The reduction of data dependencies in high level programs

Stephen J. Allan
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
Allan, Stephen J., "The reduction of data dependencies in high level programs" (1979). Retrospective Theses and Dissertations. 7185.
https://lib.dr.iastate.edu/rtd/7185
INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
ALLAN, STEPHEN JOHN
THE REDUCTION OF DATA DEPENDENCIES IN HIGH
LEVEL PROGRAMS.

IOWA STATE UNIVERSITY, PH.D., 1979
The reduction of data dependencies in high level programs

by

Stephen J. Allan

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of The Requirements for the Degree of DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1979
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>INTRODUCTION</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER I</td>
<td>Outline of Thesis</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER II</td>
<td>HIGH LEVEL LANGUAGE</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Comments on Language</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Synopsis of High Level Language</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Internal Form Table</td>
<td>14</td>
</tr>
<tr>
<td>CHAPTER III</td>
<td>FLOW ANALYSIS</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Sequential Control Flow Analysis</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Data Flow Analysis</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Collection of input and output sets</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Generating the use and definition sets</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Live value analysis</td>
<td>47</td>
</tr>
<tr>
<td>CHAPTER IV</td>
<td>REDUCING DATA DEPENDENCIES</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>Transformations</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>Constant folding</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Scalar propagation</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Strength reduction</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>Common subexpression elimination</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>Invariant code motion</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>Induction value removal</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>Loop decomposition</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Loop fusion</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Loop unrolling</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>Forward substitution</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>Tree balancing</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>Orderings of the Transformations</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>Partial ordering with respect to execution time</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>Partial ordering with respect to redundant computation</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>Partial ordering with respect to instruction space</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>Inclusion of Partial Orderings in a Compiler</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>Language Extensions for the Exposure of Parallelism</td>
<td>152</td>
</tr>
</tbody>
</table>

| CHAPTER V | CONCLUSION | 157 |
REFERENCES 160

ACKNOWLEDGEMENTS 165

APPENDIX A. GRAMMAR FOR SEQUENTIAL HIGH LEVEL LANGUAGE 166

APPENDIX B. ALGORITHMS FOR TRANSFORMATIONS 169
  Strength Reduction 169
  Induction Value Removal 176
  Loop Decomposition 180
CHAPTER I. INTRODUCTION

There is an ever increasing demand for higher performance computing systems. Some users of systems would like to solve problems which cannot be adequately simulated on conventional systems or to obtain a better solution to existing problems by increasing the amount of data handled. Existing sequential machines do not have the power necessary to solve these problems. In the past, the needed increase in computer system performance has been obtained through advances in hardware technology. But the point is being reached where physical contraints limit the improvement in performance which can be obtained through increased hardware speed along. In spite of these physical contraints there have been improvements in the performance of computing systems by the introduction of parallelism into the hardware and also exploiting the parallelism which naturally exists in problems. The parallelism in problems may be exploited by executing a number of distinct programs on different processors or utilizing the structure of the existing programs.

There are, in general, two different types of parallel processors. First, there are those which are von Neumann in nature with a centralized control. These may be classified as control-flow computers. They can be contrasted with data-flow computers which are non von Neumann in nature because there is no central control.

Control-flow computers are driven by one or more sequential instruction streams and require that any parallel activity in the
computations be explicitly indicated by the program. There are four different classes of this type of parallel machine (Treleaven 1979). They are:

a. Look-ahead (Keller 1975) - the parallelism in this type of machine is achieved by overlapping the various decision processes that make up the execution of instructions. According to Flynn (1972), this type of machine may be classified as having a single instruction stream and a single data stream. Examples of machines of this nature are the CDC 6600 and IBM 360/91.

b. Vector (Ramamoorthy 1977) - the resources in this type of machine are organized into a pipeline. Vector instructions apply a single pipelined operation to sets of vector operands. This type of machine may be classified as having a single instruction stream and a single data stream. Examples of this type of machine are the CDC Star 100, the Cray-1 and the TI ASC.

c. Array (Kuck 1977) - in this type of machine, a single control unit is used to drive a number of synchronous processing units which perform the same operation simultaneously on a number of data streams. This type of machine may be classified as having a single instruction stream and a multiple data stream. An example of this type of machine is the ILLIAC IV.

d. Multiprocessor (Enslow 1977) - in this type of machine, there exists a number of asynchronous processing units each of which can be performing a different calculation. This type of machine
may be classified as having a multiple instruction stream and a multiple data stream. An example of this type of machine is the C.mmp.

On the other hand, data-flow computers are not driven by a sequential instruction stream. These machines execute concurrent operations according to the data dependencies which exist in the computation. No explicit indication of those operations which may execute in parallel need be given by the computation. Operations are allowed to execute as soon as their input data is available. There have been at least two prototypes constructed (Davis (1978) and Texas Instruments\(^1\) 1979) while several others have been proposed or are under construction (Arvind and Gostelow 1976, Dennis 1974, Dennis and Misunas 1975, Plas et al. 1976, Rumbaugh 1977, Watson and Gurd 1979).

The level of parallelism which can be exploited varies greatly with the architecture. It varies from supporting parallelism at the procedure level to the statement level to the operation level. It is ultimately desirable, however, for an architecture to support the parallelism available at all levels.

The same problem, the level of parallelism supported, arises when considering a high level language which will be able to utilize the resources of the underlying architecture. There have been proposed a number of mechanisms which would allow the programmer the opportunity to express parallelism (Conway 1963, Dijkstra 1968b, Brinch Hansen 1975).

\(^1\)Texas Instruments, Dallas, Texas.
These mechanisms may be a natural extension to the sequential high level language, but there is a limit to the level of parallelism they can show.

How, then, is the parallelism detected? The compiler can be given the task of examining a program written in some sequential high level language and analyzing the parallelism which exists. This approach has been used by others (Millstein and Muntz 1975, Presberg and Johnson 1975, Wedel 1975). In addition, the compiler may also examine the program to see if the algorithm may be restructured in order to enhance the parallelism which already exists in the program. Methods for restructuring loops have also been studied by others (Cohagan 1973, Kuck 1975, Lamport 1973, Lamport 1974, Lo 1976, Muroaka 1971, Ramamoorthy and Gonzalez 1969, Schneck 1972, Schneck 1975).

Ultimately, the compiler needs to detect the parallelism which exists in a program and attempt to enhance this parallelism. But for the compiler to be more effective, it must receive information about the parallelism which exists from the programmer through the programming language. Consider the process of solving a problem as depicted by Figure 1.1 (Lamport 1975).

![Figure 1.1. Process of problem solving](image-url)
Given the statement of a problem, an algorithm is specified after which a program is written in some programming language which expresses the problem. This program is then given to a compiler which translates the program into some intermediate form, optimizes the intermediate form if necessary and generates the code for the proper machine. But in Figure 1.1, as one moves from left to right, each step represents a loss of information about the problem as the solution to the program is defined. This is because there may be many solutions to a given problem, but just one solution is finally chosen from which the ultimate algorithm is written. Once again there are many different ways the program may be written in the chosen programming language, but only one is chosen. By the time the compiler receives the program, information has been lost in choosing a given solution and in writing a particular program for that solution. Information which might have been useful to the compiler and might have been readily available in the original statement of the problem may have been lost. For example, the information as to whether given computations may be executed in parallel (independent iterations of a loop) may have been obvious in the statement of the problem. If the programming language offers no facilities which allow the programmer to pass this information along to the compiler, this information about parallel execution may be lost. The compiler can make an attempt to recover this information about the parallelism which previously existed, but the chances of doing this are not as great as if it had been explicitly indicated in the program. This does not mean that the programmer should have to specify parallelism at the operation level since the
compiler can detect parallelism at that level. But he should be able to indicate parallelism which exists between iterations of a loop stating if they are independent of one another. This allows an appropriately constructed compiler to exploit this parallelism in its object code. This of course depends upon the architecture of the host machine. Nothing at all is lost if the architecture cannot exploit this type parallelism because the parallel loop may be executed as a sequential loop.

This all comes down to the following: (a) There is much parallelism which exists in a program that can be detected by the compiler; (b) There is also much the compiler can do to enhance the parallelism that is detected. But even more could be done to enhance the parallelism in the program if the programmer could explicitly indicate the parallelism.

This research will address a two part problem. First, methods which permit the programmer to explicitly indicate the existence of parallelism in a high level program. Second, given a high level program, compiler techniques which detect the parallelism and data dependencies in the program and reduces the number of these data dependencies. Both of these problems are discussed in the chapters which follow with a heavy emphasis on the latter.

