Decompilation of Binary Programs

Cristina Cifuentes* K. John Gough

cifuente@fit.qut.edu.au gough@fit.qut.edu.au

School of Computing Science

Queensland University of Technology

GPO Box 2434, Brisbane, QLD 4001, Australia

---

*Present address: Department of Computer Science, University of Tasmania, GPO Box 252C, Hobart TAS 7001, Australia. Email: C.N.Cifuentes@cs.utas.edu.au

Summary

The structure of a decompiler is presented, along with a thorough description of the different modules that form part of a decompiler, and the type of analyses that are performed on the machine code to regenerate high-level language code. The phases of the decompiler have been grouped into three main modules: front-end, universal decompiling machine, and back-end. The front-end is a machine dependent module that performs the loading, parsing and semantic analysis of the input program, as well as generating an intermediate representation of the program. The universal decompiling machine is a machine and language independent module that performs data and control flow analysis of the program based on the intermediate representation, and the program's control flow graph. The back-end is a language dependent module that deals with the details of the target high-level language.

In order to increase the readability of the generated programs, a decompiling system has been implemented which integrates a decompiler, *dcc*, and an automatic signature generator, *dccSign*. Signatures for libraries and compilers are stored in a database that is read by the decompiler, thus, the generated programs can make use of known library names, such as `WriteLn()` and `printf()`.

dcc is a decompiler for the Intel 80286 architecture and the DOS operating system. *dcc* takes as input binary programs from a DOS environment and generates C programs as output. Sample code produced by this decompiler is given.

Key Words: decompiler, reverse compiler, compiler signature, library signature, i80286, C language

Introduction

A decompiler, or reverse compiler, is a program that attempts to perform the inverse process of the compiler: given an executable program compiled in any high-level language, the aim is to produce a high-level language program that performs the same function as the executable program. Thus, the input is machine dependent, and the output is language dependent.

Several practical problems are faced when writing a decompiler. The main problem derives from the representation of data and instructions in the Von Neumann architecture: they are indistinguishable. Thus, data can be located in between instructions, such as many implementations of indexed jump (`case`) tables. This representation and self-modifying code practices makes it hard to decompile a binary program.

Another problem is the great number of subroutines introduced by the compiler and the linker. The compiler will always include start-up subroutines that set up its environment, and runtime support routines whenever required. These routines are normally written in assembler and in most cases are untranslatable into a higher-level representation. Also, most operating systems do not provide a mechanism to share libraries, consequently, binary programs are self-contained and library routines are bound into each binary image. Library routines are either written in the language the compiler was written in or in assembler. This means that a binary program contains not only the routines written by the programmer, but a great number of other routines linked in by the linker. As an example of the amount of extra subroutines available in a binary program, a "hello world" program compiled in C generates 23 different procedures. The same program compiled in Pascal generates more than 40 procedures. All we are really interested in is the one procedure, `main`.

Despite the above-mentioned limitations, there are several uses for a decompiler, including two major software areas: maintenance of code and software security. From a maintenance point of view, a decompiler is an aid in the recovery of lost source code, the migration of applications to a new hardware platform, the translation of code written in an obsolete language into a newer language, the structuring of old code written in an unstructured way (e.g. "spaghetti" code), and a debugger tool that helps in finding and correcting bugs in an existing binary program. From a security point of view, a binary program can be checked for the existance of malicious code (e.g. viruses) before it is run for the first time on a computer, in safety-critical systems where the compiler is not trusted, the binary program is validated to do exactly what the original high-level language program intended to do, and thus, the output of the compiler can be verified in this way.

Different attempts at writing decompilers have been made in the last 20 years. Due to the amount of information lost in the compilation process, to be able to regenerate high-level language (HLL) code, all of these experimental decompilers have limitations in one way or another, including decompilation of assembly files[1, 2, 3, 4, 5] or object files with or without symbolic debugging information[6, 7], simplified high-level language[1], and the requirement of the compiler's specification[8, 9]. Assembly programs have helpful data information in the form of symbolic text, such as data segments, data and type declarations, subroutine names, subroutine entry point, and subroutine exit statement. All this information can be collected in a symbol table and the decompiler would not need to address the problem of separating data from instructions, or the naming of variables and subroutines. Object files with debugging information contain the program's symbol table as constructed by the compiler. Given the symbol table, it is easy to determine which memory locations are instructions, as there is a certainty on which memory locations represent data. In general, object files contain more information than binary files. Finally, the requirement to have access to the compiler's specifications is impractical, as these specifications are not normally disclosed by compiler manufacturers, or do not even exist.

Our decompiler, *dcc*, differs from previous decompilation projects in several ways; it analyses binary programs rather than assembler or object files, performs idiom[1] analysis to capture the essence of a sequence of instructions with a special meaning, performs data flow analysis on registers and condition codes to eliminate them, and structures the program's control flow graph into a generic set of high-level structures that can be accommodated

[1] An idiom is a sequence of instruction that forms a logical entity and has a meaning that cannot be derived by considering the primary meanings of the individual instructions

into different high-level languages, eliminating as much as possible the use of the `goto` statement.

The rest of this paper is structured in the following way; a thorough description of the structure of a decompiler is given, followed by the description of our implementation of an automatic decompiling system, and conclusions. The paper is followed by the definitions of graph theoretical concepts used throughout the paper (Appendix A), and sample output from different phases of the decompilation of a program (Appendix B).

