basic block node in an action.
2. *Composition*: a sequence of two structures in a composition.
3. *Conditional*: a structure of the form **`if`** *p* **`then`** *s1* **`else`** *s2*, where *p* is a predicate and *s1*, *s2* are structures is a conditional structure.
4. *Pre-tested loop*: a loop of the form **`while`** *p* **`do`** *s*, where *p* is a predicate and *s* is a structure.
5. *Single branch conditional*: a conditional of the form **`if`** *p* **`then`** *s*, where *p* is a predicate and *s* is a structure.
6. *Post-tested loop:* a loop of the form **`repeat`** *s* **`until`** *p*, where *s* is a structure and *p* is a predicate.
7. *Endless loop*: a loop of the form **`loop`** *s* **`end`**, where *s* is a structure

6.3.7.1 Loops

A Loop is a collection of nodes in a flow graph such that

1. *All nodes in the collection are strongly connected, that is, from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and*
2. *The collection of nodes has a unique entry, that is, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.*

A loop that contains no other loops is called an *inner loop*.

Every interval has at most one loop.

The follow node of a structured or unstructured loop is the first node that is reached from the exit of the loop. In the case of unstructured loops, one

node is considered the loop exit node, and the first node that follows it is the follow node of the loop.

6.3.7.2 Conditionals

A 2-way conditional is a directed sub-graph with a 2-way conditional header node, one entry point, two or more branch nodes, and a common end node that is reached by both branch nodes. This final common end node is referred to as the follow node, and has the property of being immediately dominated by the header node.

In an if..then conditional, one of the two branch nodes of the header node is the follow node of the sub-graph. In an if..then..else conditional, neither branch node is the follow node, but they both converge to a common end node. The figure below shows these two control structures, with the values of the out-edges of the header node; true or false.

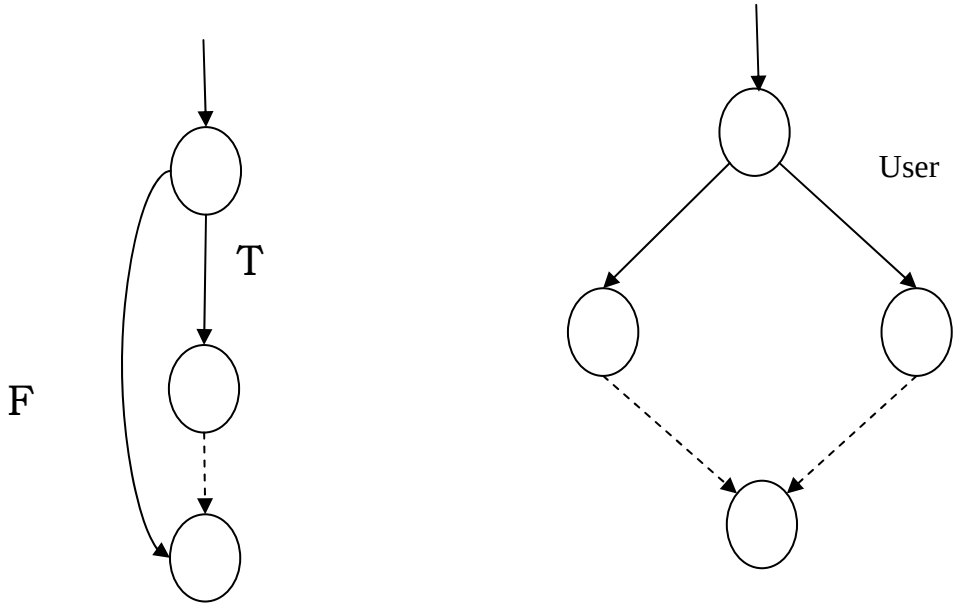

Fig 6.11: if and if...else conditionals

6.3.8 Structuring Algorithms

In decompilation, the aim of a structuring algorithm is to determine the underlying control structures of an arbitrary graph, thus converting it into a functional and semantically equivalent graph.

6.3.8.1 Structuring Loops

In order to structure loops, a loop in terms of a graph representation needs to be defined. This representation must be able to not only determine the extent of a loop, but also provide a nesting order for the loops. The use of intervals provides a representation that satisfies the abovementioned conditions: one loop per interval and a nesting order provided by the derived sequence of graphs.
Given an interval I(h) with header h, there is a loop rooted at h, if there is a back-edge to the header node h from latching node n ∈ I(h). Consider the below control flow graph which is the graph (above no) without intermediate information, and with intervals delimitated by dotted lines.

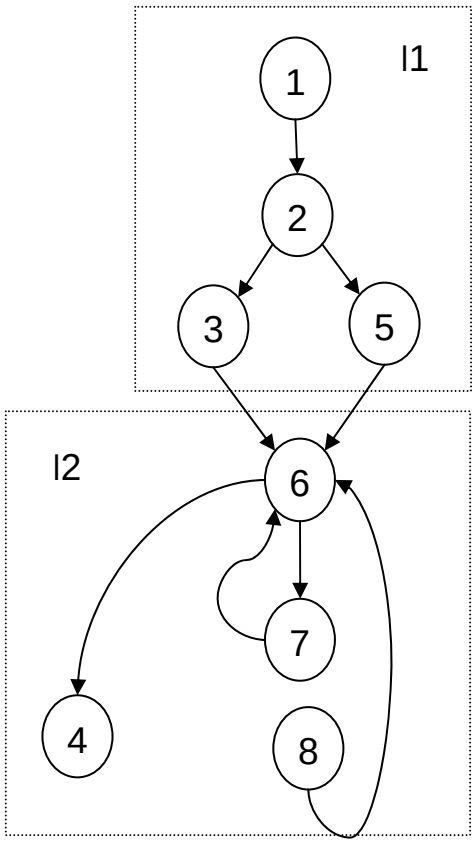

Fig 6.12: Structuring Loops

There are 2 intervals: I1 rooted at basic block B1 and I2 rooted at basic block B6.

