F. E. Allen
J. L. Carter
J. Fabri
J. Ferrante
W. H. Harrison
P. G. Loewner
L. H. Trevillyan

The Experimental Compiling System

The Experimental Compiling System (ECS) described here represents a new compiler construction methodology that uses a compiler base which can be augmented to create a compiler for any one of a wide class of source languages. The resulting compiler permits the user to select code quality ranging from highly optimized to interpretive. The investigation is concentrating on easy expression and efficient implementation of language semantics; syntax analysis is ignored.

1. Introduction

The Experimental Compiling System (ECS) uses a new compiler construction methodology [1] the fundamental goal of which is to provide a system on which customized compilers for a variety of source languages and a variety of target machines can be developed. The compilers are intended to be easy to build, modify, and maintain and to produce optimized object code if desired. In our investigation we assume the existence of a general parsing system, which is therefore not considered. Instead, we concentrate on the design and development of a system which permits easy expression of language semantics in a form amenable to analysis and optimization.

The meanings of most of the constructs in a language are given in a collection of procedures, which are essentially identical to user procedures and can thus be subjected to the same analyses and optimizations. In this way specific characteristics of the source language can be deduced by the system. Modifying or extending the language involves changing or augmenting the collection of procedures defining it.

The basic system is designed to minimize the constraints imposed on languages. This increases the range of possible constructs which can be supported when the system is customized to compile a given language. The primary interface between the basic system, the procedures defining a language, and the programs written in it is the internal language in which the procedures and their characteristics are expressed. The basic system provides a schema for this language and understands its semantics. The meaning of a given language is built on this schema.

These ideas (the internal language schema, semantic definition by procedures, and procedure characteristics derived by analysis, as well as the mechanism for code expansion and selection) are fundamental to the Experimental Compiling System approach and significant departures from conventional approaches to compiler design.

The internal language (IL) schema is a framework for expressing various languages. One of two unique aspects of the schema is that attributes, including information normally provided by data declarations, are variables. ECS has no *a priori* knowledge about attributes, their possible values, or when such values are bound to attributes. Furthermore, the usual dictionary in which such source-specific information is directly encoded for use during the compilation process does not exist in ECS.

The other unique feature of the schema is that all operations are references to procedures which implicitly define

and elaborate the meaning of the operation. The procedure reference mechanism is used to express declarative information as well as the executable statements in a given source language. Unlike conventional programming languages in which a syntax and semantics are specified, the IL schema provides a form but only limited semantics; it does not attach meaning to most of the operations.

The collection of procedures (called *defining procedures*) which elaborate the operations of a language must eventually reference primitive operations. These operations are defined by *degenerate procedures*—procedures with no elaborating text—and are abstractions of the target machine.

The term *dialect* is used to refer to a language—including its semantics—expressed in the form provided by the IL schema. The notation IL/X is used to refer to the dialect of IL supporting the language X. The terms IL/S and IL/P are used to refer to the particular source and primitive dialects employed in an ECS compiler. Because, in our experimental work, we use PL/I as the source language and the IBM System/370 as the target machine, the output of the PL/I translator is called the IL/PLI dialect, and the primitive language is called the IL/370 dialect.

The defining procedures for nonprimitive operators are written in an external syntax of IL called DP. Although it would be most convenient to use the same IL language for elaborating the operators of a source language as is used for the source language itself, this does not work in general. PL/I does not, for example, permit the testing of parameter types, which is needed to select code alternatives.

When processing a program, it is necessary to get answers to such questions as: Does this instruction branch? What variables are used by this instruction? What variables are defined? Normally such information about the operators of a language is built into a compiler. Furthermore, compilers generally make worst-case assumptions about an operation which refers to an external procedure. In order to answer such questions and mitigate worst-case assumptions, ECS performs both intra- and inter-procedural analyses. Given a collection of procedures, certain control and data flow characteristics of each procedure are found by an in-depth analysis of the procedure in the context of the collection. Externally interesting information, such as how global variables or parameters are used, is summarized and retained with the procedure in a library. Since all operators are defined by procedures, most of the interesting operator characteristics are mechanically derived by the system. Certain characteristics must be given, however. Summary information for the degenerate procedures which define the primitive operators cannot be derived automatically, since the body of such a procedure does not exist. Such properties as operand commutativity are also not deducible by our analysis. The system provides a mechanism whereby the summary information can be supplied or augmented by the definer of the procedure.

Code expansion is accomplished by procedure integration. This contrasts with code generation in most compilers, which basically involves scanning an input text string and, depending on what is found there and in a dictionary, selecting and emitting code sequences to an output text string. In ECS code emission occurs before code selection and is usually accomplished by replacing a procedure reference with the procedure itself. It is similar to macro expansion: the actual arguments replace occurrences of parameters in the text, local names are distinguished, and external names are resolved. This transformation, therefore, can be used to replace the code emission function of the usual code generator. If desired, integration can be done selectively; for example, it can be made dependent on the projected frequency of execution of an operation instance. When a procedure reference is not replaced with the procedure itself, it becomes a CALL to the executable version of the procedure. A single definition of an operation thus suffices for use in systems having quite disparate optimization goals.

When global program analyses and optimizations are applied to the text after procedure integration, a general procedure frequently becomes tailored to the particular reference. ECS exploits this idea to effect the code selection usually accomplished by emitting code sequences after interrogating contextual and dictionary information.

An analyze-optimize-integrate cycle reduces an IL/S program to an IL/P program consisting of primitive operations. These operations reflect the functions of the target machine but not its resource constraints. Thus storage is not mapped and registers are not allocated. The system provides a table-driven mechanism to perform machine-tailoring functions.

Given the basic ECS system, a particularization to support a given target machine P can be constructed by first defining IL/P. Then the register requirements and alternative code skeletons for the IL/P operations are defined and a storage mapper written.

A compiler for a particular source language S can be constructed from this particularization by defining IL/S. A translator is then written which translates S programs into IL/S. The defining procedures for all the nonprimitive IL/S

operations are written next. (These are written in DP, the defining procedure language.) The defining procedures are compiled and the compiling process augments the IL library of procedures with these new defining procedures. (The procedures in the IL library contain summary information and are optimized.) The compilation also generates optimized object modules for these procedures and augments the object library.

The resulting compiling system can compile an S program in any of several modes. A complete optimization can be requested in which all or most procedure references are expanded in line, and the analyze-optimize-integrate cycle is performed as often as necessary to achieve full optimization. Less optimization and faster compilation is achieved by decreasing the number of iterations through the cycle. If no iterations are performed, the result is a program in which every instruction becomes a reference to the compiled form of the defining procedure.

In order to realistically evaluate the approach we are, as already indicated, using PL/I as the source language and the 370 as the target machine. The run-time environment of the IBM PL/I Optimizing Compiler [2] and its run-time library are being used. This not only obviates the need to develop a new environment and library but allows more accurate comparative evaluations to be made regarding the relative efficiency of the code produced by the PL/I Optimizer and by ECS.

In the next section of this paper, we discuss the IL schema on which various dialects can be constructed to express source and target language constructs and their semantic interpretation. Section 3 describes the organization of the ECS compiler which is being constructed to evaluate the methodology as it applies to PL/I. Section 4 gives an example of a (hand-simulated) application of the approach to string concatenation. Although this application has been reported elsewhere [3], it is repeated here to provide a specific basis for evaluating the approach. The last section includes some observations regarding the relevance of the ECS approach in extensible language systems, program development systems, program maintenance, and data isolation. An Appendix elaborates some of the technical mechanisms developed to support the ECS approach.

2. IL schema and its dialects

In this section the objectives and the constructs and concepts of the intermediate language schema are given. Following that the two dialects of IL (IL/PLI and IL/370) and the IL external form (DP) being used in the current ECS implementation are discussed.

- *Objectives*

The IL schema [4] is designed to support a class of languages which includes PL/I, FORTRAN, COBOL, ALGOL-60 and 68, as well as low-level languages close to the assembly language level. While the IL schema is capable of supporting APL, the rest of the system would require additional analysis and transformation components to effectively compile that language.

The number of built-in IL constructs is small. Since the schema (and ECS) is independent of any particular source language, a minimal schema both avoids precluding constructs in source languages and avoids including constructs in the base system which are not required for a given language.

Since the notion of procedures, their definition, invocation, and integration is central to ECS, the IL schema necessarily supports a wide class of definitional and invocational mechanisms, including all the usual call-by-reference, call-by-value, and call-by-name argument-parameter association forms. Furthermore the schema lets the ECS procedure integrator be a mechanical, language-independent transformer which can preserve the semantics of an invocation.

In addition to these objectives, which are central in determining the form of the IL schema, several practical considerations are factored in. The most important one is the need to collect and retain storage mapping and aliasing information. The actual representation of IL within the system is also very much dictated by practical, primarily efficiency, considerations.

- *Constructs and concepts*

The objectives of the IL schema are supported through a number of constructs and assumptions regarding the expression of a language, S, in IL. These represent our conclusions as to what constitutes a practical, "lowest common denominator" schema on which a class of languages can be expressed.

Variables

Most source languages explicitly or implicitly associate with each variable rules governing attributes, storage mapping, aliases, name scope, and legal usage. The IL schema contains mechanisms for the expression of these rules but does not imbed them in the schema.

