A METHODOLOGY FOR MACHINE LANGUAGE DECOMPILATION

by

Barron C. Housel*
IBM Research Laboratory
San Jose, California

Maurice H. Halstead
Purdue University
Lafayette, Indiana

Machine language decompilation is the translation of machine (assembly) language instruction sequences into statements in a high-level algebraic language such as PL/1. This process can be viewed as the inverse of compilation. Decompilation can be used as an aid for program conversion and program documentation. A general methodology for decompilation that is independent of a particular source and target language is presented. The basic approach is to map the source machine language to a high-level representation, which is relatively machine and language independent, and then translate to the chosen target language. An experimental decompiler was implemented to translate Knuth's MIXAL assembly language into PL/1.

Key Words and Phrases: Decompiling, Inverse Translation, Machine Language Translation, Optimization, program documentation.

CR Categories: 4.12, 4.43, 5.24.

1. INTRODUCTION

Machine language decompilation is the translation of machine or assembly language into a high-level machine independent language. This process can be viewed as the inverse of compilation.

Decompilers were written as early as 1960 [1], and yet there has been a conspicuous absence of articles describing decompilation technology. One reason for this is that most existing decompilers have been developed commercially [2,3], and are proprietary in nature. Also, many previous decompiler implementations were ad hoc, limiting the applicability of their techniques.

Decompilation is valuable for program conversion and program documentation. Although the increased popularity of high-level languages has lessened the conversion problem, these languages do not totally satisfy all computing needs. Furthermore, the number of machine language programs in the field represents a sizeable investment to be protected. As a documentation aid, decompiling enables a machine language program to be translated to a higher level representation, thus increasing the clarity of the program logic and reducing the task of reprogramming and maintenance. This is especially helpful when the programs are old and their documentation is nonexistent or obsolete.

Past efforts in decompilation [4,5] have shown that decompiling is in general an incomplete process. While it is theoretically possible to decompile an arbitrary program completely, assuming its environment is available, it is generally conceded that it is economically infeasible to do so because of the large number of instruction sequence combinations. Prior translation techniques have consisted largely of classifying common types of code sequences (e.g. arithmetic sequences) and providing the appropriate translation rules. Source code sequences which did not qualify for automatic translation were handled manually. If it was learned by experience that a particular construct occurred frequently, then the decompiler was extended to handle another "special case." The use of this ad hoc approach to achieve total automatic translation is clearly infeasible.

Based on the above considerations, it seems that the development of general principles and methodologies for decompilation is both timely and relevant. There is a need for a systematic approach which is applicable to a large class of source-target language pairs and for more general translation techniques which will reduce the "special case" problem.

The philosophy of our methodology is to map the source machine language up to a high-level representation, which is relatively machine and language independent, and then to translate down to the chosen target language. A generalized view of this process is given in Figure 1.

*This work was done as a Ph.d. thesis at Purdue University.