In this graph I2 contains the loop (B6, B7, B8) in its entirety.

Consider the derived sequence of graphs

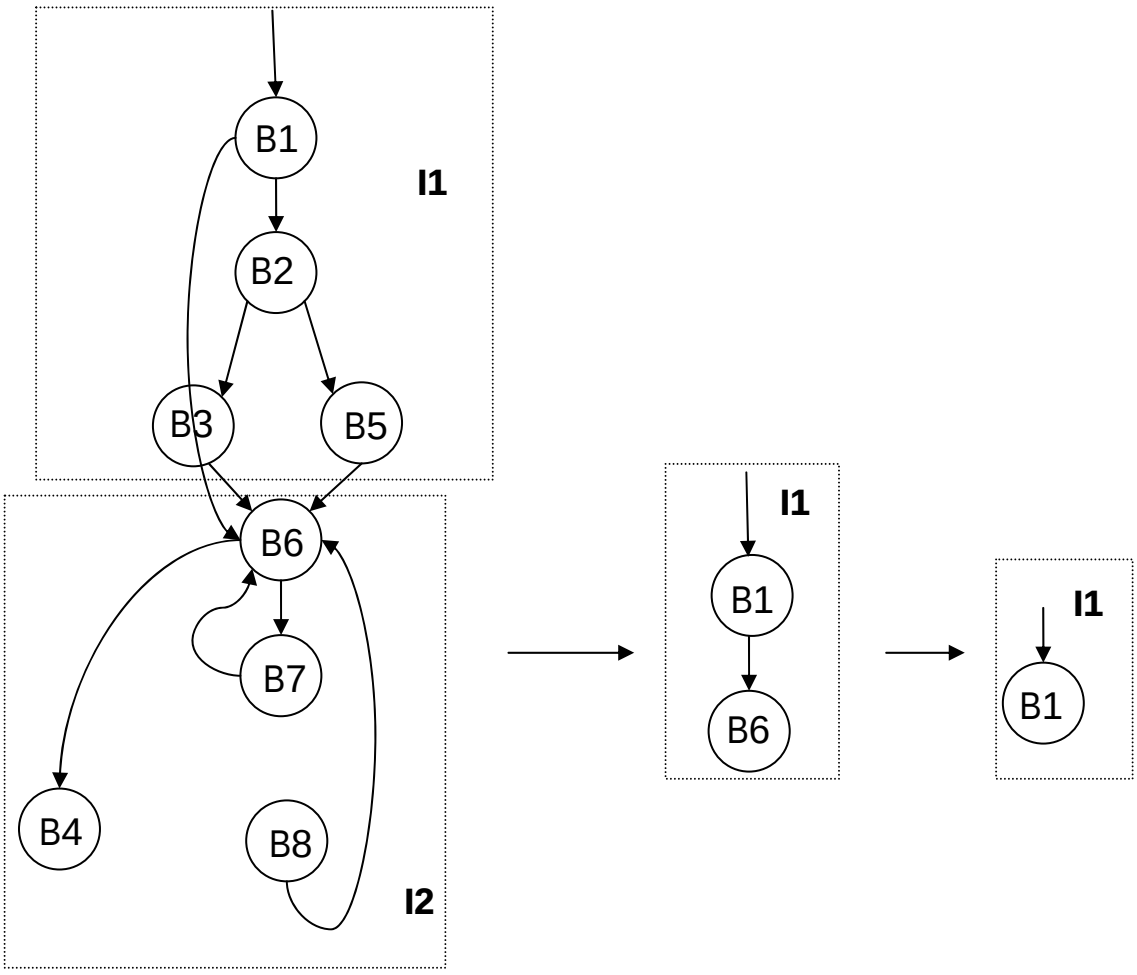

Fig 6.13: Reduction of graph into a single node

It is noted that there are no more loops in the initial graph. It is noted that the length of the derived sequence is proportional to the maximum depth of nested loops in the initial graph.

Once a loop has been found the type of the loop is determined according to the type of the header and the latching nodes. Also, the nodes that belong to this loop are marked. These methods are explained in the following sections, for now we assume that there is a procedure that determines the type of the loop, and mark the nodes that belong to that loop.

Given a control flow graph *G = G1* with interval information, the derived sequence of graphs *G1, G2, G3, .., Gn of G*, and the set of intervals of these graphs, I1, I2, I3,..,In, an algorithm to find loops is as follows: each header of an interval in G1 is checked for having a back-edge from a latching node that belong to the same interval. If this happens, a loop has been found, so its type is determined, and the nodes that belong to it are marked. Next, the intervals of G2, I2 are checked for loops, and the process is repeated until intervals in In have been checked. In this algorithm no `goto` jumps and target labels are determined. This algorithm is described below:

6.3.8.2 Algorithm For Reduction Of A Graph Into A Single Node

for each of the derived sequence of graphs G_1,G_2, .. G_n
    for each interval I(i) that belongs to this graph,
    check if the header i of this interval has a back-edge from a node n belongs to I(i)
        if such an edge is found,
            find the nodes that belong to this loop
            find the type of this loop

6.3.8.3 Finding The Nodes That Belong To A Loop

A good way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a -> b is an edge, b is the head and a is the tail.)
Given a back-edge n -> d, we define the loop of the edge to be d plus the set of nodes that can reach n without going through d. Note d is the header of the loop.

