D. Boyle
P. Mundy
T. M. Spence

Optimization and Code Generation in a Compiler for Several Machines

This paper describes optimization techniques that have been implemented in a compiler which was designed to produce code comparable to that produced by hand. Additional optimization methods were incorporated into successive versions of the compiler. It was found that no single method was effective with all compiled programs but that each of the techniques described was effective for some programs.

Introduction

This paper describes the optimization and code generation techniques which have been implemented in a compiler which generates code for several different machines. The compiler was designed to produce code which, in terms of space, is no more than 10% larger than that produced by hand. One version of the compiler has been used extensively to produce microcode for various peripheral devices. In these cases, the target machines have proprietary instruction sets. A second version of the compiler produces code for the IBM 8100, and it was used to develop the entire multiprogramming operating system for that machine. A description of the language of the latter version of the compiler can be found in [1].

The development of the compiler extended over a period of several years. Periodically, a new version of the compiler was made available to the user community, with each successive version increasing the degree of optimization. The section on "Results" shows how the code size decreased with successive releases for a representative sample of programs.

The compiler achieves a high degree of optimization by applying a number of techniques. The salient techniques are presented below. We have chosen to present these in the order in which they were implemented. The "Results" section gives an idea of the effectiveness of each technique. We believe that the handling of special register sets is unique, as is the table-driven implementation of built-in instructions. Although well known, some of the other methods are described in this paper because of their significant influence on the effectiveness of the optimization strategy as a whole. The language is frequently described as a subset of PL/I, from which it derives its terminology and syntax. However, there are a number of semantic differences which make it an improper subset. The target machine(s) are exposed in the language by providing the REGISTER storage attribute and Built-in Instruction (see below). Furthermore, there are many restrictions placed on the language to prevent the generation of cumbersome code sequences. In spirit, the language resembles the programming language C [2].

The structure of the compiler follows the classical front-end, optimization, and code generation approach. The optimization phase can be bypassed at the user's option.

Code generation

The code generation phase performs one pass through the program text to generate code. This phase keeps a table describing the current register contents for scalars and constants [3]. This form of history is limited to basic blocks, and there is no attempt to keep interstatement his-

Table 1 Improvement with succeeding compiler versions.

Module name	*Size in bytes of sample programs*						
	Version 1	*Version 2*	*%*	*Version 3*	*%*	*Version 4*	*%*
LYNXOOP	2,000	880	56.0	872	0.9	868	0.5
LEVEL7X	800	670	16.3	596	11.0	520	12.8
TEST	112	108	3.6	84	22.2	76	9.5
XKEALSCN	3,248	3,096	4.7	2,120	31.5	2,084	1.7
XKEAINFO	1,997	1,877	6.0	1,393	25.8	1,309	6.0
XKECGADD	4,728	4,744	−0.3	3,316	30.1	3,276	1.2
HGSCIDV	1,405	1,345	4.3	1,001	25.6	969	3.2
IKT0940A	218	202	7.3	190	5.9	190	0.0
IKT09405	134	122	9.0	122	0.0	122	0.0
IKT00413	328	280	14.6	242	13.6	242	0.0
HANS3	356	352	1.1	240	31.8	236	1.7
HANS5	1,211	1,191	1.7	659	44.7	651	1.2
IGRMGRV2	3,172	2,772	12.6	2,412	13.0	2,380	1.3
IGARIOCS	968	848	12.4	780	8.0	696	10.8
KDBFLIH	448	364	18.8	352	3.3	276	21.6
BOYBLUE	448	408	8.9	408	0.0	336	17.6
TOTALS	21,573	19,259	10.7	14,787	23.2	14,231	3.8

The % column is the percentage size decrease between each version and the preceding one.

tory of expressions. There is a great deal of machine-dependent special casing in this phase. We believe that such special casing is the first step to effective optimization.

Version 1 of the compiler lacked the optimization phase, but its code generation phase had most of the special casing which is in the current version of the compiler. Therefore, the version-1 column of Table 1 presented later in "Results" gives a rather good measure of the best that one can do on a single pass down the internal text.

Holding scalar variables in registers

In version 2 we introduced the optimization phase (which grew over the next several releases). This meant that the compiler began to take program topology into account.