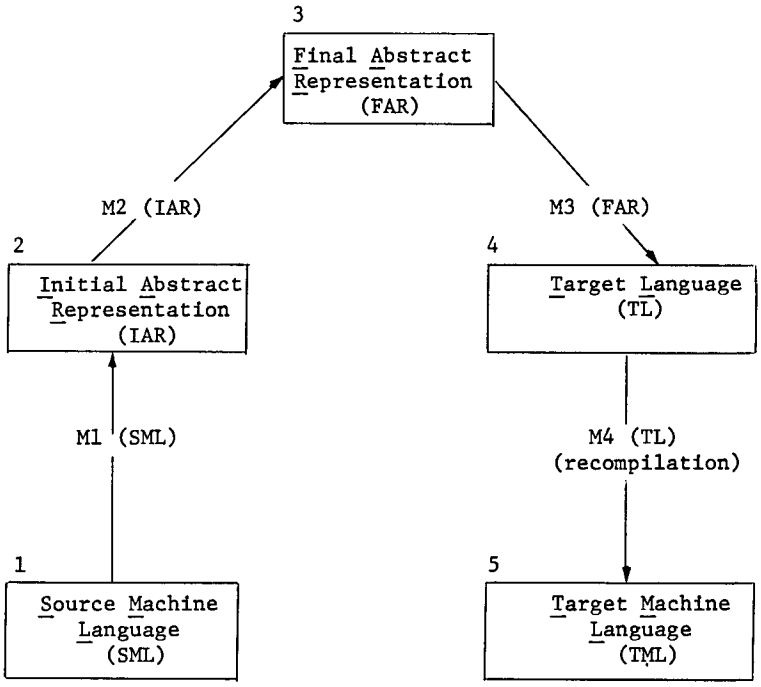

Figure 1 - The Decompilation Process Model

An example of an SML would be the IBM 7094 MAP language. The IAR (block 2) produced by M1 (SML) consists of an intermediate language and the control flow graph (CFG) of the source program. These components are analyzed (M2(IAR)), using decompilation analysis techniques, in order to detect data structures and to simplify and reorganize the program. This "optimized" version of the program (FAR) is still in intermediate language form and is the domain for applying the translation mappings (M3(FAR)). This results in statements in a high-level target language (TL) such as PL/1. These statements can then be "recompiled" (M4(TL)) to produce the target machine language (TML) statements. This effects the conversion of a machine language program for the source machine (e.g. IBM 7094) to an equivalent version for the desired target machine (e.g. IBM/360). The decompilation process consists of the mappings M1, M2, and M3, which are discussed below.

2. DETECTING DATA AND INSTRUCTIONS

One of the basic steps in decompilation of the source program P is to determine the set of instructions I(P) and the set of data D(P) referenced by I(P), where P consists of a sequence of statements (i.e. assembler or object deck) $S[1],\cdots,S[n]$, an entry point EP(P), and the memory extent M(P). The complications that arise when P contains self-modifying code or dynamically computed JUMP instructions are detailed elsewhere [6].

If the assembly language text is available, one approach for distinguishing instructions from data would be to scan all S[k] and check the opcodes. In general this is insufficient for the following reasons:

(1) Self-modifying code may exist (i.e. $I(P) \cap D(P) \neq 0$).

(2) Portions of the code and data may no longer be used. This sometimes occurs after a period of time due to maintenance activity.

(3) Data structures are not readily apparent simply by examining the data storage declarations alone.

(4) In order to do sophisticated analysis, it is necessary to determine the control flow graph of the program (CFG(P)).

(5) Frequently, the assembly text is not available, and it is not immediately discernible whether a word of object text is an instruction or assembled data.

Our method consists of scanning P, beginning with its entry point in order to partition I(P) into a set of instruction blocks: B(1),•••,B(m), where each B(k) has the following properties:

(1) Let A(I[k]) denote the address of I[k]. Then B(k) consists of an instruction sequence I[j],I[j+1],•••,I[q], such that A(I[i+1]) = A(I[i])+1, for i=j,j+1,•••,q-1.

(2) B(k) has one and only one entry point.

(3) B(k) terminates with:

a) an absolute "JUMP" instruction, or a conditional JUMP [17] sequence.

b) the entry point of an adjacent block B(j) (i.e. A(I[q])+1 = A(first instruction of b(j))).

Conditions 3a and 3b define the boundary conditions of instruction blocks. The next block to be scanned is determined by selecting a block entry point from the unscanned block list. This list is generated by recording the jump instruction references during prior block scans. Property 3b describes a "fall through" condition where if B(i) falls through to B(j), then the execution of B(i) implies execution of B(j), but the converse is not necessarily true.

During the block detection process the interrelationships among the blocks can also be recorded by associating a set of immediate successors (IS) and immediate predecessors (IP) for each B(i), similar to the technique described by Lowry and Medlock [7]. This, in effect, defines the control flow graph (CFG(P)) [8], where the instruction blocks constitute the nodes and the control flow paths define the directed arcs. The result of the block detection process is a block table, an example of which is given in Table 1 for the program in Figure 2. A CFG representation of Table 1 is given by Figure 3.

```
i          S[i] (Knuth's MIXAL [9])
-----------------------------------

     *  ADD ARRAYS X AND Y:  X = X+Y
     X       EQU   100
     Y       EQU   200
1    START   ENT3  0
2    LOOP    CMP3  =100=
3            JGE   DONE
4            LDA   X,3
5            ADD   Y,3
6            STA   X,3
7            ENT3  1,3
8            JMP   LOOP
9    DONE    HLT
             END   START
```

Figure 2 - Simple MIXAL Program.