A sequential von Neumann type high level language is chosen in this work since this type of language is currently in use on most machines. Although procedure oriented on the surface, the language is equivalent to applicative forms since global references and explicit transfer of control are not allowed. The language is sequential in that the use of
an identifier in right context must be preceded by an assignment of a value to the identifier. The idea of single assignment is not implemented (Ackerman and Dennis 1978, Arvind et al. 1976, Chamberlin 1971, Plas et al. 1976, Tesler and Enea 1968). Extensions to a von Neumann language have been studied by others (Basili 1975, Brinch Hansen 1975, Wedel 1975, Zosel 1975, Zwakenberg 1975). This work will present similar extensions and discusses their usefulness.

In order to detect parallelism, data dependencies, and perform optimizations on the program, a complete data flow analysis needs to be performed. The results of this data flow analysis are needed for at least two different purposes. First, they are needed to give sufficient information to be able to perform optimizing transformations. Second, results are needed to provide sufficient information for generating machine code for most types of parallel computers including a data flow machine. A data flow analysis technique which meets these two requirements is presented in this work.

A "typical" set of optimizing transformations are also introduced and analyzed. Most transformations fall into one of four categories:

1. reduction of data dependencies in the program thereby decreasing the execution time on a suitably parallel machine.
2. analysis of the iterative constructs used in the program with attempts to restructure the program so that parallel constructs may be used.
3. increase of the size of the body of a loop which has to be executed sequentially. Whenever the number of instructions in the body of a loop is small, the result is that there is
very little parallelism available. Transformations of this type expand the number of instructions inside the body of the loop and create potentially independent operations.

4. reduction of the number of resources used by the program.

The second category given above is the most important since the greatest reduction in the execution time of a program happens when this is possible. An existing transformation which performs this restructuring is extended in this work to allow a broader base of application.

These transformations are further studied and a partial ordering is derived which reduces the execution time of a program if the transformations are applied according to the partial ordering. Previous work has indicated the order in which some of the transformations should be applied (Allen 1969, Loveman and Faneuf 1975, Ottenstein 1978), but few results are available. The transformations are further ordered according to two other criteria: the amount of redundant computation introduced in the program, and the instruction space used by the program. Finally, the manner in which these partial orderings would be incorporated into an optimizing compiler is specified.

Outline of Thesis

Chapter II introduces the sequential high level language which is used throughout the dissertation. Chapter III presents the data flow analysis technique which is used in implementing the algorithms in Chapter IV and also could be used to generate machine code. Chapter IV
presents the transformations along with their algorithms, their partial orderings and the extensions to the high level language. Chapter V contains the conclusions.
CHAPTER II. HIGH LEVEL LANGUAGE

Introduction

This chapter introduces the high level language used in this research and the intermediate form of the language produced by the compiler. The general nature of the high level language is discussed along with those features which were included in the language and other features which were not included. The internal form table (IFT), which is the intermediate form of the program, is then introduced and the relationship between the high level language and the IFT is shown.

Comments on Language

A high level language has been designed as a basis for the work described in Chapter I. The techniques for reducing data dependencies in high level programs which are important in a parallel machine environment have been demonstrated using this language. A language such as Pascal could have been chosen, but the language needs to be devoid of those features which are incompatible with the notion of functionality of programs. The language looks much like Pascal and has about the same expressive power as Algol 60 but global references and goto's are prohibited. This language allows a thorough control flow analysis and data flow analysis without the complications that arise from the use of goto's and global references. The elimination of arbitrary transfers of control and side effects from procedures is not regarded as a limitation since
such features are considered by many to be among the most harmful of programming features (Dijkstra 1968a, Wirth 1974).

Programming languages will continue to evolve, and in the evolution they will retain many of their present features such as support of modular program construction and data structuring facilities. The concept of variables, block structuring and variations of standard structured control constructs were included in this idealized language because they will probably continue as long as von Neumann style computers exist.

Global references over procedure boundaries and goto's will be deemphasized in the future regardless of the host computer. This is because global references allow undesirable side effects and goto's destroy the readability of the programs and complicate unnecessarily their proof of correctness. These features have been excluded from this language.

Single assignment languages have been proposed (Ackerman and Dennis 1978, Arvind et al. 1976, Chamberlin 1971, Plas et al. 1976, Tesler and Enea 1968) which would further simplify the data flow analysis. This is because, simplistically stated, an identifier is only allowed to appear once in left hand context and thus finding where a value was produced for a given use would be relatively easy. The language was not restricted in this manner at this time.

Backus (1978) has said that programming languages of today are "fat and weak" and all are complex models of the von Neumann type of machine they run on. He advocates that future languages be non
von Neumann in nature and more in the nature of LISP or in the nature of his functional programming system. The high level language adopted for the study is restricted to the commonly used sequential form. Although procedure oriented on the surface, the language may be regarded as applicative since global references and explicit transfer of control are not allowed. Sequential means that the use of an identifier in right context must be preceded by an assignment of a value to the identifier. A language of this nature was developed since the majority of the languages in current use are of this type and the desire to study the translation and optimization of these sequential high level languages to highly parallel data flow languages (Arvind et al. 1976, Dennis 1974, Dennis and Misunas 1975, Plas et al. 1976). The language reported on here is not proposed as a new language, but rather it is used as a vehicle to help achieve the objectives of the research (Oldehoeft et al. 1978). It is similar to Algol 60 as defined by Naur (1963).

Synopsis of High Level Language

The high level language is a structured, procedure oriented programming language. A program consists of a main procedure with declarations, including the definition of other procedures and functions, and a body of statements.

Integer, real and boolean data types are currently supported along with a full complement of operators and intrinsic functions which can operate on identifiers declared with the above data types. An identifier which is declared boolean can only be true or false. Mixed mode arithmetic is allowed only with identifiers declared to be real or integer.
The only data structuring facility which exists at the present time is arrays. Arrays may have any number of dimensions, consist of scalars of one of the above declared homogenous types, and have lower and upper bounds of each dimension declared by the programmer. Arrays may be dynamically declared at run time upon procedure entry.

Procedures and functions may be declared in any procedure and may be called recursively. There are no entry points to procedures other than at the procedure name. Abnormal exits from procedures and functions are not allowed in that all exits must be through the end statement of the procedure or function. All parameters to procedures, both formal and actual, must carry a directionality attribute specifying whether they are input (referenced before being defined inside the procedure or function) or output (defined inside the procedure). A function can only have parameters with the input directionality attribute and a single value is returned by the function by assigning a value to the name of the function inside the body of the function. There are no global references possible in procedures or functions.

Statements in the language include assignment, conditional (i.e., if-then and if-then-else), iterative (i.e., while-do and repeat-until), procedure call and I/O (i.e., input and output).

The grammar for the complete language appears in Appendix A.

Some features which were not included, but which could be added are block structure, strings and string operators, procedure variables, case statement and generalized data structures.
Internal Form Table

The language compiler translates the source text of a program written in the high level language into an intermediate form table, which is a table of relatively high level entries and is hereafter referred to as the IFT. The IFT is used in later phases to perform the data flow analysis and the application of the optimizing transformations.

Each entry in the IFT consists of four fields as shown in Figure 2.1.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>I</th>
<th>O</th>
<th>TREE</th>
</tr>
</thead>
</table>

Figure 2.1. Entry in the IFT

Each field is briefly described below:

1. **TYPE** is the field that indicates the type of statement represented by this entry in the IFT. The different types are: procedure, function, end, input, output, assignment, if, conditional, then, else, while, repeat, call and close. Close is used to indicate the end of a repeat, while of if construct.

2. **I** is the input set for this entry and is referred to by $I(E_i)$ and means the input set for entry $E_i$. The calculation of this field for the different types of entries appears in Chapter III.

3. **O** is the output set for this entry and is referred to by $O(E_i)$ and means the output set for entry $E_i$. The calculation of this field for the different entry types appears in Chapter III.
4. TREE is the field where a syntax tree for the statement is placed if one exists.

The entries in the IFT are threaded to represent the ordering of the statements as they are encountered in a sequential scan of the high level program.

