Programming Languages

J. J. Horning*
Editor

# A Case Study of a New Code Generation Technique for Compilers

J. Lawrence Carter
IBM Thomas J. Watson Research Center

**Recent developments in optimizing techniques have allowed a new design for compilers to emerge. Such a compiler translates the parsed source code into lower level code by a sequence of steps. Each step expands higher level statements into blocks of lower level code and then performs optimizations on the result. Each statement has only one possible expansion – the task of tailoring this code to take advantage of any special cases is done by the optimizations. This paper provides evidence that this strategy can indeed result in good object code. The traditionally difficult PL/I concatenate statement was investigated as a detailed example. A set of fairly simple optimizations was identified which allow the compiler to produce good code. More elaborate optimizations can further improve the object code. For most contexts of the concatenate statement, the code produced by a compiler using the expansion-optimization strategy described above compares favorably with the code produced by a conventional PL/I optimizing compiler.**

**Key Words and Phrases: compiler structure, optimizing compiler, code generation, PL/I compiler, concatenation, program optimization, optimization techniques, data flow analysis**

**CR Categories: 4.12, 4.13, 4.22**



Author's address: IBM Thomas J. Watson Research Center, P. O. Box 218, Yorktown Heights, N. Y. 10598

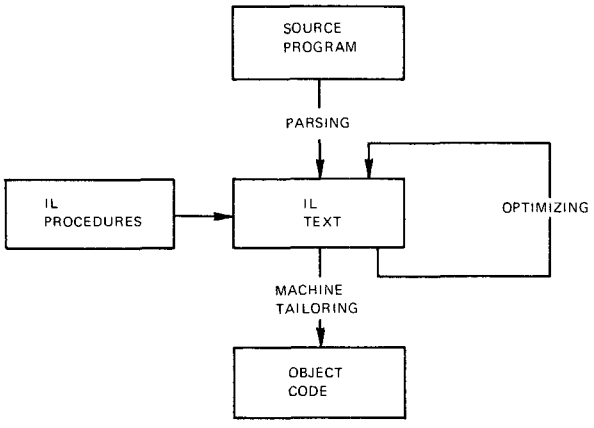
## Introduction

Figure 1 sketches a new design for compilers. Standard parsing techniques are used to translate the source code into an intermediate language (IL). Each IL statement consists of an operation name followed by a list of operands. The meaning of the IL statements produced by the parser is intended to be exactly that of the corresponding source language statements. These "high level" IL statements are defined in terms of "lower level" IL statements – ones closer to machine language – by *IL defining procedures* in the IL procedure library.

The procedures in this library have been written in IL. An IL defining procedure can be viewed as an interpretive routine for the corresponding operation. However, as will be explained in the next paragraph, the compiler can use the defining procedures in a different way. The IL statements used in a defining procedure may themselves be defined in the IL procedure library; however, recursive calls among the IL defining procedures are not allowed. The IL procedure library may also contain the IL versions of some user written procedures. For each procedure in the library, there must be some summary information which describes the procedure's effect on the program (e.g. which parameters are defined, what error routines may be invoked, etc.) This information is needed by the optimization routines to determine when the optimizations are legal. IL operations which do not have a procedure in the library are called *IL primitives*. The primitives are either operations that are close in meaning to machine instructions or invocations of procedures that are external to the compiler (such as system routines).

The IL text is subjected to a series of optimizations. *Procedure integration* replaces an IL operation (or a call to a user-supplied procedure) with the sequence of statements which it represents [2, 7]. Other optimizations can eliminate unnecessary temporaries, remove code which is not executed, and so on. The optimizations can be repeated until the IL program contains

Fig. 1.



only primitive statements, though this is not necessary. Leaving nonprimitive IL statements unintegrated makes the compilation faster but reduces the quality of the object code.

The optimized IL text is then sent to the machine tailoring phase, which assigns storage and registers, chooses machine instructions for the primitive IL statements, translates the remaining IL statements into calls to the corresponding IL defining procedures, and attends to the details needed to produce object code.

