MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 203 September 1980

Formalizing the Expertise of the Assembly Language Programmer

Roger DuWayne Duffey II

Abstract:

A novel compiler strategy for generating high quality code is described. The quality of the code results from reimplementing the program in the target language using knowledge of the program's behavior. The research is a first step towards formalizing the expertise of the assembly language programmer. The ultimate goal is to formalize code generation and implementation techniques in the same way that parsing and code generation techniques have been formalized. An experimental code generator based on the reimplementation strategy will be constructed. The code generator will provide a framework for analyzing the costs, applicability, and effectiveness of various implementation techniques. Several common code generation problems will be studied. Code written by experienced programmers and code generated by a conventional optimizing compiler will provide standards of comparison.

I. Introduction

This research proposes a novel compiler strategy for generating high quality code. The strategy evolved from studying the differences between the code produced by experienced programmers and the code produced by conventional optimization techniques. The strategy divides compilation into four serial stages: analyzing the source implementation, undoing source implementation decisions, reimplementing the program in the target language, and assembling the object modules. This strategy is termed the knowledge based reimplementation (KBRI) strategy. The KBRI strategy differs from conventional approaches in three major ways:

1. The program is reimplemented in the target language based on knowledge of the program's behavior, rather than translated into the target language on a statement by statement basis and then transformed to improve performance.
2. Descriptions of the program's behavior, its source implementation, and its target implementation are expressed in the same language. They are related in terms of a hierarchial structure linking each implementation feature to the descriptions of the behaviors that it implements.
3. Knowledge of source or target implementation techniques can be used for either analysis or generation.

This research is a first step towards formalizing the expertise of the assembly language programmer. The ultimate goal is to formalize code generation and implementation techniques in much the same way that parsing and flow analysis techniques have been formalized. The immediate objective is to understand how knowledge of the program's behavior, knowledge of source implementation techniques, and knowledge of target implementation techniques can be used in generating high quality code.

An experimental code generator, named Cobbler, will be constructed from the reimplementation strategy. It will translate Pascal into PDP11 assembly language. Cobbler will provide a framework for analyzing the costs, applicability, and effectiveness of various implementation techniques. It will be used to test how different pieces of knowledge affect the generated code. Several common code generation problems will be studied. Code written by experienced programmers and code generated by a conventional optimizing compiler will provide standards for comparison.

The remainder of this paper discusses the research in more detail. The following section illustrates the differences between hand generated code and compiler generated code with an example program. Section III defines the limits of the present study. The next section outlines the architecture of the KBRI compiler. The fifth section uses the example program to explain how the KBRI compiler works. Section VI contrasts the KBRI compiler's approach to code generation with the approach of a conventional optimizing compiler. The final section discusses related work.

II. What Does a Programmer Do?

Consider the problem of coding a high level language program in assembly language. An experienced programmer analyzes the program's behavior to decide how to implement the program in the target language. The result is a high quality target implementation which is efficient in time and space. In contrast, a conventional optimizing compiler has only a limited notion of program behavior and produces a lower quality implementation.

An example program will help to explain the differences between the code written by an assembly language programmer and the code generated by a conventional optimizing compiler. Figure II.1 shows a small Pascal program to initialize a square, two dimensional array to the identity matrix. The program takes each row in the array, sets each element in the row to 0, and then sets the diagonal element to 1. There are two things to note about this example. First, the program is implemented with two nested loops because the data structure is viewed as a two dimensional array. Second, setting each diagonal element to 0 before setting it to 1 is useless, but simplifies the Pascal implementation of the program.

```
line    { Initialize the array A to the identity matrix }

  1     const DIM = 4;
  2     var   I : 1..DIM; J : 1..DIM;
  3           A : array[1..DIM, 1..DIM] of 0..255;
  4     begin
  5       for I := DIM downto 1 do  { For each row in the array }
  6         begin
  7           for J := DIM downto 1 do  A[I,J] := 0;  { Set each element to 0 }
  8           A[I,I] := 1  { Then set the diagonal element to 1 }
  9         end
 10     end
```

Figure II.1 : The source language program

A conventional compiler replaces each source operation with one of several general target code sequences which implement the source operation. A conventional compiler misses many better target implementations because the general code sequences do not take advantage of the special properties of the context where the operations occur. Figure II.2 shows the PDP11 code produced by a conventional compiler for the Pascal program. Note that the structure of the target implementation directly reflects the structure of the source implementation.

```
line    label   instruction         comments

  1             MOV   #4,R0     ;init I, outer loop index in R0
  2     L1:     MOV   #4,R1     ;init J, inner loop index in R1
  3     L2:     MOV   R0,R3
  4             ASL   R3
  5             ASL   R3
  6             ADD   #A-5,R3
  7             ADD   R1,R3     ;form address (A+(I-1)*4-1+J) in R3
  8             CLRB  @R3       ;zero the element A[I,J]
  9             DEC   R1        ;decrement the inner loop index
 10             BGT   L2        ;loop until row has been zeroed
 11             MOV   R0,R3
 12             ASL   R3
 13             ASL   R3
 14             ADD   #A-5,R3
 15             ADD   R0,R3     ;form address (A+(I-1)*4-1+I) in R3
 16             MOVB  #1,@R3    ;set diagonal element A[I,I] to 1
 17             DEC   R0        ;decrement the outer loop index
 18             BGT   L1        ;loop until all rows initialized
```

Figure II.2 : Code produced by conventional compiler without optimization

A conventional optimizing compiler improves the generated code by applying a sequence of transformations to the program implementation. Each transformation must preserve the meaning of the program. Each application is based on information gathered from a limited region of the implementation. Figure II.3 shows the code produced for the example by a hand simulation of a conventional optimizing compiler. The simulated compiler is equivalent to the Bliss SIX/12 compiler developed by Wulf [46]. The optimizing compiler improves the target implementation in two ways:

1. The unoptimized code recomputes the expression A+(I-1)*4-1+J for each repetition of the inner loop. This is wasteful because the subexpression A+(I-1)*4-1 is constant in the inner loop. *Rho motion* moves the code for the subexpression out of the inner loop and introduces a compiler created variable to maintain the value of the subexpression.
2. The subexpression A+(I-1)*4-1 is needed to compute the address of both A[I,J] and A[I,I]. The unoptimized code recomputes the subexpression each time. This is wasteful because the value of the subexpression is available when the address of A[I,I] is computed. *Common subexpression elimination* replaces the code which recomputes the subexpression with code referencing a compiler created variable.

The optimized target implementation uses 1 more register, but executes 30% faster and resides in 15% less space than the unoptimized target implementation. The basic structure of the target implementation remains the same. Again note that the target implementation consists of two nested loops and that each diagonal element is set to 0 before it is set to 1.

```
line    label   instruction         comments

  1             MOV   #4,R0     ;init I, outer loop index in R0
  2     L1:     MOV   #4,R1     ;init J, inner loop index in R1
  3             MOV   R0,R2
  4             ASL   R2
  5             ASL   R2
  6             ADD   #A-5,R2   ;form address (A+(I-1)*4-1) in R2
  7     L2:     MOV   R2,R3
  8             ADD   R1,R3     ;form address (A+(I-1)*4-1+J) in R3
  9             CLRB  @R3       ;zero the element A[I,J]
 10             DEC   R1        ;decrement the inner loop index
 11             BGT   L2        ;loop until this row is zeroed
 12             ADD   R0,R2     ;form address (A+(I-1)*4-1+I) in R2
 13             MOVB  #1,@R2    ;set diagonal element A[I,I] to 1
 14             DEC   R0        ;decrement the outer loop index
 15             BGT   L1        ;loop until all rows initialized
```

