Code optimization techniques for micro-code compilers

by C. J. TAN
IBM T. J. Watson Research Center
Computer Sciences Department
Yorktown Heights, N.Y.

INTRODUCTION

With the ever expanding volume of system functions directly implemented in microcode and the growth of microprocessor applications, it has become necessary to design high level language compilers for these machines to achieve high programming productivity. However, because of the very low level architecture of many of these machines, compilers that generate efficient code for these machines have not been produced.

An optimizing compiler, called PL/MP, has been designed which is capable of supporting a variety of microprocessors such as the Motorola M6800 as well as machines like the IBM 370 model 145, and several microprocessors designed and used internally at IBM. PL/MP uses a set of novel, machine dependent, and the more conventional high level, machine independent optimization techniques to achieve high object code quality. Experimental results have indicated that the machine dependent optimizations play a very significant role in enhancing the quality of the object code produced by the compiler. The most effective of these optimizations are

1. Register allocation: optimized binding of compiler source and temporary variables to fast machine registers.
2. Low level addressing code optimization: code motion and common expression elimination for addressing code.
3. Code consolidation: combining a set of scattered simple instructions into a single more complex instruction supported by the object machine.

Basic to all of these optimization techniques are the algorithms for obtaining information concerning the use of variables throughout the program. These data flow analysis algorithms are used throughout the compiler for both machine independent and machine dependent optimizations. In this paper we will describe the code consolidation process, the environment in which it is carried out, and its dependence upon global data flow analysis. Algorithms for register allocation, code motion and hoisting have been published elsewhere, and will not be discussed in this paper.

THE PL/MP COMPILER

The PL/MP compiler accepts a high level language such as PL/I and generates efficient object code for a micro-machine. The compiling process takes the source code through several levels of internal text as shown in Figure 1. Note that the front end of the compiler is concerned with the machine independent transformations and optimizations, while the back end is concerned with the machine dependent transformations. The interface between them is the A-text. Operands in A-text are generic, symbolic registers of arbitrary length. Frequently, micro-machines have a large number of fast registers. Even though these registers often have inhomogeneous usage constraints, it is important to optimize the use of these resources. Hence an inherent design philosophy in PL/MP is to assume, early in the front end, a machine with an indefinite set of registers. Source level variables are loaded into the register space and kept there. Their values as well as those for the compiler generated temporaries are stored into memory only if absolutely necessary. The traffic between memory and register space is thus kept to a minimum.

The internal A-text is basically a register transfer language. However, it is general enough to support a large class of high level languages and low level micro-machines. It consists of primitive and complex operations, which may or may not be all supported by a given object machine. Operands in A-text are constants, labels or generic registers. All computational operations are register-to-register. The only operations addressing memory space are loads and stores. Thus all addressing code has been exposed at this level. A sample list of primitive and complex A-text units is shown below.

Note that the complex A-text operations, such as add indirect, load indirect, etc., can all be replaced by a sequence of equivalent primitive operations. For most ma-
chines an add-indirect-with-offset, for instance, will be more cost effective than the equivalent set of primitive instructions. However, if the variables are used at several places in the program then under certain conditions many primitive addressing instructions can be optimized out by code motion or hoisting.\(^2\)

THE ENVIRONMENT FOR CODE CONSOLIDATION

Assume that we have a machine with a set of general registers \( R_1 \) through \( R_{16} \) such that only \( R_1 \) through \( R_4 \) can be used as memory address registers. We further assume that the machine supports indirect addressing with and without
TABLE I.—Primitive and Complex A-text Operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitive Operations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>R1, R2, R3</td>
<td>add registers 2 and 3 and store result in 1</td>
</tr>
<tr>
<td>ADC</td>
<td>R1, R2, R3</td>
<td>add with carry</td>
</tr>
<tr>
<td>BLE</td>
<td>L, R1, R2</td>
<td>branch to L if R1 less than or equal to R2</td>
</tr>
<tr>
<td>LOAD</td>
<td>R1, p</td>
<td>load into R1 from location p</td>
</tr>
<tr>
<td>NOT</td>
<td>R1, R2</td>
<td>store into R1 one’s complement of R2</td>
</tr>
<tr>
<td>CALL</td>
<td>L</td>
<td>call subroutine L</td>
</tr>
</tbody>
</table>