Attributes As mentioned earlier, attributes are treated as variables, and no assumptions are made regarding kinds of attributes, their values, or when values are bound to attributes. The attributes of a source program are expressed as additional qualifiers to names. Thus the PL/I structure component B in

```
DECLARE 1 A,
        2 B CHAR (2) VARYING,
        2 C FIXED BIN (15);
```

might result in such IL/PLI variables as A.B.FORM, A.B.TYPE, and A.B.VARY. These might take on such values as

```
MOVE (A.B.FORM = 'STRING')
MOVE (A.B.TYPE = 'CHAR')
MOVE (A.B.VARY = 'TRUE')
```

(Here and throughout most of the paper we use the external form of IL: the operation, *i.e.*, the procedure referenced, followed by the operands, separated by such delimiters as = + * ( ). All such delimiters are equivalent to a comma or a blank and have no semantic implication.) Since these are truly variables—not reserved symbols—the IL schema contains no restrictions as to when they can be tested, changed, or initialized. A representational expedient has been introduced internally, however. When translating a program in S into IL/S, invariant assignments such as MOVE (A.B.FORM = 'STRING') may be expressed in a "constants dictionary" rather than directly in text. This is done to save processing time. The dictionary does not, however, have the usual form in which specific bits and fields hold specific attribute values, but is used in this context to associate variables with their program-invariant constant values.

Storage mapping information The size, alignment, and storage class (*e.g.*, static, controlled, etc.) are used in the machine tailoring component of ECS. Most of this information is not used by the IL but is "passed through" and is not normally referenced until storage is mapped. It is expressed in a language-dependent table.

Aliasing information The process of deducing relationships in a program and transforming the program based on such information requires complete knowledge of the aliasing relationships in the program to avoid making very pessimistic and limiting assumptions. There are various types of aliases ranging from the static sharing of storage, exemplified by the FORTRAN EQUIVALENCE statement, to the dynamic sharing, which can occur by using PL/I POINTER variables. Some of the aliasing information is best gleaned or at least refined by analyzing the program; other forms are explicit in the source program and must be expressed in the IL dialect of the source language. A table describes the static storage relationships which may exist between variables. Another table is used to hold the more dynamic aliasing possibilities by expressing the potential values of language-dependent variables such as pointers, entry variables, and label variables.

Name scoping In order to support block structured languages and to do procedure integration, it is expedient to incorporate a "weak" form of name scoping in the IL schema. This form assumes that all names are resolved so that identical names in different blocks are identical in IL if and only if they are the same object; otherwise, the names are different. Consider the example given in Fig. 1.

If the procedure integrator replaces the reference to INNER by the body of the procedure, it must adjust the names within INNER. Since A in this case belongs to OUTER, it should not be changed; however, if A were declared in INNER, it would have to be given a new name to prevent conflict with other copies of INNER.

Operands
Each IL operand is a single variable or constant and, with the exception of their use in built-in operations, each is actually an argument in a procedure reference. The variables can be qualified (*e.g.*, A.TYPE) or indicate a location (*e.g.*, addr A). If an operand is an address of a variable, then the operand contains a level of indirection to the variable. Constants can be labels, entries, the value constants of the source language, or symbolic constants. *Symbolic constants* are items which do not change but whose actual constant representation is irrelevant. 'FALSE,' 'FLOAT,' 'SCALAR' are examples.

Instructions
All instructions have a uniform structure: the name of the procedure to be invoked followed by the list of arguments. (The external syntax used for printing or programming purposes may be more elaborate, of course.)

BIND built-in operation
The IL schema has four built-in operations. The meaning of these operations is known to the system; they are not specified as defining procedures. BIND is one of them and has the form

```
BIND (X, P)
```

which is read "bind (associate) the address of X to be the value of P." In other words the variable X now has as its address the value of P.

Suppose A were a string of characters declared in PL/I by

```
DECLARE A CHAR (50);
```

Now suppose the value of variable C, a single character, were assigned to the tenth position in A. This can be done by first calculating the address of the character to be changed, then setting the value at that address to C. How long is the item at the tenth position in A? The reader knows it is a single character, but the compiling system must be told that fact. To do this a BIND is used to explicitly name the tenth position in A. The total calculation is

```
ADD (P = addrA + 9)
BIND (X, P)
MOVE (X = C)
```

The attributes associated with X (*e.g.*, X.LENGTH = 1) are ascribed to the storage at the tenth byte of A. (Later in this section the use of BIND in supporting various PL/I constructs is shown.)

BUY built-in operation

BUY is used to obtain storage for variable-sized temporaries. It is built into the schema so that the allocation can be easily removed from the execution string and aggregated with other variables if the size of the temporary becomes known during compilation.

LABEL and ENTRY built-in operations

These two built-in operations are simply syntactic markers in the text. They are needed for control flow analysis. Their operands are the label or entry symbols associated with that text point.

Parameter passing

The IL schema must support procedure integration and a variety of argument-parameter association conventions. When a procedure is integrated, one of the changes made to the integrated procedure involves substituting actual arguments for parameters. Thus the IL schema has a call-by-name convention. However, parameter passing is restricted so that there is no difference between call-by-name and call-by-reference. A source language translator must generate the IL appropriate to the language convention. Consider the source program fragments given in Fig. 2.

If the source language uses a call-by-reference convention (as does PL/I), then the desired result of the CALL is I = 5 and A(2) = 10. Figure 3 shows an IL expression of the source language which supports this convention. The INDEX defining procedure puts the location of A(I) in the locator variable P.

Figure 4 shows the result of integrating the two procedures. If the source language has a call-by-name parameter passing mechanism (*à la* ALGOL-60), then the source translator will create procedures to compute dynamically the location of arguments when referenced in the called procedure. The names of these procedures are passed instead of the actual arguments.

- *IL dialects*

The IL dialects used in the current ECS development effort include the DP language, IL/PLI, and IL/370.

DP language

The defining procedures for a given source language can be written in any convenient language for which a translator to IL exists. However, most users would find the constraints of such languages as PL/I too restrictive. For example, PL/I does not provide direct mechanisms for setting and interrogating the attributes of variables. Therefore, it is necessary to allow defining procedures to be written in IL. For this purpose, an external representation of IL programs, called DP, has been developed. Two important guidelines were applied in its design. In order to keep the underlying form accessible and transparent to the writer of a defining procedure, there should generally be a direct, one-for-one correspondence between external

```
PL/I
OUTER :PROC;
        DCL A · · ·
        · · ·
        CALL INNER;
        · · ·
      INNER:PROC;
              · · ·
              = A· · ·
              · · ·
        END INNER;
END OUTER;
```

Figure 1 PL/I procedure showing name scoping.

```
I = 2;                          R: PROC (X,Y);
CALL R (I, A (I));                 X = 5;
                                   Y = 10;
                                END;
```

Figure 2 PL/I call with related arguments.

```
                                R:PROC (X,Y)
MOVE (I = 2)                       MOVE (X = 5)
INDEX (P, A, I)                    MOVE (Y = 10)
BIND (T, P)                     END
R (I,T)
```

Figure 3 IL/PLI form of the procedures in Fig. 2.

```
MOVE (I = 2)
INDEX (P, A, I)
BIND (T, P)
MOVE (I = 5)
MOVE (T = 10)
```

Figure 4 Result of integrating the procedures in Fig. 3.

and internal text. (A few statements, notably the control statements, are exceptions.) The number of language constructs actually needed in writing a defining procedure is quite small. Only those considered necessary to elaborate a definition or highly desirable for expressibility are included.

IL/PLI

The decision to use PL/I as the language for testing the feasibility of the ECS approach was made for several reasons, the most important being the richness of the PL/I language. By establishing the technology required to handle such constructs as pointer variables, ON conditions, etc., ECS will be able to support similar facilities occurring in many other languages.

In choosing PL/I we were able to take advantage of an available translator: the "front end" of the IBM PL/I Checkout Compiler [5]. This translates PL/I into an internal form called HTEXT, which is actually a text and dictionary suitable for interpretation. It is this form that is transformed into IL/PLI.

In the discussion of the IL schema several PL/I-related examples were used to illustrate schema constructs. The PL/I features now discussed are some examples of the strategy used to express interesting PL/I constructs in IL.

ON condition enablement The PL/I ON conditions which are enabled at any time are established by their lexical scopes. A defining procedure for an operation may need to find out what conditions are enabled. This information is passed to the procedure as an explicit argument which has been established by the translator.

ON units The PL/I program units used to define the actions to be taken when an enabled condition occurs in the executing program are treated as procedures. The ON condition name itself is treated as a local entry variable.

Procedures In multiple-entry procedures the relation between parameters and their order or existence in a parameter list can vary between entries:

```
A: PROC (X, Y);
   X=1;
B: ENTRY (Y, Z);
   . . .
   Y=2;
   . . .
   END;
```

To avoid making parameter operands entry-dependent, an alternate solution to the PL/I method of renaming parameters at different entries was chosen. When a multiple-entry procedure is encountered whose entry lists specify different parameters, the procedure is modified so that each parameter is assigned a fixed position in a canonical list to be used at all entry points. All entry points are altered to accept this canonical list. A series of dummy procedures is then created at the same lexical level as the procedure being modified. Each of these dummy procedures reorders the arguments to the canonical form and invokes the corresponding entry point in the modified procedure. Thus the above PL/I procedure becomes the equivalent of

```
A: PROC (XX, YY);
   CALL AA (XX, YY, 0);
   END;
B: PROC (YY, ZZ);
   CALL BB (0, YY, ZZ);
   END:
AA: PROC (X, Y, Z);
   X=1;
BB: ENTRY (X, Y, Z);
   . . .
   Y=2;
   . . .
   END;
```

The procedure integration optimization can generally remove the introduced CALL.