The philosophy behind the compiler is that the IL procedures are written to handle the most general case, and the optimizations will tailor the code to take advantage of special cases. The purpose of this paper is first to explore whether such a philosophy can actually work, and second to identify some of the requirements on the various parts of the compiler necessary to produce good object code. We do not describe the compiler design in detail, but it is hoped that the reader will gain an understanding of it through the examples. More information on the design can be found in [5].

## Concatenation

In order to study the worth of the above compiler scheme, we decided to write the IL procedures needed to handle the PL/I statement A = B||C; for character strings. These procedures were then used to generate object code for various examples, and this code was evaluated.

We chose concatenation since it typically is a difficult operation for compilers. In A = B||C;, B or C may overlap A in storage. If so, it may not be possible to directly move B or C to A since during the move the unmoved portion may be destroyed. Yet substantial time can be saved by making a direct move in cases where it is safe to do so. Another useful special case arises when A is the same as B, for then B need not be moved at all. Yet another consideration is that (on the IBM 360), only 256 bytes can be moved at a time; thus the primitive move operation must be in a loop. Still, we want to remove the loop in cases where it is known that no more than 256 characters are to be moved. Finally, there is the classical problem [4] of compiling LENGTH(B||C) so that only the code needed to do LENGTH(B) + LENGTH(C) is executed.

Traditional compilers handle these problems by examining the context of the concatenate statement to decide which code sequence to generate. Our approach is to have only one IL procedure which performs the operation, but also to have optimizations which can modify the procedure to take advantage of the special cases. A result of this study is that fairly simple optimizations are indeed powerful enough to do as well or better than a traditional compiler and more sophisticated optimizations can improve the code further. Before the IL procedures to handle concatenation are presented, a few optimizations will be outlined.

## Optimizations

*Procedure integration* is an optimization which replaces a procedure call with the set of statements it represents and makes the appropriate substitutions for the parameters and labels. One of the advantages of integrating a procedure is that the compiler will then have more information about the procedure's variables. For instance, if a parameter of a particular invocation of a procedure were a constant, and the procedure contained a test of that parameter, then the test could be eliminated. The next two optimizations are particularly useful when procedures are being integrated.

*Constant propagation* recognizes when all the operands of an IL primitive are known at compile time, and it can replace the primitive with a simpler primitive. For instance, if A is known to have the value 3, then constant propagation will replace B = A with B = 3, B = A + A with B = 6, and IF A = 3 GO TO LABEL with GO TO LABEL. *Dead code elimination* removes statements which cannot be reached from the program's entry points or which define variables which are not used before being redefined.

Another optimization which is needed to produce reasonable code for concatenation is *variable propagation*. If there is a statement which sets T equal to A, then, as long as A and T are not redefined, variable propagation can replace a subsequent use of T with a use of A and eliminate any subsequent statements which set T to A or which set A to T. One must be careful in applying variable propagation to ensure that the semantics of the program are not changed. In particular, if $s_1$ is a statement which sets the first N bytes of T equal to A and $s_2$ another statement, then we can replace a use of T with A in $s_2$ if all of the following criteria are met:

(a) A and T are not aliases; that is, the memory cells assigned to A and T cannot overlap. This insures that when T is set to A, A is not changed.

(b) $s_1$ backdominates $s_2$; i.e. any possible path through the control graph which reaches $s_2$ must go through $s_1$.

(c) No intermediate statement in any path which goes from $s_1$ to $s_2$ without repeating $s_1$ can modify A, T, or any variable aliased with A or T.

(d) If $s_2$ is a use of T which we wish to change to a use of A, then $s_2$ may not modify A or T (or any of their aliases) and may not use more than N bytes of T.

(e) If $s_2$ is an assignment of K bytes of A to T (or vice versa) which we wish to eliminate, then K must be no greater than N.

