Strategies for Machine Instruction Transformation in an Expert System

Man L. Li and Willis K. King
Department of Computer Science
University of Houston
Houston, TX 77204

Abstract
To build an expert system which can transform machine language programs between any pair of computers, an intermediate language is used. Several strategies are developed to ensure the proper and efficient transformation. This paper discusses local transformation, basic block transformation, and instruction side effect handling. Example of program transformation between the VAX and the IBM 370 is given.

1 Introduction
Two sequences of machine instructions are equivalent, if after execution they have the same effect on two different computers which have the same initial states. Given a sequence of instructions for a particular computer, how to find a sequence of equivalent instructions in another computer is the problem of machine instruction transformation.

The main motivation for machine language transformation is an attempt to improve software portability. Software in machine language could be ported to new hardware platforms using the program transformation method.

Program transformation in high level languages have been investigated by many researchers [4, 7]. It is reported in the comprehensive survey by Partsch [12]. To transform programs in assembly or machine language, different schemes have been tried [10, 13, 8]. Mostly they deal with the translation between a particular pair of computers. There is so far no general approach for the transformation of a machine language program to any arbitrary target machine architecture.

To achieve this more ambitious goal, our approach is to use a common intermediate language as the transformation medium. A program is first converted to an equivalent program in this intermediate language, which in turn is converted to the assembly language and machine language of the destination machine as shown in Figure 1.

2 Overview
The conversions between a machine language and an assembly language are well understood [5]. The transformation between the assembly language and the intermediate language is therefore the main focus of this paper.

In our transformation system, the architectural definitions of each computer are entered in a knowledge base. Using a rule-based expert system approach, transformation rules describing the transformation strategies are also stored in the knowledge base. The transformation process can then be done automatically while the information and strategies of
transformation can be utilized and maintained in a systematic manner [11]. Organization of our transformation system is represented in Figure 2.

3 Intermediate language

The purpose of intermediate language is to provide a standardized representation of assembly language programs. From the intermediate language, the assembly language for various destination machines can be generated. To simplify the process, the intermediate language should be able to express the operations and data types of most common machine instructions in a straightforward manner. Furthermore, the language should provide representations for registers, memory elements, and other processing elements in the machine. Although a high level programming language can be used as an intermediate language, it would be more convenient to use an universal assembly language which have close resemblance to typical assembly languages. The forward and backward transformations between assembly languages and the intermediate language can then be performed using similar strategies.

We choose the proposed IEEE standard microprocessor assembly language [1] as the base for the intermediate language. This standard, although originally intended for microprocessors, contains most features typically found in all classes of machines. Conceptually, the intermediate language can be viewed as the assembly language for an idealized machine.

4 Machine description

To carry out the program transformation, the knowledge base must contain descriptions for the real machines and the specification of the intermediate language. This includes the op-code value, operation mnemonic, operand number, operand addressing-modes, operand data types, and effects of the operation. The description of machine architectures should include the number and size of general and special registers, number of program status flag, memory unit, the range of address space, and input/output port size and addresses.

Machine description languages have been used to describe computer architecture for product specification, machine simulation, performance evaluation, design verification, test generation, logic synthesis, microcode generation, software generation and other applications [3, 6, 9]. However, many of the languages have complicated language syntax making them difficult to be processed automatically. For our purpose, we have developed a machine description language based on ISPS[2] using the syntax of LISP. An interactive knowledge acquisition system is also under development to automate the process of entering machine description[14].

The main emphasis of our machine description language is to represent the effect of each instruction in expression called action-sequence. The action-sequence is a LISP expression representing simple action or multiple actions in parallel or in serial. A simple action has the first item in the expression as the operator and the other items as operands. Operands can be constant value or address of hardware elements (register or memory). For example, the move instruction

\[
\text{move } \text{src, dst}
\]

is represented in the action-sequence as a single action

\[
\text{let dst dst src}
\]

where let is the operator, dst and src are the addresses of hardware storage elements. Full details of our machine description language can be found in [14].

5 Program transformation

The transformation process between assembly languages and the intermediate language are similar in both directions. One approach is the local transformation in which the source program is analyzed one instruction at a time and equivalent target machine instructions are generated. On the other extreme, the whole program can be analyzed and “understood” and then rewritten in the target machine language. This automatic understanding of program at the global level although desirable is not easily achievable. Our strategies as described in the following sections try to accomplish this goal by taking an approach between these two extremes.
5.1 Mapping of hardware elements

The first step in program transformation is the mapping of hardware space to find the equivalent hardware elements of the two machines. An actual machine has a fixed number of registers and a limited size of memory. On the other hand, the idealized machine for the intermediate language is assumed to have unlimited number of registers which can be accessed as an unit with the low-order 16 or 8 bits or with consecutive registers. The machine is byte addressable with unlimited memory space and status flags.