Constructing a loop of a back-edge:

Given a flow graph of G and a back edge n -> d.
Beginning with node n, we consider each node m != d that we know is in loop, to make sure that m's predecessors are also placed in loop. The algorithm is given below. Each node in loop, except for d, is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessor, and thus find only those nodes that reach n without going through d.

```
procedure insert( m ) begin
        if m is not in loop then
        begin
                loop = loop U { m }
                mark m
                push m onto stack
        end
end

stack = empty
loop = { d }
insert( n )
while stack is not empty do begin
        pop m, the first element of stack, off stack;
        for each predecessor p of m do
                insert( p )
end
```

At the end of the algorithm all the nodes that belong to a loop have been found and marked.

6.3.8.4 Determining The Type Of The Loop

The type of a loop is determined by header and latching node of the loop.
In a pre-tested loop, the 2-way header determines whether the loop will be entered for the next iteration or not and the latching node transfers control back to the header node.
A post-tested loop is characterized by a 2-way latching node that branches back to the header node of the loop or out of the loop, and a 1-way header node. Finally, an endless loop has a 1-way latching node that transfers control back to the header node, and any type of header node.
The type of the loop in the above example is as follows: the loop B6,B7,B8 has a 2-way header and a latching node. But since for a pos-tested loop it is required to have a 1-way header node, this loop must be pre-tested loop.
The algorithm is given below:

Given an interval I(x) with a loop L determined by a set of latching nodes
x = {x1, x2, ... }

```
if no_latching_nodes >= 2 then
        type = PRE_TESTED
else
   if nodeType( x1 ) == 2-way then begin
        if nodeType( y ) == 2-way then begin
                if both children of y belong to L then
                        type = POST_TESTED
                else
                        type = PRE_TESTED
        else
                type = POS_TESTED
   else if nodeType( y ) == 2-way
        type = POST_TESTED
```

else

type = ENDLESS

6.3.9 Structuring 2-Way Conditionals

Both a single branch conditional (i.e., `if…then`) and a conditional (i.e., `if…then…else`) subgraph have a common end node, from here onwards referred to as the follow-node, which has the property of being immediately dominated by the 2-way header node. Whenever these subgraphs are nested, they can have different follow nodes or share the same common follow node.
Consider the graph in Fig 5.12.

This graph has 4 2-way nodes namely B1, B2, B6 and B7. As seen during loop structuring a 2-way node that belongs to either the header or the latching node of a loop is marked as being so, and must not be processed during 2-way conditional structuring given that it already belongs to another structure. Hence the node B6 is not considered in this analysis. Whenever two or more conditional are nested, it is always desirable to analyse the most nested conditional first, and then the outer ones. In this case the among conditional B1 and B2, B2 must be analysed first since it is nested in the sub graph headed by B1; in other words the node that has greater reverse post order numbering needs to be analysed first since it was visited first in the search traversal. When this sub graph is analysed it is found that the two children of B2 share the same follow node, B6 and hence B2 must be a `if…then…else` conditional. After this the nodes B2, B3 and B5 are merged to form a single node B2. Now in this reduced graph the node B1 has two children and of the two children leads to the other. Thus the node B1 must be an `if…then` conditional. After this the nodes B1 and

B2 are merged to form a single node B1. In a similar manner the `if…then` conditional node can be reduced.

The idea of the algorithm is to determine which nodes are header nodes of conditionals, and which nodes are the follow of such conditionals. The type of the conditional can be determined after finding the follow node by checking whether one of the branches of the header node is equivalent to the follow node. Inner conditionals are traversed first, then outer ones, so a descending reverse postorder traversal is performed (i.e., from greater to smaller node number).

6.3.9.1 Algorithm for structuring 2 way conditionals

Given an interval I(x) the below algorithm finds all the conditionals of this interval.

```
for all nodes m in descending order begin
        if nodeType( m ) == 2-way and m is not a latching  node or a header node then
                if both the children of m belong to I(x)
                        if one of the children of m leads to the          other then
                                type = IF_THEN
                        else if both the children of m lead to the same node then
                                type = IF_THEN_ELSE
end
```

6.4 CODE GENERATION

After data flow analysis, the instructions in a basic block are all high-level instructions.

We “include” a header file “ourHeader.h” that contains all the C header files that could have been included in the original C program.

After this the symbol table is interpreted and the declarations for all the global variables in the program are generated. Then the function table is interpreted and the function prototypes for all the functions defined in the program is generated.

Then for each function in the function in function table, that definition of that function is generated. Here we first have to generate the function's parameter declaration list and the return type of the function.

Then the declarations for the local variables used in the program are generated by interpreting the activation record table and finally the body of the function is generated by interpreting the block table associated with this function.

