An Environment for the Reverse Engineering of Executable Programs

Cristina Cifuentes*
Department of Computer Science, University of Tasmania
GPO Box 252C, Hobart Tas 7001, Australia
C.N.Cifuentes@cs.utas.edu.au

Abstract

Reverse engineering of software systems has traditionally centered upon the generation of high-level abstractions or specifications from high-level code or databases. In this paper we report on a reverse engineering environment for low-level executable code: a reverse compilation or decompilation environment that aids in the understanding of the underlying executable program.

The reverse compilation process recovers high-level code from executable programs at a higher representation level than that produced by disassemblers; in fact, disassembly is part of the first stage in this process. Several tools aid in the process of reverse compilation, these are: loaders, signature generators, library prototype generators, disassemblers, library bindings, and language to language translators. The integration of these tools in the whole process is presented in this paper.

The results obtained by the prototype reverse compilation system **dcc** *are encouraging: high-level code is regenerated with correct use of expressions and control structures, and the complete elimination of registers and condition codes. An elimination rate of low-level instructions of over 75% was reached, representing the overall improvement this decompiler system has made over previous decompilers and disassemblers (where the rate tends to be nil). A sample decompilation program is given.*

Keywords: *reverse engineering, reverse compiler, disassembler, signatures, DOS, i80286, C language.*

1 Introduction

Reverse engineering of software systems has been defined as the analysis of a subject system to identify the system's components and their interrelationships, and to create a representation of the system in another form or at a higher level of abstraction [1]. The aim of reverse engineering of software systems is to gain an *understanding* of the system and its structure for the purposes of maintenance, which is unlike the case of reverse engineering of hardware systems where the system is normally analyzed to create a duplicate copy of it.

Most reverse engineering environments reported in the literature concentrate on the generation of high-level specifications or graphical representations for Cobol and Fortran source code [2, 3], and the extraction of design representations and specification documents from source code [4]. Also, the recovery of design information from a database has been incorporated into CASE (Computer-Aided Software Engineering) tools such as ADW. In this paper we concentrate on reverse engineering tools at a lower level: tools that recover high-level source code from executable programs. Tools in this category include disassemblers, debuggers, decompilers, disk editors, and unpacking utilities; we are interested in disassemblers and decompilers. It has long been argued that decompilers are difficult to write due to the variety of output of each vendor's compiler [5], in this paper we present a suite of tools that help in the process of decompilation and disassembly.

A disassembler is a program that reads an executable program and translates it into an equivalent assembler program; a decompiler goes a step further by translating the program into an equivalent high-level language (such as C or Pascal) program. In both cases, the executable program to be translated is an arbitrary program compiled from any high-level language. In general, a decompiler comprises a disassembler since this step is intermediate in the generation of high-level code. As with most reverse engineering tools, disassemblers and decompilers are semi-automated tools rather than fully automated tools.

1.1 Uses of decompilation in the 90s

Traditionally, decompilation has been used to help in the migration of computer software from an old to a new machine architecture. In the 90s, with the advent of high-performance machines, a transitional migration platform is needed to provide users with the same soft-

*This research was partly funded by Australian Research Council (ARC) Grant No.A49130261 while the author was with the Queensland University of Technology, Brisbane, Australia.

ware they run on their soon-to-become-old-machines, on the new machine. A technique known as binary translation was developed which makes use of disassembly and decompilation techniques, but which also provides a run-time support environment to run code that could not be successfully translated to the new machine; this environment was first developed by Digital to aid in the translation of VAX and `mx` software to the new Alpha machine [6].

As with the Alpha binary translation environment, several other uses of decompilation require other tools to help in the process of extraction of high-level information. This is mainly due to the fact that software has become very large and complex, and also because decompilation on its own is an incomplete process.

In the UK, the Nuclear Electric plc developed a decompiler to verify the code produced by a proprietary compiler; such code was to be run in PROM in a safety critical environment and therefore the compiler was not trusted. Their environment incorporated a decompiler that made use of data type information from the original high-level language program, decompiled the executable program, and determined whether the high-level code regenerated was similar in functionality to the initial high-level code [7].

Also in the UK, a group at Oxford University has worked on the automatic generation of decompilers based on compiler specifications using logic and functional languages [8, 9], and recently such approach was used to generate a decompiler for a subset of a C++ compiler [10]. This research was aimed at the provision of reverse engineering tools for maintenance.