The mapping of the hardware elements between the actual machine and the idealized machine is to assign each register of the actual machine to a general register with comparable characteristics of the idealized machine in a mapping table. Other hardware elements such as the program counter, the stack pointer, and status flags are also included in the mapping table. If the required number of registers in the idealized machine exceeds that of the actual machine, they will be mapped to fixed memory spaces.

5.2 Local transformation

Local transformation is done by examining the source program one instruction at a time and searching for its equivalent instruction(s) in the target machine. This involves matching the action-sequence of the source instruction with the action-sequences of the target instruction set. The algorithm for the local transformation can be represented as followed:

Algorithm 1

Repeat
Find the action-sequence of the source language instruction in the machine description knowledge base.
Substitute operands by the actual source machine hardware elements.
Change the source machine hardware elements to the equivalent target machine hardware elements.
Find the equivalent action-sequence in the target machine.
If the equivalent action-sequence is a single instruction then
Substitute operands with actual hardware elements.
else /* serial actions */
Repeat
For each action in the action-sequence
find its equivalent action-sequence in the target machine.
If the equivalent action-sequence is found then
Substitute operands with actual hardware elements.
endif
until no more action
endif
until end of the source program.

For example, the move instruction in the VAX machine with autodecrement addressing

\[ \text{MOV} L-(R5), A \]

is represented in action-sequence as

\[ (\text{next} \ (\text{let} \ R5 \ (\text{sub} \ R5 \ 4))) \]
\[ (\text{let} \ (\text{loc} \ M \ A) \ (\text{loc} \ M \ R5)) \]

where the keyword \text{next} is used for serial actions. Assuming that the mapping table to intermediate language contains the equivalent register pair \([R5, R5]\), the following intermediate language instructions can be generated:

\[ \text{SUB} \ R5, #4 \]
\[ \text{MOV} [R5], /A \]

This approach of transformation has only limited application since it requires the source and target machines to have similar instruction sets. In considering single instruction one at a time and disregarding the effect of adjacent instructions, local transformation will normally produce inefficient target program. Nevertheless, using the instruction expansion technique as in Section 5.4, local transformation may be modified to handle most kind of program.

5.3 Basic block transformation

A more efficient strategy is to analyze a segment of the source program first, and then the equivalent target machine instructions of the whole segment is produced. One way to achieve this is to analyze each basic block of the source program by using a Direct Acyclic Graph (DAG). A basic block is a sequence of instructions with a single entry point at the beginning and no branching within the block except at the end. A DAG can be constructed easily from the action-sequence of the instructions after the program is partitioned into basic blocks. The basic block transformation using DAG is shown in Algorithm 2.

Algorithm 2

Repeat
Get a source program instruction.
Find the instruction's action-sequence.
Substitute action-sequence operands by the actual source machine hardware elements.

Change the source machine hardware elements to the equivalent target machine hardware elements in the action-sequence.

If new basic block then
   Start basic block with new instruction.
else
   Add instruction to basic block.
endif
until end of program.
Repeat
   Build a DAG from a basic block.
until no more basic block.
Repeat
   Decompose a DAG into action-sequences and generate target machine instructions.
until no more DAG.

In the basic block transformation, DAG is constructed from the action-sequences of the source program then the target machine action-sequences are generated by decomposition. In contrast, in local transformation action-sequences for the target machine are generated directly from their counterparts in the source program. Both of these transformations are shown in Figure 3.

Figure 4 shows an example of a source program of single basic block in the intermediate language. DAG for the source program is constructed as shown in Figure 5. The result target machine instructions for a three-address machine using basic block transformation are shown in Figure 6.
5.4 Instruction expansion

So far in both the local and basic block transformation algorithms, it is assumed that for a given action-sequence it is always possible to find equivalent instructions in the target machine instruction set. However, due to differences in machine architecture, exact equivalent target instruction might not be available for a given action-sequence. In such cases, sequence of target machine instructions will be required to produce the equivalent effect of the action-sequence. For the action-sequence of an instruction

\( (\text{let } \text{dst} \ \text{exp}) \)

where \( \text{dst} \) is the destination operand and \( \text{exp} \) is the source operand or an expression of operation, the action-sequence can be expanded by finding another instruction with the action-sequence

\( (\text{let } \text{dst}' \ \text{exp}') \)

where \( \text{dst}' \) is the same as \( \text{exp} \) or a term in \( \text{exp} \). For example, the VAX-11 computer has three-address instruction

\[ \text{SUBBS} \ #1, \ \text{SIZE}, \ \text{LIMIT} \]

where \( \text{SIZE} \) and \( \text{LIMIT} \) are memory addresses and \#1 is operand in immediate addressing. The action-sequence of this instruction contains a single action

