2016 5th IIAI International Congress on Advanced Applied Informatics

Reverse Engineering from Mainframe Assembly to C Codes in Legacy Migration

Daisuke Fujiwara and Nagisa Ishiura
School of Science and Technology
Kwansei Gakuin University
2–1 Gakuen, Sanda, Hyogo, 669–1337, Japan

Ryo Sakai, Ryo Aoki, and Takashi Ogawara
SYSTEM'S Co., Ltd.
7–24–5 Nishigotanda
Shinagawa-ku, Tokyo, 141–0031 Japan

***Abstract*—This paper presents a method of constructing C programs from mainframe assembly programs. IBM mainframe assembly programs, which are called as subroutines from programs written in high-level language such as COBOL, are automatically translated into equivalent C programs. The assembly programs are converted into intermediate representation (IR) of the SSA form on which dataflow analysis, recognition of control structures, and pattern match based transformation are applied to produce codes with readability. Our method features documentation of the translation process. Along with translation, correspondence between the source assembly codes and the resulting C codes are generated as documents, which plays very important role in manually correcting incomplete C codes from architecture dependent codes or self morphing codes. Furthermore, comments in the assembly programs are embedded into appropriate positions in the resulting C programs. A prototype system based on our method successfully translated some assembly codes into C program with function, if, and do-while structures.**

I. Introduction

In many business and enterprise systems, mainframe computers have long been used as core computing systems, for their high reliability and fault tolerance. In recent years, however, as lower cost open systems such as Linux and Windows servers have gained higher performance, there has been motivation to move from the legacy systems to the open systems. Since the legacy systems accumulate business know-how of many years, it is often a rational choice to port the existing systems to operate on the open systems than to re-develop equivalent systems. Such kind of porting is called "legacy migration," on which there is a growing demand.

One of the core technologies in legacy migration is porting of programs on the mainframe computers, written in COBOL, PL/1, assembly languages, etc., to run on the modern systems. Programs written in high-level languages may work by re-compilation, or they can be auto-converted [1] or auto-corrected to run on the target systems. Since they are relatively easy to understand, manual modification on the resulting codes is also easy. On the other hand, assembly programs needs manual translation to some languages, after understanding their behavior. Although those assembly programs are usually small, a single legacy system may contain hundreds, or sometimes thousands of assembly codes, which needs enormous man-hours for migration.

To solve this problem, there have been several attempts on automated conversion of assembly codes to high-level codes. Literatures [5], [6], [7] have presented methods of translating IBM mainframe assembly codes into to C programs. Assembly codes are converted into intermediate representation, to which reverse engineering such as reconstruction of control structures are performed to generate C codes with reasonable readability.

However, it should be noted that static translation does not always succeed; architecture specific codes and self morphing codes may not always be converted to correct codes. In such cases, the resulting codes must be first inspected to see what went wrong and be modified manually. In this situation, resulting C codes alone are not enough; it is very important that the process of translation or correspondence between original assembly code fragments and the resulting code sequence should be well documented.

This paper proposes a method of translating IBM assembly programs to equivalent C programs, with emphasis on generating auxiliary information to understand the resulting C codes and the conversion process. C codes with high readability are generated via intermediate representation extracted from the given assembly codes, in the similar way as [5], [6], [7]. At the same time, a table to show the correspondence between the assembly instructions and the resulting C code fragment is generated. Furthermore, the comments in the assembly codes, which hold important information to understand the program such as the authors of the codes, how the codes should be used, the intent of each code fragment, etc., are embedded into the proper position of the resulting C programs.

The tool based on the proposed method is implemented in Perl5, which has successfully converted some assembly programs consisting of about 100 lines into working C programs.

II. Migration of Assembly Programs

A. Target of Conversion

In this paper, we deal with the problem of converting hand written assembly programs of the IBM 370–390 mainframes into C programs, for there is a big demand for migration from this architecture. As shown in Fig. 1, we assume that the assembly programs are called as subroutines from the other programs written in assembly, COBOL, etc. We also assume that C libraries equivalent to the library routines and macros called from the assembly programs are already prepared.