In China, a decompiler for 8086 executable programs compiled with the Microsoft C Version 5.0 compiler was written [11]. This decompiler made use of the recognition of library functions by a hand-crafted method. An attempt at high-level data type recognition, such as arrays and structures is mentioned but not much detail is given in the paper. The use of this decompiler is not mentioned either.

In Australia, the **dcc** decompiler was written to help in the detection of virus code in an executable program, and to aid in the maintenance of legacy code by the extraction of source code from these systems [12]. In the former case, the decompiler reports on any suspicious malicious code in the comments of the generated high-level language program, and in the latter case the decompiler produces a high-level program that aids in the understanding of the legacy system, which can then be rewritten in a different (newer) language.

1.2 The need for extra tools

There are several theoretical and practical limitations for writing decompilers. Some of these problems can be solved by the use of heuristic methods, others cannot be determined completely. Due to these limitations, a decompiler performs automatic program translation of *some* source programs, and semi-automatic program translation of other source programs. This differs from a compiler, which performs an automatic program translation of all source programs, but is inline with the semi-automation of reverse engineering tools [13].

The main problem in disassembly 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 along with idioms[1] and self-modifying code practices makes it hard to disassemble an executable program. In fact, the separation of data and instructions is unsolvable in general as if there was an algorithm to determine such separation, this algorithm would also solve the halting problem [14].

Another problem is the great number of subroutines introduced by the compiler and the linker, and bound in the executable program. The compiler introduces 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, in operating systems that do not provide a mechanism to share libraries, executable 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 assembler. This means that an executable program contains not only the routines written by the programmer, but a great number of other routines linked in by the linker. For example, a “hello world” program compiled in C generates over 25 different subroutines in the executable program. The same program compiled in Pascal generates more than 40 subroutines. When decompiling this example, we are only interested in the `main()` subroutine, rather than the other 25 or so subroutines.

1.3 Legal issues

Several questions have been raised in the last years regarding the legality of decompilation. A debate between supporters of decompilation who claim fair com-

[1] An idiom is a sequence of instructions which form a logical entity, and which taken together have a meaning that cannot be derived by considering the primary meanings of the instructions.

petition is possible with the use of decompilation tools, and the opponents of decompilation who claim copyright is infringed by decompilation, is currently being held; in fact, this debate has been reported in the literature since 1984 [15]. The law in different countries is being modified to determine in which cases decompilation is lawful. At present, commercial software is being sold with software agreements that ban the user from disassembling or decompiling the product. For example, part of the Lotus software agreement reads like this:

> You may not alter, merge, modify or adapt this Sofware in any way including disassembling or decompiling.

It is not the purpose of this paper to debate the legal implications of decompilation. This topic is not further covered in this paper.

The rest of this paper is structured in the following way: §2 describes a complete reverse compilation system, §3 reports on results achieved by the implementation of the **dcc** system, §4 provides a sample decompilation of a benchmark multiplication program, §5 compares this work with previous work, and §6 provides the conclusions of this work.

2 A reverse compilation system

The steps involved in a typical "decompilation" of a contemporary executable program are shown in Figure 1. In this figure, all text in boxes represent tools that aid in the process of decompilation; all other text represents data files. This system is completely automated and provides partial results which aid in the understanding of the underlying program. In general, one can consider the user to be another source of information in this process, in particular when separating data from code; this option was not included in this system.

2.1 Loader

Typical information contained in an executable program is: the header (which contains information on the sizes of segments, default values for registers, and the entry-point to the program), the relocation table, the code segment and the data segment. The loader reads the header information of the executable program, determines the amount of memory required to load the binary image of the program (both code and data segments), allocates memory, loads it, and performs relocation of the data elements found in the relocation table. The loader also determines the entry-point to the program.

2.2 Signature generator

The signature generator is a program that automatically determines library signatures. A signature is a binary pattern that uniquely identifies each compiler and each library subroutine of a particular library file. The use of these signatures attempts to reverse the task performed by the linker, which links in compiler start-up code and library subroutines into the executable program. The disassembler and decompiler uses this information to determine linked-in subroutines that are not required to be analyzed but rather eliminated (in the case of start-up code) or replaced by their name (in the case of library routines). All other subroutines are likely to have been user compiled from the initial high-level language program. Library signatures are needed for statically-linked libraries, which is the case of all DOS executable programs and of Windows programs that do not use DDLs (dynamic-link libraries).