These conditions are not all necessary—for instance,

criterion (d) could be weakened to allow $s_2$ to modify A or T provided that the modification occurred after the use of T. Also, the conditions could be rewritten to allow several statements which set T to A. However, the conditions given here are sufficient for our purposes in this paper.

We wish that as much as possible, the writer of the IL procedures need not be aware of how the optimizations work. For example, consider the two sequences of code given in Figure 2 (which for clarity are depicted as flow graphs.) If constant propagation were to be applied to the first sequence, it would recognize that the first test could be replaced by a GOTO. However, since there are two different definitions of T.TYPE reaching the second test, it could not proceed any further until the dead code eliminator got rid of the unreachable T.TYPE = BIT statement. In fact, to determine that Y = X could be replaced by Y = 2, one would have to perform constant propagation three times, with dead code elimination in between. On the other hand, the second code sequence requires only two applications of constant propagation. We could therefore insist that the IL programmer always use the latter construction. However, we feel that this would be putting an unnatural constraint on the programmer. It is better to assume that constant propagation, dead code elimination, and variable propagation are all done together or repeated enough times that no further application of any of them would change the code. Incidentally code such as this could easily arise in the course of expanding the IL procedures.

## ECS Compiler

This section briefly describes a compiler (called the Experimental Compiler System compiler) which is currently under development at the IBM T. J. Watson Research Center. This compiler will use the procedure integration-optimization scheme described earlier. The following sections describe the results of hand-simulating the ECS compiler on the PL/I concatenation statement.

Figure 3 lists the IL statements which are needed to handle the concatenation of character strings. The concatenation statement does not need to worry about data conversions since the parser puts out separate instructions to do the conversions when they are needed. We have taken the liberty of writing our IL statements in a flexible format for the sake of clarity. Of course the actual compiler has a more restricted format. The IL defining procedures for some of the nonprimitive IL statements are given in Figure 4.

The target machine of the ECS compiler is the IBM 360 or 370; hence the IL primitives reflect their machine language. For instance, the primitive assignment statement A = B (N + 1 BYTES) corresponds to the MVC instruction. Thus it moves one more byte than specified by the parameter N, and N must be less than 256. On the other hand, the higher-level IL statement ASSIGN A FROM B (N BYTES) moves N bytes and N can be any nonnegative integer.

Each variable X of the source program has several attribute variables, such as X.LEN (which is the current length of a string), X. MAXLEN (the maximum length of a varying string), and X.VARY (whether the string is varying or not.) At the IL level, these attribute variables are treated just like other variables. If A is a variable, then :A: means the location that A is stored in and (A) means indirect on A, that is, the contents of

Fig. 2.

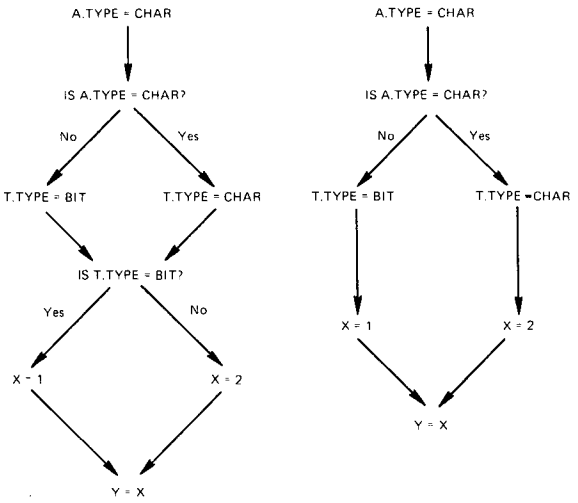



Fig. 3. Some IL statements.

