Abstract

An assembly-language program often has more than one valid machine-language encoding. Furthermore, a program’s different encodings usually have different time and space requirements at run time. This paper examines the problem of translating an assembly-language program into an optimum machine-language encoding. The optimization problem is NP-complete, so a polynomial-time algorithm that always finds an optimum encoding is unlikely [4]. Some suboptimum algorithms are available; however, they place significant restrictions on assembly-language programmers and compilers, and some of the algorithms can produce invalid machine-language encodings of programs. This paper discusses a new optimization algorithm that (1) always generates correct encodings, (2) does not place any restrictions on programmers and compilers, and (3) eliminates the need for programmers to optimize machine-language encodings of programs manually.

1 Introduction

Assemblers translate assembly-language programs into machine code. A programmer or a compiler produces an assembly-language program, and the assembler translates the program into absolute machine code or relocatable machine code. Assemblers are therefore important, although often invisible, tools even for applications that use high-level languages. An assembler may actually be part of a compiler.

1.1 Optimization

Most assembly-language programs have more than one possible machine-language encoding. Multiple encodings are possible because some assembly-language instructions are multiple-form instructions: they have two or more machine-language forms. For example, many computers have a branch instruction that has both a short form and a long form, where the short form requires fewer bytes in memory than the long form requires. Besides consuming less memory, the short form usually executes faster than the long form.

Programmers and compilers can specify the form an assembler should use for each multiple-form instruction in a program. In general, however, programmers and compilers cannot accurately determine the shortest possible forms without actually assembling the program; ideally, assemblers should decide which forms to use.

When assembling a multiple-form instruction, an assembler can generate a particular form of the instruction if and only if the instruction has an operand with an expression whose value is within a certain range. If the expression is not within the proper range, the assembler must generate a longer form of the instruction (or report an error if there is not a longer form). Since the longest form of an instruction is more general than the shorter forms, an assembler could simply always choose the longest forms for instructions. To ensure the shortest encoding of a program, however, an assembler must choose the shortest possible form for each multiple-form instruction in the program.

The optimization problem is difficult because instructions can depend on each other. The expression that determines which forms are possible for a particular instruction can depend on the forms that the assembler selects for other instructions in the program.

1.2 Contributions and Limitations

Previous work in the area of optimizing assemblers ignores some requirements that are important for a general-purpose optimizing assembler. This paper specifies these important requirements.

Besides specifying the requirements, this paper describes a new algorithm that meets those requirements. The algorithm does not place any restrictions on assembly-language programs that people and compilers generate.
2 The Optimization Problem

The examples in this paper use a subset of the assembly language that ASMK, an M68000 assembler at Arizona State University, accepts for the Motorola M68000 processor. In particular, we use an EQU statement that assigns a value to a label, IF and ENDIF statements that provide support for conditional assembly, and a BRA instruction that has a one-word form and a two-word form.

The IF and ENDIF statements delimit conditional-assembly blocks. The assembler assembles the statements in a conditional-assembly block only if the expression for the IF statement is true.

If the value of a BRA instruction’s expression is in the range *+2-128 through * or the range *+4 through *+2+126 (where * is the address of the instruction), an assembler can translate the BRA statement into a one-word machine-language instruction; otherwise, the assembler must translate the BRA statement into a two-word machine-language instruction. An assembler can choose to generate the long form even when the short form would work.

2.1 Observations

The optimization problem in general does not include just branch instructions. Most computers have several types of multiple-form instructions, and an optimizing assembler should optimize all of them.

Some computers have multiple-form instructions with more than two forms, so a general-purpose optimization algorithm must be able to handle instructions that have three or more forms.

A greedy algorithm that optimizes each instruction individually will not always generate the shortest possible program because optimizing a particular instruction might make a program longer than not optimizing the instruction. Figure 1 provides an admittedly contrived example that, in spite of being contrived, illustrates the challenge that conditional assembly presents. Figure 2 provides a more realistic example that does not use conditional assembly.

Figure 3 shows that a multi-pass optimizing assembler must be careful to guarantee that it always terminates. We use conditional assembly in this contrived example to illustrate the point simply, but similar circumstances arise in real programs, even without conditional assembly. The assembler must choose the long form of the BRA instruction in Figure 3 to avoid an infinite loop.