Whereas in version 1 all user-declared data were mapped into main storage, in version 2, user-declared scalar variables were automatically mapped into registers so that no load/stores are needed. Each variable is given a "profit," which is a measure of the payoff for placing the variable in a register. Assignments are done to maximize total profit. A given variable is bound to a particular register throughout the life of the program being compiled. If variable activity permits, several names are bound to the same register. The actual register assignment algorithm used is one described by Day [4]. The version-2 column of Table 1 gives us an idea of the benefits from this approach.

Placing common expression computations in registers

The next compiler version treated expressions instead of merely dealing with simple scalars. As text is read, a "candidate table" of expressions found in the text is constructed. This table shows where a given expression is referenced or killed. By using techniques described by Ernst [5], we are able to determine the program points at which each expression value must be reestablished or can simply be used. Each entry in the expression candidate table is then treated as if it were a user-declared scalar, and it is assigned to a register using the same methods as are used for scalars. Expression candidates and scalar candidates vie for the same register. The internal text and dictionary are then modified to make it appear to the code generation phase as if the original user program were written in terms of user-written variables. The version-3 column of Table 1 reflects this addition.

Special register sets

The optimization techniques described so far are machine-independent. The detection of expression or variable usage, the counting of profit, and the tracing of candidate activity have little or nothing to do with a particular machine. Machine dependence is simply reflected in a table which lists the available registers.

Machine dependence is introduced by the fact that some of the supported machines have nonhomogeneous

registers. For example, although bits can be tested in any register, it is, on one machine, more efficient to test bits if the datum is in register number 2. Several such machine-dependent register characteristics were implied by the instruction sets.

This problem was addressed by introducing the notion of special register sets. All registers sharing a particular property (*e.g.*, "able-to-do-fast-bit-test") are marked as being in a special register set. (Some register numbers may be in several special register sets.) While reading text, an occurrence of a particular candidate for assignment is checked to determine whether this occurrence would imply that inclusion in a given register set is desirable. If so, the candidate is marked as being "desirable to place in special register set *n*."

When candidates are actually assigned to registers, the special register sets are tried first using only those candidates which are so marked. This tends to put candidates into register numbers which allow the code generation phase to exploit the use of special registers. Note that a marked candidate which fails to be assigned to the preferred special register set may still be assigned to a register not in that set. Also, unmarked candidates may be assigned to a special register set when no marked candidate is active at the particular program points at which the unmarked candidate is active.

Pre-optimization transforms

We then introduced machine-dependent transforms at the start of optimization. This can be viewed as recoding the original program so that features of the subsequent optimization and code generation phases tend to be exploited. We have incorporated a number of such transformations in the compiler, but the one described below should give an idea of the approach.

```
DCL C(4) FIXED(15) BASED;
DCL PT POINTER;
⋮
PT→C(J) = 0;
```

The compiler's front end converts the subscript expression into a byte offset expression. This can be represented as

```
PT→C[(J*2)-2] = 0;
```

where [(J*2)-2] is the offset to add to PT to address PT→C(J).

If we are compiling for a machine which has index registers, this is a good form for internal text. The optimization phase will tend to assign PT to a base register, and [(J*2)-2] can be assigned to an index register. However, sometimes we are compiling for a machine which lacks index registers. In this case the text is transformed to look like the following:

```
(PT+((J*2)-2))→C = 0;
```

This exposes the full addressing calculation to optimization and allows the expression (PT+((J*2)-2)) to be assigned to a single register. When the text transformation phase encounters a bit assignment statement where the source is a bit constant, it checks whether the previous statement was a bit assignment that refers to the same byte or halfword. If this is the case, the previous statement is deleted and a single character operation is produced which combines the effects of the two statements. This process is repeated until this bit history is terminated by the occurrence of any statement which is not a bit assignment referring to the same byte or halfword.