| | |
|---|---|
| | **Nonprimitives** |
| $a = b \mid\mid c$ | Performs the concatenation of the character strings $b$ and $c$ and puts the result into $a$. |
| BUY $t$ AND SET $t = b \mid\mid c$ | Allocates a temporary $t$ into which $b$ and $c$ are then concatenated. |
| ASSIGN $a$ FROM $b$ ($n$ BYTES) | Moves the first $n$ bytes of $b$ to $a$. $n$ may be any non-negative integer. Results are undefined if the storage allocated to $a$ overlaps with that allocated to $b$. |
| PAD $a$ ($n$ BYTES) | Fills the first $n$ bytes of $a$ with blanks. |
| BUY CHAR STRING $t$ ($n$ BYTES) | Allocates storage and makes the necessary declarations for $t$ to be a new fixed-length character string of length $n$ bytes. |
| | **Primitives** |
| $a = b$ ($n + 1$ BYTES) or $a = b$ | Assigns the first $n + 1$ bytes of $b$ to $a$. $n$ must be between 0 and 255 inclusive, since this may be translated into an MVC instruction. The default when $n$ is not specified is 4 bytes. |
| $a = b + c$ or $a = b - c$ | Addition or subtraction of 4-byte integer variables. |
| IF *exp* GOTO *label* | An abbreviation for a variety of IL primitives doing conditional branching. |
| EXTEND $a$ ($n + 1$ BYTES) | Duplicates the leftmost byte of $a$ into the next $n + 1$ bytes. $n$ must be between 0 and 255. |
| NOOP | Does nothing. |
| GOTO *label* | Unconditional branch to *label*. |
| RETURN | Return statement for IL procedures. |

memory location A. Thus the sequence of P = :A:, (P) = B would have the same effect as an A = B statement. The compiler will change any occurrence of (:A:) to simply A.

The optimizations used by the ECS compiler on the IL text are procedure integration, constant propagation, dead code elimination, and variable propagation. The variable propagation optimization is able to recognize that (A) is a use of the variable A, as well as possibly being a use of (A). Thus, given the sequence P = :A:, (P) = B, variable propagation would replace the second statement by (:A:) = B, which would then be changed to A = B. In the examples, some additional optimizations (such as range analysis) are mentioned to show how they could further improve the compiler.

An interesting question arises as to what is the best order in which to apply the optimizations, but we largely avoid that question in this paper. Our strategy is to integrate one procedure at a time and after each integration to apply the other optimizations until no further applications can be made. Then another procedure will be integrated, and so on. In all the examples presented, it makes no difference on the final code which optimization is applied when there is a choice. However, there are programs where the order of optimization does make a difference; so some research is needed to find heuristics to guide this choice.

The ECS compiler also applies constant propagation and dead code elimination after the storage mapping phase. This allows expressions like :A: + 1 which really are constants at compile time to be recognized as such. We assume that the register allocator of the ECS compiler is good enough to avoid instructions which are obviously unnecessary, but we do not require it to find any sophisticated simplifications (such as combining two IL primitives into a single machine instruction.)

In general, writing the IL procedures was straightforward. Little thought was given to details of the machine or optimizations. There were two exceptions to this claim. The loops of the ASSIGN and PAD procedures were partially unrolled so that the first iteration was done before entering the loop. This lets the optimizations remove the loop if the length is a constant less than 256. A side advantage of this construction is that the loops have only one branch instruction. It does not seem unreasonable to request the IL programmer to use this form for loops.

The other time that an IL procedure had to be written carefully to be sure that the optimizations would be able to work effectively was in the expansion of A = B||C. As mentioned earlier, one cannot boldly move B and C to A because B or C may overlap A in memory. The most obvious way to avoid the overlap problem is to create a new, separate temporary T, then concatenate B and C into T, and finally move T to A. One may hope that the optimizations will eliminate the extra motion in cases when it is not needed, but this would take an extremely sophisticated optimization since the distinction between B and C becomes lost once they are combined into T. However, a much better method was discovered: First move B and C into separate temporaries T1 and T2, and then concatenate these variables into A. As the examples show, the variable propagation optimization described earlier is then able to eliminate the use of the temporaries when the variables are not aliased and to avoid moving A in the statement A = A||C.