The Decompiler Structure

Figure 1 about here

A decompiler can be structured in a similar way to a compiler, that is, a series of modules that deal with machine or language dependent features. Three main modules are required: a machine-dependent module that reads in the program, loads it into virtual memory and parses it (the front-end), a machine and language independent module that analyses the program in memory (the universal decompiling machine), and a language-dependent module that writes formatted output for the target language (the back-end) (see Figure 1). This modular representation makes it easier to write decompilers for different machine/target language pairs, by writing different front-ends for different machines, and different back-ends for different target languages. This result is true in theory, but in practical applications is always limited by the generality of the intermediate language used.

The Front-end

The front-end module deals with machine dependent features and produces a machine independent representation. It takes as input a binary program for a specific machine, loads it into virtual memory, parses it, and produces an intermediate representation of the program, as well as the program's control flow graph (see Figure 2).

Figure 2 about here

The **parser** disassembles code starting at the entry point given by the **loader**, and follows the instructions sequentially until a change in flow of control is met. Flow of control is changed due to a conditional, unconditional or indexed branch, or a procedure call; in which case the target branch label(s) start new instruction paths to be disassembled. A path is finished by a return instruction or program termination. All instruction paths are followed in a recursive manner. Problems are introduced by machine instructions that use indexed or indirect addressing modes. To handle these, heuristic methods are implemented. For example, while disassembling code, the parser must check for sequences of instructions that represent a multiway branch (e.g. a `switch` statement), normally implemented as an index jump in a jump table[10, 11]. Finally, the intermediate code is generated and the control flow graph is built.

Two levels of intermediate code are required; a low-level representation that resembles the assembler from the machine, and a higher-level representation that resembles statements from a high-level language. The initial level, or *low-level intermediate code*, is a simple $m : 1$ mapping of machine instructions to assembler mnemonics. Compound instructions (such as `rep movs`) are represented by a unique low-level intermediate intruction (e.g. `rep_movs`). The second level, or *high-level intermediate code*, is generated by the interprocedural data flow analysis, as explained in Section , and maps $n : 1$ low-level to high-level instructions. The front-end generates a low-level representation.

The **semantic analysis** phase performs idiom analysis and type propagation. Idioms are replaced by an appropriate functionally equivalent intermediate instruction. For example, Figure 3 illustrates two different idioms: the one on the left-hand side is a negation of a long variable, represented in this case by registers `dx:ax`. The idiom on the right-hand side is the prologue code of a high-level language procedure. In this case, space for 6 bytes is being reserved on the stack for local variables. There are a number of different idioms widely known in the compiler community, and the decompiler must code them in order to generate clearer high-level code.

Figure 3 about here

Type information is propagated after the idioms have been recognized. For example, if a long local variable is found at stack offsets -1 to -4, all references to `[bp-2]` and `[bp-4]` must be merged into references to such a long variable, e.g. `[bp-2]:[bp-4]`. Other type information can be propagated in the same way, such as fields (offsets) of a record.

An optimization phase is performed on the control flow graph as well. Due to the nature of machine code instructions, the compiler might need to introduce intermediate branches in an executable program, because there is no machine instruction capable of branching more than a certain maximum distance in bytes (architecture dependent). An optimization pass over the control flow graph removes this redundancy, by replacing the target branch location of all conditional or unconditional jumps that branch to an unconditional jump (and any recursive branches in this format) with the final target basic block. While performing this process, some basic blocks are not going to be referenced any more, as they were used only for intermediate branches. These nodes must be eliminated from the graph as well.

The Universal Decompiling Machine

The universal decompiling machine (UDM) is an intermediate module that is totally machine and language independent. It deals with flow graphs and the intermediate representation of the program, and performs all the flow analysis the input program needs (see Figure 4).

Figure 4 about here

Data Flow Analysis

The aim of the **data flow analysis** phase is to transform the low-level intermediate representation into a higher-level representation that resembles a HLL statement. It is therefore necessary to eliminate the concept of condition codes (or flags) and registers, as these concepts do not exist in high-level languages, and to introduce the concept of expressions, as these can be used in any HLL program. For this purpose, the technology of compiler optimization has been appropriated.

The first analysis is concerned with condition codes. Some condition codes are used only by hand-crafted assembly code instructions, and thus are not translatable to a high-level representation. Therefore, condition codes are classified in two groups: HLCC which is the set of condition codes that are likely to have been generated by a compiler (e.g. overflow, carry), and NHLCC which is the set of condition codes that are likely to have been generated by assembly code (e.g. trap, interrupt). The HLCC set is the one that can be analysed further. Instructions that use condition codes in the NHLCC set mean that the subroutine is most likely non high-level, and is therefore flagged as being so; assembler is all that can be generated for these subroutines.

A use/definition, or reaching definition, analysis is performed on condition codes. In this way, for a given use of a condition code c at instruction j, the use/definition chain (`ud-cc()`) determines which instruction(s) defined this condition. In practical cases, there is always only one instruction that defined the condition(s) used in the instruction at j; (i.e. `ud-cc`$(x) = \{i\}$). This set can be computed by a forward-flow any-path algorithm[12, 13]. Once the set of defined/used instructions is known (i.e. (i, j)), these low-level instructions can be replaced by a unique conditional high-level instruction that is semantically equivalent to the given instructions. Let us consider the example in Figure 5, which illustrates this point. Instruction 2 uses the sign (`SF`) and zero (`ZF`) condition codes. These flags were previously defined by instruction 1, which also defines the carry flag (`CF`). Given that this instruction defines all flags used by the conditional jump, they can be merged into one high-level instruction that compares the registers for the greater condition; i.e. `JCOND (ax > bx)`.