The first priority in this kind of conversion is that the resulting C programs should work correctly. This needs the same technology as binary translation [2]. The second priority is readability of the reconstructed C programs. This is because the new programs must be maintained, or sometimes be

978-1-4673-8985-3/16 $31.00 © 2016 IEEE
DOI 10.1109/IIAI-AAI.2016.37

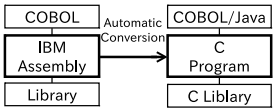


Fig. 1. Target of conversion.

debugged, on the new platforms. This needs the technologies used in decompilation [3], [4].

There have been several attempts to convert mainframe assemblies to higher-level programs. Feldman [5] translates IBM assemblies to C programs via an intermediate representation named HLL. Ward [6], [7] makes use of a formal intermediate model FermaT to generate C programs with high readability from IBM assembly codes.

However, automatic translation does not always succeed. Since the character codes used on the mainframe computers and the open systems are different, routines that directly manipulate character bit patterns may not be converted to intended programs. Differences of the address spaces and addressing conventions, such as the use of the most significant bit on the IBM mainframes, cause the same problems. Many unexpected coding techniques are often used in hand written assemblies. Furthermore, it is impossible to convert self morphing codes completely by static translation.

Thus, in a practical point of view, resulting codes must be investigated to see if the translation was correct or to see what went wrong. For this purpose, documentation of how codes are translated would be very important. At the same time, comments in the original assembly codes would be very helpful, for they describe the intentions behind the codes or give explanations to complicated logic.

B. IBM Assembly

The IBM mainframes have 32bit architecture. It has 16 general purpose registers numbered 0 through 15. It deals with 32bit, 16bit, and 8bit binaries as well as character strings and packed/zone decimals. The instruction set consists of 631 instructions with 0, 1, 2, or 3 operands.

An example of the IBM assembly program is shown in Fig. 2. This code defines a subroutine named EMONTH which receives the address to a date in YYYYMMDD form as the first parameter, and write the last day of the month of the date in the same form at the address passed as the second parameter.

```
 1: EMONTH   CSECT                                  00000000
 2:          USING EMONTH,12                        00000100
 3:          STM   14,12,12(13)                     00000200
 4:          LA    12,0(,15)                        00000300
 5:          LA    15,SAVE                          00000400
 6:          ST    13,4(,15)                        00000500
 7:          ST    15,8(,13)                        00000600
 8:          LR    13,15                            00000700
 9:          LM    3,4,0(1)                         00000800
10:          CLC   0(6,4),=C'00000000'              00001100
11:          BNE   ER_FMT                           00001200
12:          B     FC_EOM                           00001300
13: ER_FMT   EQU   *                                 00001400
14:          ABEND 0999,DUMP                        00001500
15: FC_EOM   BAL   14,EOM                           00001600
16: EOM      EQU   *                                 00001700
17:          MVC   0(8,4),=CL9' '                   00001800
18:          LA    0,6                              00001900
19:          LR    1,3                              00002000
20: EOM1     EQU   *                                 00002100
21:          CLI   0(1),C'0'                        00002200
22:          BL    RET                              00002300
23:          CLI   0(1),C'9'                        00002400
24:          BH    RET                              00002500
25:          LA    1,1(,1)                          00002600
26:          BCT   0,EOM1                           00002700
27:          CLC   4(2,3),=C'01'                    00003000
28:          BL    RET                              00003100
29:          CLC   4(2,3),=C'12'                    00003200
30:          BH    RET                              00003300
31:          PACK  WORK,4(2,3)                      00003400
32:          CVB   2,WORK                           00003500
33:          BCTR  2,0                              00003600
34:          MH    2,=H'4'                          00003700
35:          LA    5,MLAST                          00003800
36:          AR    2,5                              00003900
37:          LH    0,4(,2)                          00004000
38:          SH    0,0(,2)                          00004100
39:          CVD   0,WORK                           00004200
40:          UNPK  6(2,4),WORK                      00004300
41:          OI    7(4),X'F0'                       00004400
42:          MVC   0(6,4),0(3)                      00004500
43:          B     RET                              00004600
44: RET      EQU   *                                 00004700
45:          L     13,4(,13)                        00004800
46:          LM    14,12,12(13)                     00004900
47:          SLR   15,15                            00005000
48:          BR    14                               00005100
49: MLAST    DC    H'000',H'000'                    00005200
50:          DC    H'031',H'031'                    00005300
51:          DC    H'059',H'060'                    00005400
52:          DC    H'090',H'091'                    00005500
53:          DC    H'120',H'121'                    00005600
54:          DC    H'151',H'152'                    00005700
55:          DC    H'181',H'182'                    00005800
56:          DC    H'212',H'213'                    00005900
57:          DC    H'243',H'244'                    00006000
58:          DC    H'273',H'274'                    00006100
59:          DC    H'304',H'305'                    00006200
60:          DC    H'334',H'335'                    00006300
61:          DC    H'365',H'366'                    00006400
62:          LTORG                                  00006500
63: SAVE     DS    18F                              00006600
64: WORK     DS    D                                00006700
65:          DROP  12                               00006800
66:          LTORG                                  00006900
67:          SPACE                                  00007000
68:          END                                    00007100
```