| **Complex Operations** | | |
| LOADO | R1, p, n | load with offset [R1, <MEM(p+n)] |
| LOADIO | R1, R2, n | load with offset indirect [R1, =MEM(R2)+n] |
| LOADI | R1, R2 | load indirect |
| ADDI | R1, R2, R3 | add indirect |
| ADDIM | R1, p, k | add immediate [R1, =p+constant k] |
| ADDII | R1, R2, R3, n | add indirect with offset |

offsets. Thus, in this respect the target machine is at a higher semantic level than that of the primitive A-text.

Figure 2 shows a segment of an example in which we want to add a variable v to an array A of length 100 bytes. Both A and v are stored in a data area whose origin is at address p. The array’s first element coincides with the area’s origin and variable v is stored at an address 100 bytes beyond this origin. Thus, as shown in the figure, we first load into a register @a the address of array A. At program point p4 the array element is then loaded into register x through the load indirect instruction. The variable v is similarly loaded into register y, and it is added to x and stored in register z. Finally, the contents of z is stored back into the array, and the array element address @a is incremented by one to get to the next element. After iterating through the loop 100 times the program branches to point p7 where we proceed to other parts of the overall program. Figure 3 shows, in primitive A-text, the program after addressing code optimization by code commoning and moving the addressing code outside of the loop.

Notice that the sequence of code (p2,p3) which loads the variable v into the symbolic register y can be combined into the single instruction ‘load indirect with offset’, i.e.,

LOADIO y,@b,100  [Ry <-MEM((@b)+100)]

Once the above consolidation is performed the code sequence (p2,p3,p5) can be combined, so resulting, at p5, in a single add-indirect-with-offset instruction

ADDIO z,x,@b,100

Figure 4 shows the sample program after code consolidations at p2, p3, p5, and p7.

Notice that in the above consolidation process we have eliminated the register y containing the value of the variable v. Hence these consolidations are effective only because the symbolic register y, itself, is not referenced elsewhere in the program. Consider a slightly different situation, shown in Figure 5, where the register y is used at point p8. The subtract immediate instruction involves the register y and cannot be consolidated with the addressing code for y. This implies that we must keep the value of v in this registry.
Figure 3—Sample program in A-text

Figure 4—Sample program after code consolidation
Therefore, the instruction first consolidated from the sequence at (p2,p3), i.e.,

\[ \text{LOADIO y,b,100} \]

cannot be deleted in the next consolidation iteration. Hence, for this case, replacing the code at p5 and p7 with the more complex instructions would actually increase the object code size.

It is clear from the above example that the decision to perform code consolidation can be made intelligently only after examining the global usages of the variables in question. It is not surprising, therefore, that the compiler strategy is to first generate A-text with only primitive instructions. The primitive A-text, with addressing code fully exposed, is then subjected to global code optimizations such as commoning and code motion. Finally, the possibility of code consolidation is explored to generate an equivalent A-text using complex instructions where advantageous.

CODE CONSOLIDATION ALGORITHM

We have seen, in the previous section, instances where scattered code sequences can be consolidated. We will now describe the procedure for performing code consolidations. The procedure involves: (1) recognizing a possible consolidation candidate, and (2) performing the actual consolidation if it is profitable to do so.

Code pattern classification

For many representative machines, the primitive A-text code patterns that correspond to complex instructions consist of an addressing code sequence followed by a computational operator, which we will call the terminal operator. Furthermore, it is often possible to find a set of terminal operators that share an identical addressing code sequence. Sequences having this commonality are grouped together to form a pattern class. For instance, the code sequence p2, p3 in Figure 3 is a member of the class that contains indirect-with-offset operations such as LOADIO, STOREIO, and so forth. Similarly, the sequence at p3, p5 is a member of the indirect operations class.