Figure II.3 : Code produced by conventional optimizing techniques

An assembly language programmer reimplements the Pascal program by applying knowledge of target language implementation techniques to achieve the same results as the source implementation. Implementation decisions are based on knowledge of the program's behavior and tradeoffs between different parts of the implementation. Figure II.4 shows the target implementation produced by an assembly language programmer for the Pascal program. There are two things to note about the target implementation. First, the structure of the implementation does not reflect a two dimensional array. The loop divides the elements of A into the last diagonal element and 3 groups of 5 elements. Each group consists of 4 non-diagonal elements followed by 1 diagonal element. This suggests that A is being viewed as an element and a 3 by 5 array. Further, each element of A is accessed once using an auto-decrement pointer. This suggests that A is being viewed as a one dimensional vector. Second, the restructuring of the target implementation simplifies the address computations and eliminates the need to set each diagonal element twice. The programmer's implementation uses 2 less registers, executes three times faster, and resides in 40% less space than the optimizing compiler's implementation.

```
line    label   instruction         comments

  1             MOV   #A+17,R3   ;R3 points to last element of A + 1
  2             MOVB  #1,@R3     ;set diagonal element A[4,4] to 1
  3             MOV   #3,R0      ;no. nondiagonal element sequences
  4     L1:     CLRB  -(R3)      ;zero elements between the last
  5             CLRB  -(R3)      ;  diagonal element A[I+1,I+1] and
  6             CLRB  -(R3)      ;  the next diagonal element A[I,I]
  7             CLRB  -(R3)      ;
  8             MOVB  #1,-(R3)   ;set the diagonal element A[I,I] to 1
  9             DEC   R0         ;loop until all nondiagonal element
 10             BGT   L1         ;  sequences have been zeroed
```

Figure II.4 : Code produced by an assembly language programmer

These differences pose three research questions. First, what knowledge about the program is used by the programmer? Second, what knowledge about the target machine is used by the programmer? Third, how is this knowledge used in the process of reimplementing the program? In attempting to answer these questions note that there are several differences between reimplementing the program by refining a high level specification and improvment by applying transformations to a (low level) program.

- The transformation approach suffers from the classical search problems and phase ordering problems. The reimplementation approach avoids these problems by not enforcing a strict order for considering problems. Instead, it uses parallel exploration of distinct possibilities and eliminates poor choices through comparison and the recognition of tradeoffs.

- The transformation approach suffers from code fragmentation and dispersion with the code motion transformations. This makes it difficult to recognize when particular features of the target language can be exploited. (eg. the lack of use of auto-decrement addressing) The reimplementation approach avoids this problem by describing the implementation in terms of a hierarchy. The hierarchy relates a high level concept to its implementation in terms of the machine's primitive operations.

- The transformation approach's notion of behavior is limited to local operation groups and single, standard loop structures. The reimplementation approach embodies a strong notion of the program's behavior, such as describing the action of the nested loops as storing a sequence of 1's and 0's in memory.

III. An Overview of the Research Program

The competence to be studied is the programmer's ability to efficiently implement a high level language program in assembly language. An account of the reimplementation process must identify how knowledge of the program's behavior and assembly language implementation techniques are employed in reimplementing the program. The study will focus on understanding how various implementation options are recognized, evaluated, and selected. Therefore, the account of the reimplementation process concentrates on the ability to characterize the applicability and desirability criteria of implementation options, on the ability to compare alternatives at differing levels of detail, and on the ability to control resource tradeoffs among implementation options. A code generator which implements this process will be constructed for test purposes. The result of the study will be an understanding of how an implementation decision affects the quality of the generated code.

The test domain is the implementation decisions relating data representation and program structure. Implementation problems in this domain range from high level considerations such as data flow mechanisms for repetitive and procedural constructs, to low level considerations such as global register allocation or access mode determination (ie. determination of how an instruction will access an operand). The implementation problems to be considered will focus on the interactions between the structure of the implementation and the features of the target machine architecture. This domain was chosen because the implementation decisions offered by programmers and conventional compilation techniques differ widely and often have pronounced effects on the quality of the target implementation.

The source language and target machine architecture for the proposed study were selected for their ability to illustrate implementation problems which are common to a wide range of languages and machines, and to facilitate comparison with the conventional optimization techniques described in the literature. The source language is a subset of Pascal [20] designed around the common features of the algebraic programming languages. The target machine architecture is the common instruction set of the PDP11 processor family [14].

This study's goal is to identify how knowledge of the program's behavior, knowledge of source implementation techniques, and knowledge of target implementation techniques can be employed to generate high quality code. Work by Rich, Shrobe, and Waters [29,30,35,41] is considering the problems in analyzing the source program automatically. Work by Barbacci, Joyner, and Wick [8,21,43] is approaching the problem of automatically deriving the target implementation techniques from descriptions of the target language. This study concentrates on how this knowledge is used in making implementation decisions rather than on how this knowledge is derived. Cobbler will demonstrate how different pieces of knowledge can be used in reimplementing programs.

IV. The Architecture of a KBRI Compiler

A KBRI compiler is divided into four basic parts (see Figure IV.1). The data base records the state of the program implementation at each moment. It maintains descriptions of the program, the source language implementation of the program, potential target language implementations of the program, and their relationships to each other. It also serves as the communications medium for the procedural experts which represent the knowledge of how to implement a program behavior.

The source and target blocks contain Cobbler's knowledge of how to implement a program behavior in the source and target languages. Each block is divided into two parts separating data structure and procedural structure implementation techniques. Note that although the source knowledge is used exclusively for analysis and the target knowledge for generation in a compiler, there is nothing inherent in the knowledge or the representation to enforce this asymmetry. Any piece of knowledge can be used for either analysis or generation.

Control of the implementation process is vested in the Conflict Resolution Monitor (CRM). The CRM controls the selection of defaults, the analysis of different tradoffs, and the expansion of potential implementations.

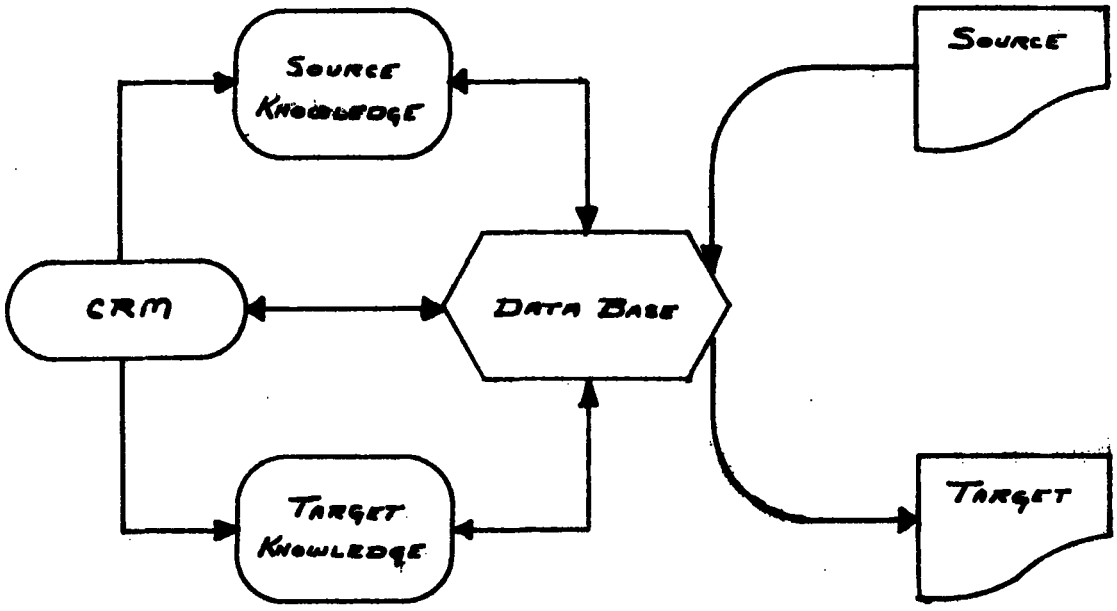

Figure IV.1 : A KBRI compiler can be divided into four sections

A. Representing the program in the data base

The data base represents the state of the program's source and target language implementation in terms of a Plan Implementation Language (PIL). PIL is based on the plan representation developed by Rich, Shrobe, and Waters [29,30,35,41]. PIL plans are language independent. They divide a program's implementation into hierarchial segments and describe their data flow, their control flow, and their purposes. They do not introduce non-essential constraints into the representation of the program's implementation. PIL makes all of the information about a program's implementation explicit. This means that it can be read directly from the representation without reasoning. The program and its source and target language implementations form an implementation hierarchy which relates each aspect of one implementation to the corresponding aspect of the other implementations (see Figure IV.5).