6.4.1 FINAL CODE FOR THE EXAMPLE 6.1 (Generated By relipmoC)

```
#include<ourHeader.h>

long aa = 262;
long b = 262;
long ss;

int function(long par_8, long par_12);
int main(void);

int main(void)
{
    long local_4;

    local_4 = 262;
    while (local_4 != 0) {
     if (local_4 != 0) {
         ++local_4;
         local_4 = function((aa) + local_4, aa);
         if (aa != -1) {
          while (ss <= 0) {
              aa += 1000;
          }
         }
     }
     else {
         aa = 34;
         while (aa != 12) {
          ss = aa;
          --aa;
         }
         aa = ((ss) + b) ^ local_4;
     }
    }

    return 0;
}

int function(long par_8, long par_12)
{
    while (par_8 >= par_12) {
     par_8 ^= par_12;
    }
    return par_8;
}
```

6.4.2 Original C File

```
long aa = 262;
long b = 262;
long ss;

int function( int a, int b );

int main( void )
{
long a = 262;
    while ( a != 0 )
     if ( a != 0 ) {
         ++a;
         a = function( a + aa, aa );
         if ( aa != -1 )
          while ( ss <= 0 )
              aa += 1000;
     }
     else {
         aa = 34;
         while ( aa != 12 ) {
          ss = aa;
          --aa;
         }
         aa = b + ss ^ a;
     }
     return 0;
}

int function( int a, int b )
{
     while ( a >= b )
          a = a ^ b;
     return a;
}
```

6.4.3 Equivalence Of Both Codes

Analyzing the codes in Section 5.4.1 and 5.4.2, we can say that both these codes are equivalent in terms of output. Hence, the correctness of design of relipmoC can be justified.

8 TESTING AND EXPERIMENTAL RESULTS

relipmoC was extensively tested for defects. A detailed description of all the procedures followed for this, the test cases and results is given below.

8.1 Black Box Testing

In this method of testing, the code is considered to be a black box, which takes a set of inputs and generates outputs. The inputs provided need to be carefully picked and sampled, as a subset of all possible inputs. The code is then tested for bugs or erroneous outputs. An equivalence partitioning is used to determine the sample set of inputs. These test cases are discussed here:

Input to relipmoC: 80386 code obtained by compiling a C program by gcc 3.2.2 compiler.

Test cases:

1. Arithmetic, Logical, Relational and Bitwise operators:
   - Test case: A program that consists of all these operators, in different combinations.
   - Result: relipmoC successfully generated output C code for a program consisting of all such operators.

2. Conditionals .. If..else and switch statements:
   - Test case: A program that consists of multiple and compounded if..else statements.
   - Result: relipmoC generated back the equivalent C program, without bugs for if...else and if statements, but failed to recognize

switch statements, which is not a bug but rather an implementational limitation. Coding has not been done in v1.0 to recognize switch statement because of the enormous complexities involved.

3. Loops:
   - Test Case: Code consisting of for, while and do – while loops that are nested and intermixed.
   - Result: Decompilation successful. Both while and for loops are reduced to equivalent while loop because in assembly language, there is no way of distinguishing one from the other.

4. Local Variables:
   - Concept: In assembly language, local variables are stored on stack and there isn't a way to know their actual names. As a result of which, it is mandatory to generate variable names for local variables.
   - Test Case: A program consisting of local variables of different types, both initialized and uninitialized.
   - Result: Works fine. relipmoC generates the local variables and uses them appropriately. But if the declared local variable is never used, then we cannot infer its type and our program does not declare any variable for the same, which is perfectly sensible.

5. Floating point numbers:
   - Concept: Floating point numbers are stored in32 bit register in the IEEE format, which is different from the normal and hence requires further processing.
   - Test Case: Program involving floating point numbers and various operations involving such variables.

- Result: OK. The floating point numbers used are generated, format recognized and floating point instructions from ASM language executed correctly in C.

6. Function Calls:
   - Concept: The compiler processes function calls like this – First, all the arguments are pushed on to the stack and then the function is called. Naturally, arguments passed to a function, variables used in it are all local and they need to be generated. This needs to be taken care of.
   - Test Case: Input program consists of multiple program calls, with different types of arguments and return types.
   - Result: Produced predicted output, which is an equivalent C program, with appropriate arguments, return types and variable generation.

7. Recursion:
   - Test Case: Program generating a Fibonacci sequence, or giving the nth term of a Fibonacci sequence is a classic example of testing for recursion, as it involves double recursion.
   - Result: The program generated by relipmoC produced the same Fibonacci sequence and generated the same number, for the given input; as the source program. This means that an equivalent C program is generated, and hence relipmoC supports Recursion, without any bugs.

8.2 Structural Testing

In this phase of testing, we test each module for the functionality supported by it, whether it is defective or not.
At every phase, relipmoC was tested completely for defects. Bugs were discovered and rectified. And after each stage of development, a combination of inputs discussed in the previous section was given; output observed. In the process, certain defects were found and fixed appropriately.

With the result of these two techniques, we say that relipmoC is tested OK, and a reliable software system.

9 CONCLUSION

Decompilation is used in two main areas of computer science – software maintenance and security. A decompiler is used in software maintenance to recover lost or inaccessible source code, translate code written in an obsolete language into a newer language, structure old code written in an unstructured way (i.e. spaghetti code), migrate applications to a new hardware platform and debug binary programs that are known to have a bug. In security, a decompiler is used to verify binary programs and the correctness of the code produced by a compiler for safety - critical systems, where the compiler is not trusted to generate correct code and to check for the existence of malicious code such as viruses.