The general form of the different high level statements and the types of entries in the IFT generated by the compiler are given in Figure 2.2. A separate IFT is generated for every procedure and function defined in the program. If an input or output statement involves an implied do loop, a while loop is generated and placed in the IFT with the body of the while being the input or output statement and the implied increment.
<table>
<thead>
<tr>
<th>Type of statement</th>
<th>High level statement</th>
<th>Entries in the IFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure or function declaration</td>
<td>procedure (function)</td>
<td>procedure (function)</td>
</tr>
<tr>
<td></td>
<td>statement list</td>
<td>entries for statement list</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>input/output statement</td>
<td>input/output $a_1, \ldots, a_n$</td>
<td>input (output) for $a_1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\cdot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>input (output) for $a_n$</td>
</tr>
<tr>
<td>assignment</td>
<td>$x :=$ expression</td>
<td>assign</td>
</tr>
<tr>
<td>if statement</td>
<td>if condition</td>
<td>if condition</td>
</tr>
<tr>
<td></td>
<td>then statement list</td>
<td>then</td>
</tr>
<tr>
<td></td>
<td>{else statement list2}</td>
<td>{else</td>
</tr>
<tr>
<td></td>
<td>close</td>
<td>}</td>
</tr>
<tr>
<td>while statement</td>
<td>while condition</td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>statement list</td>
<td>condition</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>entries for statement list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>entries for statement list2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>close</td>
</tr>
</tbody>
</table>

Figure 2.2. High level statements and entries in the IFT
<table>
<thead>
<tr>
<th>repeat statement</th>
<th>repeat statement list</th>
<th>repeat entries for statement list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>statement list</td>
<td>condition</td>
</tr>
<tr>
<td></td>
<td>until</td>
<td>close</td>
</tr>
<tr>
<td>procedure call</td>
<td>x(in(...), out(...))</td>
<td>call</td>
</tr>
</tbody>
</table>

Figure 2.2. Continued
CHAPTER III. FLOW ANALYSIS

Introduction

This chapter describes a procedure for flow analysis which is used in performing optimizing transformations and generating code for highly parallel computers. The technique is useful for a broad class of high level languages which include the sequential von Neumann type high level languages in common use as well as nonsequential high level languages such as single assignment languages. In order for this flow analysis technique to be applicable to single assignment languages, it is required that the definition of a value precede any use of that value in the text of the high level program. In some single assignment languages this is required by the definition of the language while others would require a preprocessor to topologically order the statements. The use of this flow analysis is performing optimizing transformations is detailed in Chapter IV.

The usefulness in generating code for highly parallel computers depends on the level of concurrency supported by the computer. For example, consider data driven machines (Arvind and Gostelow 1976, Dennis and Misunas 1975, Rumbaugh 1977, Weng 1975). The assumption underlying a data driven machine is that a program is not a sequence of instructions that cause changes to a memory space, but instead a program is a collection of computations related to each other by the need for data values that are produced and consumed. The order of execution of the computations is not directly stated by the program but rather by the partial ordering provided by the data dependencies. The purpose of the data
flow analysis is to determine this partial ordering so that code may be
generated to exploit the inherent parallelism in the program. While
other techniques (Aho and Ullman 1977, Allen and Cocke 1976, Hecht
and Ullman 1975, Hecht 1977) provide a general basis for the analysis,
additional information must be gathered to generate code for such a par­
allel execution environment. The data flow analysis technique described
in this chapter gathers this additional information.

The sequential control flow is first discussed and then the data flow
analysis is described.

Sequential Control Flow Analysis

In order to describe the manner in which the data flow information is
gathered, it is necessary to look at the structure of the IFT and analyze
the information contained therein.

Control flow graphs have been extensively studied (Allen 1970,
Hecht 1977) and they are applicable here because the high level language
being analyzed is von Neumann in nature. It will be the purpose here to
present only the relevant parts of their definitions.

The compiler produces a separate IFT for every procedure or function
encountered. A block is defined to be a sequence of entries in the IFT
which has a single entry point and a single exit point and corresponds to
a statement in the high level language. A block consists of a single IFT
entry if TYPE(entry) is assign, condition, call, procedure, function,
input or output. A block consists of more than one IFT entry if TYPE–
(entry) is repeat, while or if. In this case the block consists of the
entry itself (i.e., if, while or repeat), a condition entry, entries for the body of the construct (as they are sequentially written in the high level language) and a close entry. For a given IFT, we may consider its control flow graph to be the 4-tuple $\text{CFG} = (B, \text{entry}, \text{exit}, \text{succ})$ where $B$ is the set of blocks in the body of the procedure, entry is a unique block which has no predecessors, exit is a unique block which has no successors and succ defines a relation on the entry, exit and the blocks. Entry and exit represent the single entry point and single exit point from the block which may be the same entry in the IFT if the block consists of a single IFT entry. Succ is defined as follows: $i \in \text{succ}(j)$ if the sequential control passes from block $j$ to block $i$. Only condition blocks can have more than one (namely two) successor.

Schematically, the control flow graph for a procedure, $\text{CFG} = (B, \text{procedure}, \text{end}, \text{succ})$, can be thought to appear as in Figure 3.1 (Conway and Gries 1975). Note that the control flow graph is just a straight line sequence of blocks since the body of the procedure now consists of a number of blocks.

Each $B_i$ in Figure 3.1 can be thought of as a separate control flow graph having one of the following four forms.

1. If $B_i$ represents an assignment, input, output, condition or call entry, $B_i$ consists of just the IFT entry itself which of course has a single entry point and a single exit point. The control flow graph is $\text{CFG} = (\emptyset, B_i, B_i, \emptyset)$ and is schematically shown in Figure 3.2.
CFG = (B, procedure, end, succ)
B = \{B_1, B_2, \ldots, B_n\}
succ are indicated by the arrows

Figure 3.1. CFG for procedure
Figure 3.2. CFG for single entry block

2. If $B_i$ represents a while block, $B_i$ consists of a while entry, condition entry, entries for the body of the while and a close entry. The control flow graph, $\text{CFG} = (\emptyset, \text{while}, \text{close}, \text{succ})$, is schematically shown in Figure 3.3.

3. If $B_i$ represents a repeat block, $B_i$ consists of a repeat entry, entries for the body of the repeat, a condition entry and a close entry. The control flow graph $\text{CFG} = (\emptyset, \text{repeat}, \text{close}, \text{succ})$, is schematically shown in Figure 3.4.

4. If $B_i$ represents an if block, $B_i$ consists of an if entry, a condition entry, a then entry, entries for the body of the then, and else entry, entries for the body of the else (if the else clause if present) and a close entry. The control flow graph, $\text{CFG} = (\emptyset, \text{if}, \text{close}, \text{succ})$, is schematically shown in
CFG = (B,while,close,succ)
B = \{condition,B_1, ..., B_n\}
succ are indicated by the arrows

Figure 3.3. While CFG
CFG = (B, repeat, close, succ)
B = \{B_1, \ldots, B_n, \text{condition}\}
succ are indicated by the arrows

Figure 3.4. Repeat CFG
Figure 3.5. If the else clause is missing, all the blocks on the right hand side would not be present.

Note that in cases 2, 3 and 4, each block that appears in the body of the construct has a single entry point and a single exit point. The whole block therefore has the single entry point single exit point property.

The definition of block is recursive since any block can only be one of the four types mentioned above. Therefore, a control flow graph may be associated not only with the procedure or function itself, but also with each of the blocks represented in that procedure. Hence there is a nesting of control flow graphs and at any level of nesting, the blocks appear as if they are single entries in the IFT even though in reality they may not be.

Figure 3.6 presents a program written in the high level language. Figure 3.7 shows the entries that would be generated and placed in the IFT by the compiler and Figure 3.8 shows the blocks in the procedure. The square boxes represent blocks which are not trivial blocks (a block consisting of just one entry). The control flow graph is given for each of these non-trivial blocks.

For all practical purposes, the IFT and its associated CFG are equivalent. The information in either one is readily available in the other. For this reason, only the IFT is referred to in the chapters that follow.