Figure 5 about here

The second analysis is concerned with registers. The elimination of temporary registers leads to the elimination of intermediate instructions by replacing several low-level instructions by one high-level instruction that does not make use of the intermediate register. As with condition codes, some machine instructions are hand-crafted by assembler programmers, and are untranslatable to a higher representation. We therefore classify the low-level instructions into two sets: HLI which is the set of instructions that are representable in a high-level language (e.g. `mov`, `add`), and NHLI which is the set of instructions that are likely to be generated only by assembly code (e.g. `cli`, `ins`). This analysis is concerned only with instructions from the HLI set. Instructions from the NHLI set belong to subroutines that are likely to have been written in assembler, or are untranslatable, and therefore are flagged as being so, and assembler is generated for them.

Two preliminary analyses are required for the elimination of registers. A definition/use analysis on registers is needed to determine how many uses there are for a definition of a register. Note that register variables are not eliminated by this analysis, as they represent local variables and thus are required in the final output program.

This analysis can be solved in a backward-flow any-path problem[12, 13]. Also, an interprocedural live register analysis is required, to ascertain which registers are live on entrance and on exit from a basic block. This analysis is also solved by known algorithms[12, 13].

Figure 6 about here

The method to eliminate registers has been named *forward substitution* as, by performing a forward substitution of the symbolic contents of a defined register that is only used once on the instruction that uses it, the temporary register is eliminated, the temporary instruction that defined the register is also eliminated, and the final instruction that used the register is now defined in terms of an expression. Let us consider a modulo 10 example, described in Figure 6. The registers `si` and `di` are register variables in this example. Register `tmp` is a virtual register introduced by the parser whenever a `DIV` instruction is found, as `DIV` is equivalent to two low-level instructions; a division and a modulus. Because they use the same registers as operands, and they redefine these registers, a `tmp` register is introduced to hold the initial value of `dx:ax`. The substitution up to instruction 32 is illustrated in the following code:

```
28 MOV ax, di      ; ASGN ax, di
29 MOV bx, OAh     ; ASGN bx, OAh
30 CWD             ; ASGN dx:ax, di   (substitute 28->30)
31 MOV tmp, dx:ax  ; ASGN tmp, di     (substitute 30->31)
32 DIV bx          ; ASGN ax,tmp / bx (eliminate instruction)
```

Instruction 32 defines a register that is not going to be used, therefore this instruction is redundant and must be eliminated. Any uses of the registers in the right-hand side of the instruction that is redundant need to be backpatched to reflect the non-existence of the instruction. After this step, the code would look like this:

```
29 MOV bx, OAh     ; ASGN bx, OAh      ; du(bx) = {33}
31 MOV tmp, dx:ax  ; ASGN tmp, di
33 MOD bx          ; ASGN dx, tmp % bx
34 MOV si, dx      ; ASGN si, dx
```

and the final substitutions would give the final result in instruction 34:

```
29 MOV bx,0Ah    ; ASGN bx, 0Ah
31 MOV tmp,dx:ax ; ASGN tmp, di
33 MOD bx        ; ASGN dx, di % 0Ah (substitute 31->33)
                                     (substitute 29->33)
34 MOV si, dx    ; ASGN si, di % 0Ah (substitute 33->34)
```

A complete algorithm that describes this data flow analysis can be found in References [14] and [15].

Control Flow Analysis

The **control flow analyzer** structures the control flow graph into generic high-level control structures that are available in most languages. These are conditional (`if..then[..else]`), multiway branch (`case`), and different types of loops (`while()`, `repeat..until()`, and endless `loop`). Different methods have been specified in the literature to structure graphs, most of them dealing with the elimination of `goto` statements from the graph, by the introduction of new variables[16, 17], code replication[18, 19, 20] or the use of multilevel `exit`[21, 22]. Both the introduction of new variables and code replication modify the apparent semantics of the program, and is therefore not desirable when decompiling binary programs, given that we want to decompile the code "as is". The use of multilevel `exit` statements is not supported by commonly used languages (e.g. Pascal, C), and thus cannot be part of the generic set of high-level control constructs that can be generated. We developed an algorithm that structures the graph into the set of generic high-level control structures, and, whenever it determines that a particular subgraph is not one of the generic constructs, it uses a `goto`. Note that the minimum number of `goto`s is always used. This algorithm is described in Reference [23].

A second structuring phase can be implemented to check for short circuit evaluation graphs. These graphs can be transformed into simpler graphs that hold two or more compound conditions on the one basic block, rather than requiring to generate high-level code that uses two or more nested `if..then` statements. Figure 7 illustrates an example of a compound `or` condition. The top basic block checks for the equality of `(si * 5)` and `50`. If this is false, a `printf()` node is reached. On the other hand, if the equality is true, a second condition is checked; `di < si`. If this condition is true, the same `printf()` basic block is reached, otherwise some other code is reached. Rather than generating C code for these conditions that require a `goto` (as these conditions are not properly nested), they can be merged into a compound `or` that negates the first condition, as the `printf()` node is reached whenever the first condition is false, and leaves the second condition as it is, given that this condition also reaches `printf()` when it is true. The intermediate basic block and edges are removed from the graph. The complete structuring algorithm is described in References [24] and [15].

Figure 7 about here

The Back-end

The back-end module is language dependent, as it deals with the target high-level language. This module, optionally restructures the graph into control constructs available only in the particular target language, and then generates code for this language (see Figure 8).