The procedure statement In addition to the parameters expressed in the PL/I procedure statement, the IL/PLI form indicates the PL/I procedure statement options (*e.g.*, recursion) and contains the number of parameters, the ON-condition enablement parameter, the return variable, and the label of the initialization block. Initialization includes space acquisition, variable initialization, and the usual procedure prelude. It is separated as a procedure and referenced from each entry point of the original procedure. Again procedure integration will embed it in-line if there is only one reference or when otherwise feasible.

Computed references Addressing of the components of structures, arrays, and based variables is handled using the BIND built-in operation. A reference to P → A becomes BIND (A,P) followed by a reference to A.

An example of addressing is given in Fig. 5. (Note that in PL/I on the 370 the current length of a varying-length character string is stored in the two bytes preceding the characters.)

IL/370

The primitive dialect of IL in the current ECS is IL/370. While providing access to the 370 constructs, it differs from the machine instructions in several ways:

1. Registers are not visible.
2. Load and store instructions are not included in the repertory of IL/370 instructions.

3. Operands are IL variables and constants; they are not in base-index-displacement form. Since storage has not been mapped, addresses relative to, for example, the beginning of the dynamic storage area are not known.
4. The instructions generally have three addresses. The target operand need not be one of the source operands. However, all instructions which will not use registers in their realization (*e.g.*, the decimal instructions) have two addresses.
5. Operand lengths are those of the 370 and are encoded in the operation code if the instruction can be realized by the use of registers.
6. The raising of exceptions is modeled as a call on an external variable.
7. Condition codes and program masks are modeled as external variables.

An example of the description of an IL/370 instruction follows. (The summary information to be associated with the degenerate defining procedure is contained in the ECS library.)

FIXED_ADD_ijk(X, Y, Z)

This performs the fixed point addition X = Y + Z, where the lengths of X, Y, and Z are given by i, j, and k, respectively. Here i, j, and k may each be either 2 or 4.

Implementation: Load, Add, Store, with "Halfword" on any of the instructions if appropriate. If either X or Y is in a register, then the Load can be omitted.

Summary information: X and the condition code are defined; Y and Z and the program mask are used and preserved. A Fixed-Point Overflow exception will be raised if the appropriate bit in the program mask bit is on and a 4-byte overflow occurs.

Y and Z commute if you also switch the operand lengths, *i.e.*, FIXED_ADD_ijk (X, Y, Z) is the same as FIXED_ADD_ikj (X, Z, Y).

3. ECS compiler organization

In this section we describe the structure of the Experimental Compiling System currently being developed. Figure 6 depicts its structure.

- *Translators*

Two different translators exist: one for PL/I and one for the defining procedure language, DP. The translator for the DP language uses a general LALR(1) parser which provides a convenient tool for translating other languages to IL. The translator for PL/I is the translator used by the PL/I Checkout Compiler [5] to produce HTEXT (the internal form of PL/I which is interpreted by the back end of that system) followed by a transformer to change HTEXT into IL/PLI.

PL/I	IL/PLI
DCL Y CHAR (5) VAR BASED;	MOVE (Y.FORM = 'STRING') MOVE (Y.TYPE = 'CHAR VAR') MOVE (Y.MAXLENGTH = 5)
P = ADDR(X);	MOVE (P = addr X)
P→ Y = 'ZZ';	BIND (Y.LENGTH, P) ADD (P1 = P+2) BIND (T, P1) MOVE (Y.LENGTH = 2) MOVE (T = 'ZZ')

Figure 5 Example of the use of BIND when translating a store into a PL/I varying-length character string.

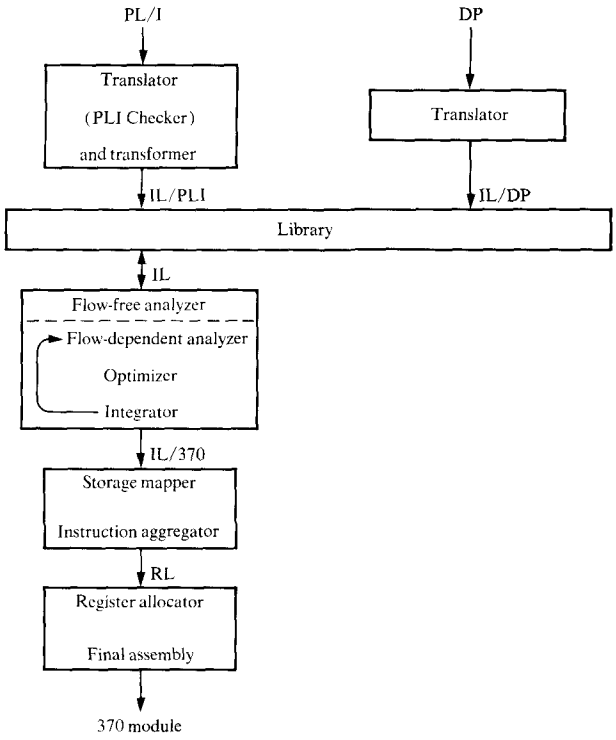

Figure 6 ECS compiler organization for PL/I on the 370.

One of the functions of a source language translator is to determine *packets*: all data objects which have a lan-

guage-dictated storage relationship to each other are mapped into the same packet; otherwise data objects are in unique packets. (The relationship of objects in the same storage class but in separate packets is not resolved until later in the compilation.) Associated with each variable is its packet number and enough mapping information to resolve to the bit level the mapping of a variable and its components. With this level of information partial overlay can be distinguished—a fact which is of interest to those components of the system doing alias analysis. Of particular importance in PL/I is the ability to note substructure independence. For example, changes to any of the 10 B's do not affect the C's in the structure:

```
DECLARE 1 A (10),
        2 B FIXED BIN (15),
        2 C FIXED BIN (15);
```

- *Library*

The library is a repository of analyzed and unanalyzed IL procedures, including both user and system procedures. In the current implementation the library contains IL/370 degenerate procedures, IL procedures which establish the meaning of the IL/PLI operations, and PL/I user procedures translated to IL/PLI. The procedures may have just been produced by the translator or they may have been retained from earlier compilations. The fact that procedures can be created during the translation of a program and can be subsequently analyzed, optimized, and integrated makes the handling of a number of source language constructs relatively easy. For example, a procedure can be created by the translator when array or structure initialization is requested.

The existence of this library has interesting implications for the flow of information through the compiler and on the relationship of the compiler to its environment.

1. All procedures associated with a problem solution and submitted for compilation at the same time are translated and placed in the library before the rest of the compilation proceeds. (Most compilers independently compile each external procedure. Furthermore, the intermediate form is usually very transient.)
2. After a procedure has been analyzed and summary information (see Appendix) has been accumulated, the augmented procedure replaces the original procedure in the library. This may happen more than once as additional knowledge is acquired about the entire collection of procedures associated with a problem solution.
3. During the analysis and optimization of a procedure, summary information for a referenced procedure may be used if it is available. With this additional information optimizations across CALLs and of CALLs can be done. For example, expressions involving global variables might be removed from loops containing CALLs if it is known that the CALLs cannot change the values of these variables. A CALL itself might be removed under the right circumstances.

- *Flow-free analysis*

In order to establish an analysis order [6] on the collection of procedures, a CALL graph—possibly disconnected—must be built. Also, since various language constructs, like CASE statements and subscripting operations, are not built into the compiler but are realized by the defining procedures, the compiler must be prepared to produce good code for procedures that manipulate labels and addresses as variables. For these and other reasons flow-free analysis [7] must be performed before flow-dependent analysis. Flow-free analysis determines the possible values of variables used to reference procedures, designate procedures and branch targets, and contain addresses. Instruction execution order is not considered. The values obtained by the analysis are used by the flow-dependent analyzer to obtain more precise control and data-dependent relationships. The Appendix contains additional material on flow-free analysis.

The call graph built by the flow-free analyzer is used to determine the order in which the subsequent analyses and optimizations will be applied to the collection of procedures. Basically it is inverse invocation order: a procedure is analyzed and optimized after all procedure references have been analyzed. This is not, however, the loop depicted in Fig. 6. Before describing the components of that loop—the flow-dependent analyzer, the optimizer, and the procedure integrator—we discuss the purpose of the loop.

ECS is designed to permit multiple applications of program analysis and transformation. This is possible because the programs which perform these functions are insensitive to the text levels so can be applied to multiple levels, and because the IL schema, which is the only language the basic system knows about, has no built-in assumptions about text levels or binding times for information.

The primary reason for the loop is the way operations are defined. The operation-defining procedures elaborate the high-level operations in terms of other operations. By successive elaborations, every IL/S instruction is reduced to a sequence of IL/P instructions. Thus an IL/S program is processed by the compiler until all instructions are IL/P instructions. This can occur in three ways:

1. IL/S can be IL/P. If S is very close to the target language, the parser for S may not generate any higher-level in-

structions. Most IL/S's will probably contain some IL/P instructions.
2. As a result of integrating procedures and thereby elaborating the high-level IL/S instructions to IL/P.
3. As a result of transforming IL/S instructions into a sequence of IL/P statements calling the IL/S defining procedure or user-supplied procedure.

The choice of when to replace a high-level instruction with a procedure and when to replace it with a calling sequence depends on many factors: the goals of the system and/or of this particular run, and the space/time tradeoffs of making a particular replacement.

Since the defining procedures are compilable, they are available for use at run time. By transforming references to these procedures into run time calls, a program is created which is executed in the object environment defined by these procedures. In this way we can get an interpreter. Furthermore, if during the optimization process all of the input operands of an operation become known, the operation can be performed at compile time using a compiled version of a defining procedure.