In the next section, it is necessary to do a backward scan of the IFT in order to generate some data flow information. It is thus necessary to know which blocks dominate (Hecht 1977) another so this information
Figure 3.5. If CFG
procedure bubblesort begin
  amax := 50;
i := 1;
while i <= amax do
  input a(i) file=in;
i := i + 1
end;
top := amax;
repeat
  flag := true;
i := 1;
  while i <= top do
    if a(i) < a(i+1)
      then begin
        flag := false;
temp := a(i);
a(i) := a(i+1);
a(i+1) := temp
      end;
i := i + 1
  end;
top := top - 1
until flag;
i := 1;
while i <= amax do
  output a(i) file=out;
i := i + 1
end
end

Figure 3.6. High level program for bubble sort

0. procedure 17. assign
1. assign 18. assign
2. assign 19. assign
3. while 20. assign
4. condition 21. close
5. input 22. assign
6. assign 23. close
7. close 24. assign
8. assign 25. condition
9. repeat 26. close
10. assign 27. assign
11. assign 28. while
12. while 29. condition
13. condition 30. output
14. if 31. assign
15. condition 32. close
16. then 33. end

Figure 3.7. IFT entries for bubble sort
Figure 3.8. Blocks in bubble sort
can be collected. Block i is said to dominate block j if every path from the entry block to j must pass through i. Looking at Figure 3.1, it can be seen that for a CFG of a procedure, if block i appears in the CFG (or IFT) before block j, block i dominates block j. Looking at Figures 3.3, 3.4 and 3.5, it can be seen that for a block in the body of the construct, block i dominates block j if block i appears in the IFT before block j. The blocks which dominate another block in a CFG graph are apparent and no special algorithm is needed to produce these sets. It should be noted that as the blocks in an IFT are being processed, all the dominators of a given block will have been processed before the given block if the processing is done according to the order of appearance in the IFT. All the information that a block needs from the dominating blocks is therefore known and it is not necessary to iterate in order to propagate the information. As a consequence, the concept of dominance will not be explicitly revisited in future discussion.

Data Flow Analysis

The main reasons for studying the sequential control flow are the analysis of the program's structure (Cocke and Schwartz 1970, Allen 1970) and the gathering of global information about the uses and definitions of values. This section which deals with the gathering of global information is referred to as data flow analysis. After the flow of data has been analyzed, the program no longer needs to be viewed as a sequence of
statements that have to be executed in the given order but may be con-
sidered as a list of statements connected to each other by the need for
data values that are produced and consumed. The program is therefore
parallel within the definition of the conditional control flow constructs
and the dependencies which exist among the data. Efficient code may be
generated from the IFT for either sequential or parallel target languages.

There are basically two general approaches to data flow analysis
that are presented in the literature, the interval approach and the
iterative approach.

The interval approach (Aho and Ullman 1977, Allen 1970, Allen and
Cocke 1976, Cocke and Schwartz 1970, Hecht 1977) partitions the control
flow graph into subgraphs, called intervals, replacing that subgraph with
a single node which contains the local information for that interval. This
process of propagating local information globally by defining such inter-
val partitions continues until the entire graph becomes a single node. The
partitioning process is then reversed in order to propagate the global
information locally.

The iterative approach (Aho and Ullman 1977, Graham and Wegman 1976,
Hecht and Ullman 1975, Hecht 1977) propagates information in a simple
iterative manner until all the required information has been gathered,
that is, until the process converges.

These methods have been compared in the literature (Fosdick and
data flow analysis is presented in this section and is compared with the
two general methods mentioned above.
The technique presented here is based on the IFT as an internal form of the program without regard to the form of the original high level program. Since the IFT is a highly structured representation of the program with side effect free procedure interfaces, the subsequent data flow analysis can be much simplified as the discussion in this section will illustrate. The technique presented here is a top-down, recursive descent, flow analysis. This particular approach has been advocated elsewhere (Hecht 1977).

In the approach presented here, data flow analysis is undertaken in three phases. In the first phase, the input and output sets for each block are collected. The second phase generates the use and definition information about each value and the third phase does the live value analysis to determine if a value is used in future computations. Each of these phases is described in detail below.

Collection of input and output sets

Entries are created in the IFT during the parse phase and are threaded to represent the ordering of the statements as they were encountered in a sequential scan of the high level program. Each high level statement results in the generation of one or more entries in the IFT where the data flow information is maintained. A simple high level statement (i.e., assignment, procedure or function call, and procedure or function heading) generates only one IFT entry in which the data flow information for that statement is maintained. For compound statements that are conditionally executed (i.e., bodies of while, repeat and if constructs) an interface entry is generated to maintain the
cumulative data flow information for the condition and block of statements within the body. An interface entry represents a staging area for the values used by the condition and block and for the values defined by the block. All information used by the block is conceptually passed from preceding statements through the interface and all information defined by the block is conceptually passed to succeeding statements through the interface. This allows for local flow analysis of blocks of statements.

Two pieces of information called the input set and the output set are associated with each block. The set of upward exposed uses of values in entry $E_i$ contains the names of all the values that are referenced by entry $E_i$ before being defined. The input set for entry $E_i$, denoted $I(E_i)$, contains all the values in the upward exposed use set and those values which are conditionally defined in the body of the block. These are the values that are used inside a given block before being defined. The output set for entry $E_i$, denoted $O(E_i)$, contains the names of all those values which are defined in entry $E_i$. A value which does not appear in $O(E_i)$ is said have its value preserved (Hecht 1977) by entry $E_i$ and will not be of concern in this analysis.

The calculation of the input set and output set for each different type of entry is given below. This discussion is within the context of the sequential high level language described in Chapter II although the ideas are applicable to any high level language.
If $E_0$ is an assignment entry in the IFT and $S$ is the high level statement corresponding to this block then
\[ I(E_0) = \{x: x \text{ is referenced by } S\} \quad \text{and} \quad O(E_0) = \{x: x \text{ is defined by } S\}. \]

If $C$ is a conditional expression and $E_0$ the condition entry then
\[ I(E_0) = \{x: x \text{ is referenced by } C\} \quad \text{and} \quad O(E_0) = \emptyset \quad \text{(i.e., null set)}. \]

As shown in Figure 2.2, the high level input (output) statement generates an input (output) entry for every item that is being read (written). The input entry reflects an operation on a sequential file, removing the first element and converting it to an internal value. Since the file is sequential, the input must necessarily be sequenced by establishing a data dependency on the filename. For this reason, the input filename is added to the input and output sets. If $E_0$ is an input entry needed in support of part of a high level input statement $S$, then
\[ I(E_0) = \{x: x \text{ is referenced by } S\} \cup \{\text{input filename}\} \quad \text{and} \quad O(E_0) = \{x: x \text{ is defined by } S\} \cup \{\text{input filename}\}. \]

If $E_0$ is an output entry needed in support of part of a high level output statement $S$, then
\[ I(E_0) = \{x: x \text{ is referenced by } S\} \cup \{\text{output filename}\} \quad \text{and} \quad O(E_0) = \{\text{output filename}\}. \]

The output filename is added to the input and output sets for the same reason as was the input filename above.
Let \( \text{in}(S) \) be those values which are referenced before being defined in a given procedure and \( \text{out}(S) \) be those values which are defined in the procedure where \( S \) is the high level procedure definition. If \( E_0 \) is a call entry or a procedure entry then
\[
\text{I}(E_0) = \text{in}(S) \quad \text{and} \quad \text{O}(E_0) = \text{out}(S).
\]

If \( S \) is a function definition with entry \( E_0 \) then
\[
\text{I}(E_0) = \text{in}(S) \quad \text{and} \quad \text{O}(E_0) = \{ \text{function name} \}.
\]

A close or end entry is generated to mark the end of a repeat, while, if, procedure or function block and contains no data flow information. The then and else entries are also generated to mark the start of the then and else bodies.

Definitions are now developed so that the input and output sets for interface entries for higher level blocks (if, while or repeat) can be constructed.

Let \( E = E_1, \ldots, E_n \) be any arbitrary set of entries in the IFT where \( i < j \) indicates that entry \( E_i \) appears in the IFT before entry \( E_j \). The set of sequential entries, \( E \), has its input and output sets defined to be
\[
\text{I}(E) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{i-1} (\text{I}(E_i) - \text{O}(E_j)) \quad \text{and} \quad \text{O}(E) = \bigcup_{i=1}^{n} \text{O}(E_i).
\]
The input set consists of those values which do not have a prior definition in the group of entries being processed and the output set contains all values defined in the group of entries.

Conditionally executed blocks (i.e., bodies of if-then, if-then-else, while-do) require special consideration since values are conditionally defined. Suppose \( x \) is conditionally defined in such a block. The value for \( x \) may or may not be produced depending on the result of the test. Suppose that \( x \) is used in some subsequent computation. The value for this use of \( x \) may depend on its conditional definition or on some previous definition. This situation is portrayed in Figure 3.9(a).

To simplify this situation, the previous definition of \( x \) is placed in the input set of the interface IFT entry for the conditional block. The conditional block now unconditionally produces the most recent definition of \( x \) whether it comes from within the conditional block or from the previous definition. This situation is portrayed in Figure 3.9(b).