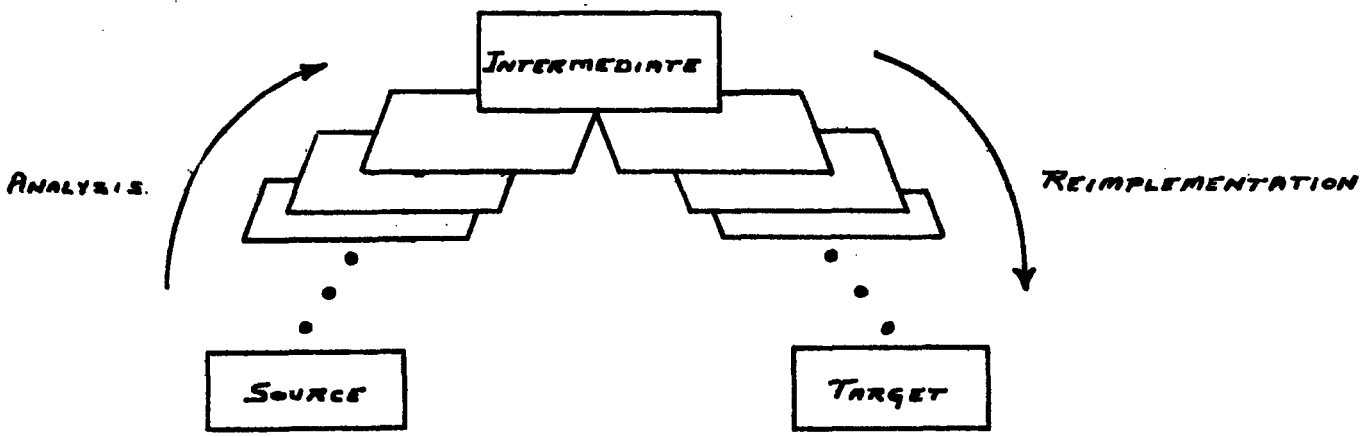

Figure IV.5 : The program's implementation hierarchies

B. Representing knowledge about implementation techniques

The computation problem is to use knowledge of how to implement a program behavior to refine a program's target language implementation. This entails the ability to recognize when an implementation technique is applicable. It also requires the ability to select from among many different implementations by comparing their desirability in terms of collaborations, tradeoffs, and resource useage.

Different ways of implementing a program behavior are represented in terms of procedural entities called sprites [22]. A sprite consists of a pattern and a body of code. They can be enabled or disabled by the CRM. If a sprite is enabled, it's body of code is executed once for each item in the data base which matches it's pattern. Sprites communicate with each other by adding new data to the data base.

Knowledge about how to control the implementation process is represented in terms of procedural entities called actors [18]. Actors communicate with each other by passing messages. An actor consists of a pattern and a body of code. An actor's body of code is executed once for each message that it receives. The CRM is actually a collection of actors which monitor the state of the implementation and control which groups of sprites are enabled.

C. An outline of the compilation process

The KBRI compiler strategy divides compilation into four serial stages: analyzing the source implementation, undoing the source implementation decisions, reimplementing the program in the target language, and assembling the object code modules.

Stage 1 translates the program's source implementation into a source plan. The source plan expresses the source implementation in a language independent form. It differs from the source language representation in two ways. First, it explicitly represents data flow, control flow, and underlying operations such as operand coercions or dynamic checking of subscript ranges. Second, complex surface constructions such as loops are expressed in terms of hierarchial structures of primitive operations. The hierarchial structures simplify implementation analysis by reducing the number of primitive operations. They avoid the conventional problem of scattering the primitive operations throughout the source implementation by retaining a high level characterization of each construction.

Stage 2 expands the source plan into the implementation intermediate plan and a set of source implementation decisions represented as plans. The source implementation decisions define a hierarchial structure which relates the intermediate plan to the source plan. This structure separates the programmer's decisions about how to implement the program in the source language from the other decisions made during the programming process. The intermediate plan differs from the surface plan in 3 ways. First, it represents the program as a partially ordered set of operations on unstructured objects. The order of operations is constrained only by the control and data flow semantics of the operations. Second, the intermediate plan represents repetitions as a sequence of states. The initial and terminal states of the sequence are distinguished from a generic intermediate state. Data flows connecting to the repetition are described as sequences in correspondence with the sequence of states. Third, the notion of variable is entirely replaced by the notion of data flow.

Stage 3 reimplements the program in the target language. Cobbler is an implementation of this stage. The reimplementation process is a breadth limited, parallel search through a space of potential implementations. The evaluation criteria include resource tradeoffs between implementation options, the desirability criteria of each implementation option, and estimates of

storage and execution time requirements. The CRM controls the search process by allocating resources to the individual experts. The structure and target resource requirements of each program segment decide when the program segment is examined. As implementation decisions are made, the search process moves to the program segments that are affected by those implementation decisions. An analysis of the global program features provides a starting point for the search process. An implementation decision defaults to the choice made in the source implementation, when the evaluation criteria and the knowledge of target implementation techniques do not force the choice. Note that this ensures that the search process terminates. It also preserves the programmer's choices which are based on the application domain of the program but are arbitrary with respect to the target language.

Stage 4 does object code assembly and relocation.

D. Testing the KBRI compiler strategy

Only Cobbler (stage 3 of a KBRI compiler) will be implemented for this thesis. Source implementation descriptions will be constructed manually and used as input to Cobbler. The reimplementation strategy will be tested by comparing the code produced by Cobbler with the code produced by several assembly language programmers and by a conventional optimizing compiler.

Cobbler will translate Pascal into PDP11 assembly language. The Bliss SIX/12 compiler developed by Wulf [45] will be used for comparison. This compiler was chosen because it is one of the very few optimizing compilers that have been fully described within the literature. It is reasonable to use the Bliss SIX/12 compiler for comparison because each of the Pascal programs to be used can be translated into an equivalent Bliss program by a trivial sequence of syntactic transformations. The test examples will be selected from the published literature on algebraic languages and code optimization. The target implementations will be compared in terms of their computational structure and the tradeoffs made to produce them, in addition to the traditional measures of execution time and storage requirements. No claims will be made regarding the psychological validity of the reimplementation strategy.

V. A Simple Example Showing How the KBRI Compiler Works

An example will clarify how Cobbler works. The example is the Pascal program presented in section II, which initializes an array to the identity matrix. The example describes all 4 stages of compilation, but concentrates on describing Cobbler's operation. Following the example, the code produced by Cobbler is compared with the code produced by a conventional optimizing compiler and an assembly language programmer.

```
{ Initialize the array A to the identity matrix }

const DIM = 4
var   I : 1..DIM; J : 1..DIM;
      A : array[1..DIM, 1..DIM] of 0..255;

begin
  for I := DIM downto 1 do  { For each row in the array }
    begin
      for J := DIM downto 1 do  A[I,J] := 0;  { Set each element to 0 }
      A[I,I] := 1  { Then set the diagonal element to 1 }
    end
end
```

Figure V.1 : the source program input

Stage 1 translates the source program into PIL. The input is a stream of characters representing the program's source implementation (see Figure V.1). Stage 1 does a lexical and syntactic analysis of the input, and also processes the program declarations. The results of stage 1 are a symbol table and the source implementation plan for the program (see Figure V.2).