Making a decompiler for assembly languages provided us a rich experience of the underlined principles of reverse engineering, Compiler Design and Optimization theory. It also provided us a feel of actually what object oriented programming is, and how it is to be implemented. Also importantly, we learnt how to manage a software project – i.e. We executed all the phases of the project systematically, starting from Literature Survey, Feasibility study, Requirement Analysis, Design, Implementation, Testing, Debugging and the Documentation of all the phases. This has provided us with an immense experience in the field of software engineering, which is definitely going to benefit us in our future professional life.

However, before concluding we would like to enlist certain Legal and Ethical issues involved with decompilation. A decompiler writer needs to be well aware of these issues. But, since relipmoC is targeted to work for Linux, which is an Open Source system and source codes of all software are available here, we don't concern ourselves too much with these legal issues. Some of these are discussed here.

9.1 Legal and Ethical Issues of Decompilation:

Decompilers aren't necessarily evil - but they do pose an ethical dilemma for many software developers. Decompilers offer great potential for legitimate purposes, but can also be used to steal the source code of proprietary software or competitors; or by hackers to determine weaknesses in the design of software. But just don't blame the decompiler - its the programmer who uses it for intellectual property theft, or the hacker that decompiles the software to find security holes that is at fault!

9.2 Future Enhancements:

Further work on relipmoC can be done in three areas: the separation of code and data, the determination of data types such as arrays, structures and pointers and generalizing relipmoC to take input from objdump or any other generic disassembler. The first area needs a robust method of determining n-way branch statements (i.e. indexed jumps) and indirect subroutine calls. The second area needs heuristic methods to identify different types of compound data types and propagate their values. Efficient implementation of the algorithms would provide a faster decompiler, although the speed of decompilation is not a concern given that a program is normally decompiled once only.

10 BIBLIOGRAPHY

[AHO86] Aho, A.V., Ravi Sethi and J.D. Ullman; "*Compilers - Principles, Techniques and Tools*", Pearson Education, 1997.

[CIFUENTES93] C.Cifuentes and K.J.Gough, "*A Methodology for Decompilation*", Proceedings of the XIX Conferencia Latinoamericana de Informatica, pp 257-266, Buenos Aires, August 1993.

[CIFUENTES94] C.Cifuentes, "*Reverse Compilation Techniques*", Ph.D. thesis, Faculty of Info Technology, Queensland University of Technology, July 1994.

[SOMMERVILLE96] Sommerville, Ian, *"Software Engineering"*, Pearson Education, 5th edition, 1996.

[BREY96] Barry B Brey, "*The Intel Microprocessors*".

[KERNIGHAN88] Kernighan B.W. and D.M.Ritchie, "*The C programming language*", Prentice Hall Inc., 2nd edition,1998.

[COC70] J Cocke, *"Global Common Subexpression Elimination", SIGPLAN* Notices 5(7): 20 – 25, July 1970.

Appendix A

ACCOMPLISHMENTS

relipmoC was awarded the 1st place at the annual technical festival of IIT MADRAS – SHAASTRA 04.

relipmoC: Project selected and presented at IEEE Aavishkar 2004, MSR Institute of Technology, Bangalore.

Our accomplishments with the corresponding paper, *Reverse Engineering: Building an Assembly to C Code Translator,* so far, as on 7th February 2005 are:

IBM Vidyut '04, Dept of E and E, RVCE, Bangalore. April 2004 – 1st place
int 04h – Dept of CSE, MSRIT, Bangalore. April 2004 - 1st place
Utsav 04 – BMS College of Engg., Bangalore. May 2004 – 1st place
Connect '04 – Dept. of Telecommunications, RVCE, Bangalore. May 2004 – 1st place
SHAASTRA 04, IIT MADRAS, October 2004 – 5th place

This paper has also been presented at the 39th Annual National Convention of the Computer Society of India (CSI) held at Bombay in December 2004. It has been published in the Proceedings of the Convention and in the January issue of CSI Communications.

Appendix B

PAPER

Reverse Engineering: Building an Assembly to C code Translator

Arjun S J and Darshan Desai
R V College of Engineering, Bangalore, India

Abstract - The process of translation of assembly language to high-level language (HLL) is part of a larger process, which converts machine code to HLL code, performed by a *decompiler*. We can write a decompiler by first converting the machine code to assembly code and then by making use of the assembly to C (asm2C) translator to get the HLL code. *In this paper, our original work of design and implementation of an i386 assembly to C code translator on Linux platform, called 'relipmoC' is described.*

Index Terms – Decompiler, Reverse Engineering, asm2C Translator

I. INTRODUCTION

A *translator* is a program that takes as input a program written in one programming language (source program) and produces as output a program in another language, which is the object or target language. If the source language is a high level language like C or PASCAL and the object language is a low-level language, then such a translator is called a *compiler* (ref. 1). A *decompiler* can be thought of as a program that converts an object program to a high-level language (HLL) program. *Reverse engineering* attempts to produce source code from object code by the use of disassemblers, decompilers, debuggers and other tools.

Consider that the processor, by executing the machine language program, can successfully figure out what to do with any input, to create the correct output. An assembly code program is a set of instructions, plus a set of data bytes. Each assembly code instruction has semantics - you can define precisely what each instruction does. A HLL is more abstract than an assembly code program – details such as individual instructions, registers, and so on are abstracted away. In a way, asm2C translation is the ***judicious deleting of unneeded information***. Machine code features such as individual registers disappear with these transformations. Finally, the intermediate representation can be structured, replacing jump instructions with loops and conditionals.