Fig. 2. IBM assembly program.

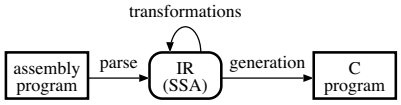


Fig. 3. Flow of translation process.

III. Conversion of Mainframe Assembly to C

A. Overview

The flow of our translation process is shown in Fig. 3. A given assembly program is compiled into an intermediate representation (IR) of the SSA (static single assignment) form, where each instruction is decomposed into atomic operations. Dataflow analysis and a various kind of transformation on the IR are performed to generate a resulting C program.

B. Intermediate Representation (IR)

The structure of the IR in our method is shown in Fig. 4. The root node *Assembly*, representing the assembly program under conversion, consists of a list of *Sections* and a *SymbolTable*. The *SymbolTable* keeps track of all the variable names, function names, label names, etc. in the program. The *Section* represents a section in the program, a group of instructions and data placed in contiguous storage locations, which has a list of *Functions*. The *Function* represents a subroutine and has a list of *Ifs*, *Loops*, and *BasicBlocks*. The *If* and *Loop* represent *if* and *do-while* structures having *then/else* parts and a body part, respectively. The *BasicBlock* represents a basic block consisting of a list of *Operations* and links to its next basic blocks. The *Operation* represents an atomic operation, such as arithmetic/logical, load/store, string and decimal operations.

Fig. 5 shows some examples of conversion from instructions to IR. A load instruction (`L`) is decomposed into a 32bit addition (`addu32`) to calculate the address and a 4-byte memory access operation (`load32`). String and decimal data manipulation to implement instructions like `AP` (add packed decimal) and `CLC` (compare logical character) are dealt with

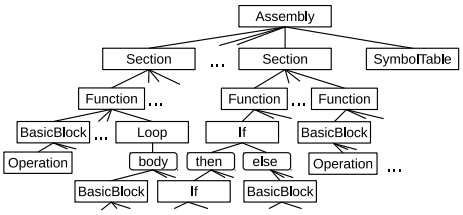


Fig. 4. Structure of IR.

assembly		IR
`L 11,4(,13)`	⇒	`addr = addu32(r13, 4)` `r11 = load32(addr)`
`AP DAT1,DAT2+5(3)`	⇒	`cc=addpack(DAT1, 8, DAT2, 8)`
`CLC 4(2,4),=C'01'`	⇒	`addr=addu32(r4, 4)` `cc=compstr(addr, C'01', 2)`

Fig. 5. Conversion from instructions to IR.

as atomic operations.

The lists of the operations translated from the instructions are converted to the SSA form. This normalizes different assembly code sequences of the same meaning to the same IR. The SSA form is also useful in unnecessary code elimination (in III-D) and transformation by pattern matching (in III-F). After the SSA conversion, dataflow analysis is performed and definition/reference relation is stored into the IR.

C. Recognition of Control Structures

1) Recognition of Functions: Firstly, all the entry points in the IR are identified. The entry points are either 1) the basic blocks starting with `nop` operations converted from `ENTRY` instructions, or 2) the basic blocks which are the targets of `jump` operations converted from `BAL`, `BALR`, and `BAS` instructions. A function is extracted by enumerating all the basic blocks reachable from each entry point. In hand written assembly codes, there are cases where a basic block is reachable from multiple entry points. For simplicity, such situation is averted by cloning the basic blocks.

2) Recognition of Loops: Do-while loops are recognized according to the following steps.