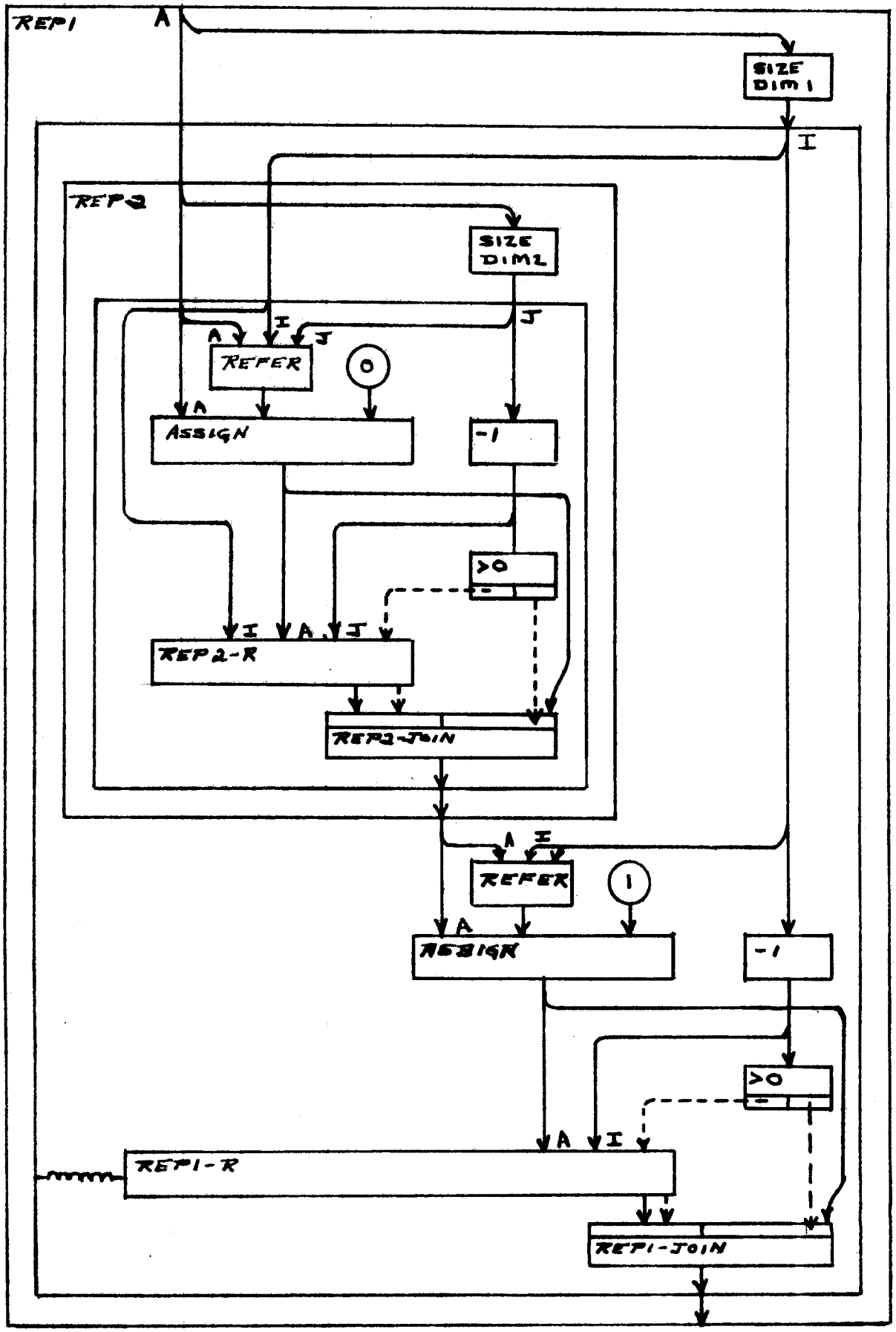

Figure V.2 : the source implementation plan

A program's source text and its source implementation plan differ in only one major way. The source implementation plan exposes the underlying operations that the source language syntax was designed to represent implicitly. The example illustrates this difference in several ways:

- Data flow arcs (solid arrows) explicitly represent the movement of the data objects during the program's execution. Data flow represents the effects of many different source implementation mechanisms. For example, the source program uses the variable J to hold the second array index. The source implementation plan uses four data flow arcs to record where the values of J are used.

- Control flow arcs (dashed arrows) explicitly represent the flow of control during the program's execution. Control flow represents the effects of many different source implementation mechanisms. For example, the source program uses the arrangement of the statements to show the two execution paths from the end of a loop. The source implementation plan uses two control flow arcs to represent the execution paths.

- Coercions are represented explicitly. Consider setting A[I,J] to 0 in the example. The ASSIGN operation requires the array object A, the constant 0, and a reference to the [I,J] component of A. It yields a new array object A in which A[I,J] is 0. The REF operation is a coercion which creates the appropriate reference object from the array object A, the value of I, and the value of J.

- The plans for high level surface constructions expose the underlying source language primitive operations. Consider the inner FOR-loop which zeroes the elements in the Ith row of A. REP2 describes the FOR-loop at two levels. The lower level describes the FOR-loop as a tail recursion terminated when a counter reaches 0. Initializing, updating, and testing the counter are all represented explicitly. The higher level describes the FOR-loop as a "black box". Its behavioral descriptions do not refer to its internal structure. Note that the structure of the plan links the high level characterization to the primitive operations. Thus PIL avoids the problem of scattering the primitive operations throughout the representation and losing the notion of a repetition.

Stage 2 analyzes the implementation decisions which formed the source implementation of the program. The results of stage 2 are an implementation intermediate plan for the program and an implementation hierarchy which relates the implementation intermediate plan to the source implementation plan. (see Figure V.3)

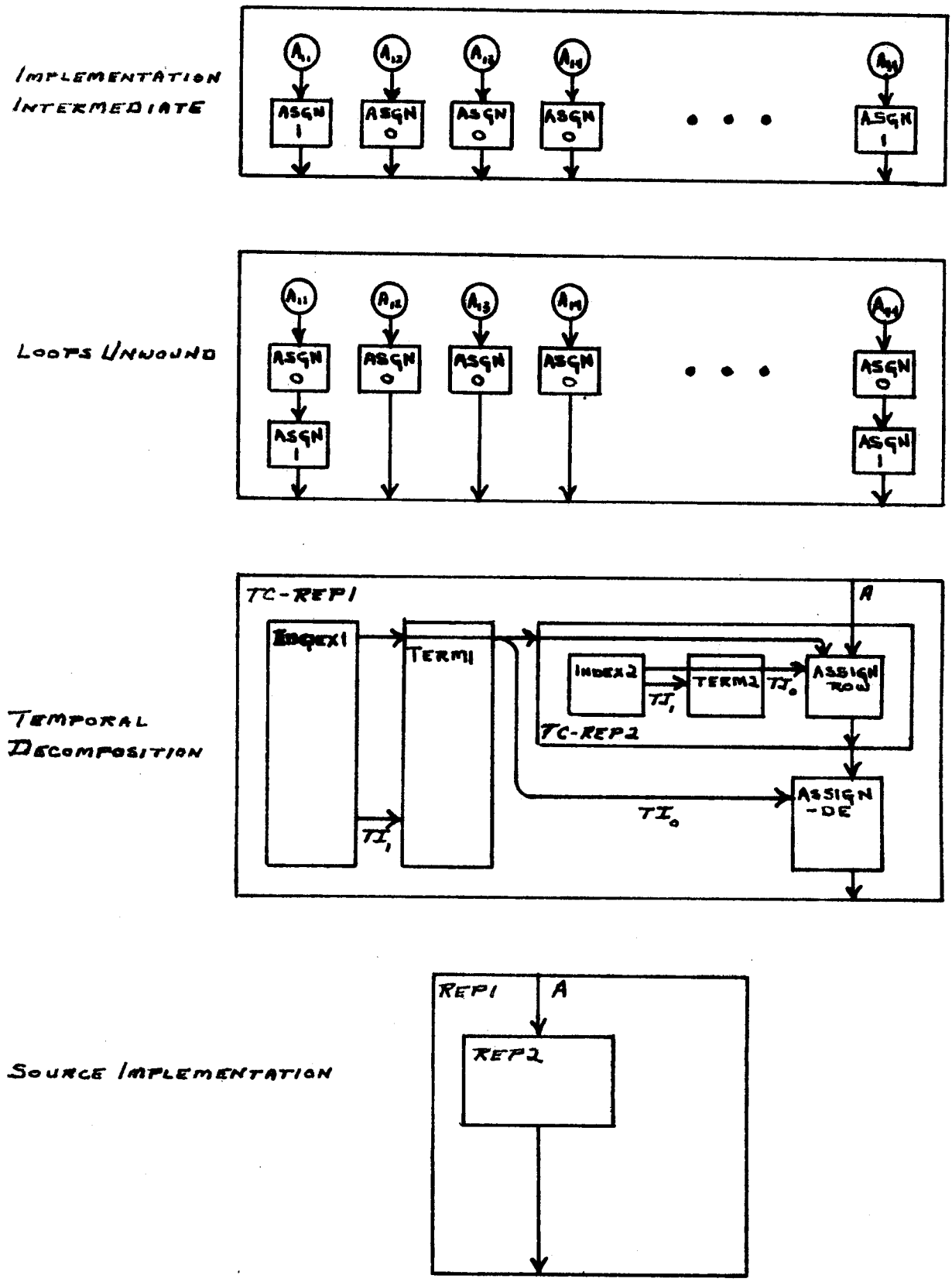

Figure V.3 : The implementation hierarchy

First, temporal decomposition is applied to the repetition REP1. The plan TC-REP1 is created which describes REP1 in terms of an augmentation INDEX1, a termination TERM1, a second augmentation ASSIGN-DE, and REP2. INDEX1 generates the sequence of values taken on by I. The sequence of values is represented by the temporal data flows TI_0 and TI_1. TERM1 controls the repetition by testing TI_1. The repetition terminates when TI_1 becomes 0. ASSIGN-DE takes an array object A, and sets the diagonal element indicated by the current value of TI_0 to 1.