Figure 8 about here

The **restructuring** phase is optional and aims at structuring the graph even further, so that control structures available in the target language but not present in the generic set of control structures of the structuring algorithm, described in Section , are utilized. For instance, if the target language is Ada, multilevel exits are allowed. After the graph has been structured, multilevel exits look like a loop with abnormal `goto` exits. The restructuring phase can check the target destination of each `goto`, and determine if an `exit(i)` statement is suitable instead. Another example is the `for` loop; such a loop is equivalent to a `while()` loop that makes use of an induction variable. In this case, the induction variable needs to be found.

The final phase is the **HLL code generation**, which generates code for the target HLL based on the cfg and the associated high-level intermediate code. This phase defines global variables, and emits code for each procedure/function following a depth first traversal of the call graph of the program. Each procedure has comments on information that was collected during the analysis of such procedure, such as whether the procedure is likely to be low-level (in which case assembler is produced for the procedure), whether there were register arguments used, which registers returned a function return value, and so on. While generating code, if a `goto` instruction is required, a unique label identifier is created and placed before the instruction that takes the label. Variables and procedures are given names of the form `loc1`, `proc2`, as there is no information concerning their initial high-level name. A user interface can be built to allow the user to name these variables and procedures with more significant names.

The Decompiling System

As mentioned in the introduction, the compiler start-up code is linked into the executable program, as well as any library routines invoked by the program. Start-up code and library routines are often written in assembler, and therefore may contain low-level machine instructions, making these routines untranslatable or difficult to translate into a HLL representation. In order to get as much information as possible on the program to be decompiled, we have developed a decompiling system that integrates a decompiler, *dcc*, and an automatic signature generator, *dccSign*, as illustrated in Figure 9.

Figure 9 about here

A signature generator is a front-end module that generates signatures for compilers and library functions of those compilers. Such signatures are stored in a database, and are accessed by *dcc* to check whether a subroutine is a library function or not, in which case, the function is not analysed by *dcc*, but replaced by its library name (e.g.

`printf()`). This module is completely automatic, and takes as input library functions[25]. A library signature is a unique series of instructions that identifies a library function for a particular compiler. Correspondently, a compiler signature is a unique series of instructions that identifies a particular version of a compiler. In practice, different compiler signatures are required for different memory models (in the Intel architecture). Determining compiler signatures helps the decompiler determine where the real entry to the program is, i.e. the `main()`, and knowing the names of libraries functions makes the final program more readable.

At present, *dccSign* has been tested with Borland Turbo C, versions 2.1 and 3.0, Microsoft C, versions 5.0 and 8.0, and Borland Turbo Pascal version 4.0 and 5.0. The start-up code for these compilers is different enough to easily differentiate them. Borland and Microsoft provide the source code for their start-up code as part of their compiler distribution, therefore, it is a matter of finding the instruction that invokes `main()` to determine the entry point to the original high-level program. This process eliminates the need to analyze about 10 procedures that set up the environment for the particular compiler, and depend heavily on low-level machine instructions.

dcc is an experimental decompiler for the DOS operating system and the Intel i80286 architecture. As input, *dcc* reads `.com` and `.exe` files, and produces C programs as output (see Figure 10 for the structure of this decompiler). A disassembler is also part of *dcc*, so the user has the option of generating assembler files, C files, or both. As seen in Figure 10, *dcc* does not implement the restructuring phase of the back-end. This is due to the choice of target language, C in this instance, for which *dcc* generates good enough code without the need for restructuring of control structures.

Figure 10 about here

A signature checker module is inbuilt in *dcc*. This module determines if a known compiler is used, and therefore returns the `main()` to that program. It also scans at each procedure entry the first n bytes of instructions with a pattern-matching algorithm to see if the instructions correspond to any of the library routines of the compiler used. If a correspondence exists, the rest of the procedure is disregarded for further HLL analysis and code generation, its name and offset are placed in the symbol table, and any references to this procedure are replaced with the procedure's name. The decompilation of a sample program is given in Appendix B. The intermediate representation and final program are given.

Summary and Conclusions

This paper presents a methodology for decompilation of binary programs, and describes the current development state of *dcc*, a decompiler for the Intel 80286 architecture. The decompiler structure resembles that of a compiler; three main modules are distinguished: the front-end which is machine-dependent, the universal decompiling machine (UDM) which is machine- and language-independent, and the back-end which is language-dependent.

The front-end deals with the loading of the binary program, parsing it, and producing an intermediate representation of the program, and the program's control flow graph. The UDM performs data flow analysis in order to eliminate non high-level language concepts, such as condition codes and registers, from the intermediate representation, and to introduce the concept of expressions. The UDM also structures the control flow graph by determining which high-level structures are used in the program. Finally, the back-end optionally performs the restructuring needed to accommodate the structures found in the program into structures available in the target high-level language, and generates high-level code for each procedure.

The introduction of a module for compiler and library signature detection, *dccSign*, has reduced the number of routines to be decompiled, making the decompilation process faster, and providing better documentation of the output C programs. Binary programs that do not match against any of the compiler signatures of *dcc* are decompiled entirely, i.e. all compiler start-up code, runtime support routines, and library subroutines are decompiled and analyzed.

This project has proven the feasibility of writing a decompiler for a contemporary machine architecture. Many uses are envisaged for this decompiler, including software maintenance and security.

Acknowledgements

We would like to thank Michael Van Emmerik and Jeff Lederman for coding some of the modules involved in this project. This research is partly funded by Australian Research Council (ARC) grant no.A49130261.