We now consider analyses and transformations applied to a procedure during a single iteration of the loop.

- *Flow-dependent analysis*

Using the aliasing and summary information provided by the flow-free analyzer and the packet mapper, control and data flow analyses are performed on a procedure. Control flow analysis builds the control flow graph of the procedure and performs a variant of the interval analysis described in [8]. The purpose of interval analysis is to codify the control flow relationships (*e.g.*, loops and loop nests), so that other analyses and transformations can be done more rapidly. The interval analysis variant is based on [9].

Data flow analysis finds all "def-use" relationships: all definitions which may affect a given use (and all uses which may be affected by a given definition) are found by the bit-vectoring methods described in [8]. In order to limit the sizes of the bit vectors and to retain the results of data flow analysis when procedures are integrated, the analysis is performed and retained within "data-flow domains." This program partition and its uses are described in the Appendix.

- *Optimization*

The collection of optimizing transformations is quite open-ended and subject to change. The initial collection includes some "classical" transformations as well as some new ones.

```
DCL A(100) INT;
DO I = 1 TO 10;                    I = 1
                                   GO TO TEST
                                LOOP:
    .                              . . .
    .                              CK[1 ≤ I ≤ 100]
    .                                = A(I)
    A(I)                           I = I + 1
END;                            TEST:
                                   IF I ≤ 10 GOTO LOOP
```

Figure 7 Subscript range check generated during translation.

Redundant expression elimination This includes both code motion and common subexpression elimination.

Constant propagation Instructions are executed at compile time if the operands which are used are constant. The ECS methodology allows the system to provide directly executable constant propagators for all operations—whether user- or system-defined. The Appendix describes this optimization in more detail.

Dead code elimination Unreachable code is eliminated. Performing this transformation after propagating constants through procedures which have been integrated has the effect of tailoring the general procedure to its specific instance of use. This transformation also eliminates useless instructions and instructions of the form A = A.

Strength reduction This is primarily aimed at changing subscript calculations to increment instructions [10].

Range analysis In [11] a method is given for determining the bounds on the ranges of values assumed by certain variables at various points in the program. Such *range information* is used to eliminate redundant tests and to expose dead code. A particularly interesting application for this analysis is in reducing the costs of checking for subscripts that are out of range. Figure 7 shows a fragment of a PL/I program on the left and on the right a schematized internal form in which a check on the range of subscript I has been expressed.

The range analyzer acquires range information from definition and test points and propagates it to use points. When applied to this example, it will find that the value of I at the point of the check is $1 \leq I \leq 10$. The check statement is unnecessary and is eliminated. By explicitly expressing such checks as instructions in the text string, they are also subject to other forms of optimization: they will frequently be redundant and can be eliminated or moved out of loops.

Variable propagation The variable propagation transformation changes an occurrence of a variable name in a program to a different name which has the same value:

X = Y use of (X)	becomes	X = Y use of (Y)

This may allow the elimination of the trivial assignment X = Y as dead code. Its most important applications in the ECS context are in removing levels of indirect addressing, particularly after procedure integration.

Renaming Renaming is a transformation in which one variable is replaced by another. The motivation is to reuse variables in order to reduce the number of temporaries required and the number of moves. There are two forms as shown in Fig. 8. In Section 4 an example is given using this transformation.

- *Procedure integration*

References to procedures are replaced by the procedures or by their calling sequences when procedure integration is performed. The Appendix discusses this transformation in greater detail.

4. Machine tailoring

Not all functions a compiler must perform fit naturally into a procedurally based specification and elaboration. Storage mapping in particular does not entirely fit into this approach.

The fundamental function of storage mapping is to change the underlying model of storage used by the program. All of the variables required by a procedure are examined and relative locations assigned to each. Since ECS does not distinguish temporary, compiler-generated variables from other variables and since it generates a new such variable whenever one is required, ECS overlays storage [12-14]. This decreases user storage as well as making temporary management unnecessary.

When storage is mapped, the references to that storage must also be changed. This transformation is accomplished by instruction aggregation which constructs the more complicated 370 base-index or base-index-displacement (BXD) operands from the simpler IL operands and the results of storage mapping.

In ECS, storage mapping and instruction aggregation are part of the target machine tailoring function. Another major function performed by the machine tailoring function is register allocation. Its input is an augmented form of IL, called RL for Register Language, which is the output of instruction aggregation. The machine tailoring functions of the compiler are now considered.

Figure 8 Effects of the renaming transformation.

Original	becomes	or
T = op (A, X)	B = op (A, X)	A = op (A, X)
. . .	. . .	. . .
B = op′ (T, Y)	B = op′ (B, Y)	B = op′ (A, Y)

- *Storage mapping*

Storage mapping in ECS involves collecting packets belonging to the same storage class into larger packets. This includes overlaying storage—determining which sets of packets in the same storage class can be assigned overlapping storage so that the overall object storage requirement is reduced. It also includes generating the instructions required to allocate and reference a packet and the objects in it.

The first task, integrating and overlaying the primitive packets into larger packets within storage classes, could be done by defining procedures which are referenced at appropriate points in the text string and provided with the necessary information by the usual analysis techniques. For several reasons, however, it is desirable to treat this function in a special way.

1. The target environment as well as the source language influences the organization of the larger packets.
2. The information required for packet construction is not that normally collected by the analysis processes. It might be necessary to make a special analysis to derive such information.
3. The integration of packets should happen after other optimizations and procedure integration. At that time the "dead variables" which need no storage will have been identified, and the coalescing of storage class membership for the integrated procedures will have occurred.
4. Since storage mapping changes the storage model from that of the source language to that of the target machine, the reference forms must also be changed. This involves, for example, transforming references to a variable X in the PL/I automatic storage class to references to an offset (off) to the base of the appropriate dynamic storage area (DSA).

For these reasons our current implementation performs packet integration as part of the machine tailoring component of the system. The storage overlay aspect of packet integration is discussed in the Appendix.

- *Instruction aggregation*

The instruction aggregation component of ECS augments the IL instructions to include the storage mapping information. The additions explicit in the IL prior to aggregation are implicit in an RL operand. Thus, the aggregator

synthesizes the complex machine operands out of the operands of several IL expressions. It is described in greater detail in the Appendix.

RL, the annotated IL instructions produced by instruction aggregation, can be characterized as follows: operations are identical with IL/370; operands are annotated to include the 370 base + index + displacement (BXD) form.

- *Register allocation*

The register allocation component of ECS not only allocates and assigns registers but makes the final code selections. Any optimizing compiler for the IBM 370 (or any computer with multiple ways of performing the same function) is faced with the dilemma caused by the fact that the selection of the instruction sequence depends on register availability and the assignment of registers depends on the instruction sequence.

If there were only one possible sequence for every higher-level operation, then the problem would be somewhat easier, though by no means trivial. The ECS register allocation component tries to select the best sequence of instructions subject to register availability and an estimate of the relative execution frequencies of various areas of the program.

The organization of the ECS register allocator is given in the Appendix.

- *Final assembly*

The last component of ECS generates the actual code and creates the load module.

5. An example

The example given in this section is taken from [3]. The study reported in that paper was designed to evaluate (by a hand simulation) the effectiveness of the ECS approach in producing good code for a hard problem. The problem chosen was the PL/I string concatenation operation: A = B||C.

A code generator has to be aware of numerous possibilities when generating code for this operation:

1. The operands may be varying or fixed-length strings.
2. The result may need to be padded or truncated.
3. One or both of the operands may alias the target variable. For example, if C is aliased with A, then moving B into A will destroy the original C. If B and A start at the same memory address, then we might be able to save a move operation.
4. Different instruction sequences are required for operands of different lengths. These can range from a simple load-store sequence to loops for long strings.

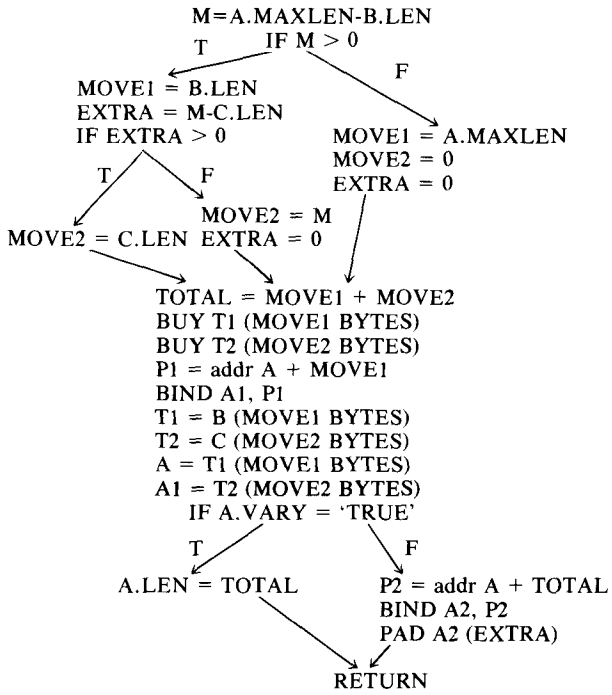

Figure 9 Defining procedure for concatenate.

5. The context of the concatenate operation may greatly affect the kind of code that should be generated. The quintessential example of this is LENGTH (A||B) in which the actual concatenation is unnecessary since the desired result is the sum of the lengths of the two operands.

The conventional strategy for producing good code for such an operation is to build into the code generators an extensive selection process which distinguishes the "special cases."