For example, in the compiled C program that displays "hello world" and has over 25 different subroutines in the executable program, 16 subroutines are added by the compiler to set-up its environment, 9 routines that form part of `printf()` are added by the linker, and 1 subroutine only forms part of the initial C program. The determination of library signatures aids in the matching of `printf()` and its complete subtree of subroutines called. Compiler signatures aid in the determination of the real entry-point to the `main` program (rather than to the code introduced by the compiler for setting up of the environment), hence allowing the disassembler and decompiler to skip all compiler set-up routines (16 in this example).

The use of a signature generator not only reduces the number of subroutines to analyze, but also increases the documentation of the final programs by using library names rather than arbitrary subroutine names. The lookup of these signatures can be implemented with a perfect minimal hashing algorithm, hence a constant O(1) lookup is expected at all time to determine whether a subroutine matches a signature of any known subroutine signatures in the library. An automatic method for generating library signatures is described in [16, 17].

2.3 Prototype generator

The prototype generator is a program that automatically determines the types of the formal arguments of library subroutines, and the type of the return value for functions. For C programs, these prototypes are derived from the library header files provided with any C compiler; in the case of Pascal, this information is stored in the library file itself and hence it can be determined by parsing this file. This information is used by the decompiler to determine the type of the arguments to library subroutines, the number of such arguments (whenever possible), and the return type of functions.

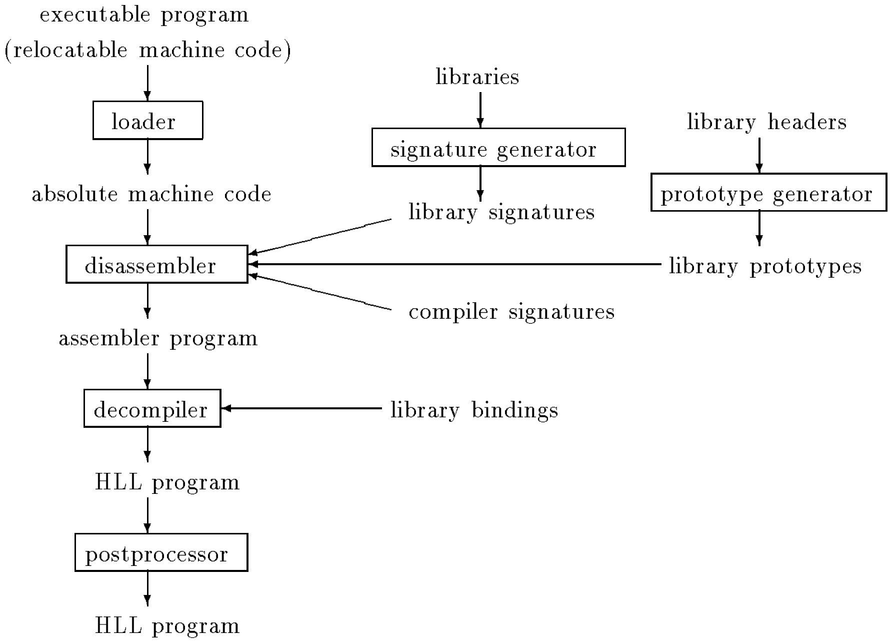

Figure 1: A decompilation system

2.4 Disassembler

The disassembler parses the binary image of the program to translate it to assembler. With the aid of compiler signatures, the disassembler firstly determines if the first bytes at the entry point provided by the loader match a known compiler signature, if so, the real entry-point to the `main` program can be determined and code is disasssembled from this point onwards. If no compiler signature is matched, code is disassembled from the given entry-point, and hence all the compiler's set-up routines are disassembled. Note that the use of compiler signatures is done without lose of generality; the disasembler performs the same parsing algorithm based on any given entry-point.

The main problem encountered by the disassembler is determining what is data and what is an instruction. Parsing of the program is performed by traversing all paths from the program's entry-point, assuming that all paths are followed in the program. New paths are determined by transfers of control to other sections of code (i.e. new entry-points) by means of conditional or unconditional jumps, or subroutine calls. The algorithm stops when all paths have been searched. Note that some paths may be only partially searched due to an instruction that cannot be decoded. In general, such is the case for indexed or indirect jumps which rely in the value in a register. In some cases, heuristics can be applied to determine the bounds of the register, for example, in the implementation of indexed `case` instructions. Whenever an instruction cannot be decoded, it is flagged for commenting during the generation of the final high-level program. This conventional approach is able to disassemble a great percentage of the executable program.