The following block table would be produced:

B(i)	EXTENT[B(i)]	IS	IP
1	1:1	2	-
2	2:3	3,4	1,3
3	4:8	2	2
4	9:9	-	2

Table 1
Block Table of Simple MIXAL Program

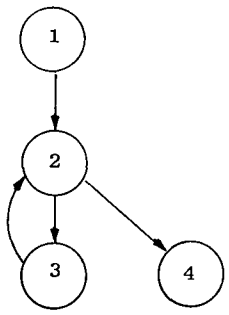

Figure 3 - CFG of Simple MIXAL Program.

3. INTERMEDIATE LANGUAGE (IL) GENERATION

The second phase of the mapping M1 (Figure 1) is to translate the instructions of P into an intermediate language. An intermediate language can be selected to realize a generalized decompiler system. For example, the PILER system [10] expresses a large class of machine languages using a very low (micro) level intermediate language. The rationale for our intermediate language (IL) has not been generality but a representation which is most suitable for program analysis, simplification, and reorganization. However, because IL is inherently a less machine dependent representation of the source program, it has provided a sufficient basis for expressing a restricted class of machine languages, namely, those for mini-computers (12-16 bit words). These restrictions are due to implementation considerations. Theoretically, it should not be difficult to provide an intermediate language that is sufficient to handle a wider class of machine architectures. Friedman and Schneider [11] have extended IL in order to decompile IBM 1130 assembler language to their language for mini-computer systems [12].

To achieve the above objectives, IL was designed to have the following properties:

(P1) All operands in IL are explicitly referenced.

(P2) All operands are treated in a uniform manner.

(P3) The representations of I(P) and D(P) are disjoint (i.e. not bound to same address space). This is achieved by making operands in I(P) refer to data tables.

(P4) The physical order of any two instruction blocks is independent of their original order in P.

Applying these properties in the translation process for the program in Figure 2 into a 3-address code IL representation would result in Figure 4.

i	S'[i]	Blk of S'[i]
1	ASSIGN IR3,ZERO	1
2	JUMP ,[B(2)]	1
3	CMP CI,IR3,=100=	2
4	JUMP GE,CI,[B(4)]	2
5	JUMP ,[B(3)]	2
6	ASSIGN AC,X(IR3)	3
7	ADD AC,AC,Y(IR3)	3
8	ASSIGN X(IR3),AC	3
9	ADD IR3,IR3,ONE	3
10	JUMP ,[B(2)]	3
11	STOP	4

Figure 4 - IL Representation of Simple MIX Program.

In Figure 4 all operands (except the "GE" condition in 4) represent pointers to operand tables. IRk, AC, and CI correspond to index register k, the MIX "A" register, and the condition indicator respectively. [B(k)] designates the address of block k. The index (register 3) associated with X(IR3) and Y(IR3) is treated as an attribute of the operands X and Y and is stored in their respective operand table entries.

The following observations clarify the above properties. First, all implied jumps (in B(1) and B(2) of the example) have been made explicit by adding absolute jump instructions (P4). This allows B(i) to be reorganized during translation to the target language (redundant jumps are eliminated at this time). Second, no operands are implicitly defined by the opcodes (P1). This reduces the number of unique IL opcodes required and provides a basis for simplifying the text (i.e. combining instructions). Third, because I(P) and D(P) are disjoint (P3), instructions can be added or deleted to I(P) without causing relocation difficulties.

4. PROGRAM ANALYSIS AND SIMPLIFICATION

At this stage, the mapping M1 (Figure 1) is complete. The next stage (M2) is to perform analyses needed to determine the program data structures and the instruction sequences which can be combined to form high level target statements. Many of these analysis techniques are substantially indebted to the framework established by program optimization theory as set forth by Lowry and Medlock [7], Allen [8,13], Allen and Cocke [14], Cocke and Schwartz [15], and Frailey [16]. Determining data structures involves the detection of all strongly connected regions (e.g. loops) of CFG(P) in conjunction with the generation of computation graphs in order to analyze data flow and compute array bounds, data types, et cetera. With regard to the example program, the analysis of CFG(P) would detect loop (2,3)(Figure 3). Upon analyzing the data flow in (2,3), it is learned that IR3 serves as an iteration control variable for the loop and also as an index for dynamic data storage references. Further analysis would reveal that two 100 word, disjoint arrays (X and Y) are implicitly defined.