\( (\text{let } (\text{loc } \text{M } \text{LIMIT}) (\text{sub } (\text{loc } \text{M } \text{SIZE}) 1)) \)

where (loc M label) represents content at memory location label. When this single action is transformed to the intermediate language, only two-address or single-address instructions are available. To find equivalent target instructions, the following expansion of action-sequence can be found:

\( (\text{let } (\text{loc } \text{M } \text{label}) .RI) \)
\( (\text{let } .RI (\text{sub } .RI 1)) \)
\( (\text{let } .RI (\text{loc } \text{M } \text{label})) \)

After substitutions, the new action-sequence can be seen as equivalent to the original instruction

\( (\text{let } (\text{loc } \text{M } \text{label}) (\text{sub } (\text{loc } \text{M } \text{label}) 1)) \).

The intermediate language instructions for such expansion using the actual register and memory addresses become:

\[ \text{LD } \text{R2}/\text{SIZE} \]
\[ \text{DEC } \text{R2} \]
\[ \text{ST } \text{R2}/\text{LIMIT} \]

5.5 Tree pruning

To avoid unnecessary instruction expansion, action-sequences which do not appear to lead to a solution will not be considered. This tree pruning is based on two criteria. The first one is to discard expansions which have operation not compatible with the target action-sequence. For example, if the source action sequence is an add operation

\( (\text{let } a (\text{ADD } e_1 e_2)) \)

the target instruction action-sequence for expansion should be

\( (\text{let } x y) \) or \( (\text{let } x (\text{ADD } y_1 y_2)) \)

where \( x \) is equivalent to \( a \).

Another type of tree pruning is to avoid cycle in tree expansion which can be detected when a new action-sequence is selected. If a tree expansion generates a new node which is the same as its ancestor, this expansion creates a cycle.

For example, the action-sequence

\( (\text{let } x (\text{OP } y_1 y_2)) \)

can be expanded with action-sequence \( (\text{let } y_1 z) \) but the action sequence \( (\text{let } z y_1) \) should not be used for further expansion since it will produce cycle in the tree. However, cycle of more than two levels is not always easy to detect. For practical purpose, we limit the tree expansion to four levels.

5.6 Side effect analysis

In our machine description language, action-sequences are used to describe the main effect of machine instructions. Most actual machines instructions will also affect the status flags. These side effects of instructions are described in the machine description language but ignored on the previous transformation algorithms for simplicity.

Most side effects do not influence the overall result of a program. Only a few instructions, such as conditional branch, addition/subtraction with carry etc., are affected by status flags. To determine the influence of instruction side effects and ensure the source program is transformed correctly, both the source and target program must be analyzed.

The side effect analysis can be carried out for the whole program or within a basic block. By comparing the side effect of each instruction in the basic...
LOAD F  
DIV G  
STORE T  
NEXT LOAD D  
MUL E  
END  

Figure 7: Source program of Single-address Machine

LD .RO /F  
DIV .RO /G  
ST .RO /T  
NEXT LD .RO /D  
MUL .RO /E  
END  

Figure 8: Target Program in Intermediate Language

For the second basic block no adjustment on the target program is required, since all flags are changed at the equivalent locations for the source and target programs.

If the whole target program containing two basic blocks is analyzed further, it is obvious that the additional instructions for side effects in the first basic block are not necessary, because none of the four flags are used in the following basic block and throughout the program.

6 Implementation

Our program transformation system was implemented on Sun Microsystems 3/60 workstations running UNIX using Sun Common LISP. The basic inference engine using a forward chaining approach contains only about 500 lines of LISP statement. The Transformation System running on top of the inference engine has about 4000 lines of LISP statement. There are 16 rule sets in approximately 3000 lines of production rule. The machine knowledge base takes 30K bytes containing information for the VAX, IBM 370, SPARC, Motorola 68000, INTEL 8080, Motorola 6800 and three hypothetical machine models for single-address, two-address, and three-address machines.

7 Results

The following example shows the transformation from an assembly language program for the VAX computer into assembly languages of the IBM mainframe computer. Similar transformations were also done, but not shown here, with the Motorola MC68000 processor and the SPARC RISC processor as the destination machine.