Optimization can adversely affect the behavior of programs that make assumptions about the sizes of instructions (Figure 4). Such assumptions are bad style.

Figure 1: Greedy Algorithms Don’t Always Optimize

<table>
<thead>
<tr>
<th>L1</th>
<th>BRA</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>IF</td>
<td>*-L1.EQ.2</td>
</tr>
<tr>
<td>BRA</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>BRA</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

L1 BRA * $$ IF *-L1.EQ.2 BRA * BRA * $$ ENDIF END

If the assembler uses the short form for the first BRA, the IF expression is true and the program assembles into three words of machine code. If the assembler uses the long form for the first BRA, the IF expression is false and the program assembles into two words.

Figure 2: A Greedy Algorithm Fails Again

Should an assembler select ADDQ or ADDI? If the assembler selects ADDQ, the program requires 20 bytes in memory (2 for ADDQ.L and 6 for each LEA.L). If the assembler selects ADDI, the program requires only 18 bytes in memory (6 for ADDI.L and 4 for each LEA.L).

Figure 2: A Greedy Algorithm Fails Again

(At best), so most programmers do not make them. Besides, a programmer never needs to assume that an assembler will generate a particular form for an instruction because a programmer can always force the assembler to generate any form that she or he wants. (With an M68000 assembler, a programmer can use BRA.B to force the short form of a branch instruction, or BRA.W to force the long form.)

2.2 Requirements for an Assembler

A general-purpose optimizing assembler for both programmers and compilers must meet the following four requirements:

1. The assembler must always generate a correct machine-language encoding of an error-free assembly-language program; the encoding might not be optimal, but it must be correct. For a program that contains assembly errors, the assembler must report the errors.

2. The assembler must not place any restrictions on the contents of assembly-language programs. Programmers and compilers must be free to generate any assembly-language program. The assembler must not refuse to assemble a program (or, even worse, generate incorrect machine code without reporting an error)
L1 BRA L2
$$ IF *-L1.EQ.2
L2 EQU L1+2+128
$$ ENDIF
$$ IF *-L1.EQ.4
L2 EQU L1
$$ ENDIF
END

If an assembler selects the short form for the BRA, the destination of the branch is too far away for a short branch. If the assembler selects the long form for the BRA, the destination of the branch is close enough for the short form to work. A multi-pass optimizing assembler might select the short form during pass one, the long form during pass two, the short form during pass three, etc.

Figure 3: An Assembly Might Never Terminate

L1 BRA L2
L2 EQU L1+4
BRA *+500
END

A non-optimizing assembler might use the long form for the first BRA instruction because it makes a forward reference; however, an optimizing assembler would use the short form.

Figure 4: Optimization Can Change a Program

simply because the program is hard to optimize.

3. The assembler must select the smallest possible form for most multiple-form instructions in most programs. Unfortunately, this requirement is not very specific. The goal is to require an acceptable amount of optimization without demanding complete optimization of all instructions in all programs because such a demand would necessitate an exponential algorithm (assuming $P \neq NP$).

4. The assembler must work with all of the computer’s multiple-form instructions, not just branch instructions; also, the assembler must take advantage of any multiple-form instructions that have more than two forms.

3 Previous Work

Traditional one-pass and two-pass assemblers optimize instructions only when they don’t include any forward references. Richards [2] describes an exhaustive-search algorithm for optimizing forward-reference instructions that have two forms. Frieder and Saal [1] extend Richards’ work, but their algorithm is still $O(2^n)$ in execution time and $O(n^2)$ in space, both of which are entirely unacceptable.

Szymanski [4] describes an algorithm from an old Unix assembler that selects long forms first and then makes one extra pass to shorten long branches where possible. This algorithm typically misses many possible optimizations since it does not consider optimizations that become possible only as a result of other (subsequent) optimizations.

Szymanski also describes an algorithm that selects short forms first and later changes them to long forms. He restricts expressions to the form $label \pm constant$ where $label$ specifies the address of an instruction, and his algorithm is $O(n^2)$ in time and $O(n)$ in space.

The existing algorithms that optimize forward branches are adequate for the back ends of some compilers, but they are not good enough for use in general-purpose assemblers. The traditional algorithm, which optimizes only backward references, is the only algorithm that does not place restrictions on the programs that people and compilers can generate. For example, an assembler that provides conditional-assembly statements could not use any of the existing algorithms (except for the traditional algorithm) without limiting programmers and compilers in some way. Most general-purpose assemblers do not optimize forward references, probably because the available optimization algorithms are not good enough.