Fig. 4. Defining procedures for some IL statements.

```
Expansion of A=B||C

     M=A.MAXLEN-B.LEN
     IF M>0 GOTO L1
     MOVE1=A.MAXLEN
     MOVE2=0
     EXTRA=0
     GOTO L2
L1:  MOVE1=B.LEN
     EXTRA=M-C.LEN
     IF EXTRA>0 GOTO L3
     MOVE2=M
     EXTRA=0
     GOTO L2
L3:  MOVE2=C.LEN
L2:  TOTAL_MOVE=MOVE1+MOVE2
     BUY CHAR STRING T1 (MOVE1 BYTES)
     BUY CHAR STRING T2 (MOVE2 BYTES)
     P2=:A:+MOVE1
     ASSIGN T1 FROM B (MOVE1 BYTES)
     ASSIGN T2 FROM C (MOVE2 BYTES)
     ASSIGN A FROM T1 (MOVE1 BYTES)
     ASSIGN (P2) FROM T2 (MOVE2 BYTES)
     IF A.VARY=true GOTO L4
     P3=:A:+TOTAL_MOVE
     PAD (P3) (EXTRA BYTES)
     RETURN
L4:  A.LEN=TOTAL_MOVE
     RETURN

Expansion of
     ASSIGN A FROM B (N BYTES)

     T=N
     IF T≤0 GOTO L1
     XT=:A:
     YT=:B:
     T=T-256
     IF T≤0 GOTO L2
L3:  (XT)=(YT) (255+1 BYTES)
     XT=XT+256
     YT=YT+256
     T=T-256
     IF T>0 GOTO L3
L2:  T=T+255
     (XT)=(YT) (T+1 BYTES)
L1:  RETURN

Expansion of PAD A (N BYTES)

     T=N-1
     IF T<0 GOTO L1
     A=blank (0+1 BYTES)
     IF T=0 GOTO L1
     P=:A:
     T=T-256
     IF T≤0 GOTO L2
L3:  EXTEND (P) (255+1 BYTES)
     P=P+256
     T=T-256
     IF T>0 GOTO L3
L2:  T=T+255
     EXTEND (P) (T+1 BYTES)
L1:  RETURN
```

## A Detailed Example

We shall walk through an example to show how the ECS compiler can tailor the IL to a special case. Suppose a PL/I programmer wrote the program

```
P: PROC OPTIONS (MAIN);
  DCL (B, C) CHAR(10), A CHAR(50);
  A = B||C;
  END;
```

This would be parsed into a set of assignments (B.TYPE = CHAR, B.LEN = 10, B.MAXLEN = 10, etc.) followed by the IL statement A = B||C. The procedure integrator would replace the concatenate statement with the code of Figure 4. Then the variable propagation optimization would notice that T1 in ASSIGN A FROM T1 could be replaced by B. Similarly the T2 in the next statement would be replaced with C. The constant propagator and dead code eliminator would make a number of simplifications, including determining the values of MOVE1, MOVE2, and EXTRA, eliminating all the conditional branch instructions, and determining that the statements defining T1 and T2 can be eliminated. The resulting code would be

```
P2 = :A: + 10
ASSIGN A FROM B (10 BYTES)
ASSIGN (P2) FROM C (10 BYTES)
P3 = :A: + 20
PAD(P3) (30 BYTES)
```

At this point, procedure integration would replace the ASSIGN and PAD statements with the procedures they represent. Let us look at what happens to the expansion of the ASSIGN A FROM B (10 BYTES) statement as the optimizations work on it. First, the result of applying constant propagation is to produce the code

```
   T = 10
   IF 10 ≤ 0 GOTO L1
   XT = :A:
   YT = :B:
   T = −246
   IF −246 ≤ GOTO L2
L3: (XT) = (YT) (255 + 1 BYTES)
  XT = XT + 256
  YT = YT + 256
  T = T − 256
  IF T > O GOTO L3
L2: T = T + 255
  (XT) = (YT) (T + 1 BYTES)
L1: NO OP
```