The ECS strategy is to write the defining procedure in as straightforward a way as possible and use the existing analysis and optimization techniques to produce good code. The next few figures elaborate the application of this strategy to a specific instance of concatenation. Figure 9 shows in schematic form most of the defining procedure for the concatenation A = B||C. Note that the overlay problem is handled by moving each input string into a temporary. (The notation used in this example differs from our usual notation, but we hope it is both clear and concise.)

Now consider the PL/I program in Fig. 10 which references the defining procedure for concatenate. The declare

```
DCL (B,C) CHAR (10);
DCL A CHAR (50);
. . .
A=B||C;
```

Figure 10 A reference to the concatenate denning procedure.

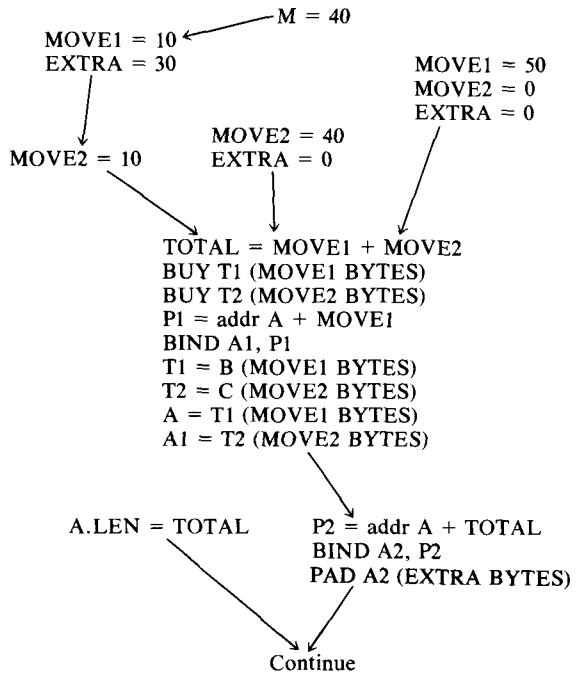

Figure 11 After integrating the concatenate defining procedure.

```
BUY T1 (10 BYTES)
BUY T2 (10 BYTES)
P1 = addr A + 10
BIND A1, P1
T1 = B (10 BYTES)
T2 = C (10 BYTES)
A = T1 (10 BYTES)
A1 = T2 (10 BYTES)
P2 = addr A + 20
BIND A2, P2
PAD A2 (30 BYTES)
```

Figure 12 After applying constant propagation and dead code elimination.

statements are translated into IL statements which assign values to a number of variables, including A.LEN, A.MAXLEN, A.VARY, etc. The PL/I concatenate statement is translated into a reference to the concatenate procedure. After procedure integration has replaced the reference to concatenate by the defining procedure, constants can be propagated. We now get the program shown in Fig. 11. (In this example arguments and parameters have the same names—this is not usually the case, of course.)

Figure 11 shows several instructions which cannot be executed. Dead code elimination removes them. The constants assigned to the remaining MOVE1, MOVE2, and EXTRA can then be propagated to their uses. Having done the constant propagation, the assignments of the constants to these variables are dead and can be eliminated. The program in Fig. 12 is left. The program now has T1 = B and A = T1. Since it can be established that A, B, and T1 are not aliases, variable propagation can transform the latter statement to A = B. This causes T1 = B to become dead, so it can be eliminated. This, in turn, makes the BUY of T1 dead. Similar analyses and transformations affect T2 and C. The result is shown in Fig. 13.

When the operations in Fig. 13 are replaced by their defining procedures, the IL/370 version of the program fragment is obtained. After more optimization, the result is the code of Fig. 14. (The number of bytes expressed in a 370 move instruction is one less than the number of bytes to be moved by the instruction.)

Storage is now mapped. A, B, and C are in automatic storage so are mapped relative to the beginning of the dynamic storage area (*i.e.*, the DSA) at constant offsets: offA, offB, and offC. The result of the storage mapping is shown in Fig. 15. (In the implementation the text is not actually expanded with the instructions for accessing the data, but the accessing information is held in a table associated with the instructions.) Only the address computations needed to address A are shown.

The IL to RL conversion is performed. The addressing computations are collected into the base-index-displacement operands of the 370. In this example, we are assuming that the offsets are < 4096. In conjunction with the formation of BXD operands, other constant components of the address computation are also collected into the displacement if possible. The result is shown in Fig. 16. The register allocator generates the result shown in Fig. 17.

Thus, the original, very general defining procedure for concatenate has been reduced by general transformations to four instructions for this particular case. What about other cases? A number of cases were considered and the results compared with the PL/I Optimizer, which contains a very large, special-case code generator.

1. For A = A||C, A will not be moved by either ECS or the Optimizer.

2. For LENGTH (B||C), the concatenation will not be done by ECS; it is done by the Optimizer.
3. In ECS, A = B||C||D will use the renaming optimization to avoid unnecessary moves to and from temporaries. The Optimizer also avoids unnecessary moves.
4. If concatenation is done on parameters of unknown length, as in

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

then the ECS code will be longer but faster in comparison with the Optimizer.

6. Conclusions

A compiler construction methodology has been described which provides a language-independent compiler framework on which language-specific compilers can be built. The approach is based upon the use of

1. An intermediate language (IL) schema to express languages.
2. Procedures to specify ("elaborate") the semantics of the language.
3. Analysis to derive the characteristics of operations.
4. Procedure integration to expand high-level code into lower-level code.
5. Analysis and optimization to tailor code to its particular context.

As a consequence of the approach, the system features

1. Interprocedural analysis and optimization, including in-line expansion ("integration") of user procedures.
2. Both interpretation and compilation within the same system and from a single semantic definition. The compiled object code can optionally be highly optimized. Interpretive code (in the form of references to the generalized procedures for each operator) and optimized code can be mixed in the same routine.
3. An extensive collection of optimizing transformations.
4. Variable binding times. Most systems expect to bind information at fixed times: attributes to variables at compile time, relative addresses at load time or execution time. The Experimental Compiling System binds information when it is known.

The approach is being validated by implementing the basic system and testing its applicability to PL/I on the 370.

In addition to providing a compiling system which should significantly reduce the cost and complexity of creating a compiler, while increasing the reliability and code quality of the programs compiled by it, the ECS approach has other advantages [15].

```
P1 = addr A + 10
BIND A1, P1
A = B (10 BYTES)
A1 = C (10 BYTES)
P2 = addr A + 20
BIND A2, P2
PAD A2 (30 BYTES)
```

Figure 13 After variable propagation and other optimizations.

```
ADDRESS_ADD   P1 = A + 10
BIND          A1, P1
MOVE          A, B, 9 bytes
MOVE          A1, C, 9 bytes
ADDRESS_ADD   P2 = A + 20
BIND          A2, P2
MOVE          A2, '_', 0 bytes /*insert pad char */
EXTEND        A2, 28 bytes /* pad end of A*/
```

Figure 14 IL/370 version of the program.

```
ADDRESS_ADD   LA = DSA + offA
              /*EST. LOC. OF A*/
BIND          A,LA
· · · /* SIMILAR INSTS FOR B AND C. */
ADDRESS_ADD   P1 = A + 10
BIND          A1, P1
MOVE          A, B, 9 bytes
MOVE          A1, C, 9 bytes
ADDRESS_ADD   P2 = A + 20
BIND          A2, P2
MOVE          A2, '_', 0 bytes
EXTEND        A2, 28 bytes
```

Figure 15 After storage mapping.

```
MOVE   A[DSA+offA],          B[DSA+offB], 9
MOVE   A1[DSA+(offA+10)],    C[DSA+offC], 9
MOVE   A2[DSA+(offA+20)],    '_'[SI+off._.], 0
MOVE   A2[DSA+(offA+21)],    A2[DSA+(offA+20)], 28
```

Figure 16 After instruction aggregation.

```
MVC   offA (9,DSA), offB(DSA)
MVC   offA+10 (9,DSA), offC(DSA)
MVI   offA+20, C '_'
MVC   offA+21 (28,DSA), offA+20(DSA)
```

Figure 17 After register allocation.

Good programming style is supported. The programmer can freely organize a problem into a functionally related, highly structured collection of procedures. The system deduces the data flow through the collection and can open procedures in line. This latter transformation not only eliminates the overhead of a call but, when fol-

lowed by optimization, tailors a general procedure to a specific instance. A particularly interesting use of this is in isolating data representations. A(I,J) can be treated as a reference to a function A which, using arguments I and J, returns a value or a pointer to a value.

Program management functions are supported. ECS can be used to check the consistency of a collection of procedures and, when one is changed, to determine the proliferation of the effects. Ideally an ECS compiler is a component of a larger system which can both use and supply information regarding the status of an entire collection of procedures. A component of this could be a design specification subsystem in which the functions of the components of a system being designed are specified. The components can be checked for consistency and as each component is developed a check made to ensure that the specified interface has been correctly implemented.

Interesting diagnostic and maintenance material is available. As a result of the extensive and intensive analysis of a collection of procedures, a great deal of information about the entire collection is available. Comments can be automatically added to a program listing at procedure call and definition points which summarize the effects of the procedure call or definition. Because of the volume of information made available by the system, an interactive mode of communicating to the user is desirable.

Extensive error checking is supported by the system. The usual overhead of in-line checks on subscript ranges, argument-parameter compatibility, variable types, etc., will largely disappear as a result of compile time analysis and optimization.

Language extensibility is supported via the procedure mechanisms.