For many machines there exists a partial ordering among the code pattern classes. For example the code sequence (p2,p3,p5) corresponds to the complex operation ADDIO. Notice that the sequence (p2,p3) forms the complex operation LOADIO, and similarly (p3,p5) forms the operation ADDI. Hence the arith-indirect-with-offset(AIO) class is greater than the class containing LOADIO as well as the class containing ADDI.

The patterns for our fictitious machine may be classified as below, where the first instruction in the sequence is called the ‘token’, and the primitive instructions are divided into arithmetic, memory-register classes, and so forth.

From this classification we can derive the substitution rules for these patterns as shown in TABLE III.
That is, the instruction in C3 (e.g., ADDIO) replaces either the sequence consisting of LOADIO followed by the corresponding arithmetic instruction (e.g., ADD) or the sequence of ADDIM and a sequence from C2 (e.g., ADDI). Notice that all tokens are specific instructions either primitive or complex. Furthermore, a code sequence in a secondary pattern class, such as C3, can always be formed from sequences of a smaller class. It is obvious that if no code pattern in C1 or C2 can be profitably consolidated, for instance, then no pattern in C3 can be profitably consolidated either. Hence it suffices to perform consolidation starting from the smallest element in the partial ordering defined by the pattern classes.

Code sequence recognition

Before we can make a judicious consolidation decision based on relative costs, for a given A-text and a specific pattern class C, we must first find a valid code sequence in that class. This involves (1) finding a sequence of code satisfying the substitution rules, and (2) checking that the operands in this sequence satisfy certain data flow constraints.

Notice that, starting from the token of a code sequence, each operation defines a new symbolic register. This defined value, unaltered, must be used in the immediate successor operation in the sequence. Furthermore, the target variable at the token, called the substitution variable, must not be altered along the path containing the code sequence so that it can be substituted at the terminal operation.

Since the elements of a sequence may be scattered throughout the program, it is a difficult task to check for these conditions. Fortunately, very efficient global data flow techniques have been developed that enable us to generate certain global relations among program variables. We will now introduce some of these data flow concepts, and then show how they can be used to find a valid code sequence for a given class.

<table>
<thead>
<tr>
<th>Table II—Pattern Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
</tr>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
</tr>
</tbody>
</table>

Let \((v, p)\) denote a variable \(v\) defined at program point \(p\). We say \((v, p)\) reaches \(q\) if there is a path from \(p\) to \(q\) such that \(v\) is not redefined along that path. If \(q\) is a program point with two or more predecessor statements then we say \((v, p)\) is available at \(q\) if \((v, p)\) reaches \(q\) through every one of its predecessor statements. Availability implies that if \(v\) is also defined at another point, say \(r\), then \((v, r)\) cannot also reach \(q\). Since the reach and availability information can be determined efficiently for any program, we can state the following:

**Variable Flow Condition:** if \(p_i\) and \(p_j\) are two consecutive code points in a code sequence then the variable defined at \(p_i\), as well as the substitution variable for the sequence, must be available at \(p_j\).

We have already mentioned that every substitution sequence starts with a known token. Hence having found a token for a given class we must start scanning the text to find the next code point in the sequence that satisfies the variable flow condition. The following constraints allow us to search through the program rather efficiently. First we will introduce two additional data flow concepts.

The first concept we need is the so-called **variable liveness.** If a variable \(v\) defined at \(p\) reaches a point \(q\) and is such that there is a subsequent use of \((v, p)\) in a path starting at \(q\) then we say \((v, p)\) is live at \(q\); otherwise we say \((v, p)\) is dead at \(q\). Thus we need not search along a path where the variable being searched is either dead or not available.

The second concept we need is that of **node dominance.** (For the following discussions we use the term node to denote a block of straight line code in a program. Hence \(p_0, p_1, p_2, p_3\) in Figure 5 constitute a node. Similarly, \(p_5, p_6\) is a node.) We say a node \(d\) in a program flow graph dominates another node \(n\) if every path from the program entry node to node \(n\) goes through node \(d\). For a given program a dominator tree can be obtained such that there is an edge connecting nodes \(d\) and \(n\) if and only if \(d\) dominates \(n\). For the program in Figure 3 the dominator tree is simply the graph minus the loop edge from \(p_6\) to \(p_4\).