The original source program in VAX MACRO assembly language is:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Argument</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPB</td>
<td>SIZE, #1</td>
<td></td>
</tr>
<tr>
<td>BLEQ</td>
<td>SORTC</td>
<td></td>
</tr>
<tr>
<td>STARTS</td>
<td>SUBR3</td>
<td>#1, SIZE, LIMIT</td>
</tr>
<tr>
<td>OUTER</td>
<td>CLRB</td>
<td>FLAG</td>
</tr>
<tr>
<td></td>
<td>MOVAVW</td>
<td>A, R10</td>
</tr>
<tr>
<td></td>
<td>ADDL3</td>
<td>#2, R10, R11</td>
</tr>
<tr>
<td>INNER</td>
<td>INCB</td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td>CMPB</td>
<td>R1, LIMIT</td>
</tr>
<tr>
<td></td>
<td>BGTR</td>
<td>EXIT</td>
</tr>
<tr>
<td></td>
<td>CMPW</td>
<td>(R10), (R11)</td>
</tr>
<tr>
<td></td>
<td>BLEQ</td>
<td>NEXTE</td>
</tr>
<tr>
<td></td>
<td>MOVW</td>
<td>(R10), TEMP</td>
</tr>
<tr>
<td></td>
<td>MOVW</td>
<td>(R11), (R10)</td>
</tr>
</tbody>
</table>

274
The program was first transformed into intermediate language as:

```
ORG 0
CMP /SIZE #1
BLE SORTC
STARTS MOVE /SIZE, /LIMIT
OUTER CLR /FLAG
LD R1, /FLAG
LD R10, $A
ADD R10, #2
MOVE R10, R11
INNER INC R1
CMP R1, /LIMIT
BR EXIT
CMP [R10], [R11]
BLE NEXTE
MOVE [R10], /TEMP
MOVE [R11], [R10]
SET /FLAG
MOVE /TEMP, [R11]
NEXTE ADD R10, #2
ADD R1, #2
BR INNER
EXIT DEC /LIMIT
TEST /FLAG
BNZ OUTER
SORTC SET R9
END
```

The result of transformation into assembly language for the IBM machine is:

```
LA R2, 1
L R3, SIZE
CR R3, R2
BNH SORTC
STARTS LA R2, 1
L R3, SIZE
SR R3, R2
ST R3, LIMIT
OUTER MVI FLAG, 0
L R1, FLAG
LA R2, 2
LA R10, A(R2)
LR R11, R10
INNER LA R1, 1(R1)
C R1, LIMIT
BH EXIT
L R2, 0(R10)
C R2, 0(R11)
BNH NEXTE
L R2, 0(R10)
ST R2, TEMP
L R2, 0(R11)
ST R2, 0(R10)
MVI FLAG, 1
L R2, TEMP
ST R2, 0(R11)
NEXTE LA R10, 2(R10)
LA R11, 2(R11)
B INNER
EXIT LA R2, 1
```

We have verified the result of the transformation which compares favorably with human translation. For example, the system automatically allocates temporary register R2 in the destination program to make up for the difference in addressing mode. The same register is used repeatedly throughout the program to minimize the number of register used.

The system might use different operation to achieve the equivalent effect. In the above program, instead of using the add instruction to change register contents, the system uses the load register instruction for the same effect, such as:

```
INNER LA R1, 1(R1)
```

```
NEXTE LA R10, 2(R10)
```

to increment register R1 and add 2 to register R10.

The system can transform program between machines with incompatible addressing modes. When register indirect addressing mode is not available for the IBM machine, the system simulates such effect with zero displacement and a base register such as:

```
LA R2, 0(R10)
```

In lacking of immediate addressing mode, the load register instruction is used

```
LA R2, 1
```

These techniques might not be even apparent to a human translator.

However, the system has difficulty in dealing with different data types. The original array of 16-bit word in the VAX program was not properly transformed in the destination program.

# 8 Conclusions

Our program transformation system for machine language has the following advantages over other known methods:

**Automatic** The biggest advantage of this system is labor saving. Doing the transformation manually will take an enormous number of manhours, even for a simple program. Such transformation will require the translator to be familiar with both the source and target assembly languages. Before the transformation, he must first analyze and
understand the whole program, then rewrite the program in the target assembly language.

**System Independent** This system is independent of the original and the destination machine architectures. All transformations are done between the intermediate language and any designated machines. The transformation rules are set up without any particular machine in mind.

**Syntax Independent** The system is independent of the syntax of the source and target assembly languages. By defining the language in the machine architecture knowledge base, the transformation system can accept assembly languages with various syntax. The transformation process is not affected when the language syntax is changed.

There are many areas related to our system which warrant more improvements and further research. One basic problem is the transformation between different data types. This requires a better description of the machine instruction and algorithm to select proper operations. Another area is machine instructions with multiple actions. In our present approach, an instruction with multiple serial actions is transformed into a sequence of simple instructions. For instructions with parallel actions, a more detailed analysis of the data dependency and the time sequence of each action would be required for correct transformation. Further research is also needed for program verification at the assembly language level. Until a systematical method is developed, it will be very difficult to prove the correctness of the transformation.

**References**