<table>
<thead>
<tr>
<th>Type</th>
<th>Input Set</th>
<th>Output Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := )</td>
<td>assign</td>
<td>( x )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td></td>
<td>( \ldots )</td>
</tr>
<tr>
<td>if condition then ( x := )</td>
<td>if,</td>
<td>( x )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td></td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( z := x )</td>
<td>assign</td>
<td>( x )</td>
</tr>
</tbody>
</table>

(a) High level segment  
(b) IFT entries

Figure 3.9. Conditional definition
With the above definitions and concepts, the calculation of the input and output sets for interface IFT entries can now be specified.

If $E_0$ is the interface entry for a block of the form if $C$ then $E$ where $C$ is the condition entry in the IFT and $E$ is the set of entries for the body of the then, then

$$I(E_0) = I(C) \cup I(E) \cup \emptyset(E) \text{ and }$$

$$O(E_0) = \emptyset(E).$$

Since all values in the output set are only conditionally defined, they are added to the input set to allow passing on previous definitions.

If $E_0$ is the interface entry for a block of the form if $C$ then $E_1$ else $E_2$ where $C$ is the condition entry and $E_1$ is the set of entries for the body of the then and $E_2$ is the set of entries for the body of the else, then

$$O(E_0) = \emptyset(E_1) \cup \emptyset(E_2) \text{ and }$$

$$I(E_0) = I(C) \cup I(E_1) \cup I(E_2) \cup \{\emptyset(E_0) - (\emptyset(E_1) \cap \emptyset(E_2))\}.$$ 

Those values which are defined only in the then side or the else side are conditionally defined and are added to the input set.

If $E_0$ is the interface entry for a block of the form while $C$ do $E$ end where $C$ is the entry for the condition and $E$ is the set of entries for the body of the while, then

$$I(E_0) = I(C) \cup I(E) \cup \emptyset(E) \text{ and }$$

$$O(E_0) = \emptyset(E).$$

The while block is only conditionally executed and thus all values produced by the block are added to the input set.
If $E_0$ is the interface entry for a block of the form repeat $E$ until $C$ where $E$ is the set of entries for the body of the repeat and $C$ is the entry for the condition, then

$$I(E_0) = I(E) \cup (I(C) - O(E))$$

and

$$O(E_0) = O(E).$$

Figure 3.10 shows the IFT for the bubble sort routine given in Figure 3.6 with the input and output sets calculated for each entry. Lines 3, 9, 12, 14 and 28 represent interface entries for the specified blocks. The values in input sets which are primed indicate those which are conditionally defined by the given block.

Array values receive special consideration since the flow analysis determines data dependency on array name rather than individual components. For example, in line 20, the value $a$ has been added to the input set even though the value was used in left context only in the high level statement. Whenever an element in an array is changed, it is assumed that the entire array has been modified, thus establishing a data dependency on the previous value of the array.

The input and output sets for an interface entry in the IFT represents an accumulation of all the information generated by the input and output sets for the body of the block represented by the interface entry. The collection of this information is readily implemented by a top-down recursive descent parse. Up to this point, the input and output sets have only been generated and no linkages exist to show the use and definition relationships between the values. This is the subject of the next section.
<table>
<thead>
<tr>
<th>Entry type</th>
<th>Input set</th>
<th>Output set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. procedure</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>1. assign</td>
<td>$\emptyset$</td>
<td>$\text{amax}$</td>
</tr>
<tr>
<td>2. assign</td>
<td>$\emptyset$</td>
<td>$i$</td>
</tr>
<tr>
<td>3. while</td>
<td>$i', \text{amax}, a'$, $\text{in}'$</td>
<td>$a, i, \text{in}$</td>
</tr>
<tr>
<td>4. condition</td>
<td>$i, \text{amax}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>5. input</td>
<td>$i, a, \text{in}$</td>
<td>$a, \text{in}$</td>
</tr>
<tr>
<td>6. assign</td>
<td>$i$</td>
<td>$i$</td>
</tr>
<tr>
<td>7. close (while)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. assign</td>
<td>$\text{amax}$</td>
<td>$\text{top}$</td>
</tr>
<tr>
<td>9. repeat</td>
<td>$\text{top}, a, \text{temp}$</td>
<td>$\text{flag, temp, a, i, top}$</td>
</tr>
<tr>
<td>10. assign</td>
<td>$\emptyset$</td>
<td>$\text{flag}$</td>
</tr>
<tr>
<td>11. assign</td>
<td>$\emptyset$</td>
<td>$i$</td>
</tr>
<tr>
<td>12. while</td>
<td>$i', \text{top, a}'$, $\text{flag}', \text{temp}'$</td>
<td>$\text{flag, temp, a, i}$</td>
</tr>
<tr>
<td>13. condition</td>
<td>$i, \text{top}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>14. if</td>
<td>$a', i$, $\text{flag}', \text{temp}'$</td>
<td>$\text{flag, temp, a}$</td>
</tr>
<tr>
<td>15. condition</td>
<td>$a, i$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>16. then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17. assign</td>
<td>$\emptyset$</td>
<td>$\text{flag}$</td>
</tr>
<tr>
<td>18. assign</td>
<td>$a, i$</td>
<td>$\text{temp}$</td>
</tr>
<tr>
<td>19. assign</td>
<td>$a, i$</td>
<td>$a$</td>
</tr>
<tr>
<td>20. assign</td>
<td>$\text{temp, i, a}$</td>
<td>$a$</td>
</tr>
<tr>
<td>21. close (if)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22. assign</td>
<td>$i$</td>
<td>$i$</td>
</tr>
<tr>
<td>23. close (while)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24. assign</td>
<td>$\text{top}$</td>
<td>$\text{top}$</td>
</tr>
<tr>
<td>25. condition</td>
<td>$\text{flag}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>26. close (repeat)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27. assign</td>
<td>$\emptyset$</td>
<td>$i$</td>
</tr>
<tr>
<td>28. while</td>
<td>$i', \text{amax, a}$, $\text{out}'$</td>
<td>$\text{i, out}$</td>
</tr>
<tr>
<td>29. condition</td>
<td>$i, \text{amax}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>30. output</td>
<td>$a, i, \text{out}$</td>
<td>$\text{out}$</td>
</tr>
<tr>
<td>31. assign</td>
<td>$i$</td>
<td>$i$</td>
</tr>
<tr>
<td>32. close (while)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33. end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.10. IFT entries for bubble sort with input and output sets
Generating the use and definition sets

After the IFT has been completely constructed and the input and output sets have been generated, the use and definition information is collected. The matching of definition with corresponding use is done by matching the name of the values (name of identifier in original source code) in the corresponding input and output sets. For all entries producing a value, a list is maintained showing all the entries where that value is used. For all entries needing a value, an ordered list (of maximum length of two) is maintained giving the entry(s) where the value was defined. The entry which defines a value can be found by a backward scan of the preceding entries until the value appears in an output set of an entry or the interface entry of the enclosing block. If it is not found, a use of the value appears before its definition.

The set of entries within the enclosing block which use the value of \( x \) defined in \( E \) is denoted \( \text{use}(x,E) = (a_1, \ldots, a_n) \). If \( x \) is used in entry \( E \), then \( \text{def}(x,E) = (a, (b,c)) \) denotes the definition set of \( x \).

For all blocks without an interface entry, this set consists of only a first element \( a \). For blocks with an interface entry, this set contains two elements \( a \) and \( (b,c) \). The element \( a \) identifies where the value was most recently defined outside the block and \( (b,c) \) identifies the last definition within the block. Except for the case of an if-then-else, \( c \) is null.

The use and definition analysis is presented in Figure 3.11 as a recursive top-down procedure which produces the use and def sets for
procedure useanddef(in(E,H), out(E,H))
    /*
    E - set of entries in the body of H
    H - interface entry or procedure/function entry
    */
    elseflag := false;
    for i = 1 to |E| do
        if TYPE(E_i) = (else or then)
            then if TYPE(E_i) = else
                then elseflag := true
            end if
        else for each x ∈ I(E_i) do
            finddef(in(i,x,E,H), out(E,H))
        end for
        for each x ∈ O(E_i) do
            if x ∈ O(H)
                then if elseflag
                    then def(x,H)(2,2) := E_i
                    else def(x,H)(2,1) := E_i
                end if
            end if
        end for
        if TYPE(E_i) = (while or repeat or if) then
            then U := {x:x is an entry of E_i} useanddef(in(U,E_i), out(U,E_i))
        end if
        if TYPE(E_i) = (while or repeat)
            then for each x ∈ (I(H) - O(H)) do
                def(x,H)(2,1) := H
                use(x,H) := use(x,H) U H
            end for.
        end if
        if TYPE(E_i) = (while or repeat or if or procedure or function)
            then for each x ∈ O(H) do
                if def(x,H)(2,1) ≠ ∅
                    then SE := def(x,H)(2,1)
                    use(x,SE) := use(x,SE) U H
                end if.
                if def(x,H)(2,2) ≠ ∅
                    then SE := def(x,H)(2,2)
                    use(x,SE) := use(x,SE) U H
            end for.
    end for.