Notice that if \(p_1, p_2, p_3, \ldots, p_i\) is a substitution sequence then in order to replace this sequence at point \(p_1\) by a more complex instruction, the node containing \(p_1\) must dominate that containing \(p_i\), and so forth. Therefore, starting from the token at \(p_1\), we need only follow the path defined by the dominator tree. Fortunately, for any flow graph a dominator tree can be obtained very efficiently as a part of the general data flow analysis. 

From the collection of the Computer History Museum (www.computerhistory.org)
The overall algorithm

Once we have found a valid substitution sequence the next question is whether or not it is advantageous to actually consolidate the code. Notice that if \((p_0, p_1, \ldots, p_n)\) is a substitution sequence, then we can always replace the operation at \(p_1\) by the complex operation that is logically equivalent to the function of the sequence. However, since the replacement operation is usually more complex than the original operation at \(p_1\), we presume it is advantageous to consolidate only if all the code at the other points in the sequence can be deleted. This is possible only if the intermediate variables are all used by terminal operations of the same class and found to be valid for consolidation. Otherwise, we cannot delete them, and hence there may be no advantage in replacing the original operation at \(p_1\) by a more complex one.

Starting with the token we search for all possible substitution sequences of the given class. If a code sequence is found to violate either the variable flow condition, or one of the intermediate variables is used in a statement not allowed by the sequence substitution rules, then the intermediate addressing code cannot be deleted. Hence no code consolidation will take place and the process terminates.

Note that if a code pattern is greater than another pattern in the pattern class ordering, as is the case with \(C_3 > C_2\) and \(C_3 > C_1\) in our previous example, then in searching for a sequence for \(C_3\) we must encounter a consolidated text unit in either class \(C_1\) or \(C_2\). Otherwise, since we always start the consolidation process for the smaller class first, this implies that no code can be advantageously consolidated for the patterns in \(C_1\) and \(C_2\). Hence no code pattern in class \(C_3\) can be consolidated either. In fact, the substitution rules for \(C_3\), as shown in Table III, clearly indicate that a valid sequence in that class must contain either a \(C_1\) or a \(C_2\) operation.

The above discussion can be summarized by the following:

Algorithm C

Input: Program expressed in A-text, program flow graph, variable liveness, availability, and dominator tree, pattern classes and substitution roles.

Output: Consolidated A-text.

Method:

1. Starting from the smallest pattern class \(C\) that has not been previously processed, perform the following steps. If no unprocessed class is left then terminate.
2. If consolidation failed for all lesser classes of \(C\) then go to step 1, otherwise proceed to step 3.
3. Starting from the root of the dominator tree, look for a previously unprocessed token of the given class. If a token is found then proceed to step 4, otherwise go to step 1.
4. Search for substitution sequences in class \(C\) along the paths defined by the dominator tree.
5. If no substitution sequence is found or if the variable flow condition is violated then return to step 1.
6. If a use of an intermediate variable is made by an operation not part of a code sequence then go to step 1.
7. Replace terminal operations found by appropriate operations in class \(C\), and delete the intermediate addressing code. Return to step 3.

CONCLUSIONS

We have discussed some of the techniques used for machine dependent code optimizations for a micro-program compiler. One of these techniques, namely code consolidation, has been discussed in detail. It should be obvious that the compile-time efficiency of these optimizations is heavily dependent upon the efficiency of the global data flow analysis package used by the compiler.

The code consolidation process, as described, may seem to be rather complex. However, for typical real machines, the number of pattern classes turns out to be rather small, and the substitution sequence length is also usually short. Hence with the use of a high-powered data flow package, the algorithm indeed can produce very efficient code with reasonable computation.

Experiments conducted for one of the commercially available micro-processors, namely the M6800, have shown that code consolidation together with addressing code optimization have contributed a 30-40 percent improvement of the object code produced by the PL/MP compiler.

ACKNOWLEDGMENT

It is a great pleasure to thank Hans Schlaeppi, manager of the PL/MP compiler development group, for providing the environment under which the research reported in this paper was carried out. Thanks are also due to many of my colleagues in the compiler group, without whose assistance this work would have been impossible.

REFERENCES