Our paper deals with building an *ASM to C code translator (asm2C),* for Linux platform, which can be thought of as a Decompiler for assembly codes.

A. Significance of Decompilation:

Translation from assembly code to high-level language (HLL) code is of importance in the maintenance of legacy code, as well as in the areas of program understanding, porting and recovery of code. It is very useful in making assembly codes platform independent. In the context of fixing the year 2000 bug, the Gartner Group estimated that many organizations are missing 3% to 5% of their source code portfolios. This means that a medium-sized information systems organization with a software portfolio of 30 to 50 millions lines of code could easily be missing a million lines or more. Further, some large organizations have thousands of lines of code written in assembly code and cannot benefit from state of the art object-oriented techniques unless the assembly code is reengineered into some HLL. When an old application needs to be upgraded or modified, and the original source code isn't available, reverse engineering through decompilation can be very useful. Also asm2C translators could be used to "crack" virus codes.

B. Previous Work:

The first decompilers were used as tools in the translation of software from second to third generation machines in the 1960s. Maurice Halstead at the Navy Electronic Labs developed the very first decompiler. This decompiler took machine code for IBM 7094 as input and produced Neliac code for Univac 1108. In the 1970s and 1980s, decompilers were used to port programs, recreate lost source code, modify and debug binaries. In 1990s, decompilation has become part of a wider area, called *reverse engineering*. A detailed study of Decompilation techniques is done by Cristina Cifuentes (ref. 2).

On the Linux platform, a disassembler (objdump) is available, but no fully functional or standard ASM to C translators are available. Here, we speak about design and implementation of such a decompiler for Linux.

II. METHODOLOGY USED

This section presents the methodologies used for reverse compilation of ASM programs. At this moment, we are not concerned with any particular machine language, all we are given is an assembly language program and we need to produce the equivalent C language program. Here, equivalency is in terms of the output generated and not the source code. The process of ASM to C code conversion can be divided into three main phases viz. Control Flow Graph (CFG) Generation, Data flow and Type analysis and Control flow analysis (see Fig 1).

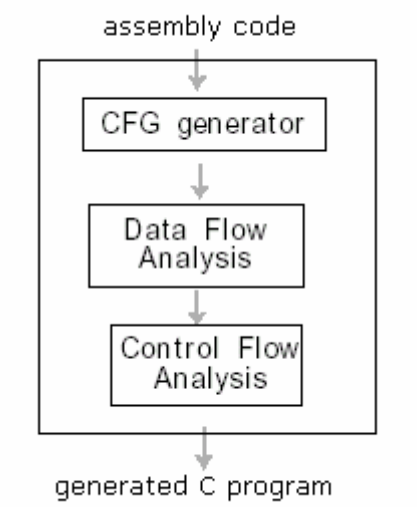

Fig.1 Structure of asm2C translator

Data flow analysis is to recover HLL expressions and statements (other than control-transfer statements), high level operators, actual parameters in a function call, function return values, and to remove hardware references from the code, such as registers, condition codes and stack references;
Control flow analysis is to recover control flow information, such as loops and conditional statements, as well as their nesting level; and
Type analysis is to recover high-level type information for variables – global and local, formal and actual parameter types, and return types of functions.

A. Control Flow Graph Generation:

A basic block is a sequence of instructions that have no jumps in except at the beginning and no jumps out except at the end (ref. 1). A *control flow graph (cfg)* is a connected, directed graph with nodes representing basic blocks and directed arcs representing flow of control from one node to another. The cfg generation phase generates a cfg for each function in the program.

Each basic block needs to record information such as its predecessors, successors, and associated assembly code. The type of a basic block is determined by the final assembly instruction that changes the flow of control, such as unconditional or conditional branch, and end of program. When any of these instructions is met, the end of a basic block is reached as well.

A *labelled basic block* is a basic block whose entry point is the target of a branch. All the basic blocks of a function following a `jump` are of these kinds. A function can contain forward jumps[2] and hence create a labelled basic block, two passes are needed to generate the graph; one to create a list of basic blocks, and the next to transform this list into a graph of basic blocks. The cfg for the sample code shown in fig 2 is shown in fig 3.

[2] Jump to a block that has not been encountered yet

```
pushl   %ebp
movl    %esp, %ebp
subl    $8, %esp
andl    $-16, %esp
movl    $0, %eax
subl    %eax, %esp
subl    $8, %esp
pushl   $a
pushl   $.LC0
call    scanf
addl    $16, %esp
movl    %eax, -4(%ebp)
cmpl    $0, a
je      .L2
subl    $12, %esp
pushl   $.LC1
call    printf
addl    $16, %esp
movl    %eax, -4(%ebp)
jmp     .L3
L2:
movl    a, %eax
imull   -4(%ebp), %eax
movl    %eax, a
L3:
movl    $0, %eax
leave
ret
```

Fig.2 A sample *i386* assembly code generated by gcc 3.2.2 compiler.