A Graph Theoretical Definitions

A *basic block* is a sequence of instructions that has a single entry point and a single exit point. These requirements give the basic block the property that, if one instruction is executed, then all other instructions are executed as well.

A *control flow graph* G is a tuple (N, E, h), where N is the set of nodes, E is the set of directed edges, and h is the root of the graph. A node $n \in N$ represents a basic block. A *path* from n_1 to n_m, represented $n_1 \rightarrow n_m$, is a sequence of edges $(n_1, n_2), (n_2, n_3), \ldots, (n_{m-1}, n_m)$.

Let $\mathcal{P}= \{p_1, p_2, \ldots\}$ be the finite set of procedures of a program. A *call graph* C is a tuple (N, E, h), where N is the set of procedures and $n_i \in N$ represents one and only one $p_i \in \mathcal{P}$, E is the set of edges and $(n_i, n_j) \in E$ represents one or more references of p_i to p_j, and h is the main procedure.

A du-chain for register x at statement i is the set of statements j where x could be used, given that x is defined at statement i.

A ud-chain for register x at statement j is the set of statements i where x was defined.

B Example

This section illustrates an example of the decompilation of a simple C program. The sample program (see Figure 16) calculates the fibonacci number of a given input number. Figure 11 illustrates the relevant machine code of this binary. No library and compiler start up code is included. Figures 13 and 14 are the disassembly of the binary program. All calls to library routines were detected by *dccSign*, and thus not included in the analysis. Figure 15 is the final output from *dcc*. This C program can be compared with the original C program in Figure 16. The decompiled program is functionally equivalent to the original C program, although some differences are noticed. First of all, the recursive procedure `proc_1()` uses 2 local variables: one to copy a parameter, and another to hold the final result of the function. The use of the first local could have been avoided if data flow analysis was done across basic blocks. The second local cannot be deleted as this is the way the compiler compiled the C program. Second, there is no unsigned use of the identifiers in `proc_1()` to be able to know that the result is an unsigned integer rather than a signed integer. Finally, the `main()` procedure makes use of a `while()` rather than a `for`. In this program, 85 low-level instructions were converted into 16 high-level instructions; a reduction in the number of instructions of 81.78%.

Figure 11 about here

Figure 12 about here

Figure 13 about here

References

[1] B.C. Housel. *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages.* PhD dissertation, Purdue University, Computer Science, August 1973.

[2] F.L. Friedman. *Decompilation and the Transfer of Mini-Computer Operating Systems.* PhD dissertation, Purdue University, Computer Science, August 1974.

[3] D.A. Workman. Language design using decompilation. Technical report, University of Central Florida, December 1978.

[4] G.L. Hopwood. *Decompilation.* PhD dissertation, University of California, Irvine, Computer Science, 1978.

[5] D.L. Brinkley. Intercomputer transportation of assembly language software through decompilation. Technical report, Naval Underwater Systems Center, October 1981.

[6] J. Reuter. URL: ftp//cs.washington.edu/ pub/decomp.tar.z. Public domain software, 1988.

[7] D.J. Pavey and L.A. Winsborrow. Demonstrating equivalence of source code and PROM contents. *The Computer Language*, 36(7):654–667, 1993.

[8] J. Bowen and P. Breuer. Decompilation techniques. Internal to ESPRIT REDO project no. 2487 2487-TN-PRG-1065 Version 1.2, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, March 1991.

Figure 14 about here

Figure 15 about here

[9] P.T. Breuer and J.P. Bowen. Decompilation: the enumeration of types and grammars. To appear in Transaction of Programming Languages and Systems, 1993.

[10] C. Cifuentes and K.J. Gough. A methodology for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informática*, pages 257–266, Buenos Aires, Argentina, 2-6 August 1993. Centro Latinoamericano de Estudios en Informática.

[11] J.R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software – Practice and Experience*, 24(2):197–218, February 1994.

[12] C.N. Fischer and R.J. LeBlanc Jr. *Crafting a Compiler*, chapter 16, pages 609–680. Benjamin Cummings, 2727 Sand Hill Road, Menlo Park, California 94025, 1988.

[13] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume II: Compiling, chapter 11, pages 907–956. Prentice Hall, 1973.

[14] C. Cifuentes. Interprocedural data flow decompilation. Technical Report 4/94, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia, April 1994.

[15] C. Cifuentes. *Reverse Compilation Techniques*. PhD dissertation, Queensland University of Technology, School of Computing Science, July 1994.

Figure 16 about here

[16] M.H. Williams and G.Chen. Restructuring pascal programs containing goto statements. *The Computer Journal*, 28(2):134–137, 1985.

[17] A.M. Erosa and L.J. Hendren. Taming control flow: A structured approach to eliminating goto statements. Technical Report ACAPS Technical Memo 76, School of Computer Science, McGill University, 3480 University St, Montreal, Canada H3A 2A7, September 1993.

[18] D.E. Knuth and R.W. Floyd. Notes on avoiding go to statements. *Information Processing Letters*, 1(1):23–31, 1971.

[19] M.H. Williams. Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal*, 20(1):45–50, 1977.

[20] A.L. Baker and S.H. Zweben. A comparison of measures of control flow complexity. *IEEE Transactions on Software Engineering*, SE-6(6):506–512, November 1980.

[21] B.S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[22] L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35(4):893–920, October 1988.

[23] C. Cifuentes. A structuring algorithm for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informática*, pages 267–276, Buenos Aires, Argentina, 2-6 August 1993. Centro Latinoamericano de Estudios en Informática.