Temporal decomposition is then applied to the repetition REP2 creating the plan TC-REP2 (see Figure V.4). Note, Waters has implemented a system which can perform all of the analysis up to this point [41].

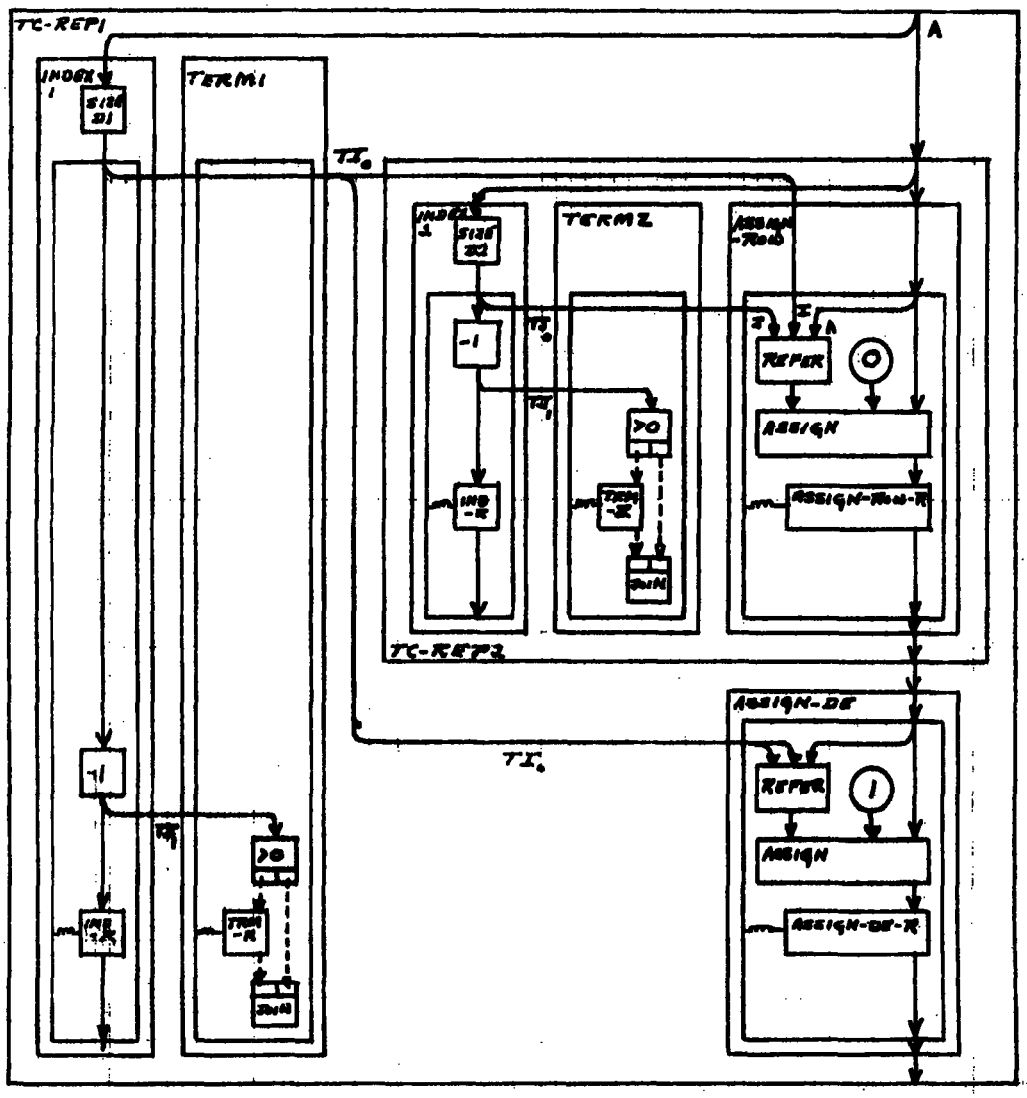

Figure V.4 : Temporal decomposition of the source implementation plan

At the next level the decision to implement the matrix A with a Pascal array is undone. The repetitions are then unwound using the temporal data flows (see Figure V.5). Note the distinction between the notion of a Pascal array and the notion of the matrix. The subscripts on the data objects convey the notion of the matrix.

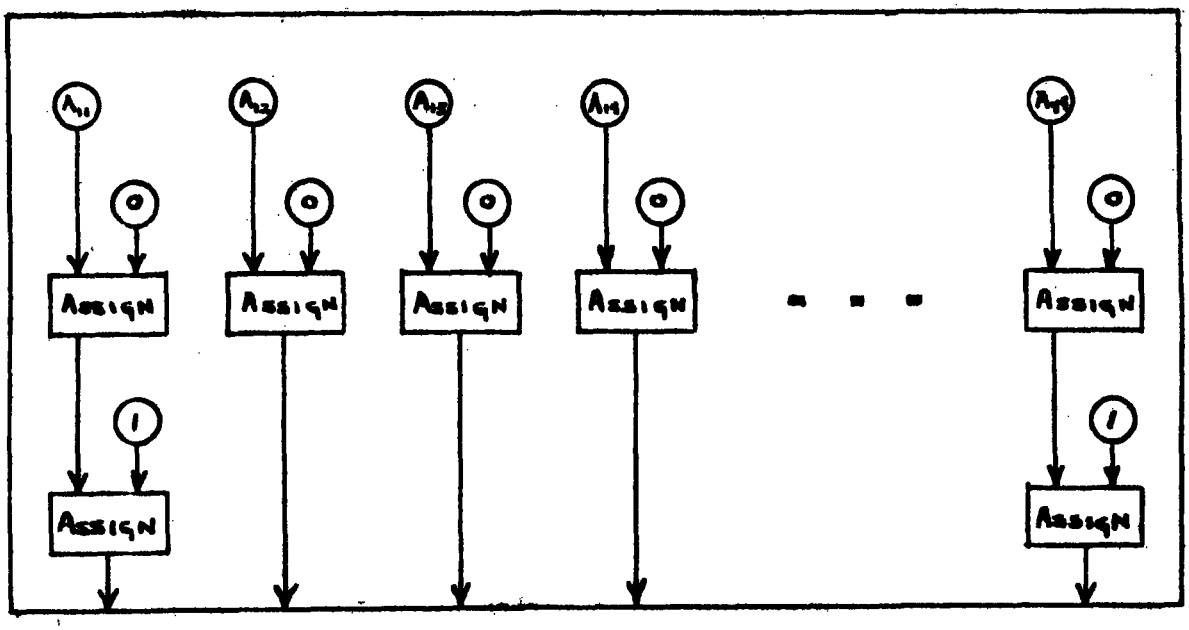

Figure V.5 : Plan after the loops are unwound

At the top level the decision to zero each diagonal element before setting it to 1 is eliminated. This decision allowed the source implementation to avoid checking whether each element was on the diagonal before setting it. This is the implementation intermediate plan for the program.

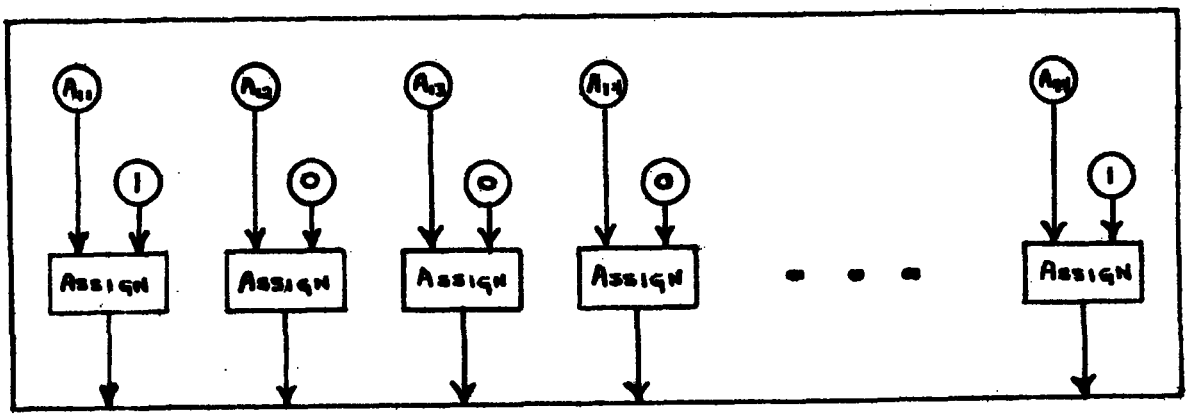

Figure V.6 : The implementation intermediate plan for the program

Cobbler (i.e. stage 3) reimplements the program in the target language. Cobbler's results are the target implementation plan for the program and an implementation hierarchy which relates the implementation intermediate plan to the target implementation plan.