A disasembler on its own normally only generates an assembler program, but, if used as part of a decompiler, it also needs to provide a control flow graph of the program (i.e. the call graph with control flow graphs attached to each subroutine node). This graph is needed for the analysis that the decompiler performs.

2.5 Decompiler

The decompiler transforms a low-level representation of a program into a high-level language representation. In this section we treat decompilation as the complete process of translation from the binary image to the high-level language program, hence, in this discussion, a disassembler is incorporated as part of the decompiler, and assembler code is used as the intermediate code which is analysed and transformed by the decompiler.

Conceptually, a decompiler is structured in a similar way to a compiler, by a series of phases that transform the source executable program from one representation to another. These phases are grouped for implementation purposes into three different modules: **front-end**, **udm**, and **back-end**; as seen in Figure 2.

In theory, the grouping of phases in a decompiler makes it easy to write different decompilers for different machines and languages; by writing different front-ends

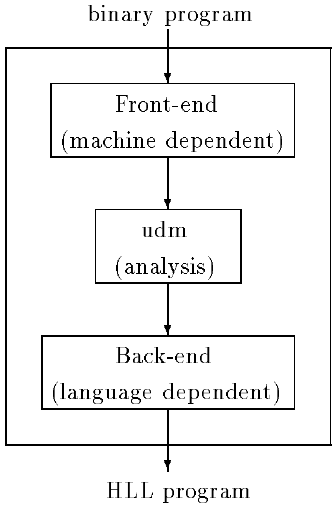

Figure 2: Decompiler modules

for different machines, and different back-ends for different target languages. In practical applications, this result is always limited by the generality of the intermediate language used.

The **front-end** consists of those phases that are machine and machine-language dependent. This module parses the binary image of the program (i.e. implements a disassembler) and stores it in an intermediate language representation, analyzes the disassembled code for the determination of base data types such as integers, long integers, and reals, and constructs the control flow graph of the program.

The **udm** is the universal decompiling machine; an intermediate module that is completely machine and language independent, and that performs the core of the decompiling analysis. Two phases are included in this module;

1. Data flow analysis: this phase improves the intermediate code by eliminating machine low-level concepts such as register and condition codes, and by regenerating high-level language concepts such as expressions. For example, the following intermediate language code:

   ```
   asgn ax, [bp-0Eh]
   asgn bx, [bp-0Ch]
   asgn bx, bx * 2
   asgn ax, ax + bx
   asgn [bp-0Eh], ax
   ```

   is converted into the following high-level expression

   ```
   asgn [bp-0Eh], [bp-0Eh] + [bp-0Ch] * 2
   ```

   which does not rely on registers any more.

   This phase makes use of traditional compiler optimization theory and not only implements an extended register copy propagation algorithm to eliminate registers, regenerate high-level expressions, and eliminate intermediate language instructions such as `push` and `pop`; but it also performs inter- and intra-procedural live register analysis to determine register variables and function return registers. Complete algorithms to perform this analysis are described in [17].

   A further analysis of data types is done at this stage to attempt to recover compound data types such as arrays and structures.

2. Control flow analysis: this analysis structures the control flow graph of each subroutine of the program into a generic set of high-level language constructs. This generic set contains control instructions available in most languages; such as looping and conditional transfers of control. In our implementation, the generic set was composed of `if..then`, `if..then..else`, and `case` conditionals, and `while()`, `repeat..until()`, and endless `loop` loops. Language-specific constructs are not allowed as they are not available in most other languages. Figure 3 shows two sample control flow graphs: an `if..then..else` conditional and a `while()` loop. The aim of the structuring algorithm is to detect the underlying generic control structures of the program, and eliminate as much as possible the use of unconditional transfers of control (i.e. `goto` statements). Note that the `goto` construct forms part of the generic set of control structures since languages such as Pascal and C allow the use of this construct; hence, the underlying graph might only be representable in a higher-level language that makes use of the `goto` statement. Complete algorithms for structuring arbitrary graphs are based in graph theory and are given in [17].

The **back-end** generates the final high-level language code based on the control flow graph and the improved intermediate code of each subroutine. Variable names are selected for all local variables (stack and register-variables), and for all arguments. Subroutine names are also selected for the different routines found in the program. Control structures and intermediate instructions are translated into a high-level language statement.