Acknowledgments

John Cocke is the originator and promulgator of many of the ideas presented here. Pat Goldberg has provided both intellectual and material support. We thank both these people for making this project possible. Many others have contributed ideas and support. In particular we wish to thank Kenneth Davies, David Lomet, Barry Rosen, Robert Tapscott, Mark Wegman, and Bill Weihl.

Appendix: Specific techniques

Several techniques which are new and/or basic to the ECS approach are discussed in this Appendix: flow-free analysis, summaries, procedure integration, data flow domains, constant propagation, storage overlay, and instruction aggregation.

- *Flow-free analysis*

Data flow analysis in ECS is complicated by the presence of procedure, label, and pointer variables. Procedure variables make it impossible to determine, from a simple scan of the program, which procedures may be called by each call statement. Label variables similarly make it impossible to determine which labels may be the targets of each goto statement. Thus a call graph and a control flow graph cannot be constructed after a simple scan of the program. Further complications occur when aliasing among variables in a program is possible. This can result from mechanisms such as pointers and call-by-reference parameter passing, both of which we must be able to handle. As an example of the problems which aliasing can cause, a call on a procedure variable using call-by-reference could have the effect, depending on the value of the procedure variable at the time of the call, of assigning a procedure value to one of the parameters of the call. This fact must be taken into account in constructing the call graph, for if a procedure A contains a call on procedure variable X, the call graph must contain arcs from the node for procedure A to the nodes for each procedure which X can have as its value. To determine the necessary information, a program analysis which is flow-free (in the sense that the call graph and control flow graph are not yet available) is required.

Given a collection of procedures, the flow-free analyzer

1. Computes range information (*i.e.*, lists of possible values) for procedure variables, thereby generating a call graph,
2. Computes aliasing patterns and range information for pointers while computing (1), since procedure variables can acquire values as a result of aliasing,
3. Computes range information for label variables for use when determining the control flow graph, and
4. Finds argument-parameter relationships.

Flow-free analysis also generates summaries for procedures, which is necessary in the case of recursive calls. This is considered in a subsequent section.

Unfortunately, the problem of determining completely precise information (precise up to symbolic evaluation [16]) is inherently difficult. The algorithm suggested here, though not precise in all cases, is safe and has a running time which is approximately bounded by the product of the number of alias relationships in the program and the number of variables and constants of pointer, procedure, or label type.

The method used extends the work of Barth [16], Banning [17], and Allen [6] to the cases we wish to handle. It is described by Weihl in [7] and is similar to that given in

[18]. In general outline, the method involves manipulating relations over a set of variables and values of interest. The code must be scanned to initialize the relations, and then a closure operation is performed on the relations. We illustrate the method by considering several examples, starting with the simplest case, a single procedure with no aliasing, and gradually considering reference parameters, pointers, and calls on procedure variables.

Let us first consider the case of a single procedure with no aliasing and no procedure calls. We first create a relation MODVAL such that

(X,A) in MODVAL means X is assigned value A.

Suppose our procedure consists of two assignments

```
B = C;
A = B;
```

Then, scanning the code, we put the pairs (B,C), (A,B) in MODVAL. We then create the relation PVAL to be such that

(X,A) in PVAL means X has possible value A.

If we take PVAL = (MODVAL)+ (where + is the non-reflexive transitive closure), then we get (A,C) in PVAL. In this limited case, the above formula is both correct and as precise as possible.

Next, let us consider the case of multiple procedures with procedure calls and no aliasing, where operands are passed to procedures in the collection by value. Suppose the body of one of the procedures contains the following code:

```
Call P(A);
B = C;
A = B;
```

where P is a procedure in the collection with a single formal parameter X. We first define a relation AFFECT to be such that

(X,A) in AFFECT

means X may be aliased to A and to every other variable which may be aliased to A. AFFECT is the set of all formal-actual parameter pairs which result from calls to procedures in the collection, and so in our example (X,A) is in AFFECT. We then take

PVAL = (AFFECT ∪ MODVAL)+.

This accounts for the transmission of values from actual parameters to formal parameters. In our example we then obtain, among other pairs, (X,C) and (X,A) in PVAL.

As a further extension, let us now allow parameters to procedures in the collection to be passed by reference. This means that when a value Y is copied into a variable X, there is an implied copy of Y into each alias of X. Following Barth [16], the ALIAS relation, which indicates possible aliasing relationships among variables, can be computed by

ALIAS = (AFFECT*) ∘ (AFFECT*)T

(where T is the transpose, * is reflexive transitive closure, and ∘ is composition). As an example, suppose our collection consists of two procedures P and Q, as follows:

```
P(X,Y)      Q
X = B;      Call P(A,A);
```

Then we initialize our relations as follows:

(X,B) in MODVAL
(X,A), (Y,A) in AFFECT.

We then obtain (X,A), (Y,A), and (X,Y) in ALIAS. We use the following to compute PVAL:

PVAL = ((ALIAS ∘ MODVAL) ∪ AFFECT)+

We note the information is correct in this case, but not completely precise. See Weihl [7] for further details.

In the case of pointer variables and procedure variables, no closed form formulas for computing PVAL can be obtained. The algorithm used to handle the case of reference parameters is not sufficiently general to handle pointers. The difference with pointers is that the variables which contain addresses can be aliased as well, and so assignments to a pointer variable must be propagated to all of the aliases of the variable. The method used to solve this problem is to incrementally iterate. For each modification to a variable, the aliasing relationships implied by that modification are added, and we iterate to see if this produces any more modifications. See Weihl [7] for details of the algorithm, which is both precise and correct in the case of pointers, given the assumption that no control flow information is available.

We next consider calls on procedure variables. The basic problem is that at the time the call is encountered in scanning the program, the possible values for the variable, and hence the actual procedures which might be called by the statement, are unknown. Therefore, it is not possible to immediately associate the actual parameters of the call with the formal parameters of the procedure being called. To avoid rescanning the program several times, we need a mechanism to keep track of the actual parameters of calls on procedure variables. When a value is determined for a procedure variable, we can then associate the actual parameters of the calls on the variable with the formal parameters of the value. The mechanism used to accomplish this is to create, for each procedure

```
PL/I Source: CALL R(I,A(I), B+C,2);
at IL level:  INDEX (P, A, I)
              BIND (T1, P)
              ADD (T2=B+C)
              MOVE (T3=2)
              R (I,T1, T2, T3)
```

Figure 18 The IL/PLI form of a call.

variable, dummy formal parameters. The algorithm in this case, like the one for pointers, involves incremental iteration.

• *Summaries*

The summary for a procedure delineates the effects of calling the procedure on all nonlocal (to the procedure) variables mentioned in the procedure and all formal parameters to the procedure. The effects to be summarized for a variable include whether it is used or modified in the procedure, whether data accessible through the variable is used or modified, and whether the variable is called and in what manner. The summary also includes what copies (*i.e.*, assignments) between variables take place as a result of executing the procedure and information about the nature of the copy (*e.g.*, whether it is actually the storage accessible through the given variable, and not the variable itself, being assigned).

The use and modify information is necessary anytime we want to examine the effects of an operand, *e.g.*, for data flow analysis. In data flow analysis, summaries are examined per instruction in the program, and bit vectors are formed based on this summary information. The information about copies is needed for flow-free analysis to propagate procedure, label, and pointer values.

The information for summaries is first collected by a flow-free analysis and then by a flow-dependent analysis. In broad outline, the flow-free collection of this information requires initializing relations and then performing a closure operation. The flow-dependent counterpart is computed as a data flow analysis problem.

We present an example to illustrate summaries as well as the differences in flow-free and flow-dependent summary generation. Consider the following procedure P:

```
P: PROC(X);
        IF A > X THEN A = X;
                 ELSE A = D;
END;
```

Collecting information in a flow-free manner, the summary would be

A may be modified
A may be used
X may be used
D may be used
X may be copied into A
D may be copied into A

Collected in a flow-dependent manner, the summary would be

A must be modified
A may be used
X may be used
D may be used
X may be copied into A
D may be copied into A

i.e., the additional information that A must be modified is detected. Reference [19] has a discussion of the differences between "may" and "must."

• *Procedure integration*

In general the term procedure integration can be used to apply to a range of transformations designed to bind calling and called procedures more intimately prior to execution. We restrict our use of the term here to mean in-line opening. By that we mean replacing a reference to a procedure with the procedure itself. There are three central considerations in this: the conditions under which it is reasonable, the order in which to perform possible sequences of such transformations, and the "mechanics" of the actual integration as related to maintaining the correct semantics for the source language.

We consider this last issue first. Here, as in all of the discussion related to the ECS transformations, it is important to remember that integration occurs *after* the program has been translated from its external form. Symbolic names have been replaced by numbers (referring to symbol table entries), and all name qualifications, scoping conventions, implicit definitions, etc., have been resolved. Thus variables local to an internal procedure have already been distinguished from other variables having the same name. The following adjustments must be made when replacing a reference to a procedure with the procedure itself:

1. The argument-parameter associations must be made. Different languages have a wide variety of different possible associations. The ECS procedure integration transformation replaces all occurrences of parameters in the text with the corresponding arguments. It is assumed that the source language translator has replaced the actual arguments given in the source program with references to actual or dummy arguments if this is appropriate. Consider the example for PL/I given

in Fig. 18. Occurrences of the parameters in the text of procedure R are replaced with the arguments I, T1, T2, and T3 if R is integrated. The result is correct according to PL/I semantics.