Figure 3.11. useanddef procedure
procedure finddef (in(i,x,E,H),out(E,H))

/*
i - index of entry defining the value
x - name of the value being searched for
E - set of entries in the body of H
H - interface entry or procedure/function entry
*/
found := false
for j = i-1 to 1 while not found do
  if x \in O(E_j)
     then def(x,E_j)(1) := E_j
     use(x,E_j) := use(x,E_j) U E_j
     found := true
  end if
end for
if not found
  then if x \in I(H)
     then def(x,E_i)(1) := H
     use(x,H) := use(x,H) U E_i
     else def(x,E_i)(1) := \emptyset
  end if
end if
end procedure

Figure 3.11. Continued
the entire IFT. Suppose \( H \) denotes the interface entry for the block to be analyzed and \( E \) denotes the set of entries within the body of \( H \). The procedure modifies the IFT entries for \( E \) and \( H \) by attaching the use and def sets. The initial call would take the form \( \text{useanddef}(\text{in}(E,H), \text{out}(E,H)) \).

Figure 3.12 shows the entries in the IFT for the bubble sort procedure with the use and def sets generated for each of the input and output sets. At this point, it is easy to determine values which have been referenced before being defined because the def set is empty. This is an error condition for scalar values, but may not be an error condition for arrays since arrays may be constructed starting from a nil structure. It would be an error for an array if a component of the array is referenced before it is defined.

The outer for loop in the useanddef procedure presented in Figure 3.11 is executed for every entry in the IFT. For a given entry, it finds where the input values were last defined. This is done by a backward scan of the output set of the blocks that may have defined the value as seen in the finddef procedure. The scan terminates when the value is found in an output set or the interface entry for the enclosing block is encountered.

Let \( e = |\text{IFT}| \) be the number of entries in the IFT. Let \( io = \max(|I(E_i)|) \) for \( 1 \leq i \leq e \) be the maximum number of unique values appearing in an input set for any \( E_i \) in the IFT. Let \( B = B_1, \ldots, B_n \) be the
<table>
<thead>
<tr>
<th>IFT Entries</th>
<th>Input set</th>
<th>Output set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>def</td>
<td>use</td>
</tr>
<tr>
<td>0. procedure</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>1. assign</td>
<td>Ø</td>
<td>amax i</td>
</tr>
<tr>
<td>2. assign</td>
<td>Ø</td>
<td>i</td>
</tr>
<tr>
<td>3. while</td>
<td>i 2,6</td>
<td>4,5,6 a</td>
</tr>
<tr>
<td></td>
<td>amax 1,3</td>
<td>3,4 i in</td>
</tr>
<tr>
<td></td>
<td>a 0,5</td>
<td>4 in</td>
</tr>
<tr>
<td></td>
<td>in 0,5</td>
<td>4</td>
</tr>
<tr>
<td>4. condition</td>
<td>i 3</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>amax 3</td>
<td></td>
</tr>
<tr>
<td>5. input</td>
<td>i 3 a 3</td>
<td>in 3</td>
</tr>
<tr>
<td></td>
<td>in 3</td>
<td></td>
</tr>
<tr>
<td>6. assign</td>
<td>i 3</td>
<td>i 3</td>
</tr>
<tr>
<td>7. close (while)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. assign</td>
<td>amax 1 top</td>
<td>9</td>
</tr>
<tr>
<td>9. repeat</td>
<td>top 8,24</td>
<td>12,24 flag</td>
</tr>
<tr>
<td></td>
<td>a 3,12 12</td>
<td>temp 0,12</td>
</tr>
<tr>
<td></td>
<td>temp 0,12</td>
<td>a 28</td>
</tr>
<tr>
<td></td>
<td>i 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>top 8,24</td>
<td></td>
</tr>
<tr>
<td>10. assign</td>
<td>Ø</td>
<td>flag 12</td>
</tr>
<tr>
<td>11. assign</td>
<td>Ø</td>
<td>i 12</td>
</tr>
<tr>
<td>12. while</td>
<td>i 11,22</td>
<td>13,14,22 a</td>
</tr>
<tr>
<td></td>
<td>top 9,12</td>
<td>12,13 flag</td>
</tr>
<tr>
<td></td>
<td>a 9,14</td>
<td>14 temp 9</td>
</tr>
<tr>
<td></td>
<td>flag 10,14</td>
<td>14 i 9</td>
</tr>
<tr>
<td></td>
<td>temp 9,14</td>
<td>14</td>
</tr>
<tr>
<td>13. condition</td>
<td>i 12</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>top 12</td>
<td></td>
</tr>
<tr>
<td>14. if</td>
<td>a 12,20</td>
<td>16,18, flag</td>
</tr>
<tr>
<td></td>
<td>i 12</td>
<td>16, temp 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18,19</td>
</tr>
<tr>
<td>15. condition</td>
<td>a 14</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td>i 14</td>
<td></td>
</tr>
<tr>
<td>16. then</td>
<td>Ø</td>
<td>flag 14</td>
</tr>
<tr>
<td>17. assign</td>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>18. assign</td>
<td>a 14</td>
<td>temp 14,20</td>
</tr>
</tbody>
</table>

Figure 3.12. IFT entries for bubble sort with use and def sets
IFT Entries | Input set | Output set
--- | --- | ---
| def | use | use | def |
19. assign | a 14 | a 20 |
20. assign | temp 18 | a 14 |
| i 14 | a 19 |
21. close (if) |
22. assign | i 12 | i 12 |
23. close (while) |
24. assign | top 9 | top 9 |
25. condition | flag 12 | Ø |
26. close (repeat) |
27. assign | Ø | i 28 |
28. while | i 27,31 | 29,30,31 | i 27,31 |
| amax 1,28 | 28,29 | out 0,30 |
| a 9,28 | 28,30 |
| out 0,30 | 30 |
29. condition | i 28 | Ø |
| amax 28 |
30. output | a 28 | out 28 |
| out 28 |
| i 28 |
31. assign | i 28 | i 28 |
32. close (while) |
33. end |

---

Figure 3.12. Continued
set of blocks in a given procedure. Let \( m = \max (|B_i|) \) for \( 1 \leq i \leq e \) be the maximum number of entries that could be searched in the finddef procedure. The time complexity for the usedanddef procedure is bounded by \( O(e \cdot io \cdot b) \). This is a worst case bound seldom, if ever, realized in practice. Typically \( io \) and \( b \) would be small in comparison with \( e \). In the study of programs done by Knuth (1971), it was found that assignment statements dominated any other single kind of statement found in the programs studied. Those assignment statements were then analyzed, and it was found that more than eighty-five percent of them had a maximum of two operands. The input set of the corresponding entry in the IFT would have at most two elements in a vast majority of the assignment statements and the output set would, of course, only have one. In light of the above facts, the \( io \) term would be considerably smaller if an average case time complexity was considered instead of a worst case. \( b \) would also be small if an average case time complexity was considered. \( b \) indicates the number of blocks (entries) that need to be searched in order to find a particular value. This, in reality, represents the number of statements in the body of the high level repeat, if and while statements along with the body of the procedure or function. If \( b \) were averaged over all the occurrences of these statements in the program, \( b \) would also be much smaller. In an average case analysis, this \( O(e \cdot io \cdot b) \) time complexity is not as overwhelming as it might appear at first sight. In the bubble sort for \( e = 34 \) entries, the worst case bound would produce \( io = 5 \) and \( b = 7 \) whereas the average case would produce \( io = 1.3824 \) and \( b = 2.2059 \).
For at least two reasons, it is very difficult to compare the above time complexity with others given in the literature. First, the technique described here uses the size of the IFT in the time complexity expression while other techniques use the number of basic blocks (i.e., maximal group of statements such that no transfer occurs into a group except to the first statement in that group, and once the first statement is executed, all statements in the group are executed sequentially) in a program. Second, the technique described here generates more information than other published techniques. Other techniques determine for a given block whether a value is defined, used and what values are available. The technique presented here specifies the precise IFT entry where the values are defined or used. If it produced only the information generated by other data flow techniques, the time complexity would be \( O(e) \) bit vector steps since no backward scan would need to be done and the input and output sets could be handled with a single bit vector operation. For the iterative algorithms using a worklist or round-robin algorithm (Hecht 1977), the worst case time complexity is \( O(n^2) \) where \( n \) is the number of nodes (basic blocks) in the control flow graph. The number of entries in the IFT typically is somewhat larger than the number of nodes in a control flow graph but it is difficult to say how much larger. Ullman's algorithm (Ullman 1973) requires at most \( O(n \log n) \) bit vector steps and the Graham and Wegman algorithm (Graham and Wegman 1976) requires at most \( O(n) \) bit vector steps if the number of exits per loop is bounded. The interval algorithm (Hecht 1977)
requires $O(kr)$ steps where $r$ is the number of arcs in the input graph and $k$ is the number of graphs in the derived sequence.