2.6 Library bindings

Whenever the target high-level language generated by the decompiler is different to the original language

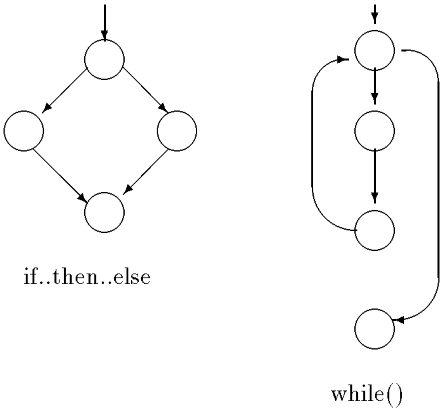

Figure 3: Generic constructs

used to compile the executable source program, if the generated target code makes use of library names (i.e. library signatures were detected), although this program is correct, it cannot be recompiled in the target language since it does not use library routines for that language but for another one. The introduction of library bindings solves this problem, by binding the subroutines of one language to the other.

ISO committee SC22 of Working Group 11 is concerned with the creation of standards for language independent access to service facilities. This work can be used to define language independent bindings for languages such as C and Modula-2. Information on library bindings can be placed in a file and used by the back-end of the decompiler to produce target code that uses the target language's library routines, instead of the ones matched during disassembly.

2.7 Postprocessor

A postprocessor is a program that transforms a high-level language program into a semantically equivalent high-level program written in the same language. For example, if the target language is C, the following code

```
  loc1 = 1;
  while (loc1 < 50) {
     /* some code in C */
     loc1 = loc1 + 1;
  }
```

would be converted by a postprocessor into

```
  for (loc1 = 1; loc1 < 50; loc1++) {
      /* some code in C */
  }
```

which is a semantically equivalent program that makes use of control structures available in the C language, but not present in the generic set of structures decompiled by the decompiler.

3 Experimental results

dcc is a prototype decompiler system written in C for the DOS operating system. **dcc** was initially developed on a DecStation 3000 running Ultrix, and was ported to the Intel architecture under DOS. **dcc** takes as input `.exe` and `.com` files for the Intel i80286 architecture, and produces target C and assembler programs. This decompiler was built using the techniques summarized in this paper and fully described in [17], and is composed of the phases shown in Figure 4. As can be seen, the decompiler has a built-in loader and disassembler, and there is no library bindings or postprocessing phase. Compiler and library signatures were generated for several compilers, including Borland Turbo C and Microsoft Visual C++. This prototype decompiler determines base types (i.e. byte, integer, long, pointer), but is not able to determine compound types such as arrays and structures because it has not been implemented yet.