2. Variables local to the called procedure must be kept distinct from those in the calling procedure; variables which are the same must be given the same identification. Since it is assumed that the translator will have resolved all names in internal procedures, name (*i.e.*, number) adjustments are made only on integrating an external procedure. Static variables must get the same identifications across all copies of a procedure.
3. Members of storage classes which require the dynamic acquisition of storage when a procedure is referenced are merged with similar storage classes in the calling procedure.
4. Statically inherited environments must be carried over. As an interesting case of this, consider the example in Fig. 19. If procedure C, which inherits A's static environment, is opened up at its reference point in procedure B, then we must ensure that C continues to inherit A's static environment.

```
A: PROC
   B: PROC
      CALL C
   C: PROC
   . . .
```

Figure 19 The static environment of C and its call are different.

ORIGINAL PL/I PROCEDURES

```
P: PROC (A);                    Q: PROC (X,Y);
   B=5;                            X=X*Y;
   C=A*B;
   CALL Q (A,B);                END;
   RETURN (C);
   END;
```

After integration

```
P: PROC(A);
   B=5;
   C=A*B;
   A=A*B;
   RETURN(C);
   END;
```

After optimization

```
P: PROC(A);
   A=A*5;
   RETURN(A);
   END;
```

Figure 20 Example showing the integration and optimization of two procedures.

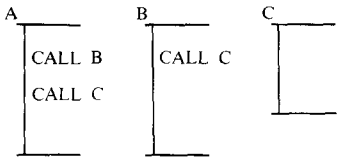

Figure 21 System of procedures.

We now turn to another of the three considerations involved in procedure integration as we are discussing it here: the conditions under which it is permitted and profitable. Surprisingly, it is nearly always permissible to integrate one procedure into another. Even if the procedure is recursive, either directly (containing a reference to itself) or indirectly, it can be integrated. (Of course, the integrator must be careful not to get into an infinite loop of integrations.)

Determining the profitability of an integration is difficult in general. Procedure size and the projected number of times a reference is executed are clearly factors. Another factor is the tailoring effect that will occur on an integrated procedure when it is optimized in the calling context. An algorithm for predicting the tailoring effects is given in [20]. In [21] it is shown that in certain contexts it is almost always profitable. Figure 20 gives an example of two procedures which are integrated and the result optimized.

It would be desirable to be able to predict at least some of the improvements to be gained by integrating a procedure and then optimizing the result. It is probable that the dramatic effects obtained in the example in Fig. 20 could not be predicted, but certain simpler cases seem promising. A particularly promising and profitable case exists when an argument is a constant and the corresponding parameter in the procedure is tested against another constant. This is often done to determine which of several alternate paths to take through a generalized procedure. Unused alternatives should disappear. Since we have found that 24% of the arguments passed in a large sample of actual programs are constants, this may be a particularly important prediction basis. In fact, it is likely that references to defining procedures will involve an even larger percentage of constants.

The third consideration related to integrating procedures is the order in which to perform the integrations. Assuming it is profitable to do a complete integration, the order in which the integration is performed can profoundly affect the optimality of the resulting code. This occurs because the optimizations which are performed after an integration work best when transforming (moving, eliminating, modifying) single expressions. A CALL is a single expression; a procedure is not. It may be possible to move a CALL out of a loop or eliminate it, but it is difficult to effect the same transformation on the expanded form. This is particularly true if the expanded form contains any control flow. Consider the little system of procedures A, B, and C given in Fig. 21.

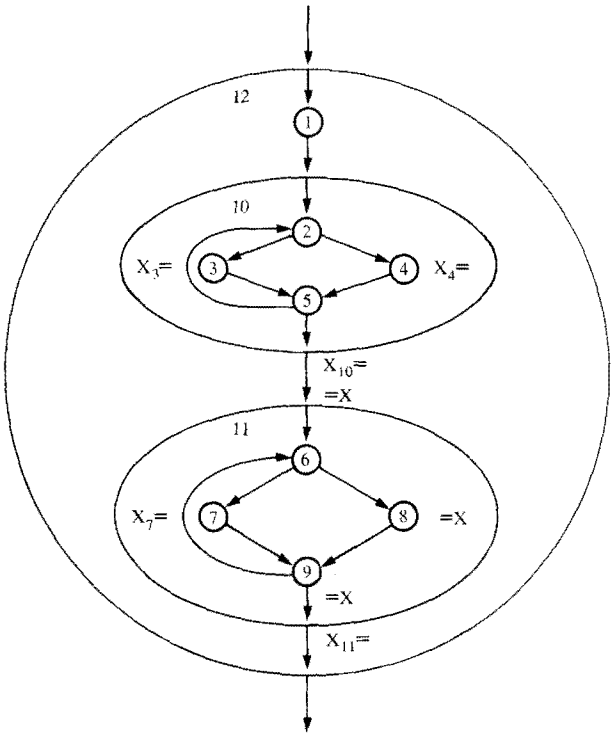

Figure 22 Control flow graph of a program partitioned into data flow domains.

Now consider applying a strategy to this system which results in integrating B into A first. A would now contain two calls to C. As a result of optimization, the second call to C might possibly be recognized as being redundant. If, however, C had been integrated into B before B was integrated into A, the possible redundancy would probably be obscured.

• *Data flow domains*

Before they are integrated, procedures have been analyzed and their internal control and data flow relationships are known. We would like to avoid redoing the entire analysis after integration. Furthermore, many of the def-use and live relationships would not be changed (parameters, global variables, and shared scopes create the exceptions), but the bit vectors would be much expanded and even sparser than before. Even within a procedure, the bit vectors could become very large, since there is one bit for every definition of interest in the program.

These and other considerations have led to *domain partitioned data flow analysis*. By this method a control flow subgraph is encapsulated and the data flow analysis completed within the subgraph. Collections of such subgraphs or *data flow domains* can be further encapsulated into larger data flow domains. Each subgraph becomes a node in its containing subgraph. Multiple definitions or uses in a data flow domain are treated as a single definition or use in the representative node.

Figure 22 shows a control flow graph consisting of basic nodes 1, 2, 3, · · ·, 9. Each of these may be a single instruction (*i.e.*, a procedure reference), a basic block, a subgraph, etc. There are definitions of X in nodes 3, 4, and 7; uses in 8 and 9. The various definitions of a variable are distinguished by subscripting the variable name with the number of the node containing the definition. Thus X_3, X_4, and X_7 appear in the example. Assume that subgraphs (2, 3, 4, 5) = 10 and (6, 7, 8, 9) = 11 form data flow domains and further that (1, 10, 11) forms the all-encompassing data flow domain represented by node 12. X_{10} then is a pseudo-definition representing all the definitions of X in node 10. Similarly we have a use and definition for node 11.

Within a data flow domain the def-use relationship is expressed directly. Thus the relationship between the definition in 7 and the uses in 8 and 9 are found directly [8] and expressed explicitly. The relationships which cross data flow domains go through one or more levels of indirection. These levels of indirection are encoded through the pseudo-definitions and uses. The data flow analysis of nodes 1, 10, and 11 finds that X_{10} can affect the use in 11. In order to find out, for example, what uses the definition in node 3 can affect, we look at the uses its representative pseudo-definition, X_{10}, can affect. Since X_{10} has a def-use relationship with the pseudo-use for node 11, we can find the actual uses of X_3 by looking at the uses represented by that pseudo-use.

The fact that domain-partitioned data flow analysis can be used to limit the lengths of the bit vectors used to do the analysis results from the fact that the number of definitions being considered at any one time can be limited. However, the number of levels of indirection needed to relate a definition and a use increases. This method is of interest in the context of integrating procedures which have been previously analyzed. The results of the analysis can be easily integrated with existing results at the time procedure integration is done.

Constant propagation

Constant propagation is a more critical optimization in ECS than in most compilers, mainly due to a larger expected proportion of compile-time constants. This is because, in ECS, an integrated defining procedure, such as the one given in the example of Section 4, will typically contain a large number of constant-valued variable references. For this reason, a rather ambitious approach to

constant propagation has been undertaken in ECS—an approach which is facilitated by the ECS methodology.

Constant propagation involves the folding of compile-time constant values into variable references. In many compilers, as in ECS, the def-use chains are used to determine all the definition points reaching a given use when constants are propagated between basic blocks. Such propagation may, in turn, give rise to an expression all of whose operands are constant, and constant propagation can proceed if the expression can be evaluated and the target of the expression is addressable.

Most constant propagators restrict the evaluation to integer arithmetic involving simple variables or temporaries. In ECS, the greatest possible latitude is provided.

Any expression that can be evaluated at object time can also be evaluated by the constant propagator at compile time. This is achieved by associating with each IL primitive a simulator that can be invoked by the ECS constant propagator to evaluate that IL operation when its operands are constant. Furthermore, any procedure whose arguments are all constant can be invoked at compile time, thus supporting the propagation of constants through built-in functions, such as SINE, and type conversion routines. The simulator for such a nonprimitive procedure is the result of integrating the more primitive simulators, and is thus an automatic product of the ECS compiler.

Propagation of constants "through storage" is supported in the ECS compiler. For example, suppose the program contains the statement

```
A(I) = J;
```

If I and J are known to have constant values 2 and 3, respectively, at this statement, then the value 3 can be propagated through A(2) to all program expressions A(K) where K is also known to be 2. The ECS BIND operation and aliasing information supports this function. The address of a bindable variable is treated by the data flow functions as a variable in its own right: a reference to the bindable variable is a use of the address, and a BIND operation is a redefinition of the address. In this way, the def-use chains are used to propagate constant "address values" as well as other values.