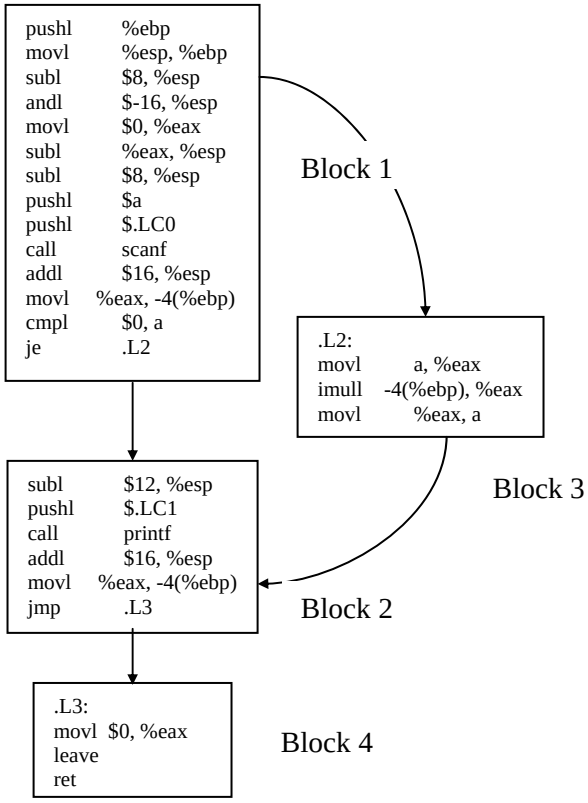

Fig.3 The output after control flow graph generation

B. Data Flow Analysis:

The data flow analysis phase makes use of compiler optimization theory to analyse the data and determine its type, temporary variables used for intermediate operations, extracting meaning from condition codes and associating them with their corresponding *"jump"* statements, arguments to functions, values returned by functions and to remove other hardware references from the code, such as stack references. Basically, **data flow analysis is the process of extracting the meaning of assembly language statements and translating that into HLL.** However, this phase does not deal with analyzing the *structure of control flow graph.*

During compilation, a high-level statement would have been translated into a sequence of assembly instructions. Registers are normally used as carriers of intermediate values to calculate a new result or transfer it to a new memory location. So one of the issues of data flow analysis is to recognize such sequences of assembly instructions and translate them into their corresponding high-level statements. This is done by propagating the register contents until an assignment to a memory location is made; after which we know that a high-level statement has been recovered.

Extraction of function parameters and return types and values is discussed in the implementation section. See fig 4 for the output of Data flow analysis for the cfg shown in fig 3.

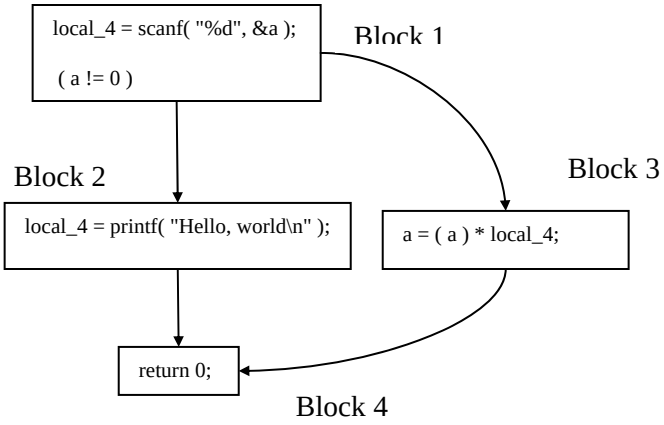

Fig.4 After data flow analysis

C. Type Analysis:

In this phase, we determine the data types of all the variables - local and global (used in the original source code, if any), types of the function arguments and reconstruct function prototypes. Ex: If a floating-point instruction is used, the operands should be of floating point type. The corresponding information is saved.

D. Control Flow Analysis:

Transfers of control are modelled in assembly code via conditional jumps and unconditional jumps (based on the value of condition codes). Forward and backward jumps denote traditional 2-way conditionals and loops respectively. The process of Control Flow analysis involves the determination of conditionals and loops, followed by restructuring them according to their nesting level.
Control flow analysis consists mainly of "*Interval analysis*". An *interval* is defined to be a set of basic blocks that contains at most one loop. The *entry node* of the interval is called the header node. The *follow node* is the node that contains the first instruction to be executed after the equivalent HLL control structure has been executed. A back edge is an edge from a higher numbered node to a lower numbered node. A *latching node* is the one with the *back edge* to the header node. Every latching node determines a loop. These are shown in Fig 5.

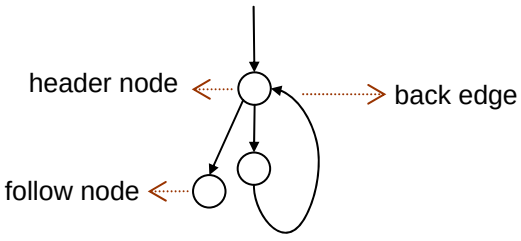

Fig.5 Description of the nodes and edges

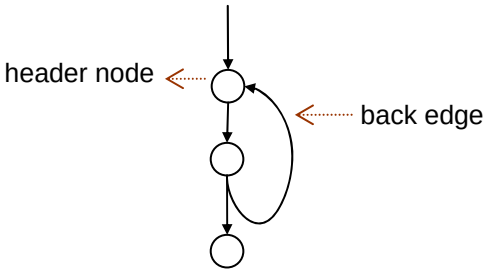

Fig.6 Structure of a post-tested loop

1) Structuring Conditionals:

For a subgraph headed by a 2-way conditional node, there exists a unique node (the follow node) such that all paths from each of the out-edges from the header converge upon it. The type of a conditional is determined by its header node. A 2-way node can be one of the following statements:

- *if...then...else* - neither successor of the header is the follow node.
- *if...then* - the false successor is the follow node.