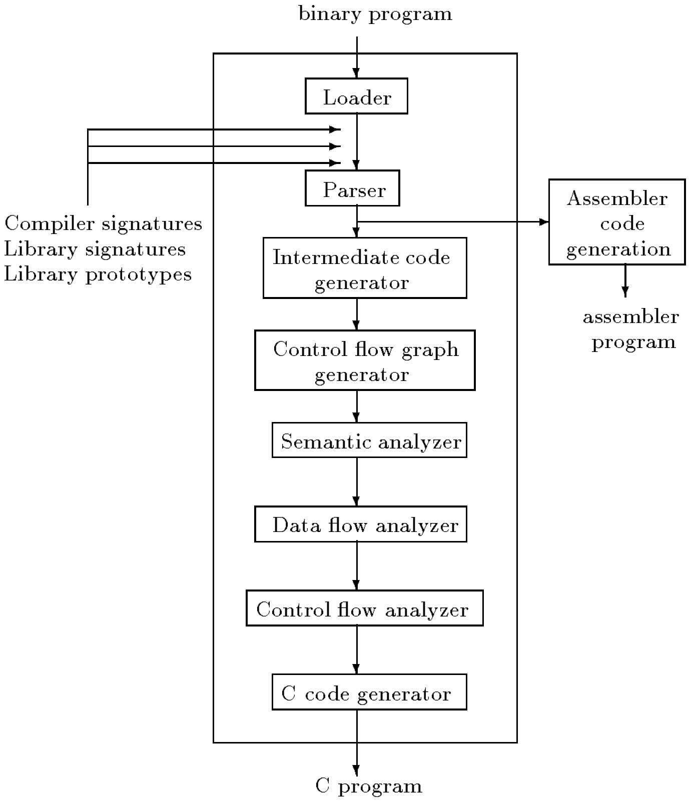

Figure 4: Structure of the dcc decompiler system

This section reports on a test suite of `.exe` programs originally written in C and compiled with Borland Turbo C under DOS. These programs make use of base type variables, and illustrate different aspects of the decompilation process. These programs were run in batch mode, generating the disassembly file `.a2`, the C file `.b`, the call graph of the program, and statistics on the intermediate code instructions. The statistics reflect the percentage of intermediate instruction reduction on all subroutines for which C is generated; subroutines which translate to assembler are not considered in the statistics. For each program, a total count on intermediate instructions before and after analysis, and a total percentage reduction is given.

The programs in the test suite are as follow: the first three programs deal with arithmetic operations on the different three base types (byte, integer, long integer); the original C programs had the same code, but their variables were defined of a different type. The next four programs are benchmark programs from the Plum-Hall benchmark suite. These programs were written by Eric S. Raymond and are freely available on the network [18]. These programs were modified to ask for the arguments to the program with `scanf()` rather than scanning for them in the `argv[]` command line array since arrays are not supported by **dcc** yet. Finally, the last three programs calculate Fibonacci numbers, compute the cyclic redundancy check (CRC) for a character, and multiply two matrixes. The latter program was used to illustrate how array address computation is left in **dcc** in terms of an expression, rather than being analyzed and type propagated as an array.

Figure 5 presents summary results for the 10 programs considered in the test suite. The total number of intermediate instructions before the analysis is 963, compared with the final 306 intermediate instructions; which gives a reduction of instructions of 76.25%. This reduction of instructions is mainly due to the optimizations performed during data flow analysis, particularly the elimination of registers across subroutines and the reconstruction of high-level expressions. The recognition of idioms in the initial intermediate code also reduces the number of instructions and helps in the determination of data types such as long integers. Decompiled programs have the same number of user subroutines, plus any runtime support routines used in the program. These latter routines are sometimes translatable into a high-level representation; assembler is generated whenever they are untranslatable.

The percentage reduction rate represents the *code explosion* performed by the compiler when translating a high-level language to a machine language. This rate needs to be fully reverted in a decompiler if a high-level of abstraction is to be expected. Note that the reduction rate for a disassembler would be zero as the disassembler does not need to generate a higher representation, but merely translate the machine instructions into assembler (i.e. a 1:1 mapping).

Program	Before	After	% Reduction
intops	45	10	77.78
byteops	58	10	82.76
longops	117	48	58.97
benchsho	101	25	75.25
benchlng	139	28	79.86
benchmul	88	12	86.36
benchfn	82	36	56.10
fibo	78	15	80.77
crc	171	38	77.78
matrixmu	84	11	86.90
total	963	306	76.25

Figure 5: Results for test suite programs

4 Example

We present the decompilation of `benchmul`, a benchmark program from the Plum-Hall benchmarks. This program benchmarks integer multiplication by executing 1000 multiplications in a loop. The disassembly program is shown in Figure 7, the decompiled C program in Figure 8, and the initial C program in Figure 9.

Benchmul makes use of two long variables to loop a large number of times through the program, and three integer variables that perform the operations; one of these variables is not actually used in the program. As seen from the disassembly, the long variables are located on the stack at offsets `-4` and `-8`, and the integer variables are at offsets `-12`, `-10`, and on the register variable `si`. The final C code is identical to the initial C code, and a reduction of 86.36% of instructions was achieved by this program, as seen in Figure 5.

5 Comparison with previous work

Decompilers have been used since the 1960s for a variety of purposes; these are summarized in Figure 10. In the 60s, decompilers were used to help in the migration of 2nd to 3rd generation machines (e.g. the Neliac decompiler [19]) and in the conversion of languages such as Autocoder to Cobol (reported in [20]).