1) Enumerate all the loops by traversing basic blocks from the starting point of recognition (which is the entry point of the function in the first iteration).
2) Choose a loop, whose bottom basic block b is the farthest from the starting point and the top basic block t is the nearest to the starting point. This is to recognize the outermost loop first and to avoid jump into the loop.
3) Determine set B of the basic blocks that form the body of the loop, which are reachable from t and reachable to b (as shown in Fig. 6).
4) Extract the continuation condition of the loop from the branch condition of the bottom basic block b, and construct the data structure of the loop.
5) Apply this process to the other basic blocks in the function and the body of the loop until no loop is detected.

3) Recognition of Conditionals: *If* statements are recognized in the following steps.

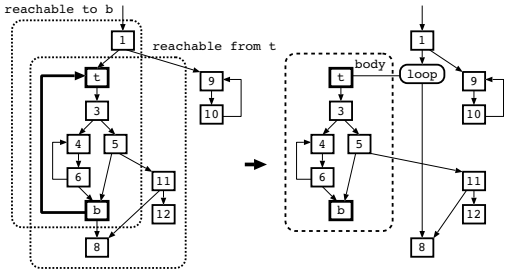


Fig. 6. Recognition of loop body.

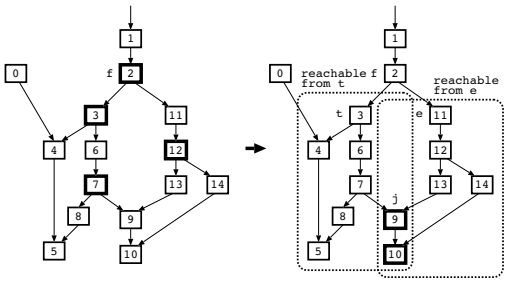


Fig. 7. Choice of branch and join nodes.

1) Choose a set of branch point f and join point j. By traversing basic blocks from the starting point of recognition (the entry point of the function in the first iteration), select the basic block f that has branch and is nearest to the starting point. Let t and e be the next basic blocks of f. Then, identify the joining basic block j that are reachable from both t and e and yet nearest to f (see Fig. 7). This is to avoid jump into *then* and *else* parts of the *if* statement. Note that there are cases where there is no joining point j.
2) Identify the sets of basic blocks T and E that belong to *then* part and *else* part, respectively, of the *if* statement. Let T' be the set of the basic blocks reachable from t. Let X be the set of basic blocks in T' that are directly reachable from the basic blocks in $\overline{T'}$, and X' be the set of the basic blocks reachable from X. Then T is defined as $T' \cup \overline{X'}$ (see Fig. 8). E is computed in the same way.
3) Extract the branch condition from the basic block f and construct the data structure of the *if* statement.
4) Apply this process to the other basic blocks in the function and the *then* part and *else* part of the *if* statement, until no branch is detected.

D. Elimination of Register Save/Restore and Dead Codes

Based on the result of the data flow analysis, codes for register save and restore of each function, which are no more necessary in C programs, are deleted. Store operation s of 32bit data is judged as save codes and is removed from the IR if 1) the address operand of s is obtained by adding offset of 0, 4, $\cdots$, or 68 to register 13, and 2) no operation defines the data operand of s. Similarly, 32bit load operation l is regarded as restore codes and is eliminated if 1) the address operand of l is obtained by adding offset of 0, 4, $\cdots$, or 68 to register 13, and 2) no operation uses the result of l.

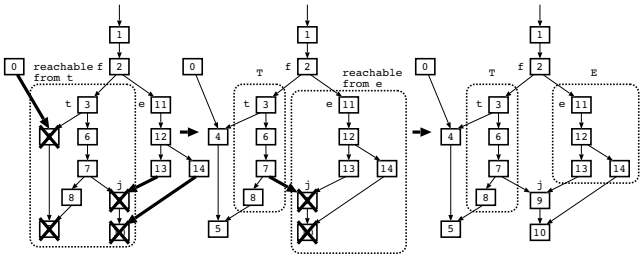


Fig. 8. Recognition of *then* and *else* parts.

IR		C
r10=add32(r10,r7)	⇒	r10=r10+r7;
cc=addpack(D1,8,D2,8)	⇒	cc=addpack(D1,8,D2,8);
cc=compstr(s,C'01',2)	⇒	cc=compstr((char*)s, "01",2);

Fig. 9. IR to C Conversion.

Dead codes, which are often created during long years' maintenance, are eliminated during the function recognition process (in III-C1).