The following examples illustrate the effect of bit commoning:

```
DCL 1 S BDY(HWORD) BASED(PT),
      2 SA BIT(8),
        3 (S1,S2,S3,S4) BIT(1),
        3 S5678 BIT(4),
      2 SB BIT(8),
        3 S9TO16 BIT(8);
S1='1'B;
S2='1'B;
S3='1'B;
S4='1'B;
S5678(1:2)='00'B;
S9TO16(5:8)='1111'B;
```

This sequence of statements would be transformed to the single character operation

```
SA = (SA & 'F3FF'X) | 'F00F'X;
```

Bit testing

In order to help the code generation phase to exploit particular instructions for testing bits in a byte, a further transformation is performed for those bit-testing operations which are one byte long. This involves the use of four special internal operators:

BON—Test specified bits equal to '1'B
BOF—Test specified bits equal to '0'B
NBON—Test specified bits not equal to '1'B
NBOF—Test specified bits not equal to '0'B

Each of these operators specifies the name of the byte under test and a bit mask which indicates which bit(s) are to be tested. This makes it simple for the code generation phase to produce the appropriate "test with mask" instructions.

Consecutive bit tests which address the same byte can be combined. For example, assume that the following statement appears for the data as declared above:

```
IF S1='1'B & S2='0'B & S3='1'B
   & S4='0'B & S5678(1:3)='101'B · · ·
```

These consecutive bit tests would be combined as follows:

```
IF (SA BON '10101010'B) & (SA BOF '01010100'B) · · ·
```

which allows us to test by merely issuing two of the special bit testing instructions.

Jump optimization

Some of the supported machines contain short jump instructions which allow a backwards or forwards branch to a point near the current location. A branch which must go a greater distance must be done with a long branch.

The compiler optimizes for time and space by generating short jumps whenever the target is in range. This is performed by the "goto-optimization" phase, which runs after the code generation phase. This phase operates on a table containing all the branch instructions and labels together with their displacements from the start of generated code. The current displacement is incremented by the size of each instruction generated. All branch instructions are initially assumed to be "short."

An iterative process then operates on the table, and any branch which is out of short jump range is marked "long." This may cause existing short jumps to become "long" and so on. The process continues until all jump instructions have been determined to be either "long" or "short."

The following pseudo-code illustrates the basic algorithm for goto optimization:

```
/*************************************************/
/* all "gxx" entries in the goto table
are initialized to "short" */
/*************************************************/
DO UNTIL one complete pass with no change;
  DO over goto table;
    IF table entry is of type "gxx" &
        entry is "short" &
        displacement needs long THEN
      DO;
        mark this "gxx" entry as "long";
        increment displacements of all following table
          entries by the difference in size between a
          short jump and a long jump;
      END;
  END;
END;
```

A further function performed by this phase is jump-to-jump optimization. This is applicable when optimizing for space rather than execution time.

For each branch which cannot be achieved with a short jump:

1. Determine whether there is a branch instruction within short jump range which causes control to (eventually) reach the desired target. If so, insert a label in front of it and change the initial branch target to be this label.
2. If there is no such branch within the range of a short jump, find a suitable place to insert an intermediate jump instruction (*e.g.*, after an unconditional branch). Then change the initial branch to be the label on the intermediate jump.

Access to built-in machine instructions

The compiler has a language feature called Built-In Instructions (BII's). This allows the user to code any instruction available on the hardware as a statement. The user codes full expressions as the instruction's arguments, and the compiler generates code as needed to perform necessary housekeeping such as loading and storing registers.

The instructions are viewed as in-line routines which require expressions with particular attributes as operands. This leads to a table-driven implementation technique which makes it possible to support all of the instructions on the target machine with little compiler implementation cost.

We defined a table structure which allows one to encode a description of any machine instruction. An instruction entry has information such as the instruction name; the number of operands required; the permissible attributes for each operand; the number and type of registers required for each operand; an indication of which operands are inputs to the instruction; an indication of which operands are outputs from the instruction.

A macro processor was used to provide a "language" which can be used to write these instruction descriptions. Persons who know little or nothing about the internal characteristics of the optimization or code generation phases can define and modify these tables.