**Live value analysis**

Live value analysis is used in certain optimizing transformations and for detection of useless computation. A useless computation is a computation whose value is never used by a subsequent computation. The live value analysis is performed after the use and def sets have been generated.

A value is said to be live at a given point in a program if the value is used by a subsequent entry or is used by an enclosing block. The information gathered in the first two phases of data flow analysis, i.e., the input sets, output sets, use sets and def sets, are used to perform the live analysis. Associated with every value $x$ is an output set which consists of a single boolean value indicating whether $x$, defined by entry $E_1$, is live after this point. If it is, live($x,E_1$) is true otherwise it is false.

A top-down recursive descent algorithm is once again used to generate this set. Starting with the entries at the procedure level, the values in the output set are analyzed to see if they are live. If an entry is encountered which represents a repeat, if or while statement in the high level program, a recursive call on the liveanalysis procedure is made to propagate the live information into the entries in the body. This continues until all the entries in the IFT have been processed.
The algorithm for live value analysis is given in Figure 3.13.

Figure 3.14 shows the IFT for the bubble sort routine with the live value analysis performed. The procedure is called with liveanalysis(in(E,H), out(E,H)) where E = E₁, ..., Eₙ are the entries of the procedure or function and H is the IFT entry for the procedure or function.

procedure liveanalysis(in(E,H),out(E,H))
/*
   E - set of entries in the body of H
   H - interface entry or procedure/function entry
*/

   for i = 1 to |E| do
      for each x ∈ O(Eᵢ) do
         live(x,Eᵢ) := false
         if use(x,Eᵢ) ≠ Ø
            then if use(x,Eᵢ) = {H}
               then if TYPE(H) = (while or repeat) and x ∈ I(H)
                  then live(x,Eᵢ) := true
               else if TYPE(H) = (procedure or function)
                  then live(x,Eᵢ) := true
                  else live(x,Eᵢ) := live(x,H)
            end if
         else live(x,Eᵢ) := true
      end if
   end for

   if TYPE(Eᵢ) = (while or repeat or if)
      then U := {x|x is an entry of Eᵢ} 
         liveanalysis(in(U,Eᵢ),out(U,Eᵢ))
   end if

end procedure

Figure 3.13. liveanalysis procedure

<table>
<thead>
<tr>
<th>Entry type</th>
<th>Input set</th>
<th>Output set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. procedure</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>1. assign</td>
<td>Ø</td>
<td>amax - true</td>
</tr>
<tr>
<td>2. assign</td>
<td>Ø</td>
<td>i - true</td>
</tr>
<tr>
<td>3. while</td>
<td>i, amax, a, in</td>
<td>a - true, i - false, in - false</td>
</tr>
</tbody>
</table>

Figure 3.14. IFT entries for bubble sort with live value analysis performed
<table>
<thead>
<tr>
<th>Entry type</th>
<th>Input set</th>
<th>Output set</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. condition</td>
<td>i,amax</td>
<td>Ø</td>
</tr>
<tr>
<td>5. input</td>
<td>i,a,in</td>
<td>a - true</td>
</tr>
<tr>
<td>in - true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i - true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. assign</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>7. close (while)</td>
<td>amax</td>
<td>top - true</td>
</tr>
<tr>
<td>8. assign</td>
<td>top,a,temp</td>
<td>flag - false</td>
</tr>
<tr>
<td>temp - false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a - true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i - false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>top - false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>flag - true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. repeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. assign</td>
<td>Ø</td>
<td>i - true</td>
</tr>
<tr>
<td>11. assign</td>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>12. while</td>
<td>i,top,a,flag,temp</td>
<td></td>
</tr>
<tr>
<td>13. condition</td>
<td>i,top,a</td>
<td></td>
</tr>
<tr>
<td>14. if</td>
<td>a,i,flag,temp</td>
<td>flag - true</td>
</tr>
<tr>
<td>temp - false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a - true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15. condition</td>
<td>a,i</td>
<td></td>
</tr>
<tr>
<td>16. then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17. assign</td>
<td>Ø</td>
<td>flag - true</td>
</tr>
<tr>
<td>18. assign</td>
<td>a,i</td>
<td>temp - true</td>
</tr>
<tr>
<td>19. assign</td>
<td>a,i</td>
<td>a - true</td>
</tr>
<tr>
<td>20. assign</td>
<td>temp,i,a</td>
<td>a - true</td>
</tr>
<tr>
<td>21. close (if)</td>
<td>i</td>
<td>i - true</td>
</tr>
<tr>
<td>22. assign</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23. close (while)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24. assign</td>
<td>top</td>
<td>Ø</td>
</tr>
<tr>
<td>25. condition</td>
<td>flag</td>
<td>Ø</td>
</tr>
<tr>
<td>26. close (repeat)</td>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>27. assign</td>
<td>i,amax,a,out</td>
<td>i - true</td>
</tr>
<tr>
<td>i - false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>out - false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28. while</td>
<td>i</td>
<td>Ø</td>
</tr>
<tr>
<td>29. condition</td>
<td>i,amax</td>
<td>out - true</td>
</tr>
<tr>
<td>30. output</td>
<td>a,i,out</td>
<td></td>
</tr>
<tr>
<td>31. assign</td>
<td>i</td>
<td>i - true</td>
</tr>
<tr>
<td>32. close (while)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33. end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.14. Continued
The outer for loop in the liveanalysis procedure presented in Figure 3.13 is executed for every entry in the IFT. For a given entry, the inner for loop processes the output set to see if the given value is live.

Let $e = |\text{IFT}|$ be the number of entries in the IFT and $os = \max(|0(E_i^1)|)$ for all $1 \leq i \leq e$ be the largest output set in an entry in the IFT where $E_i^1$ is an entry in the IFT. The time complexity for the liveanalysis procedure is bounded by $O(e*os)$.

 Typically $os$ would be small in comparison with $e$ and the same argument can be given as was done previously that the average case is not as bad as it might appear to be. In the bubble sort procedure for $e = 34$ IFT entries, the worst case bound would produce $os = 5$ whereas the average case would produce $os = 1.0588$. The other techniques time complexities are the same as given previously. This is because essentially the same algorithm is used in propagating the information inward instead of outward. If the same information was gathered by the technique presented here as by other data flow techniques, the algorithm would be $O(e)$ since the inner for loop would be taken care of by a bit vector operation.
CHAPTER IV. REDUCING DATA DEPENDENCIES

Introduction

This chapter takes a typical set of optimizing transformations and shows how these transformations may be implemented using the data flow analysis technique presented in Chapter III thereby demonstrating the usefulness and power of this technique. Special attention is given to one transformation in particular, loop decomposition. This transformation is extended so loops which previously could not have been transformed into a highly parallel form, can now be decomposed and analyzed to see if they may be done in parallel. These transformations are then further studied and a partial ordering for the transformations is derived according to three different criteria (i.e., execution time, redundant computation and instruction space). The partial ordering with respect to each of these criteria indicates the order the transformations should be applied in order to help minimize the effect of each of the criteria. Specifications are given for the incorporation of these transformations into a compiler so that the effect of the three criteria given above may be decreased. Finally, extensions to the sequential high level language are specified which allow the programmer to indicate to the compiler parallelism in the high level program.
Transformations

For this work, a typical set of transformations that are generally found in an optimizing compiler was chosen. This set does not represent every conceivable transformation that exists today, but a cross section of those which are currently used in compilers. These transformations fall into one of the following five categories.

1. reduces the number of data dependencies in a given block.
   This decreases the total execution time of the block and if the block lies on the critical path through the program, the total execution time of the whole program is decreased.