4 New Optimizing Algorithm

This section describes a new polynomial-time algorithm that optimizes many practical programs. The algorithm does not always generate optimum machine code, but it always generates correct machine code. The algorithm meets all the requirements from Section 2.2.

We have implemented this algorithm in ASMK, a full-featured assembler for the M68000 [3]. ASMK makes one more pass over a source program than the algorithm requires; during the extra pass, ASMK generates a table of contents and creates some tables for the linker.

The algorithm takes a greedy approach and optimizes each instruction individually. The greedy approach is optimal for most programs; however, Section 2.1 shows that the greedy approach is not perfect because optimizing some instructions might make a program longer than not optimizing those instructions.

The algorithm makes multiple passes over a source program. During the first pass, the algorithm uses the shortest forms for multiple-form instructions that
During pass $i$, where $2 \leq i \leq N - 1$, and $N$ is the total number of passes, the algorithm uses a longer form of an instruction (that makes forward references) if label values from pass $i - 1$ (and earlier parts of pass $i$) indicate that the instruction must be longer; also, to ensure termination, the algorithm never selects a shorter form of an instruction if the previous pass selected a long form. The algorithm continues to make passes until the algorithm makes identical decisions during two consecutive passes. For an assembly that requires more than four passes, for $3 \leq i \leq N - 2$, pass $i$ lengthens instructions that must be longer due to instructions that pass $i - 1$ lengthened. Pass $N - 1$ is the first pass during which the algorithm does not lengthen any instructions. The label values that pass $N - 1$ finds are the final values for all of the labels.

Pass $N$ (the final pass) makes the same decisions that pass $N - 1$ made. The purpose of the final pass is to generate machine code. The algorithm does not generate machine code during pass $N - 1$ because the algorithm does not know the value of $N$ until after pass $N - 1$ completes. To avoid pass $N$, the algorithm could generate machine code during pass two, and then again during each subsequent pass; however, generating machine code during multiple passes is probably less efficient than making the extra pass.

During each pass, the algorithm keeps track of GROWTH, the number of bytes by which the machine-language encoding of the program has grown during the pass. The algorithm uses GROWTH to anticipate growth in the values of labels that are forward references. Anticipating growth improves optimization. Consider, for example, a branch instruction that contains a forward reference. Since many instructions may grow from a short form to a long form from one pass to the next, the value of the label during one pass can be much greater than the value during the previous pass; if the algorithm does not anticipate growth in the value of the label, the label value could appear to require a large negative offset, even though the true label value might require a small positive offset.

We cannot describe the entire algorithm in detail here. Sterbenz [3] explains many details that we have omitted, including extensions that are necessary for relocatable assembly. He also provides a proof of termination and a proof that the algorithm always generates a correct encoding. Additionally, he analyzes the algorithm’s performance and proves it to be $O(n^2)$ in time and $O(n)$ in space.

In actual practice, the algorithm optimizes real programs with only five passes in typical cases and with seven passes in the worst case ever encountered among real programs (i.e., excluding contrived cases), so the algorithm’s execution time is $O(n)$ in practice. The assembler has neglected to optimize only one statement out of more than a million lines of code in real programs that we have assembled. For practical purposes, therefore, the algorithm optimizes programs completely.

5 Conclusion

Optimizing multiple-form instructions in assembly-language programs is a hard problem. Most optimizing compilers optimize multiple-form branch instructions; however, the available optimization algorithms are not adequate for general-purpose assemblers that assembly-language programmers and compilers use.

This paper described a new optimization algorithm that always generates correct encodings of programs without placing any restrictions on programmers and compilers. The new algorithm is suitable for use in general-purpose assemblers for both programmers and compilers. The algorithm always terminates with a correct machine-language encoding of an input assembly-language program. Also, an implementation of the algorithm in an existing assembler helped verify that the algorithm does a good job of optimization in practice.

The new optimization algorithm is significant to assembly-language programmers because it eliminates the drudgery of manually optimizing time-critical assembly-language programs.

The new algorithm is significant to compiler writers because it gives code generators the freedom to generate any assembly-language programs without having to worry about the restrictions that other optimization algorithms place on assembly-language programs.

References