For example the load-byte instruction of the IBM 8100 is described as follows:

```
?BII (L)          /* A BII DEFINITION - MNEMONIC="L"     */
   FORM (RS)      /* A REGISTER - STORAGE FORM           */
   SEQFLOW        /* "FALLS THROUGH" TO NEXT INSTR.      */
   NONEEDCC       /* DOESN'T DEPEND ON CONDITION CODE */
```

```
   NOSETSCC    /* IT DOESN'T SET THE CONDITION CODE   */
   SIZE(4);    /* NUMBER OF BYTES IN INSTRUCTION      */
?BIIARG        /* DESCRIPTION OF FIRST ARGUMENT       */
   ATTR(       /* MUST HAVE ONE OF THESE ATTRIBUTES */
          FIXED(8),CHAR(1),BIT(8))
   REGTYPE(BYTE,PRIMARY) /* TYPE OF REGISTER REQUIRED */
   ACCESS(VALUE)      /* VALUE OF ARG IS IN REGISTER */
   OUTPUT;     /* REGISTER IS SET BY INSTRUCTION      */
?BIIARG        /* DESCRIPTION OF SECOND ARGUMENT      */
   ATTR   (FIXED(8), CHAR(1),BIT(8) )/* PERMITTED ATTRS */
   REGTYPE(FULLWORD,NOTZERO)   /* REGISTER TYPE IS
                                  ANY FULLWORD SAVE
                                  REGISTER ZERO       */
   ACCESS(BASEDISP)  /* BASE-DISPLACEMENT FORM WHICH
                        MEANS THAT THE ARGUMENT NAMED
                        ACTUALLY RESIDES IN STORAGE AND
                        THE REGISTER IS TO GET A BASE
                        ADDRESS                        */
   DISP(-32768:32767) /* PERMISSIBLE DISPLACEMENT
                        RANGE                          */
   INPUT;          /* ARG IS NOT CHANGED BY INSTRUC'N */
```

When a built-in instruction appears in the text, the BII module is called. This module interprets these tables and calls the expression handling routines as needed to properly process the user's statement. There is one such *BII* module for each phase of the compiler. There is only one copy of the tables, which is used in a read-only manner by all phases.

Local block optimization

Before emitting the code for a basic block, the code generation phase makes a pass against the code which it has just generated to determine whether any final optimizations based on the peculiarities of the target instruction set can be done. The individual optimizations are defined by means of macros. These macros generate tables which are interpreted by the local block optimization module. The macros allow new optimizations to be added very easily.

Since the process is iterative, some optimizations together can have a dramatic effect. Consider the following (nonoptimized) sequence of code which would be applicable for the IBM 8100:

```
* X=X-1;
          SRI   X,1        SUBTRACT 1 FROM BYTE
                           REGISTER "X"
* IF X¬=0 THEN
          RL    X,0        ROTATE "X" SIMPLY TO SET
                           CONDITION CODE
          JZ    @RF00001   JUMP ON PROPER CONDITION
                           CODE
* GO TO    P→BLAB;
          BR    P          BRANCH UNCONDITIONALLY
                           USING ADDRESS IN REGISTER
                           "P"
@RF00001  DS    0H
```

Following are the local block optimizations performed on this sequence:

First the RL is removed since it follows SRI, which has already set the condition code according to the value of X.

```
* X=X-1;
          SRI   X,1
* IF X¬=0 THEN
          JZ    @RF00001
*    GO TO P→BLAB;
          BR    P
@RF00001  DS    0H
```

Then, the JZ/BR/@RF · · · sequence is changed to BNZR P.

```
* X=X-1;
          SRI   X,1
* IF X¬=0 THEN
*    GO TO P→BLAB;
          BNZR  P
@RF00001  DS    0H
```

Finally, the SRI/BNZR sequence is changed to BCTR.

```
* X=X-1;              SUBTRACT 1 FROM BYTE REGISTER
* IF X¬=0 THEN        "X" THEN BRANCH IF RESULT NOT
*    GO TO P→BLAB;    ZERO
          BCTR X,P
@RF00001  DS    0H
```

Thus the four instructions are eventually transformed to one BCTR.

Results

Table 1 shows the results of compiling a set of programs using different versions of the compiler. Combined with Table 2, this gives us an idea of the degree of improvement from the various optimization methods.

The results in Table 1 are the size (in bytes) of the resulting object code of each program. This includes both executable instructions and static data for literals and the like. (We have made sure that the programs do not have a large amount of user-declared static data because this would have distorted the results; *e.g.*, "DCL X CHAR (8000) STATIC;" cannot be optimized.

Table 2 Four versions of the compiler and the optimization methods added in each version.