First, Cobbler considers how to implement the matrix A. It decides to use an 8 bit byte to implement each element of A because the source implementation used the integer subrange 0 to 255 to implement them. Next three collections of sprites suggest different ways of implementing the matrix to the CRM. One group suggests treating the matrix as a 4 by 4 array and storing the elements in row major form. Another group suggests treating the matrix as a 16 element vector stored sequentially in memory. The third group suggests treating each element as an independent variable (see Figure V.7).

The CRM compares the estimated resource requirements for each suggestion. It enables the vector representation sprites. The CRM does not enable the array representation sprites because address computations are more expensive for the array representation than for the vector representation (arrow 1). It does not enable the independent variable representation sprites because of the size of the code required to set each element independently (arrow 2).

Next Cobbler considers different ways of implementing the program using the vector representation. Three different possibilities are suggested. First, each element can be initialized with a separate instruction (EXPLICIT-1). Second, the last element can be set to 1 followed by the repetition (REP-DIAGONAL). The repetition body zeroes four non-diagonal elements and then sets the following diagonal element to 1. It is repeated three times. Third, a repetition (REP-ZERO) can be used to zero each element in the vector. This repetition is executed 16 times. Then the diagonal elements must be set to 1. The CRM enables EXPLICIT-1 and REP-DIAGONAL. The CRM does not enable REP-ZERO because the overhead in repeating a segment 16 times is much greater than the overhead in repeating a segment 3 times (arrow 3).

A vector can be accessed in two ways: with an index pointer or with a base address pointer. EXPLICIT-1 and REP-DIAGONAL are divided into two parts based on how the vector is accessed. Cobbler then refines the index pointer into an auto-decrement pointer because each element is accessed once and in address order. An auto-increment pointer is never suggested because Cobbler recognizes this as equivalent to the auto-decrement pointer in this case. Then the CRM disables the base address implementations in EXPLICIT-1 (arrow 4) and REP-DIAGONAL (arrow 5) because a base address instruction is more expensive in time and space than an autodecrement instruction.

Next Cobbler considers different ways of zeroing the four non-diagonal elements in REP-DIAGONAL. There are two possibilities. Either each element can be zeroed with a separate instruction (EXPLICIT-2), or a repetition (REP-OFFDIAGONAL) can be used to zero the four non-diagonal elements. The CRM enables EXPLICIT-2. It does not enable REP-OFFDIAGONAL

because it requires another register and takes longer to execute than EXPLICIT-2 (arrow 6)

Lastly the CRM disables EXPLICIT-1 because the code for EXPLICIT-1 will take up much more space than the code for REP-DIAGONAL (arrow 7). The resulting plan for REP-DIAGONAL is at the level of the target machine instructions. Cobbler (ie. stage 3) is done.

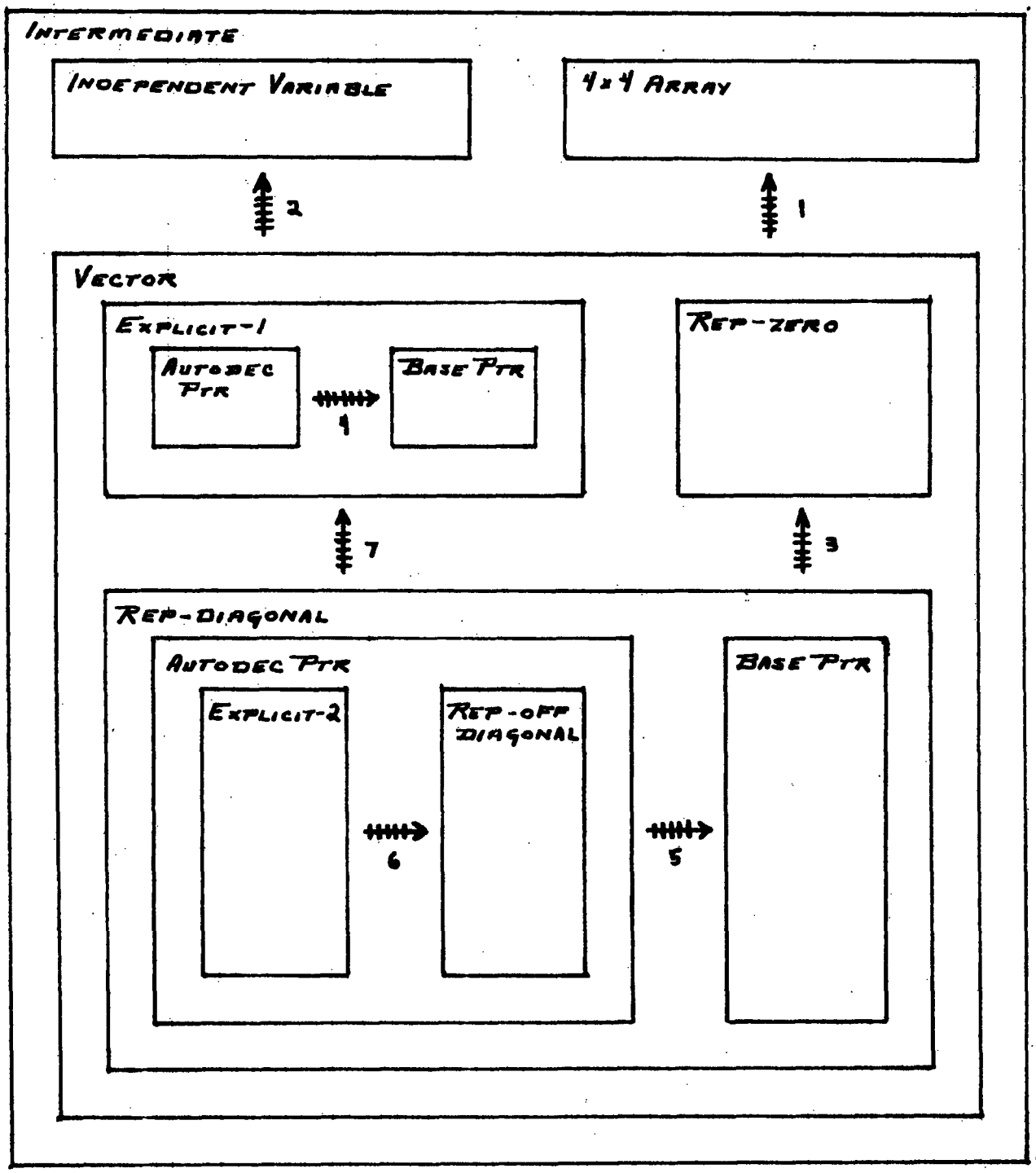

Figure V.7 : The target implementation hierarchy

Stage 4 takes the target implementation plan and performs object code assembly and relocation. The result is an executable file image. The symbolic assembly code which would be generated for this example is shown in Figure V.8. Note that the code produced by Cobbler is the same as the code produced by the assembly language programmer.

There are two things to note in comparing Cobbler's code with the code produced by a conventional optimizing compiler (see Figure V.9). First, the structure of the implementation does not reflect a two dimensional array. Cobbler selected the vector representation over the array representation based on the implementation intermediate plan and general implementation knowledge about vectors and arrays. This choice avoided all of the problems with the address computations and the need to set each diagonal element twice. Second, the structure of the implementation evolved from considering the resource tradeoffs between different implementation options. Cobbler does not transform one implementation into an equivalent but improved one as a conventional optimizing compiler does.

```
line    label   instruction          comments

  1             MOV   #A+17,R3   ;R3 points to last element of A
  2             MOVB  #1,@R3     ;set diagonal element A[4,4] to 1
  3             MOV   #3,R0      ;no. nondiagonal element sequences
  4     L1:     CLRB  -(R3)      ;zero elements between the last
  5             CLRB  -(R3)      ;  diagonal element A[I+1,I+1] and
  6             CLRB  -(R3)      ;  the next diagonal element A[I,I]
  7             CLRB  -(R3)      ;
  8             MOVB  #1,-(R3)   ;set the diagonal element A[I,I] to 1
  9             DEC   R0         ;loop until all nondiagonal element
 10             BGT   L1         ;  sequences have been zeroed
```