E. IR to C Conversion and Handling of Dummy Sections

The IR is almost straightforwardly converted into a C program. The registers and variables in the original assembly program are treated as global variables in the C program. Arithmetic/logic operations and load/store operations are also converted into the corresponding operations in the C program. Operations on strings and decimals are translated into function calls to the the support library. Fig. 9 shows examples of the conversion, where `r10` and `r7` are registers, `cc` is the condition code, `addpack` and `compstr` are support library calls for addition on packed decimals and comparison on strings, respectively.

In order to enhance readability, arithmetic operations are collected into a single expression whenever possible. Fig. 10 shows examples. If the results of all the operations are referenced only once, they are grouped into a single expression.

A dummy section of the IBM assembly, initiated by `DSECT` instruction, is a section that results in no machine instruction nor data area but is used to specify the layout of the aggregate data passed between subroutines. In our method, the memory layout described by `DSECT` is converted to definition of the corresponding struct type and the data are accessed using the register variable designated by `USING` instruction as a base pointer. Fig. 11 shows an example. The `DSECT` instruction defines data layout with two items `A` and `B`, and the `MAIN` routine accesses the data area pointed by register 2 according to the layout. The dummy section is expressed by type `DAT1_t` and the data are accessed using `r2` as a base pointer.

F. Readability Improvement by Pattern Matching

In our method, further readability improvement is attempted by pattern matching based transformation. This is realized by defining rewriting rules on tree structures consisting of IR operations. Fig. 12(a) defines a tree rewriting rule to make a conditional statement based on string comparison more readable. IR in Fig. 12(b) is converted into IR' by the rule, resulting in program C' which should be better than C.

C		C'
r1_1=mul32(r1,X) r1_2=add32(r1_1,Y) r1_3=mul32(r1_2,Z)	⇒	r1 = ((r1*X)+Y)*Z;
C		**C'**

r1_1=mul32(r1,X)
r2_1=load32(r1_1)
r1_2=add32(r1_1,Y)
r1_3=mul32(r1_2,Z)
r2_2=add32(r2_1,T)
	⇒	r2 = r1*X;
		r1 = (r2+Y)*z;
		r2 = r2+T

Fig. 10. Generation of expression with multiple operations.

```
DAT1 DSECT                          typedef struct {
A DS F                                int A;
B DS 4C                               char B[4];
 ...                                } DAT1_t;
MAIN CSECT                    ⇒
     USING DAT1,2                   int main(void){
     MVI   A,123                      ((DAT1_t*) r2)->A = 123;
     MVC   B,=C'abcd'                 strncpy( ((DAT1_t*) r2)->B,
                                                 "abcd", 4);
...                                 ...
```

Fig. 11. DSECT Conversion

IV. Documentation

Due to limitation of static translation and to architectural issues, C programs generated from assembly programs may not always work. In such cases, the C programs must be inspected and modified manually. Even after the new C programs run successfully, they must be altered for maintenance. In such a situation, understanding of the code must be important. This paper proposes (1) to generate a table to show correspondence between the original and the resulting codes, and (2) to embed comments in the original assembly program into proper positions of the translated C program.

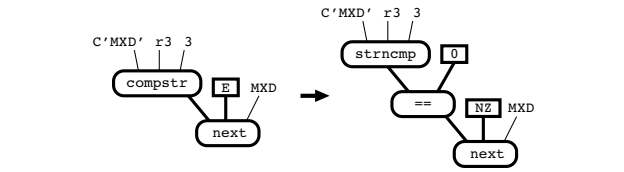
A. Correspondence between Instructions and Statements

Along with a working C programs for a given assembly program, a table to show the correspondence between original instructions and resulting C code fragments is generated in an HTML file. Fig. 13(a) shows an example. The `STM` instruction in line 113 has no corresponding C code because it is recognized as a save code and deleted. The `L` instruction in line 114 has been translated into the two C statements. The resulting C statements for the `AH` and `SH` instructions in lines 115–116 are grouped because their relation is many-to-many. Note that the rows of the table is based on the order of the statements in the C program, so the assembly instructions are reordered as 123, 124, 127, 125, 126, and 128 as the result of if statement recognition.