In the 70s decompilers were used in the automatic migration of programs, with systems such as Piler which attempted to support a large number of source and target languages [21], the design of a language for an aircraft system [22], and several theses described different methodologies for decompilation; the most complete ones are [20, 23].

```
      main  PROC  NEAR
     PUSH          bp
     MOV           bp, sp
     SUB           sp, 0Ch
     PUSH          si
     LEA           ax, [bp-4]
     PUSH          ax
     MOV           ax, 194h
     PUSH          ax
     CALL  near ptr scanf
     POP           cx
     POP           cx
     PUSH  word ptr [bp-2]
     PUSH  word ptr [bp-4]
     MOV           ax, 198h
     PUSH          ax
     CALL  near ptr printf
     ADD           sp, 6
     LEA           ax, [bp-0Ch]
     PUSH          ax
     MOV           ax, 1B2h
     PUSH          ax
     CALL  near ptr scanf
     POP           cx
     POP           cx
     LEA           ax, [bp-0Ah]
     PUSH          ax
     MOV           ax, 1B5h
     PUSH          ax
     CALL  near ptr scanf
     POP           cx
     POP           cx
     MOV   word ptr [bp-6], 0
     MOV   word ptr [bp-8], 1
L1:  MOV           dx, [bp-6]
     MOV           ax, [bp-8]
     CMP           dx, [bp-2]
     JL            L2
     JG            L3
     CMP           ax, [bp-4]
     JBE           L2
L3:  PUSH  word ptr [bp-0Ch]
     MOV           ax, 1B8h
     PUSH          ax
     CALL  near ptr printf
     POP           cx
     POP           cx
     POP           si
     MOV           sp, bp
     POP           bp
     RET
L2:  MOV           si, 1
L4:  CMP           si, 28h
     JLE           L5
```

Figure 6: Benchmul.a2

In the 80s decompilers were written to recover inaccessible source code, such is the case of Decomp [24], and in the documentation of assembler code migrated from one machine to another [25]. Sample decompilers written in the 90s were previously mentioned in §1.1.

```
     ADD   word ptr [bp-8], 1
     ADC   word ptr [bp-6], 0
     JMP           L1             ;Synthetic inst
L5:  MOV           ax, [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MUL   word ptr [bp-0Ch]
     MOV           dx, 3
     MUL           dx
     MOV           [bp-0Ch], ax
     INC           si
     JMP           L4             ;Synthetic inst
       main  ENDP
```

Figure 7: Benchmul.a2 – Continued

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 source assembly files [20, 22, 23, 25] or object files with or without symbolic debugging information [24], simplified compiler source high-level language used to compile the source executable program fed into the decompiler [20], and the requirement of the compiler's specification [8, 9].

Our decompilation system, **dcc**, differs from previous decompilation projects in several ways; it analyses source executable programs rather than assembler or object files, performs idiom 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 regenerate high-level expressions, and structures the program's control flow graph into a generic set of high-level structures that can be accommodated into different high-level languages, eliminating as much as possible the use of the

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

#include "dcc.h"

void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;
long loc3;
long loc4;
int loc5;

   scanf ("%ld", &loc4);
   printf ("executing %ld iterations\n", loc4);
   scanf ("%d", &loc1);
   scanf ("%d", &loc2);
   loc3 = 1;
   while ((loc3 <= loc4)) {
       loc5 = 1;
       while ((loc5 <= 40)) {
           loc1 = (((((((((((((((((((((((((loc1
                  * loc1) * loc1) * loc1) * loc1)
                  * loc1) * loc1) * loc1) * loc1)
                  * loc1) * loc1) * loc1) * loc1)
                  * loc1) * loc1) * loc1) * loc1)
                  * loc1) * loc1) * loc1) * loc1)
                  * loc1) * loc1) *
                  loc1) * loc1) * 3);
           loc5 = (loc5 + 1);
       }
       loc3 = (loc3 + 1);
   }
   printf ("a=%d\n", loc1);
}
```

Figure 8: Benchmul.b

goto statement. **dcc** also makes use of compiler and library signatures, and library prototypes, if available at decompilation time.

```
/* benchmul - benchmark for  int multiply
 * Thomas Plum, Plum Hall Inc, 609-927-3770
 * If machine traps overflow, use an unsigned type
 * Let T be the execution time in milliseconds
 * Then average time per operator =  T/major usec
 * (Because the inner loop has exactly 1000
 * operations)
 */
#define STOR_CL auto
#define TYPE int
#include <stdio.h>

main (int ac, char *av[])
{  STOR_CL TYPE a, b, c;
   long d, major;