Figure V.8 : Code produced by Cobbler

```
line    label   instruction         comments

  1             MOV   #4,R0         ;init I, outer loop index in R0
  2     L1:     MOV   #4,R1         ;init J, inner loop index in R1
  3             MOV   R0,R2
  4             ASL   R2
  5             ASL   R2
  6             ADD   #A-5,R2       ;form address (A+(I-1)*4-1) in R2
  7     L2:     MOV   R2,R3
  8             ADD   R1,R3         ;form address (A+(I-1)*4-1+J) in R3
  9             CLRB  @R3           ;zero the element A[I,J]
 10             DEC   R1            ;decrement the inner loop index
 11             BGT   L2            ;loop until this row is zeroed
 12             ADD   R0,R2         ;form address (A+(I-1)*4-1+I) in R2
 13             MOVB  #1,@R2        ;set diagonal element A[I,I] to 1
 14             DEC   R0            ;decrement the outer loop index
 15             BGT   L1            ;loop until all rows initialized
```

Figure V.9 : Code produced by conventional optimizing techniques

VI. Comparison with Conventional Optimization Techniques

A. A KBRI compiler works in terms of a uniform representation language.

A KBRI compiler replaces a conventional optimizing compiler's collection of distinct intermediate languages with PIL, a uniform intermediate language based on plans. Phase ordering problems are eliminated because a KBRI compiler can consider applicable implementation options at any time. Parallel elaboration, the implementation options' desirability criteria, and the programmer's source implementation choices are used to guide the selection of implementation options which interfere with each other. As an additional benefit, PIL provides flexible support for many combinations of source and target languages. Therefore the same analysis and manipulation routines can be used in many different compilers.

B. A KBRI compiler is independent of the semantics of the source language's nonprimitive operations.

The meaning of each source language construction is expressed in terms of PIL rather than by special programming within the compiler. This allows the compiler to derive the meaning of these operations from its standard analysis. Further the compiler can tailor the implementation of the source language constructions to the properties of the contexts where they are used. Restrictions designed to preserve the machine independence of particular constructions can be eliminated. In addition Cobbler can adapt the library routines from any language if they are expressed in terms of PIL.

C. The modularity of the implementation knowledge facilitates the maintenance of a KBRI compiler.

Each item of a KBRI compiler's implementation knowledge is divided into three parts: the applicability criteria for the item, the desirability criteria for the item, and the specification of how to modify the implementation plans. The CRM decides between the applicable implementation options. The modularity of this decomposition permits individual items of implementation knowledge to be introduced or deleted without a major revision of the compiler. A KBRI compiler can evolve incrementally as the language changes. In addition the modularity of the implementation knowledge avoids the inefficiency and difficulties of deriving special case instruction sequences from general algorithms.

D. A KBRI compiler permits flexible control of the type and degree of improvement.

A KBRI compiler can implement varying degrees of code improvement by controlling the knowledge of source and target language coding techniques that is available. For example an unoptimized, "line by line" translation will be produced if the implementation knowledge is limited to definitions of each source language statement in terms of the target language. Reimplementation can be limited to particular circumstances by only providing the knowledge relevant to those circumstances.

E. Compilation cost can be reduced by exploiting prior analysis.

The reimplementation strategy is more expensive than conventional optimization strategies because more extensive analysis is required. The compilation costs over the life cycle of a program can be reduced in three ways. First, the source implementation decisions for a program module do not change unless the program module is changed. Saving the source implementation decisions with the definition of each module avoids the cost of reapplying stages 1 and 2 to the unchanged program modules. Second, the target implementation options for a program module do not change unless the program module is changed. Saving the expanded target implementation options avoids the cost of rederiving those options in stage 3. Lastly, incremental changes in a program will leave a large part of the source and target implementation analyses unchanged. A KBRI compiler can use the previous analyses as defaults and simply propagate the attendant differences through the PIL descriptions. This technique can substantially reduce the cost of compiling the altered program.

F. A KBRI compiler encourages the programmer to use good programming style.

A KBRI compiler allows the programmer to write modular programs without incurring the inefficiency associated with module interconnections or the use of heavily parameterized, overly general procedures. Abstract specification techniques may be used while retaining the ability to exploit the underlying representation of the objects. Further, the quality of the generated code will improve if the source program is written in a logically clear manner which the compiler can analyze and understand in detail. These abilities should lead to improved programming languages. These languages will emphasize the programmer's ability to evolve constructions adapted to solving particular problems, rather than a general purpose set of operations which are easy to compile for a wide range of machines.

G. The reimplementation strategy allows the programmer to participate in the code generation phase.

The organization of a conventional compiler makes it difficult to involve the programmer in the code generation process. First, the intermediate representation does not represent the current state of the implementation in terms of familiar concepts. Second the reasons for particular decisions are not manifest, but are procedurally embedded in the compiler.

A KBRI compiler represents the state of the implementation in terms of concepts which are useful to the programmer. It is also able to relate each target implementation segment to the corresponding source implementation segment. A conventional compiler cannot do this because the representation of the source implementation is destructively transformed into the target implementation. The reasons for selecting an implementation option are expressed explicitly by the option's desirability criteria and the resource tradeoffs relating different options. Therefore, a KBRI compiler can involve the programmer in the code generation process.

As an additional benefit a KBRI compiler can be integrated with a Programmer's Apprentice. A programmer's apprentice is a system which can support the programmer at each stage of program development. This integration is important because the choices that a programmer makes in terms of algorithm selection and operations structure can drastically affect the efficiency of the implementation. Improvements to target implementation cannot overcome the inefficiency of poor high level choices. Integrated program support systems like the programmer's apprentice are being studied by several research groups. [10,19,25,29,30,35,41,44]

VI. Relationship to Other Work

This research seeks to apply recent advances in program understanding to problems in the area of code generation and optimization. The result is a system which shares characteristics with both automatic programming systems and conventional optimizing compilers.

Cobbler's approach to program understanding stems from the research on a Programmer's Apprentice (PA) being done by Rich, Shrobe, and Waters [29,30,35,41]. The PA will be an integrated program development system which supports the programmer during each stage of the programming process. Program understanding is viewed as a process of describing the program in terms of its data flow, its control flow, its segmentation, and the purposes of its components.

PIL is a synthesis of the work on program description done for the Programmer's Apprentice [29,30,35,41]. Waters work on plan types and describing the temporal properties of loops is directly reflected within PIL. Using symbolic descriptions to characterize the parts of a data structure and their relationship to each other appears in Rich and Shrobe [30], Rich [29], and independently in Leverett [23]. The notion of overlays and partial matching to describe the relationship between a program description and its implementation is discussed by Rich [29]. Cobbler uses overlays to describe the implementation relationships at each level in the intermediate representation. This allows Cobbler to maintain descriptions of the program at each level of implementation.

Cobbler's reasoning component is related to recent work done on procedural reasoning systems by Doyle [15], by Kornfeld [22], and by McAllester [27]. Each system is similar in that knowledge is represented in terms of simple procedural entities. The systems differ in the way that the entities are controlled and the data base primitives that the system provides. Using parallelism and resource allocation to explore decisions that cannot be reliably distinguished has been discussed in Kornfeld [22] and Hewitt [18]. Shrobe [35] discusses how to use Doyle's Truth Maintenance System (TMS) to reason about program descriptions. Cobbler uses these techniques to avoid recomputing implementation decisions that are common to different implementations.

An automatic programming system is a system which can develop an efficient, correct implementation of a program given the program's specifications in some convenient form. The PSI automatic programming system developed by Barstow [9] will serve to illustrate the similarities and differences between an automatic programming system and a KBRI compiler. PSI accepts program specifications written in a very high level language. PSI's knowledge of programming is expressed in the form of rewrite rules. For PSI programming is a search process employing the rewrite rules to refine the program's very high level language implementation into a lower level implementation.

The compiler's implementation intermediate plans (II plans) are equivalent to PSI's program specifications. PSI must make all implementation decisions below the level of the algorithmic choices

embodied in the specification. These implementation decisions will involve decisions about the application domain of the program as well as decisions about the language implementation techniques. Consequently, PSI must possess extensive knowledge of the program's application domain. A KBRI compiler only has to make decisions about target language implementation techniques. It draws on the source implementation decisions embodied in the source implementation for knowledge about the application domain. In short, a KBRI compiler must solve a simplified form of many of the research problems facing automatic programming research.