The structure of the two conditionals is shown in fig 7 below:

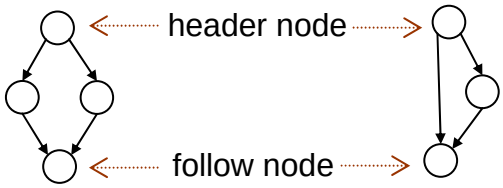

Fig.7 structure of conditional statements

2) Structuring loops:

A loop is defined by a *back edge* to the header node of the interval. The type of the loop is determined as follows:

- *A pre-tested loop* header will be a 2-way node that controls the loop iterations and will have a latching node that has a back edge to the loop header. One of the edges from a pre-tested loop header will lead to the follow node. ex: for, while. See fig V.
- *A post-tested loop* may have either a 2- or 1-way node as the header node or a 2-way latching node. The loop will have a 2-way header node if the first statement within the loop is a 2-way conditional. This is shown in fig 6.

```
local_4 = scanf( "%d", &a );

if ( a != 0 ) {
        local_4 = printf( "Hello, world\n" );
}
else {
        a = ( a ) * local_4;
}
return 0;
```

Fig.8 The output after control flow analysis

III. IMPLEMENTATION: THE "relipmoC" PROJECT

The project "*relipmoC*"[3] *ver 0.01*, which is an i386 ASM to C-code translator for Linux environment, is implemented and successfully tested for a large number of test cases, whose source code contained C expressions, conditionals, and switch statements, different kinds of loops, function calls and floating point variables.

[3] read as reverse of Compiler

A C compiler for Intel 80386, like gcc, implements a function-call-with-arguments by first pushing the arguments of the call in the reverse order (right to left) onto the stack, which would become the activation record of the callee, and then generating the call instruction. The callee then allocates space for its local variables on the stack. So the arguments that are passed to the callee can be recovered by examining the instructions before the call instruction. Since the type analysis for the arguments passed would have already been done in the caller, we know the types of the actual parameters and thus determine the function signature.

The return value is stored in the *accumulator – AX,* so the type and the return value can be recovered by examining its contents, thus completely determining the function prototype. The floating-point data that would have been used in the source program will be represented in the assembly program in the IEEE floating-point format. So such data can be recovered by reversing the process.

The three phases discussed in section 2 are implemented in C++, using the *Standard Template Library* provided by the GNU g++ compiler. We have made use of GNU's *LEX and YACC* programming tools for parsing the input assembly program.

A. Assumptions:

- The assembly language program used for translation was obtained by compiling a C program,
- The instruction set used (in the assembly language program) is that of Intel 80386 processor,
- The compiler used for obtaining the assembly program is gcc 3.2.2.

B. Limitations:

Some factors that hinder the asm2C translation process are:

- User-defined data types like *structs, typedefs,* unions and bit-fields add more to the confusion of the asm2C translator. Though it is easy to use structures while writing the code, it is almost impossible to figure out if a variable is a part of a structure or is it a basic type on its own, by looking at the compiled output.
- Use of processor-specific instructions/optimizations.
- The C code generated might need certain minor changes before execution.

IV. SUMMARY AND CONCLUSION

Before winding up, here is a brief discussion on the important issues of Reverse Engineering.

A. Ethical issues (of Reverse Engineering):

Decompilers aren't necessarily evil - but they do pose an ethical dilemma for many software developers. Decompilers offer great potential for legitimate purposes, but can also be used to steal the source code of competitors, or by hackers to determine weaknesses in the design of software. But just don't blame the decompiler - it’s the programmer who uses it for intellectual property theft, or the hacker that decompiles the software to find security holes that is at fault!

B. Future improvements:

- relipmoC could be generalized to translate object code to C language, rather than just i386 assembly; and thereby making it a fully fledged decompiler.
- Structures, pointers, multi-dimensional arrays and such constructs need to be accurately reconstructed according to original source code.
- The C code generated by this asm2C translator could be optimized.

C. Summary:

In this paper, we have discussed the concepts of Reverse Engineering with the perspective of building an Assembly to C language translator. The major design phases of such a translator viz. Control Flow Graph Generation, Data Flow Analysis and Control Flow Analysis are identified and discussed in detail. Then, we move on to describe our implementation of the asm2C translator in C++. Also we discuss the Assumptions and Limitations of the same. Finally we highlight some of the Ethical issues and future enhancements in our implementation and design.

V. REFERENCES

1. Aho, A.V., Ravi Sethi and J.D. Ullman; "*Compilers - Principles, Techniques and Tools*", Pearson Education, 1997.
2. C.Cifuentes and K.J.Gough, "*A Methodology for Decompilation*", Proceedings of the XIX Conferencia Latinoamericana de Informatica, **pp 257-266**, Buenos Aires, August 1993.
3. C.Cifuentes, "*Reverse Compilation Techniques*", Ph.D. thesis, Faculty of Info Technology, Queensland University of Technology, July 1994.
4. Barry B Brey, "*The Intel Microprocessors*".
5. Kernighan B.W. and D.M.Ritchie, "*The C programming language*", Prentice Hall Inc., 2nd edition, 1998.