[24] C. Cifuentes. Structuring decompiled graphs. Technical Report 5/94, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia, April 1994.

[25] M. Van Emmerik. Signatures for library functions in executable files. Technical Report 2/94, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia, April 1994.

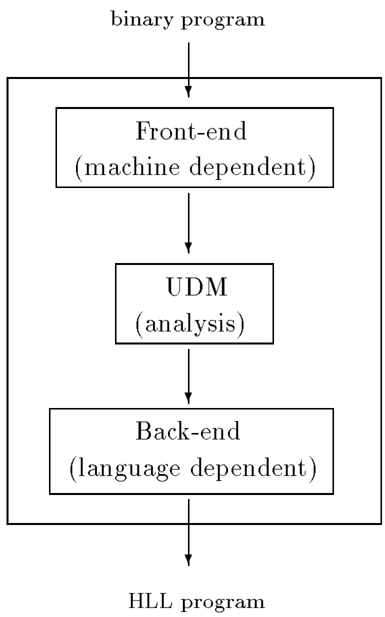

Figure 1: Decompiler Modules

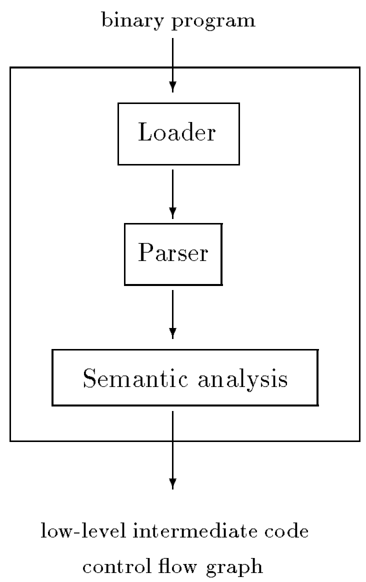

Figure 2: Front-end Phases

```
neg dx                push bp
neg ax                mov  bp, sp
sbb dx, 0             sub  bp, 6

           ⇓

neg dx:ax             enter 6,0
```

Figure 3: Sample Idioms and their Transformation

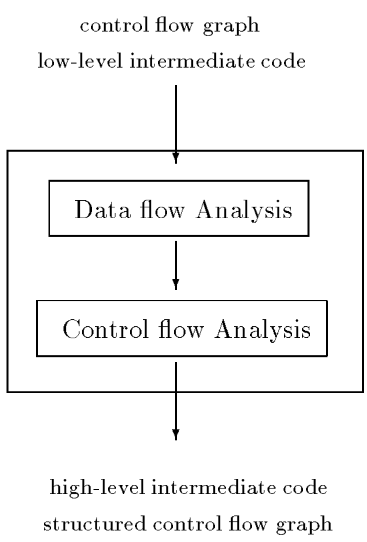

Figure 4: UDM Phases

```
1  cmp ax, bx ; def: SF,ZF,CF
2  jg  labZ   ; use: SF,ZF  ; ud-cc(SF,ZF) = {1}

                     ⇓

             JCOND (ax > bx)
```

Figure 5: Condition Code Example

```
..  ...              ; other code here
28  MOV  ax, di      ; ASGN ax, di       ; du(ax) = {30}
29  MOV  bx, 0Ah     ; ASGN bx, 0Ah      ; du(bx) = {32,33}
30  CWD              ; ASGN dx:ax, ax    ; du(ax) = {31}, du(dx) = {31}
31  MOV  tmp, dx:ax  ; ASGN tmp, dx:ax   ; du(tmp) = {32,33}
32  DIV  bx          ; ASGN ax, tmp / bx ; du(ax) = {}
33  MOD  bx          ; ASGN dx, tmp % bx ; du(dx) = {34}
34  MOV  si, dx      ; ASGN si, dx
..  ...              ; other code here, no use of ax

                              ⇓

                       ASGN si, di % 0Ah
```