Dead code elimination would reduce this to

```
XT = :A:
YT = :B:
T = −246
T = T + 255
(XT) = (YT) (T + 1 BYTES)
```

Another iteration of constant propagation and dead code elimination would yield

```
XT = :A:
YT = :B:
(XT) = (YT) (9 + 1 BYTES)
```

Now variable propagation would change the code to

```
XT = :A:
YT = :B:
(:A:) = (:B:) (9 + 1 BYTES)
```

Finally, dead code elimination and (: :)-removal would reduce this to the single IL statement

```
A = B (9 + 1 BYTES)
```

The optimizations would act in a similar way on the other ASSIGN statement and on the PAD statement. Thus the original concatenate statement would become

```
P2 = :A: + 10
A = B (9 + 1 BYTES)
(P2) = C (9 + 1 BYTES)
P3 = :A: + 20
(P3) = blank (0 + 1 BYTES)
EXTEND (P3) (28 + 1 BYTES)
```

The constant propagation which occurs after storage locations have been assigned would recognize that P2 and P3 are constants. The final code would be

```
A = B (9 + 1 BYTES)
(:A: + 10) = C (9 + 1 BYTES)
(:A: + 20) = blank (0 + 1 BYTES)
EXTEND (:A: + 20) (28 + 1 BYTES)
```

Presumably, the machine tailoring stage would translate this into three MVC's and an MVI instruction in 360 machine code.

This example shows that the ECS compiler gives quite good code on a simple concatenation. In this case, traditional compilers also do well. In fact, the IBM OS PL/I optimizing compiler (hereafter called PLIOPT) produces slightly better code. Since B and C happen to be stored contiguously, a peephole optimization is able to recognize that the first two move statements can be combined into one statement. Of course, if it is felt that combining adjacent moves is useful, then this peephole optimization can be applied in the ECS compiler as well.

## Performance on Other Sample Programs

In this section we present a number of other situations involving concatenation and comment on how well the ECS compiler performs.

**Example 2.**

A = B || C; where the declarations are DCL (A, B, C) CHAR(100) VARYING;

Since it is known that A is not aliased with B or C, the moves to the temporaries can be eliminated as in the first example. However, B.LEN and C.LEN are not known; so the constant propagator cannot eliminate the loops in the assign statements. The machine level code produced by a hand-simulation of the ECS compiler had 46 instructions. On the same example, PLIOPT produced code having 29 instructions, including a call to the compiler-generated subroutine IELCGMV which contains 33 instructions. Thus the ECS compiler produces code which is 17 statements longer or 16 shorter, depending on whether you count the length of IELCGMV. If B.LEN and C.LEN are both

nonzero and B.LEN + C.LEN ≤ 100, then 32 statements would be executed using the ECS-produced code, while PLIOPT's code would have 35 instructions executed. PLIOPT avoided the usual overhead in calling IELCGMV by using a nonstandard linkage. The calling program put the parameters in the exact registers that IELCGMV expected to find them. Further, IELCGMV does all its work in those registers and therefore doesn't need to save registers or allocate itself memory.

PLIOPT recognized that the move of B to A could be performed without using a loop since B.LEN ≤ 256. However, it failed to see that the move of C to A doesn't require a loop either. This is mentioned to suggest the difficulty of "special casing." A traditional compiler which is to be able to handle a large number of different cases must contain code to recognize and deal with each of those cases. In the ECS compiler the task of tailoring the code is done by just a few optimizations.

