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 redevelop 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/I, 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 automatically translated 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
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 input. The structure of the IR in our method is shown in Fig. 4. The flow of our translation process is shown in Fig. 3.

The IBM assembly program.

![Fig. 2. IBM assembly program.](image)

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
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.

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 T̄, and X' be the set of the basic blocks reachable from X. Then T is defined as T' ∪ X̄ (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, ..., 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, ..., or 68 to register 13, and 2) no operation uses the result of l.
Fig. 8. Recognition of then and else parts.

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 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.

Fig. 10. Generation of expression with multiple operations.

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 1 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
**Rewriting rule**

(a) Rewriting rule

Fig. 12. Readability improvement by pattern matching.

(b) Application of the rule

---

**Translation report**

Fig. 13. Generation of correspondence table.

**Assembly**

(c) Establishing the correspondence

---

**Operand assignment**

Fig. 14. Classification of comments.

---

**Comment embedding**

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.

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.
A migration system based on the proposed method has been implemented in Perl 5. It operates on Ubuntu 14.04 LTS, 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 64-bit machines, for the memory layout of the original IBM assembly is based on 32-bit 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 64-bit 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