These functions are provided in a completely machine-independent manner. The constant propagator "knows nothing" about the storage characteristics of the object machine when it propagates constant addresses. Program variables, for the most part, are "typeless" (*i.e.*, bit strings) as far as the constant propagator itself is concerned; the IL simulators provide the type interpretation. The exceptions here are values that have been ascertained to be of pointer, label, or entry type. These values are represented in a stylized form that conveys information about the variables or program points referenced by the value. This form supports the simulation of such functions as indirect addressing and transfers to constant labels.

- *Storage overlay*

In the ECS defining procedure approach, there is no distinction between program variables and generated temporary variables. In general, the storage requirement of an ECS-compiled program before storage mapping will be considerably greater than the typical compiler's output. Furthermore, procedure integration produces enhanced opportunities for a storage overlay algorithm to determine storage-sharing opportunities for temporaries and program variables alike, in a uniform, systematic manner. Such an algorithm is described in detail in [12–14].

To illustrate the storage overlay problem, consider the PL/I program in Fig. 23(a).

Most compilers would produce the storage layout in Fig. 23(b).

Improved storage utilization would result if the compiler could observe that the first reference to E follows the last use of A and the first reference to G follows the last uses of C and D, as shown in Fig. 23(c).

An even better solution results [Fig. 23(d)] from the observation that B and G are not simultaneously live, nor are C and D. Thus, the overlay problem consists of finding sets of overlayable variables and juggling their sizes so that the total storage requirement is minimized.

Briefly described, the key to the algorithm is the concept of a *conflict graph*. The nodes of the conflict graph are the variables in a given storage class. An edge connects a pair of variables X, Y if and only if there is some node in the program flow graph where X and Y are simultaneously live and, hence, may not share storage. The minimum assignment of overlapping storage to the variables in a storage class can be formulated as an extended coloring problem. This formulation suggests the use of a simple overlay heuristic.

The nodes of the conflict graph (*i.e.*, variables in a storage class) are selected for extended coloring (*i.e.*, storage assignment) according to a figure of merit which measures the relative urgency of each node. The extended color (storage interval) is chosen from the set of available col-

(a)

```
P: PROC;
     DCL A(100), B(100), C(50), D(50), E(100), G(100);
     GET LIST( B, C );
     A = F1( B, C );
     D = F2( A, B );
     E = F3( B, D );
     G = F4( E );
     PUT LIST( E, G );
END P;
```

(b)

A(100)	B(100)	C(50)	D(50)	E(100)	G(100)

Total: 500 cells

(c)

A(100)	B(100)	C(50)	D(50)
E(100)		G(100)	

Total: 300 cells

(d)

A(100)	B(100)	C(50)
E(100)	G(100)	D(50)

Total: 250 cells

Figure 23 Effects of improved storage overlays.

ors according to a storage selection strategy, such as first-fit.

• *Instruction aggregation*

An important part of the machine tailoring phase is a process for recognizing that certain groups of instructions compute a value which can be computed by a single instruction of higher complexity. Instructions to be aggregated are related by their data flow—not by their physical proximity. In order to deal with the aggregation of instructions which are not immediately adjacent, the machine tailoring phase of the ECS compiler makes use of data flow analysis which has already been performed by the semantic elaboration phase.

For example, most computers allow for operand addressing via some kind of base/index register arrangement in which an implicit add/subtract operation is used to derive an effective address which points to the actual data to be manipulated. On the IBM System/370, storage operands may be addressed by summing the value in a base register (B), the value in an index register (X), and a displacement (D) which must be a compile-time constant.

This information is expressed by writing a *pattern*. The pattern characterizes the real machine's complex instruction by expressing their functions as a set of simpler IL instructions related by data flow. For example, a pattern for this BXD sequence would begin as

```
BXD pattern:    ADD    (T1 = B + D)
                ADD    (T2 = T1 + X)
                BIND   (T, T2)
```

To match the patterns against the program, a pass is made through the program. Each instruction is matched against all pattern points (the simpler IL instructions appearing in the patterns) which are applicable to its operation code.

For pattern points whose inputs come from other instructions in the pattern, determining the success of a match requires determining whether some other pattern-point/instruction match is successful. This situation is dealt with recursively. A collection of "already-tried" flags is used to prevent repeated attempts.

If the pattern point is successfully matched against the instruction, a resolution is constructed for use in substitution. The resolution is a map from the identifiers used in the pattern description to the actual variables used in the program fragment that matches the pattern. For pattern points which have several alternatives, the "best" alternative is selected.

Having determined the matches for all program points, instruction aggregation chooses which productions are to be executed. This choice can be accomplished by numerous algorithms, the simplest of which is a bottom-up "greedy" algorithm. Code production requires that a value be assigned to each pattern point whose match causes code to be produced. Such pattern points are called terminal pattern points. The value measures the time or space saved by using the higher-complexity instruction to be generated instead of its expansion. In addition, each terminal pattern point must have a production rule, and each operation code must have a default production rule. These are used to form the replacement for matched and unmatched pattern points, respectively.

• *Register allocation*

The ECS register allocator is based on the approach given in [22]. It consists of five phases:

1. The relative frequencies of program points (*i.e.*, RL instructions) are estimated. In the absence of real frequencies, this is necessarily determined by such control flow patterns as nested strongly connected regions.
2. The displacement priorities of the variables at each point are established. These priorities are based on a

frequency-weighted measure of the distance to the next use. These priorities are used when determining which variable to displace when the allocation phase finds that it is out of registers.

3. Variables are allocated to registers in that a decision is made as to which variables at each program point are contenders for registers. The actual decision as to which symbolic register they will get is made in the next phase. In this phase we note when the value of a potential register contender is also "home," *i.e.*, the current value for the variable also exists in storage.
4. Registers are assigned symbolically and the skeletal code sequences are determined. This does not designate the absolute register. An infinite supply of symbolic registers is assumed overall, but no more than the actual number of registers may be in use at any point.
5. The symbolic registers are given absolute designations.

By separating the allocation of variables to symbolic registers from the assignment of variables to actual registers, we can permute the allocations to decrease mismatches and the consequent register moves.

Having selected the absolute registers, the code skeletons chosen earlier can be finalized.

References

1. William Harrison, "A New Strategy for Code Generation—The General Purpose Optimizing Compiler," *IEEE Trans. Software Engineering* **SE-3,** 243-250 (1977).
2. *OS PL/I Optimizing Compiler: General Information*, Order No. GC33-0001; available through the local IBM branch office.
3. J. Lawrence Carter, "A Case Study of a New Code Generation Technique for Compilers," *Commun. ACM* **20,** 914-920 (1977).
4. William Harrison, "Formal Semantics of a Schematic Intermediate Language," *Research Report RC6271*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1976.
5. *OS PL/I Checkout Compiler: General Information*, Order No. GC33-0003; available through the local IBM branch office.
6. F. E. Allen, "Interprocedural Data Flow Analysis," *Proceedings, IFIP Congress 74*, North-Holland Publishing Co., Amsterdam, 1974, pp. 398-402.
7. William E. Weihl, "Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables," *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 83-94.
8. F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Commun. ACM* **19,** 137-147 (1976).
9. R. Endre Tarjan, "Testing Flow Graph Reducibility," *J. Computer Syst. Sci.* **9,** 355-365 (1974).
10. F. E. Allen, J. Cocke, and K. Kennedy, "Reduction of Operator Strength," *Data Flow Analysis*, Neil D. Jones and Steven S. Muchnick, Eds., Prentice-Hall, Inc., Englewood Cliffs, NJ, in press.
11. William Harrison, "Compiler Analysis of the Value Range of Variables," *IEEE Trans. Software Engineering* **SE-3,** 243-250 (1977).
12. J. Fabri, "Automatic Storage Optimization," *Research Report RC7967*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1979.
13. J. Fabri, "Automatic Storage Optimization," *Proceedings of the SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices* **14,** 83-91 (1979).
14. J. Fabri, "Automatic Storage Optimization," Ph.D. Thesis, *Courant Computer Sciences Report No. 14*, Courant Institute of Mathematical Sciences, New York University, 1979.
15. F. E. Allen, "Interprocedural Analysis and the Information Derived by It," *Programming Methodology: Lecture Notes in Computer Science*, Vol. 23, Springer-Verlag, Heidelberg, Germany, 1975, pp. 291-321.
16. Jeffrey M. Barth, "Interprocedural Data Flow Analysis Based on Transitive Closure," *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Santa Monica, CA, January 1977, pp. 119-131.
17. John Banning, "An Efficient Way to Find the Side Effects of Procedure Calls and Aliases of Variables," *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, TX, January 1979, pp. 29-41.
18. T. C. Spillman, "Exposing Side Effects in a PL/I Optimizing Compiler," *Proceedings, IFIP Congress 71*, North-Holland Publishing Co., Amsterdam, 1971, pp. 376-381.
19. B. K. Rosen, "Data Flow Analysis for Procedural Languages," *J. ACM* **26,** 322-344 (1979).
20. J. Eugene Ball, "Predicting the Effects of Optimization on a Procedure Body," *Proceedings of the SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices* **14,** 214-220 (1979).
21. Robert W. Scheifler, "An Analysis of Inline Substitution for a Structured Programming Language," *Commun. ACM* **20,** 647-654 (1977).
22. William Harrison, "A Class of Register Allocation Algorithms," *Research Report RC5342*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1975.

Received June 8, 1979; revised June 4, 1980

J. Fabri is deceased. The other authors are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.