The Experimental Compiling System (ECS) project [4,17] proposes a highly modularized, systematic approach to generating high quality code. The ECS model divides compilation into three major stages. First, the source program is translated into a uniform intermediate language (IL). IL expresses the semantics of the source language operations in terms of procedures. Second, a sequence of flow analysis, machine independent optimization, and procedure integration is applied to the program. This sequence is repeated until the program is expressed in terms of operations near the level of the target language operations. Lastly, the resulting IL program is transformed into a target language oriented version of IL followed by register allocation and object code assembly.

The ECS and KBRI compiler strategies differ widely. The ECS strategy uses program transformations to improve the quality of the generated code. However the ECS approach depends on a few powerful, machine independent optimization techniques to derive the optimizations that a conventional optimizing compiler discovers through more specialized methods. In contrast the KBRI approach can directly employ any implementation technique if its resource requirements, applicability criteria, and desirability criteria can be adequately expressed. The program representations also reflect the differences between the ECS and KBRI compiler strategies. IL is a linear representation like a conventional programming language. Each compilation step refines the program by rewriting part of the representation. PIL is a network structure incorporating several different kinds of program description in addition to defining the program's operations. In particular note that PIL separates the descriptions that are dependent on either the source or target language into two distinct hierarchies. An ECS compiler works from a high level description based on the source language. The final stage of compilation translates the low level program description into a description oriented towards the target language.

Bibliography

[1] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, 1972.

[2] F. E. Allen, *Annotated Bibliography of Selected Papers on Program Optimization*, Technical Report RC 5889, IBM Thomas J. Watson Research Center, March 1976.

[3] F. E. Allen and J. Cocke, A Catalogue of Optimizing Transformations, In R. Rustin ed., *Design and Optimization of Compilers (Courant Computer Symposium 5)*, pp. 1-30, Prentice-Hall, 1972.

[4] F. E. Allen et al., *The Experimental Compiling Systems Project*, Technical Report RC 6718, IBM Thomas J. Watson Research Center, September 1977.

[5] F. E. Allen, Interprocedural Data Flow Analysis, in *Proceedings of the IFIP Congress 74*, Amsterdam, The Netherlands: North-Holland, 1974, pp. 398-402.

[6] F. E. Allen and J. Cocke, A Program Data Flow Analysis Procedure, *Communications of the ACM*, vol. 19, no. 3, pp. 137-47.

[7] H. Baker, *Actor Systems for Real-time Computation*, MIT/LCS/TR-197, MIT Cambridge MA, March 1978.

[8] M. Barbacci and D. Siewiorek, *Some Aspects of the Symbolic Manipulation of Computer Descriptions*, Technical Report, Carnegie-Mellon University, 1974.

[9] D. Barstow, *Automatic Construction of Algorithms and Data Structures*, PhD Thesis, Stanford University, September 1977.

[10] F. L. Bauer, etal., *Notes on the Project CIP: Outline of a Transformation System*, TUM-INFO-7729, Technische Universitaet Muenchen, July 1977.

[11] J. Bruno and R. Sethi, Code Generation for a One-Register Machine, *Journal of the ACM*, vol. 23, no. 3, pp. 502-10.

[12] V. A. Busam and D. E. Englund, Optimization of Expressions in Fortran, *Communications of the ACM*, vol. 12, no. 12, pp. 666-74.

[13] J. L. Carter, A Case Study of a New Code Generation Technique for Compilers, *Communications of the ACM*, vol. 20, no. 12, pp. 914-20.

[14] Digital Equipment Corporation, *PDP11/34 Processor Handbook*, Digital Equipment Corporation, 1976.

[15] J. Doyle, *Truth Maintenance Systems for Problem Solving*, MIT/AI/TR-419, MIT Cambridge MA, January 1978.

[16] W. Harrison, Compiler Analysis of the Value Ranges for Variables, *IEEE Transactions on Software Engineering*, vol. SE-3, no. 3, pp. 243-50.

[17] W. Harrison, A New Strategy for Code Generation - The General Purpose Optimizing Compiler, *IEEE Transactions on Software Engineering*, vol. SE-5, no. 4, pp. 367-73.

[18] C. Hewitt, *Viewing Control Structures as Patterns of Passing Messages*, MIT/AI/WP-92, MIT Cambridge MA, December 1975.

[19] C. Hewitt, and B. Smith, Towards a Programming Apprentice, *IEEE Transactions on Software Engineering*, vol. SE-1, no. 1, pp. 26-45.

[20] K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, 1975.

[21] W. H. Joyner Jr., W. C. Carter, and D. Brand, *Using Machine Descriptions in Program Verification*, Technial Report RC 6922, IBM T. J. Watson Research Center, 1978.

[22] W. Kornfeld, *Using Parallel Processing for Problem Solving*, MIT/AI/TR-561, MIT Cambridge MA, 1979.

[23] B. W. Leverett et al., *An Overview of the Production Quality Compiler-Compiler Project*, Technical Report CS-79-105, Carnegie-Mellon University.

[24] D. Lomat, Data Flow Analysis in the Presence of Procedure Calls, Technical Report RC 5728, IBM T. J. Watson Research Center, November 1975.

[25] D. B. Loveman, Program Improvement by Source to Source Transformation, *Journal of the ACM*, vol. 24, no. 1, pp. 121-45.

[26] E. S. Lowry and C. W. Medlock, Object Code Optimization, *Communications of the ACM*, vol. 12, no. 1, pp. 13-22.

[27] D. A. McAllester, *The Use of Equality in Deduction and Knowledge Representation*, MIT/AI/TR-550, MIT Cambridge MA, January 1980.

[28] W. McKeeman, Peephole Optimization, *Communications of the ACM*, vol 8. no. 7, pp. 443-44.

[29] C. Rich, *A Library of Plans with Applications to Automated Analysis, Synthesis, and Verification of Programs*, PhD thesis, Massachusetts Institute of Technology, January 1980.

[30] C. Rich, and H. Shrobe, *Initial Report on a LISP Programmer's Apprentice*, MIT/AI/TR-354, MIT Cambridge MA, December 1976.

[31] B. K. Rosen, Data Flow Analysis for Procedural Languages, Technical Report RC 5948, IBM T. J. Watson Research Center, April 1976.

[32] B. K. Rosen, Data Flow Analysis for Recursive PL/I Programs, Technical Report RC 5211, IBM T. J. Watson Research Center, January 1975.

[33] M. Schaefer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, 1973.

[34] B. R. Schatz, *Algorithms for Optimizing Transformations in a General Purpose Optimizing Compiler: Propagation and Renaming*, Technical Report RC 6232, IBM Thomas J. Watson Research Center, October 1976.

[35] H. E. Shrobe, *Dependency Directed Reasoning for Complex Program Understanding*, MIT/AI/TR-503, MIT Cambridge MA, April 1979.

[36] T. A. Standish et al., *The Irvine Program Transformation Catalog*, Technical Report, University of California at Irvine, Department of Information and Computer Science, 1976.

[37] T. B. Steel, A First Version of UNCOL, in *Proceedings of the Western Joint Computer Conference*, pp. 371-77.

[38] T. G. Szymanski, Assembling Code for Machines with Span-Dependent Instructions, *Communications of the ACM*, vol. 21, no. 4, pp. 300-8.

[39] G. Urschler, Complete Redundant Expression Elimination in Flow Diagrams, Technical Report RC 4965, IBM T. J. Watson Research Center, April 1974.

[40] W. Waite, Optimization, in F. Bauer and J. Eickel ed., *Compiler Construction - An Advanced Course*, Springer-Verlag, 1976.

[41] R. C. Waters, *Automatic Analysis of the Logical Structure of Programs*, MIT/AI/TR-492, MIT Cambridge MA, December 1978.

[42] B. A. Wichmann, How to Call Procedures, or Second Thoughts on Ackermann's Function, *Software: Practice and Experience*, vol. 7, pp. 317-29.

[43] J. D. Wick, *Automatic Generation of Assemblers*, PhD thesis, Yale University, December 1975.

[44] T. Winograd, Breaking the Complexity Barrier (Again), *ACM SIGIR-SIGPLAN Interface Meeting*, November 1973.

[45] W. A. Wulf, D. B. Russell, and A. N. Habermann, BLISS: a Language for Systems Programming, *Communications of the ACM*, vol. 14, no. 12, pp. 780-90.

[46] W. Wulf et al., *The Design of an Optimizing Compiler*, American Elsevier, 1975.

[47] S. N. Zilles and B. H. Liskov, Specification Techniques for Data Abstractions, *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 7-18.