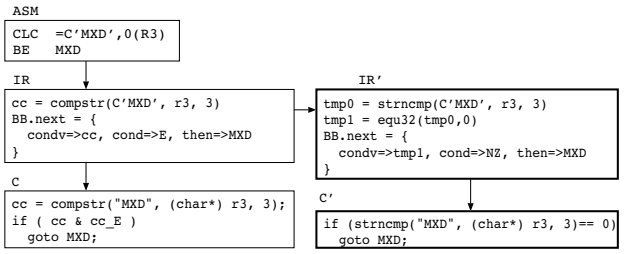
The table is generated by referencing the back pointers (depicted in dashed arrows in Fig. 13(b)) from the IR operations to the assembly instructions. If there is no IR operation for an assembly instruction (as ①), a virtual IR operation NOP is generated. When one or more IR operations have links to a single assembly instruction (as ① and ②), a table entry is created after all the IR operations for the assembly instruction are translated to C statements. If multiple IR operations are grouped to form a single C statement, all the instructions linked to the operations are put into an entry of the table (as ③).

B. Embedding Assembly Comments to C

A comment of the IBM assembly are either 1) a string placed in the same line of an instruction at the right of the



(a) Rewriting rule



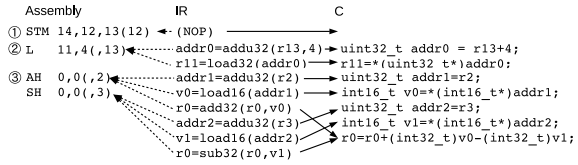
(b) Application of the rule

Fig. 12. Readability improvement by pattern matching.

Translation report

assembly	C	notes
...	...	
113 STM 14,12,12(13)		save code deleted
114 L 11,4(,13)	01 uint32_t addr0 = r13+4; 02 r11 = *(uint32_t*) addr0;	
115 AH 0,0(,2) 116 SH 0,0(,3)	03 uint32_t addr1 = r2; 04 int16_t v0 = *(int16_t*) addr1; 05 uint32_t addr2 = r3; 06 int16_t v1 = *(int16_t*) addr2; 07 r0 = r0 + (int32_t) v0 - (int32_t) v1;	ops grouped
...	...	
123 CLC ='MXD',0(R3) 124 BE L1	31 if (strncmp("MXD", (char*)r3,3)==0) {	
127 L1 SR 1,2	32 r1 = r1 - r2; 33 }	
125 AR 1,2 126 B BB1	34 else { 35 r1 = r1 + r2; 36 }	
128 BB1 ...	37 BB1: ; 38 ...	
...	...	

(a) Correspondence table (HTML)



(b) Establishing the correspondence

Fig. 13. Generation of correspondence table.

operands of the instruction, or 2) a string in a line starting from character '*'. In our method, assembly comments are lined to instructions, and are embedded into the resulting C program as their corresponding C statements are generated.

Assembly comments are classified into the following 3 types and are linked to instructions. Two instructions `END` (marking the end of the program) and `EJECT` (forcing a page break in source code listing) plays different role in comment classification than the other instructions, so in this paper, they are referred to as *delimiter instructions* and the other instructions as *normal instructions*.

1) Upper comments
   Comments between normal instruction i and the previous instruction of i are defined as the upper comments of i. In Fig. 14, for example, lines 1–2 are the upper comments of the `SH` instruction in line 3.
2) Right comments
   A comment placed at the right of normal instruction i is defined as the right comment of i. In Fig. 14, string `SUB` in line 3 is the right comment of the `SH` instruction.
3) Lower comments
   The lower comments of normal instruction i exist only when i's next instruction is a delimiter instruction. Let k be the next normal instruction of i, and j be the previous (delimiter) instruction of k. Then comments between i and j are defined as lower comments of i. In Fig. 14, line 5 is the lower comment of the `AH` instruction in line 4.

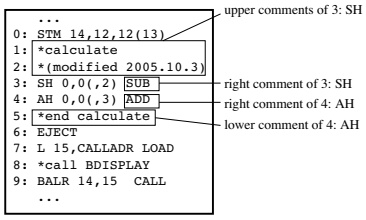


Fig. 14. Classification of comments.

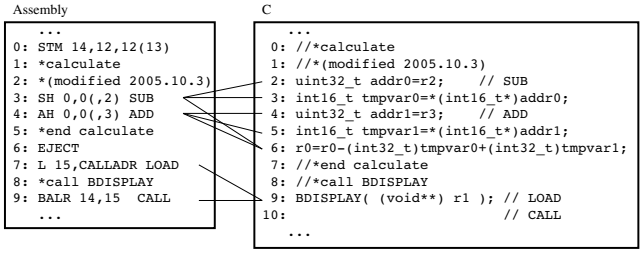


Fig. 15. Comment embedding.