It would also be concluded that lines 6, 7, and 8 (Figure 4) could be combined to form:

```
ADD    X(IR3),X(IR3),Y(IR3)
```

The result is to completely eliminate reference to the working storage register AC, thus, removing a constraint imposed by the machine architecture. In general, to combine instructions and eliminate "temporary" operands involves a global study of the operand usage, often referred to as "busy" analysis [7], in order to guarantee equivalence of the simplified program with the original. A variable V is busy at location L in P if it is fetched before it is redefined along some control path emanating from L.

5. TARGET LANGUAGE (TL) GENERATION

The last mapping (M3) of the decompilation process is that of applying translation rules to IL to generate TL. The IL->TL translation rules are a function of the target language itself and the level of translation desired. One aspect of the TL generation involves the structural translation of the program (interblock relationships). Another phase concerns translating computational instruction sequences within B(i).

If PL/1 is used as the target language, one reasonably high level translation would be that of mapping the original control flow into a block structured representation using PL/1 "DO-groups". For example, when analyzing a conditional jump sequence (e.g. lines 3,4,5, Figure 4) of some B(k), it may be possible to generate DO-group constructs for the various alternatives of an "if-then-else" construct.

If IS[B(k)] is greater than 1, then B(k) must be bounded by a sequence of conditional jump instructions (which define IS[B(k)]). Let SG[n] denote the subgraph referenced by B(k). If IP[SG[n]]=B(k), then SG[n] [18] may be translated as a DO-group, which represents the action to be taken if the corresponding JUMP in B(k) is executed. Since SG[n] may also contain DO-groups, the procedure is recursive and the resulting translation may be a nested block structure, resulting in a more structured representation, frequently with many fewer transfer instructions than the orignal P.

General algorithms have been developed [6] for translating arithmetic and relational expressions into high level statements in the target language. This is done by eliminating intermediate operands whenever possible in order to combine subexpressions. The result is a tree representation of the expression, which can later be translated into an appropriate infix expression in the target language. Consider the following IL sequence and assume that no operands are busy past IL[5] (i.e. busy on exit).

k	IL[k]	
1	ADD	T1,B,C
2	MUL	T2,T1,D
3	SUB	T1,T1,E
4	DIV	T3,T2,G
5	ADD	X,T3,T1

Figure 5 - IL Arithmetic Expression.

If we try to combine instructions subject to the operand usage constraints described in [6], we discover that IL[3] and IL[5] can be combined by eliminating T1. This is realized by replacing T1 in IL[5] by a pointer to IL[3] and replacing T1 in IL[3] with a "null" operand. Now, IL[3] is treated as being a subexpression which defines the third operand for IL[5].

```
3    SUB  null,T1,E          3    NOP
     ...            =>       ...
5    ADD  X,T3,(3)           5    ADD   X,T3,(Sub T1,E)
```

The operand (3) can be viewed as a nonterminal operand. A complete reduction of Figure 5 would result in:

```
1    ADD  null,B,C
2    MUL  null,(1),D
3    SUB  null,(1),E
4    DIV  null,(2),G
5    ADD  X,(4),(3)
```

Figure 6 - Reduced IL Arithmetic Expression.

The expansion of the nonterminal operands, beginning with IL[5], results in an expression tree, which is easily converted into the following polish postfix expressions:

=X+/*+BCDG-+BCE.

Using the usual arithmetic operator precedence relations and the algebraic properties of the operators, the above expression can be converted to an infix expression with nonredundant parentheses as shown below:

X=(B+C)*D/G+B+C-E.

A second class of translation rules are heavily dependent on the source machine languages. One useful approach is to develop a set of machine language idioms. Gaines [19] defines an idiomatic expression as "a sequence of instructions which form a logical entity and which cannot be derived by considering the primary meaning of the instructions." For example, one such idiom would be incrementing the exponent of a binary floating point number to effect multiplication by powers of two. Idioms differ from the "special case" situations in that they comprise a repertoire of programming techniques commonly employed by programmers of the particular machine language.