Figure 6: Simple Expression Example

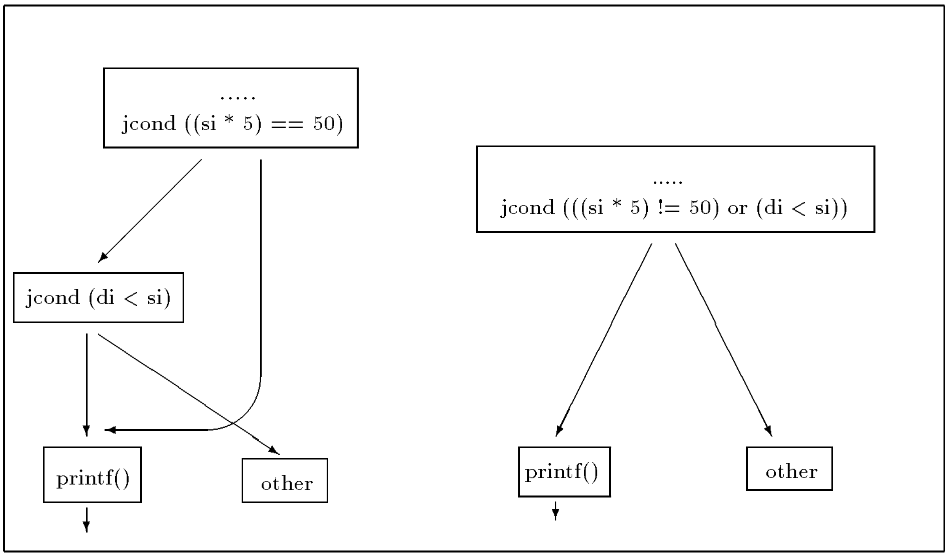

Figure 7: Short-circuit Evaluation Graph

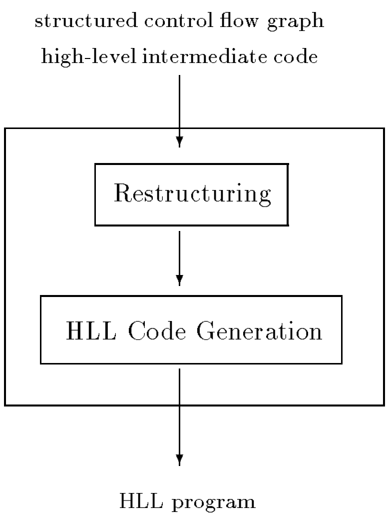

Figure 8: Back-end Phases

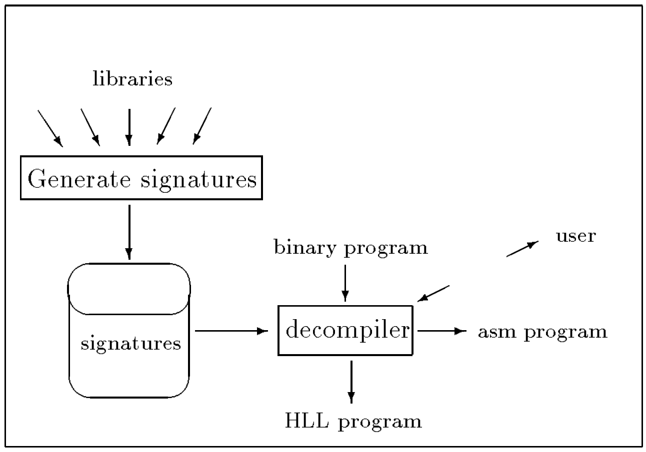

Figure 9: Decompiling System

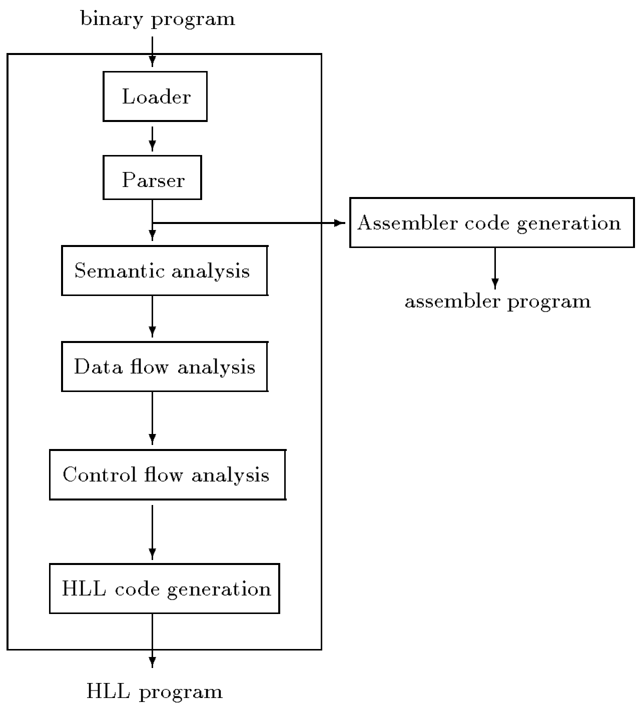

Figure 10: Structure of dcc

```
      55 8B EC 83 EC 04 56 57 1E B8 94 00 50 9A
0E 00 3C 17 59 59 16 8D 46 FC 50 1E B8 B1 00 50
9A 07 00 F0 17 83 C4 08 BE 01 00 EB 3B 1E B8 B4
00 50 9A 0E 00 3C 17 59 59 16 8D 46 FE 50 1E B8
C3 00 50 9A 07 00 F0 17 83 C4 08 FF 76 FE 9A 7C
00 3B 16 59 8B F8 57 FF 76 FE 1E B8 C6 00 50 9A
0E 00 3C 17 83 C4 08 46 3B 76 FC 7E C0 33 C0 50
9A 0A 00 49 16 59 5F 5E 8B E5 5D CB 55 8B EC 56
8B 76 06 83 FE 02 7E 1E 8B C6 48 50 0E E8 EC FF
59 50 8B C6 05 FE FF 50 0E E8 E0 FF 59 8B D0 58
03 C2 EB 07 EB 05 B8 01 00 EB 00 5E 5D CB
```

Figure 11: Machine Code (hexadecimal format)

```
File type is EXE
Signature             = 4D5A
File size % 512       = 0176
File size / 512       = 0018 pages
# relocation items    = 006A
Offset to load image = 0020 paras
Minimum allocation    = 0000 paras
Maximum allocation    = FFFF paras
Load image size       = 2D76
Initial SS:SP         = 02D9:00E6
Initial CS:IP         = 0010:0000
```