    scanf ("%ld", &major);
    printf("executing %ld iterations\n", major);
    scanf ("%d", &a);
    scanf ("%d", &b);
    for (d = 1; d <= major; ++d)
    {
        /* inner loop executes 1000 selected
           operations */
        for (c = 1; c <= 40; ++c)
        {
            a = 3 *a*a*a*a*a*a*a*a * a*a*a*a*a*
                a*a*a * a*a*a*a*a*a*a*a * a;
                                     /* 25 * */
}
    }
    printf("a=%d\n", a);
}
```

Figure 9: Benchmul.c

	Use	Example
60	Migration: 2nd → 3rd Language conversion	Neliac Autocoder
70	Migration Language design Methodology description	Piler System F4 trainer aircraft 4 theses
80	Recover inaccessible code Modify binaries Documentation	Decomp Zebra
90	Migration Security Automatic generation Maintenance	Binary translation PROM comparator Decompiler-compiler dcc

Figure 10: Uses of decompilation throughout the decades

6 Conclusions

This paper presents an overview of a set of tools used in the reverse engineering of executable programs; such tools are: loader, disassembler, signature generator, prototype generator, and decompiler. We report on results obtained from the implementation of this system in a prototype decompiler system for the Intel i80286 architecture and the DOS operating system: **dcc**. The structure of the **dcc** decompiler is based on the structure of a compiler; several phases which are implemented in machine- or language- dependent or independent modules: the front-end which is machine-dependent, the universal decompiling machine which is machine- and language-independent, and the back-end which is language-dependent.

The results reported in this paper are based on a test suite of programs that make use of base data type variables such as byte, integer, and long integer. The statistics obtained show that on average there is a reduction in the number of intermediate instructions of 76.25%. This reduction is only possible by the complete data flow analysis of the program, and it represents the overall improvement this decompiler has made over previous decompilers which generate "high-level code" that looks like assembler (and hence their reduction rate is almost nil).

The resultant C programs make use of variables (as opposed to registers) and the correct type of control structures (with the minimum use of `goto` statements). Some comments are also provided by the system. The decompiler generates assembler code for untranslatable subroutines, and C for the rest. The decompiler makes use of compiler and library signatures if they are available; otherwise it disassembles all code and analyses all subroutines (including low-level subroutines included by the compiler for setting up the environment, and all library routines linked in by the linker).

Acknowledgements

I would like to thank Michael Van Emmerik for coding the signature and prototype generators, and for modifying the disassembler, and Professor John Gough for discussions of decompilers. This research is partly supported by Sun Microsystems Laboratories. For further information refer to
http://crg.cs.utas.edu.au/

References

[1] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7:13–17, January 1990.

[2] K. Lano, P.T. Breuer, and H. Haughton. Reverse-engineering cobol via formal methods. *Journal of Software Maintenance: Research and Practice*, 5(1):13–35, March 1993.

[3] J. Mallory. Reverse engineering of fortran programs now possible. *Newsbytes*, 16 July 1991.

[4] S.E. Varney. IDE adds reverse engineering, code generation to C package. *Digital Review*, 8(25):9–10, 24 June 1991.

[5] R Swanke. The art of reverse engineering. *Computer*, June 1991.

[6] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[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. From programs to object code and back again using logic programming: Compilation and decompilation. *Journal of Software Maintenance: Research and Practice*, 5(4):205–234, 1993.

[9] P.T. Breuer and J.P. Bowen. Decompilation: the enumeration of types and grammars. *Transaction of Programming Languages and Systems*, 16(5):1613–1647, September 1994.

[10] P.T. Breuer and J.P. Bowen. Generating decompilers. To appear in Information and Software Technology Journal, 1994.

[11] C. Fuan, L. Zongtian, and L. Li. Design and implementation techniques of the 8086 C decompiling system. *Mini-Micro Systems*, 14(4):10–18,31, 1993.

[12] C. Cifuentes and K.J. Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, July 1995.

[13] P. Bailes, M Chapman, M Chia, and I Peake. Generic re-engineering environment design criteria: An evaluation of Software Refinery (TM). *Australian Computer Journal*, 26(4):151–157, November 1994.

[14] R.N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1979.

[15] H. Swartz. The case for reverse engineering. *Business Computer Systems*, 3(12):22–25, December 1984.

[16] 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.

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

[18] E.S. Raymond. Plum-hall benchmarks. URL: ftp//plaza.aarnet.edu.au/usenet/comp.sources.unix/ volume20/plum-benchmarks.gz, 1989.

[19] M.H. Halstead. *Machine-independent computer programming*, chapter 11, pages 143–150. Spartan Books, 1962.

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

[21] P. Barbe. The Piler system of computer program translation. Technical report, Probe Consultants Inc., September 1974.

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

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

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

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