The current state of the art has not completely eliminated the "special case" problem. However, the extent of automatic translation can generally be increased if one is willing to sacrifice level of the target translation (e.g. m to n instead of m to 1). The reduction of translation level implies that the target language must be capable of expressing lower level semantic constructs (e.g. shift, mask, etc.). This capability is usually found in systems programming languages but is often not included in the more common algebraic languages (e.g. Fortran). A high-level PL/1 translation of Figure 4 is given in Figure 7.

```
SIMPLE_MIXAL:  PROCEDURE;
               DCL  (X(0:99), Y(0:99)) BINARY FIXED;
               DCL  IR3 BINARY FIXED INIT(0);
          LOOP: DO IR3=0 TO 99;
               X(IR3)=X(IR3)+Y(IR3);
               END LOOP;
          END SIMPLE_MIXAL:
```

Figure 7 - PL/1 Translation of Simple MIXAL Program.

A higher level translation would be to recognize that the iterative DO-group is equivalent to X=X+Y.

6. CONCLUSIONS

While no attempt has yet been made to use this methodology in a sophisticated production model, it has been tested by programming a simple (5 man-month) experimental version. The chosen source and target languages of the decompiler are Knuth's [9] MIX assembly language (MIXAL) and PL/1 respectively. Some measure of the effectiveness of this simple decompiler can be obtained from the following: Six algorithms in MIXAL as published by Knuth [9] were converted automatically to PL/1 programs, manually edited, and verified by execution. Two of the six algorithms required no manual editing. In general, 88% of the

statements were correct as produced, and the remaining 12% were readily corrected without reference to the intent of the algorithm. An exception occurred in one case, in which an examination of the PL/1 version revealed that the published MIXAL version lacked an essential line.

As expected, the 88% value obtained is below results obtained with ad hoc, proprietary decompilers. The important point, however, is that the methodology described has demonstrated a more solid, general purpose base from which one may proceed to attach further problems not discussed here.

REFERENCES

[1] M. H. Halstead, Machine Independent Computer Programming, Spartan Books, Washington D.C., 1962.

[2] M. H. Halstead, Machine Independence and Third Generation Computers, Proceedings FJCC, 1962, 587-592.

[3] IBM, 1400 Autocoder to COBOL Conversion Aid Program (360A-SE-19X), White Plains, New York, 1967.

[4] M. H. Halstead, Using the Computer for Program Conversion, Datamation, May 1970, pp. 125-129.

[5] William A. Sassaman, A Computer Program to Translate Machine Language into Fortran, Proceedings SJCC, 21966, pp. 235-239.

[6] Barron C. Housel, A Study of Decompiling Machine Languages into High-Level Machine Independent Languages, CSD TR 100, Purdue University (Thesis), 1973.

[7] E. S. Lowry,C. W. Medlock, Object Code Optimization, CACM, Vol. 12, No. 1, January 1962, 13-22.

[8] F. E. Allen, Program Optimization, Annual Review in Automatic Programming, Vol. 5, Pergamon, 1969, pp. 239-307.

[9] D. E. Knuth, The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading Massechusetts, 1969.

[10] P. Barbe, Techniques for Automatic Program Translation, Software Engineering, Vol. 1, Academic Press Inc., 1970, pp. 151-165.

[11] Frank Friedman, Private Communications, Purdue University, 1973.

[12] F. L. Friedman, V. S. Schneider, A Programming Language for Mini-Computer Systems, Sigplan Notices, Vol. 9, No. 1, January 1974.

[13] F. E. Allen, Control Flow Analysis, Sigplan Notices, Vol. 5, No. 7, July 1970, pp. 1-19.

[14] F. E. Allen, J. Cocke, Graph-theoretic constructs for Program Control Flow Analysis, IBM Report RC3923, Yorktown Heights, July 1972.

[15] Cocke and Schwartz, Programming Languages and their Compilers (preliminary notes), Courant Institute of Mathematical Sciences, New York University, 1969.

[16] Dennis J. Frailey, Expression Optimization Using Only Complement Operators, Sigplan Notices, Vol. 5, No. 7 (July 1970) pp. 67-85.

[17] A jump sequence which terminates an instruction block is subject to a number of constraints as discussed in Reference 6.

[18] SG[n] may be a single block or a strongly connected region (SCR) subject to certain constraints as described in Reference 6.

[19] R. S. Gaines, On the Translation of Machine Language Programs, CACM, Vol. 8, No. 12, December 1965, 769-773.