Suppose the ECS compiler also included a range analysis optimization (see Harrison [6]) which knew that if A is a string, then A.LEN ≤ A.MAXLEN. This optimization could determine that MOVE1 and MOVE2 can be no greater than 100; so the loops in the ASSIGN procedures would never be executed. Thus more dead code could be eliminated, and the use of XT and YT would be avoided. The resulting code would have 30 instructions, 26 of which would be executed when B.LEN and C.LEN are positive and sum to no more than 100. If we throw in another optimization—symbolic interpretation—which recognizes that the successive statements T = T − 256 and T = T + 255 could be changed to T = T − 1, then two more instructions would be eliminated. Figure 5 summaries the data of this example. The "best possible" code that we could find had 15 instructions, including a BXLE, a BCT, and an LA instruction, each of which does the work of two or three others.

**Example 3.**

N = LENGTH(B||C); where B and C are varying length strings

One hopes the compiled code will simply perform N = LENGTH(B) + LENGTH(C), but many compilers (including PLIOPT) actually concatenate B with C. In the ECS compiler, the parser will produce the IL statements BUY S AND SET S = B||C, N = S.LEN. What happens now depends on the expansion of the first of these statements. If it is expanded as

```
K = B.LEN + C.LEN
BUY CHAR STRING S(K BYTES)
S = B||C
```

then after these procedures are expanded, variable propagation would substitute K for S.LEN in the BUY CHAR STRING procedure. Then the dead code eliminator would eliminate the entire concatenation procedure since neither S nor any of its attributes would have any uses. In fact, the entire code is simplified to the IL statements K = B.LEN + C.LEN, N = K. On another hand, if the BUY S AND SET S = B||C procedure uses a varying length temporary, then things are not handled quite as well. As before, variable propagation would eliminate the use of S.LEN, this time replacing it by TOTAL_MOVE, and all the ASSIGN procedures building up S would be eliminated. However, the remaining code would be too tangled up for the optimizations to completely straighten out, and the resulting IL program would have 13 primitive statements, eight of which would actually be executed.

Fig. 5. Summary of example 2.

| | Number of instructions in object code | Number of instructions executed when B.LEN > 0, C.LEN > 0 and B.LEN + C.LEN ≤ 100 |
|---|---|---|
| PLIOPTcompiler | 29 | |
| | + | 35 |
| | 33 in IELCGMV | |
| ECS compiler with basic optimizations | 46 | 32 |
| plus range analysis | 30 | 26 |
| plus symbolic interpretation | 28 | 24 |
| Best possible code | 15 | 14 |

**Example 4.**

A = A||C;.

Both ECS and PLIOPT recognize that the move of A is unnecessary. In the ECS system, this is accomplished because the expansion of A = A||C includes the IL statements

```
ASSIGN T1 FROM A (MOVE1 BYTES)
. . .
ASSIGN A FROM T1 (MOVE1 BYTES)
```

Variable propagation eliminates the latter statement; then the dead code eliminator recognizes that since T1 is not used, the former statement can also be eliminated.

**Example 5.**

```
P: PROCEDURE (A, B, C);
   DCL (A, B, C) CHAR(*);
   A = B||C;
```

Both the ECS and PLIOPT compilers put B and C into temporaries to prevent their possible overlap with A from producing erroneous results. The code produced by the ECS compiler would have about 115 machine instructions. PLIOPT code is shorter (43 instructions plus 33 more in IELCGMV) since it has three calls to IELCGMV. It should be mentioned that if the 370's MVCL instruction were used, the ECS produced code sequence could be reduced by 40 to 50 statements since the ASSIGN procedure would be simplified. The code from PLIOPT would remain the

same, however, except that the MVCL instruction would replace the call of IELCGMV.

In terms of running time, the code produced by the ECS compiler is superior. PLIOPT allocates one temporary T of length A.LEN and performs T = B||C, then A = T. Thus, whenever A.LEN > B.LEN + C.LEN, it pads the end of T with blanks and then moves the blanks to A. As a result, if A is a long string while B and C are short, it will move nearly twice the number of characters needed. The ECS produced code allocates two separate temporaries to hold B and C. Thus the allocation costs a little extra time, but only the exact required amount of storage is allocated, and the temporaries are not padded.