Based on the above classification, the positions of the comments in the C program are determined as follows.

1) The upper comments of instruction i are placed above the first C statement generated from i. In Fig. 15, for example, since `SH` instruction in line 3 is expanded to statements in lines 2, 3, and 6, the upper comments of `SH` (assembly lines 1–2) are embedded into lines 0–1 of the C program.
2) The right comment of instruction i is placed at the right of the first C statement generated from i, as `// SUB` in line 2 of the C program.
3) The lower comments of instruction i are placed below the last C statement generated from i. The assembly comments in line 5 goes to line 7 of the C program.

Comment generation according to this policy is implemented by making use of the back pointer from the IR operations to the assembly instructions.

V. IMPLEMENTATION

A migration system based on the proposed method has been implemented in Perl 5. It operates on Ubuntu 14.04LTS, Mac OSX 10.10, and Cygwin on Windows. Currently, translation of 86 instructions out of 631 has been supported.

Fig. 16(a) is the result of conversion from the assembly program in Fig. 2. Subroutine EOM (in lines 16-48) of the assembly was converted to the function in lines 30–75. The code is structured with a *do-while* loop (in lines 34–46) and if statements (in lines 36–45, 48–71, and 51–70). In line 54, multiple operations are grouped into a single statement. Pattern matching based transformation proposed in III-F was applied to the string comparison in lines 36, 39, 48, and 51. The behavior of the converted program was confirmed by the driver program in Fig. 16(b).

There are some limitations in our current implementation. The resulting C programs do not run properly on 64bit machines, for the memory layout of the original IBM assembly is based on 32bit architecture. The current system does not support EX instruction which modifies other instructions.

VI. CONCLUSION

This paper has proposed a method of converting IBM assembly programs to C programs, and at the same time generating a document of conversion and embedding assembly comments into C programs. Future work includes support of the remaining instructions, improvement of readability by adding tree rewriting rules, and migration to 64bit architecture.

ACKNOWLEDGMENT

The authors would like to thank Kenji Okamoto of SYSTEM'S Co., Ltd. for valuable advice for this research. We would also like to thank all the members of Ishiura Lab. of Kwansei Gakuin University for their help developing the prototype systems. This work was partly supported by Small and Medium Enterprise Agency's "Services for reforming SMEs and micro-businesses, manufacturing, commerce and services" of fiscal 2013.

REFERENCES

[1] T. Ogawara: "Information processing apparatus, information processing method, and program," Japanese patent, 2014–215938 (Nov. 2014).

[2] C. Cifuentes, M. Van Emmerik, D. Ung, D. Simon, and T. Waddington: "Preliminary experiences with the use of the UQBT binary translation framework," in *Proc. Workshop on Binary Translation*, pp.12–22 (Oct. 1999).

[3] M. Van Emmerik: *Static single assignment for decompilation*, PhD Thesis, University of Queensland (2007).

[4] G. Chen, Z. Wang, R. Zhang, K. Zhou, S. Huang, K. Ni, Z. Qi, K. Chen, and H. Guan: "A refined decompiler to generate C code with high readability," in *Proc. Working Conference on Reverse Engineering*, pp.150–154 (Oct. 2010).

[5] Y. A. Feldman: "Portability by automatic translation: A large-scale case study," in *Proc. Knowledge-Based Software Engineering Conference*, pp.123–130 (Nov. 1995).

[6] M. P. Ward: "Assembler to C migration using the FermaT transformation system," in *Proc. IEEE International Conference on Software Maintenance 1999 (ICSM ' 99)*, pp. 67–76 (Aug.–Sept. 1999).