Figure 12: Information provided by the Loader

```
proc_1  PROC  FAR
000 00053C 55                   PUSH          bp
001 00053D 8BEC                 MOV           bp, sp
002 00053F 56                   PUSH          si
003 000540 8B7606               MOV           si, [bp+6]
004 000543 83FE02               CMP           si, 2
005 000546 7E1E                 JLE           L1
006 000548 8BC6                 MOV           ax, si
007 00054A 48                   DEC           ax
008 00054B 50                   PUSH          ax
009 00054C 0E                   PUSH          cs
010 00054D E8ECFF               CALL  near ptr proc_1
011 000550 59                   POP           cx
012 000551 50                   PUSH          ax
013 000552 8BC6                 MOV           ax, si
014 000554 05FEFF               ADD           ax, 0FFFEh
015 000557 50                   PUSH          ax
016 000558 0E                   PUSH          cs
017 000559 E8E0FF               CALL  near ptr proc_1
018 00055C 59                   POP           cx
019 00055D 8BD0                 MOV           dx, ax
020 00055F 58                   POP           ax
021 000560 03C2                 ADD           ax, dx
023 00056B 5E              L2:  POP           si
024 00056C 5D                   POP           bp
025 00056D CB                   RETF
026 000566 B80100          L1:  MOV           ax, 1
027 000569 EB00                 JMP           L2
proc_1  ENDP

main  PROC  FAR
000 0004C2 55                   PUSH          bp
001 0004C3 8BEC                 MOV           bp, sp
002 0004C5 83EC04               SUB           sp, 4
003 0004C8 56                   PUSH          si
004 0004C9 57                   PUSH          di
005 0004CA 1E                   PUSH          ds
006 0004CB B89400               MOV           ax, 94h
007 0004CE 50                   PUSH          ax
008 0004CF 9A0E004D01           CALL   far ptr printf
009 0004D4 59                   POP           cx
010 0004D5 59                   POP           cx
011 0004D6 16                   PUSH          ss
012 0004D7 8D46FC               LEA           ax, [bp-4]
013 0004DA 50                   PUSH          ax
014 0004DB 1E                   PUSH          ds
```

Figure 13: Low-level Intermediate Code

```
015 0004DC B8B100             MOV            ax, 0B1h
016 0004DF 50                 PUSH           ax
017 0004E0 9A07000102         CALL   far ptr scanf
018 0004E5 83C408             ADD            sp, 8
019 0004E8 BE0100             MOV            si, 1
021 000528 3B76FC         L3: CMP            si, [bp-4]
022 00052B 7EC0               JLE            L4
023 00052D 33C0               XOR            ax, ax
024 00052F 50                 PUSH           ax
025 000530 9A0A005A00         CALL   far ptr exit
026 000535 59                 POP            cx
027 000536 5F                 POP            di
028 000537 5E                 POP            si
029 000538 8BE5               MOV            sp, bp
030 00053A 5D                 POP            bp
031 00053B CB                 RETF
032 0004ED 1E             L4: PUSH           ds
033 0004EE B8B400             MOV            ax, 0B4h
034 0004F1 50                 PUSH           ax
035 0004F2 9A0E004D01         CALL   far ptr printf
036 0004F7 59                 POP            cx
037 0004F8 59                 POP            cx
038 0004F9 16                 PUSH           ss
039 0004FA 8D46FE             LEA            ax, [bp-2]
040 0004FD 50                 PUSH           ax
041 0004FE 1E                 PUSH           ds
042 0004FF B8C300             MOV            ax, 0C3h
043 000502 50                 PUSH           ax
044 000503 9A07000102         CALL   far ptr scanf
045 000508 83C408             ADD            sp, 8
046 00050B FF76FE             PUSH  word ptr [bp-2]
047 00050E 9A7C004C00         CALL   far ptr proc_1
048 000513 59                 POP            cx
049 000514 8BF8               MOV            di, ax
050 000516 57                 PUSH           di
051 000517 FF76FE             PUSH  word ptr [bp-2]
052 00051A 1E                 PUSH           ds
053 00051B B8C600             MOV            ax, 0C6h
054 00051E 50                 PUSH           ax
055 00051F 9A0E004D01         CALL   far ptr printf
056 000524 83C408             ADD            sp, 8
057 000527 46                 INC            si
058                           JMP            L3      ;Synthetic inst
main  ENDP
```

Figure 14: Low-level Intermediate Code – cont

```
/* Input file : fibo.exe
 * File type : EXE
 */

int proc_1 (int arg0)
/* Takes 2 bytes of parameters.
 * High-level language prologue code.
 * C calling convention.
 */
{
int loc1;
int loc2; /* ax */

    loc1 = arg0;
    if (loc1 > 2) {
        loc2 = (proc_1 ((loc1 - 1)) + proc_1 ((loc1 + -2)));
    }
    else {
        loc2 = 1;
    }
    return (loc2);
}


void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;
int loc3;
int loc4;

    printf ("Input number of iterations: ");
    scanf ("%d", &loc1);
    loc3 = 1;
    while ((loc3 <= loc1)) {
        printf ("Input number: ");
        scanf ("%d", &loc2);
        loc4 = proc_1 (loc2);
        printf ("fibonacci(%d) = %u\n", loc2, loc4);
        loc3 = (loc3 + 1);
    } /* end of while */
    exit (0);
}
```

Figure 15: Final C Program

```
#include <stdio.h>

int main()
{ int i, numtimes, number;
  unsigned value, fib();

   printf("Input number of iterations: ");
   scanf ("%d", &numtimes);
   for (i = 1; i <= numtimes; i++)
   {
      printf ("Input number: ");
      scanf ("%d", &number);
      value = fib(number);
      printf("fibonacci(%d) = %u\n", number, value);
   }
   exit(0);
}

unsigned fib(x)  /* compute fibonacci number recursively */
int x;
{
   if (x > 2)
      return (fib(x - 1) + fib(x - 2));
   else
      return (1);
}
```

Figure 16: Initial C Program