In this example, the ECS compiler has a large advantage over other compilers. Interprocedural data flow analysis [1, 8] might determine that in all calls of P, A could not be an alias for B or C. If so, the code for P could be substantially simplified. Also, if a particular call of P were to be integrated, then the code would reflect any favorable special cases which might be present in that call, such as a parameter's being shorter than 256 bytes.

**Example 6.**

A = B||C||D;.

This is a case where the ECS compiler does not do very well. Assuming the parser translates this into code which creates a temporary T and then performs T = B||C, A = T||D, the optimizations described so far are not sufficient to remove the use of the temporary T. Things are even worse if more arguments are to be concatenated, since each concatenation creates a new temporary into which the entire partial string must be moved. PLIOPT, however, does this example without extra moves.

To overcome this difficulty in the ECS compiler, an additional optimization could be introduced which changes the sequence T = B||C, A = T||D to A = B||C, A = A||D. As in example 4, the second statement requires only a movement of D. This optimization is investigated in [3].

## Conclusions

The experimental compiler methodology described here is characterized by having a built-in expansion of each high level statement into a lower level code sequence and also having a set of optimizations which can tailor the code to take advantage of any special cases which may occur. This methodology has several advantages described in more detail in [5]. They include ease of compiler construction and maintenance, complete compatibility between interpretive and compiling versions, and support for run-time error checking.

We believe that the examples of this paper provide evidence that the code produced by such a compiler will be quite good. In comparisons with a PL/I optimizing compiler, hand simulation of the ECS compiler with only very basic optimizations produced code for A = B||C which was typically a little longer but somewhat faster than the PLIOPT produced code. With additional optimizations, both the speed and length of the ECS code was reduced. Also, interprocedural data flow analysis enables the ECS compiler to substantially improve its object code in places where traditional compilers have to make the most pessimistic assumptions about the aliasing and attributes of the operands.

PLIOPT benefitted from using a nonstandard linkage to the compiler-supplied subroutine IELCGMV. This enabled it to reduce the length of its object code without the usual time cost involved in the calling sequence. Such a scheme should probably be employed in the ECS compiler as well.

To obtain good code for concatenation, the optimizations of procedure integration, constant propagation, dead code elimination, and variable propagation were required. Range analysis also considerably improved the code in some cases, and there may be a need for some other optimization to improve the code resulting from A = B||C||D. For source statements other than concatenation, additional optimizations such as redundant expression elimination, code motion, and strength reduction (see [2]) may be useful.

*Acknowledgments.* I would like to thank Fran Allen, Bill Harrison, and David Lomet for many patient, helpful discussions. I also wish to thank Hania Gajewska and the referees for suggesting improvements to the paper.

Received December 1975; revised September 1976.

**References**

1. Allen, F.E. Interprocedural data flow analysis. Information Processing 74, North Holland Pub. Co., Amsterdam, pp. 398–402.
2. Allen, F.E., and Cocke, J. A. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 1–30.
3. Carter, J.L. Some useful renaming transformations for optimizing compilers. To appear.
4. Elson, M., and Rake, S.T. Code-generation technique for large-language compilers. *IBM Syst. J. 9*, 3 (1970), 166–188.
5. Harrison, W. A new strategy for code generation—the general purpose optimizing compiler. Conf. Rec. *Fourth ACM Symp. on Principles of Programming Languages*. Los Angeles, Jan. 1977, pp. 29–37.
6. Harrison, W. Compiler analysis of the value ranges for variables. *Res. Rep.* RC 5544, IBM T. J. Watson Res. Ctr., Yorktown Heights, N.Y., 1975.
7. Loveman, D.B. Program improvement by source to source transformation. Conf. Rec. *Third ACM Symp. on Principles of Programming Languages*, Atlanta, Jan. 1976, pp. 140–152.
8. Rosen, B.K. Data flow analysis for recursive PL/I programs. *IBM Res. Rep.* RC 5211, T.J. Watson Res. Ctr., Yorktown Heights, N.Y., 1975.