[7] Martin Ward: "Assembler restructuring in FermaT," in *Proc. IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2013)*, pp. 147–156 (Sept. 2013).


```
  1: #include <stdio.h>
  2: #include <stdlib.h>
  3: #include <string.h>
  4: #include <stdint.h>
  5: #include "miglib.h"
  6:
  7: static uint32_t r0, r1, r2, r3, r4, r5, r6, r7;
  8: static uint32_t r8, r9, r10, r11, r12, r13, r14, r15;
  9: static int cc;
 10: static char savearea[72];
 11:
 12: int16_t MLAST[] = {
 13:   0,  0,
 14:   31,  31,
 15:   59,  60,
 16:   90,  91,
 17:   120,  121,
 18:   151,  152,
 19:   181,  182,
 20:   212,  213,
 21:   243,  244,
 22:   273,  274,
 23:   304,  305,
 24:   334,  335,
 25:   365,  366
 26: };
 27: uint32_t SAVE[18];
 28: uint64_t WORK;
 29:
 30: void EOMfunc(void){
 31:   strncpy( (char*) r4,  "          ", 8 );
 32:   r0 = 6;
 33:   r1 =  r3;
 34:   do {
 35:     EOM1: ;
 36:       if ( strncmp( (char*) r1, "0", 1 ) < 0 ) { goto RET; }
 37:       else {
 38:         BB1: ;
 39:           if ( strncmp( (char*) r1, "9", 1 ) > 0 ) { goto RET; }
 40:           else {
 41:             BB2: ;
 42:             r1 =  r1 +  1;
 43:             r0 =  r0 -  1;
 44:           }
 45:       }
 46:   } while ( r0 );
 47: BB3: ;
 48:   if ( strncmp( (char*) r3+4, "01", 2 ) < 0 ) { goto RET; }
 49:   else {
 50:     BB4: ;
 51:       if ( strncmp( (char*) r3+4, "12", 2 ) > 0 ) { goto RET; }
 52:       else {
 53:         BB5: ;
 54:           zone_to_pack( (char*) &WORK, 8, (char*) r3+4, 2 );
 55:           r2 = pack_to_int32( (char*) &WORK, 8) - 1;
 56:         BB6: ;
 57:           r2 = (r2 * 4) + (uint32_t) MLAST;
 58:           uint32_t addr27 = r2 + 4;
 59:           r0 = * (uint16_t* ) addr27;
 60:           int16_t tmpvar29 = * (int16_t*) r2;
 61:           r0 = r0 - (int32_t) tmpvar29;
 62:           int32_to_pack( (char*) &WORK, 8,  r0);
 63:           pack_to_zone( (char*)  r4+6, 2, (char*) &WORK, 8 );
 64:           uint32_t addr31 = r4  + 7;
 65:           unsigned char tmp32 = * (unsigned char *) addr31;
 66:           unsigned char tmp33 = tmp32 & 0x0f;
 67:           unsigned char tmp34 = tmp33 | 0x30;
 68:           * (unsigned char*) addr31 = tmp34;
 69:           strncpy( (char*) r4, (char*) r3, 6 );
 70:       }
 71:   }
 72: RET: ;
 73:   r15 =  r15 -  r15;
 74:   return;
 75: }
 76: int32_t EMONTH( void** param ){
 77:   r1 = (uint32_t) param;
 78:   r13 = (uint32_t) savearea;
 79:   r14 = (uint32_t) &&RETURN;
 80: EMONTHbb: ;
 81:   r12 =  r15;
 82:   r15 = (uint32_t) SAVE;
 83:   uint32_t addr15 =  r15 +  4;
 84:   * (uint32_t*) addr15 = r13;
 85:   uint32_t addr16 =  r13 +  8;
 86:   * (uint32_t*) addr16 = r15;
 87:   r13 =  r15;
 88:   r3 = * (uint32_t* ) r1;
 89:   r4 = * (uint32_t* ) (r1 + 4);
 90:   if ( strncmp((char*) r4, "00000000", 6 ) ) {
 91:     ER_FMT: ;
 92:       abend( 999 );
 93:   }
 94:   else {
 95:     FC_EOM: ;
 96:       EOMfunc();
 97:   }
 98: RETURN: ;
 99:   return r15;
100: }
```

(a) Generated C program from the assembly of Fig. 2

```
 1: #include <stdio.h>
 2: #include <stdint.h>
 3: #include <string.h>
 4:
 5: int32_t EMONTH(void **param);
 6:
 7: int main(void)
 8: {
 9:   char gdate[] = "20141215";
10:   char result[] = "00000000";
11:   void* param[2] = {gdate, result};
12:   int32_t rc = EMONTH(param);
13:
14:   printf("rc = %d, result = \"%s\"\n", rc, result);
15:   return 0;
16: }
```

(b) Test driver

Fig. 16. Resulting C program.