Version	*Optimization methods*
1	No optimization (except for code generation special casing)
2	Scalar variables kept in registers
3	Common expressions kept in registers Jump optimization Special register sets
4	Pre-optimization transformations Bit assignment commoning Bit testing commoning Local block optimization

Summary

The optimization methods described in this paper have been successful. The degree of optimization has been such that users have found it unnecessary to resort to hand code even in the most critical parts of their systems. Our original goal was to have the compiler produce code which (in terms of space) was no more than 10% larger than hand-produced code. On several occasions, we have taken existing assembler code and reprogrammed it to demonstrate that the compiler can match hand-produced code for time and space. The 10% objective has been met, based on the results of such exercises.

Our results demonstrate the effectiveness of the register assignment method by which variables and expression values are assigned to registers. We also believe that the importance of special casing, local block optimization, and other techniques described here cannot be ignored. No single method effectively optimizes all programs. For each technique described here, there exist modules which were highly optimized only when that particular technique was implemented in the compiler.

During the development of the compiler we made conscious efforts to implement the compiler so that it would be relatively simple to support new machines. We have partially succeeded. We say "partially" because such modifications can only be done by members of our own development group who are familiar with the compiler implementation.

We were pleasantly surprised at how well the special register set approach handled the problem of nonhomogeneous registers. The method as implemented has an obvious flaw: a relatively unprofitable item may be assigned to a special register, preventing some more profitable item from being assigned to any register. In practice, this does not seem to happen often. We have inspected many samples of user code, and there appears to be a strong affinity between profitability and special register desirability. We have had only two known cases in three years where users had code which exhibited this problem. Still, we would be happier if we could discover a theoretically sound solution to the nonhomogeneous register problem.

The problem of work register allocation by optimization has not been satisfactorily solved. It is possible that the optimization phase can assign items to so many different registers that the code generation phase is unable to evaluate a particular expression because optimization has left no work registers for code generation. This is addressed by having the optimization phase estimate the work register requirements of the later code generation phase. This must be a worst-case estimate because it is not known at the time that this estimate is done whether or not various components of expression will be precomputed or not. Hence, the optimization sometimes ends up leaving a few registers free which it could have used (given the post-code-generation aftersight that people reading the code have). Arriving at the proper estimate function proved to be a delicate exercise in fine tuning. We hope that some researcher can find a solution to this problem.

We are not sure how well the methods described would work on nonregister machines. Optimization techniques could be made to map noninterfering variables and expression values onto storage instead of into registers. We believe that this would give excellent results. However, we have no experimental evidence to validate this because all machines supported by this compiler were register machines.

Acknowledgments

We want to acknowledge the important contributions of other members of our development group, namely R. J. Gauvareau and B. J. Sawchuck of IBM Kingston and D. W. Knowles of IBM United Kingdom. We also are grateful to the project management for establishing an environment which made this activity possible. This includes G. G. Leffler, who managed the project in its initial phases, D. A. Kilpatrick, who later took over, and our department manager, R. J. Battista. We also owe a special debt to the late James Beatty, who gave us a great deal of valuable advice during the early phases of this project.

References

1. *Programming Language for Distributed Systems (PL/DS)*, Order No. SC27-0446-2, available through the local IBM branch office.
2. B. W. Kernighan and D. M. Ritchie: *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

3. J. Cocke and J. T. Schwartz, *Programming Languages and Their Compilers*, Second Revised Edition, Courant Institute of Mathematical Sciences, New York University, New York, 1970.
4. W. H. E. Day, "Compiler Assignment of Data Items to Registers," *IBM Syst. J.* **9**, 281–317 (1970).
5. C. P. Earnest, K. G. Balke, and J. Anderson, "Analysis of Graphs by Ordering of Nodes," *J. ACM* **19**, 23–42 (1972).

Received January 21, 1980; revised June 12, 1980

D. Boyle is located at IBM Information Services Limited, P.O. Box 41, North Harbour, Portsmouth, Hampshire P06 3AU, England. P. Mundy is located at the IBM United Kingdom Laboratories Limited, Hursley Park, Winchester, Hampshire SO21 2JN, England. T. M. Spence is at the IBM System Communications Division laboratory, Neighborhood Road, Kingston, New York 12401.