2. analyzes the iterative constructs used in the program to see if their iterations are independent. If they are, the loop is transformed from an iterative loop into a forall loop indicating that the iterations can be executed in parallel. If the iterations are not independent, the loop is decomposed to see if some subset of statements in the body of the loop may be done as a forall loop. If all the iterations are independent, all iterations may be done in parallel thus greatly decreasing the execution time of the loop.

3. increases the size of the body of a loop which has to be executed sequentially. Whenever the number of instructions in the body of a loop is small, the result is that there is very little parallelism available. Transformations of this type expand the body of the loop and create potentially independent operations.
4. reduces the number of resources used by the program. Transformations of this type cannot decrease the execution time of a program, in fact there are cases in which the execution time can be increased as a result of this type of transformation. These transformations reduce the amount of redundant computation performed in the program.

5. replaces more expensive operators with less expensive operators. The transformation that falls into this category replaces multiplications with additions under certain circumstances. This type of transformation is performed under the assumption that a multiplication takes more time to execute than does an addition.

Those transformations included in this study are given in Figure 4.1.

1. Constant Folding
2. Scalar Propagation
3. Strength Reduction
4. Common Subexpression Elimination
5. Invariant Code Motion
6. Induction Value Removal
7. Loop Decomposition
8. Loop Fusion
9. Loop Unrolling
10. Forward Substitution
11. Tree Balancing

Figure 4.1. Set of transformations

There are many other transformations that are useful in sequential machines (Allen 1969, Allen and Cocke 1971) or in parallel machines such
as if-node balancing (Kuck et al. 1972), unswitching, clustering of similar operations for pipelined machines, precomputed conditional removal and introduction of staging into a pipe (Boyd 1977) which were not included in this study.

Many of the transformations presented here yield a local speedup factor of less than two. This speedup, however, may occur in a portion of the program that is a bottleneck as in the case of vector or stream oriented programs, thereby making a local transformation globally significant. Consider the example in Figure 4.2. For the small program segment presented in (a), if each statement is considered to be a stage in the pipe, the program being the whole pipe, statement (1) is the bottleneck, taking three time steps to execute. By applying the tree balancing transformation, the speedup of which is generally not very significant, this section of the pipe can execute in two time steps as seen in (b) making this particular transformation globally significant.

(1) a := ((b + c) + b) + d a := (b + c) + (b + d)  
(2) e := a + g e := a + g  
(3) f := a + e f := a + e

Figure 4.2. Bottleneck in a program

The sections that follow elaborate on each of the transformations shown in Figure 4.1 by explaining its purpose, giving an example of it, explaining its usefulness, presenting an algorithm to apply the transformation using the data flow information generated in Chapter III and
analyzing the work done by the compiler in applying each of these algorithms. If the algorithm is large, only a summary of the algorithm appears in this chapter and the full algorithm is found in Appendix B. The algorithms given may not be optimal, but if other algorithms were used (Cocke and Schwartz 1970, Frailey 1971, Gries 1971, Ottenstein 1978), they could be readily implemented using the data flow analysis already performed.

Most of the transformations given here are only applied to statements appearing in the same block. When a program is to have these transformations applied, the transformations are applied recursively in a highly structured manner. The innermost block of statements is considered first, and then the enclosing block of statements is considered and so on. When the transformations have all been applied to a block of statements, that block is treated as a single statement by the enclosing block and nothing more is done to the inner block. These transformations are only applied intra-procedurally. No attempt has been made for the application of these transformations inter-procedurally as has been advocated elsewhere (Barth 1978). It is realized that there might be further gain possible if the application of the transformations was not structured as such, but the structured approach was chosen because of the simplicity added in the algorithms and in the derivation of the partial orderings on these transformations. The only transformations which are allowed to cross the boundaries of a block are invariant code motion and strength reduction.

The IFT may be modified by these transformations in a number of ways. Entries in the IFT may be added and deleted. The data flow
information in the IFT is updated by all transformations which make any modifications to the IFT.

The algorithms given for each of the transformations specify the manner in which the transformation is to be performed. No algorithms are specified for manipulating the syntax trees. The work expression for a given transformation is actually a measure of the number of operations performed on the IFT by the compiler. This expression is given in terms of constant work factors which do not contribute to the order of the transformation along with variable factors which actually determine the order.

**Constant folding**

The constant folding transformation replaces a named value with its constant value if it can be computed at compile time and performs all other computations which may be done at compile time rather than at run time.

An example of constant folding appears in Figure 4.3. The value a is assigned a constant value 6 in statement (1). The constant replaces the value a in the two places it is used, in statements (3) and (4). The replacing of a in statement (3) allows a multiplication to take place giving the value of 18. The value b assigned in statement (2) is used in one place, statement (3). In statement (3), the addition takes place giving c a value of 23 which is now a constant and may be folded. Since all the uses of a, b and c in this block of statements have been replaced by their constant value (it is assumed they are not live outside this block), statements (1), (2) and (3) are no longer necessary and may be eliminated giving Figure 4.3 (b).
(1) \ a := 6;  
(2) \ b := 5;  
(3) \ c := 3 \ast a + b  
(4) \ do \ i=a \ to \ c  \quad do \ i=6 \ to \ 23  

(a) before  \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (b) after

Figure 4.3. Example of constant folding

The advantages of constant folding are that a data dependency is eliminated, the calculation of expressions at compile time rather than at run time and the possible elimination of entries and values in the IFT.

The algorithm for constant folding is presented in Figure 4.4. A call to the algorithm is made with a statement of the form constantfolding (in(E,T),out(E,T)) where T is an interface IFT entry or procedure/function entry and E = E_1, ..., E_n is the set of entries in the body of T. The procedure constantfolding first performs any calculations in the syntax tree of an entry where both operands are constant values. It then examines each entry in a given block to see if the entry is able to be folded. When an entry is found which can be folded, the procedure fold is called and performs the actual folding of the value. If the value folded has no more uses inside this block and the value is not live outside the block, the entry is deleted.
procedure constantfolding (in(E,T), out(E,T))
/*
E - set of entries in the body of T
T - interface entry or procedure/function entry
*/
o := ∅
/* for each entry in the block do */
for i = 1 to |E| do
/* perform any constant calculations in the syntax tree */
if TYPE(E_i) ≠ (if or while or repeat or forall)
then calculate(in(TREE(E_i), out(TREE(E_i)))
/* if a constant is assigned, fold the constant */
if I(E_i) = ∅ and TYPE(E_i) = assign
then fold(in(i,E), out(E, useless))
else useless := false
end if
/* if the entry is now useless, remove it */
if useless and not live(0(E_i), T)
then o := o U 0(E_i)
E := E - E_i
end if
end for
/* if T is an interface entry, remove the useless values */
if TYPE(T) ≠ (procedure or function)
then 0(T) := 0(T) - o
end if
end procedure

procedure fold (in(i,E), out(E, useless))
/*
i - index of entry to be folded
E - set of entries being considered
useless - flag indicating if entry E_i in now useless
*/
x := 0(E_i)
const := "constant value assigned to x in right hand side of TREE(E_i)"
/* for all uses of the value x, see if it can be replaced by its constant value */
for a ∈ use(x, E_i) do
if TYPE(a) ≠ {while or repeat or if or forall)
then replace(in(x, const, TREE(a)), out(TREE(a))
I(a) := I(a) - x
use(x, E_i) := use(x, E_i) - a
end if
end for

Figure 4.4. Algorithm for constant folding
/* if all the uses of x have been replaced, set useless
to true otherwise set useless to false */
if use(x,E_i) = \emptyset
    then useless := true
    else useless := false
end if
end procedure

The algorithm for the procedure "replace" is not given since
it deals only with the syntax tree, but it has the following two
functions. First, it replaces all uses of x in TREE(a) with its constant
value const and second, it calls "calculate" which performs any constant
calculation in the syntax tree.

The work involved in performing the algorithm for constant folding is
given by the following expression:

\[ W_{cf} = \sum_{i=1}^{\left|E\right|} \sum_{j=1}^{\left|u_i\right|} (c_1 + k_i \cdot c_2). \]

c_1 is the constant amount of work involved in performing the body of the
procedure constantfolding, c_2 is the constant amount of work involved in
folding a given constant to a given place. \left|u_i\right| is the number of uses
the value defined by entry E_i has and k_i is 1 if entry E_i can be folded
and 0 otherwise. It is anticipated that for most programs, \( W_{cf} \sim O\left(\left|E\right|\right) \) since \left|u_i\right| is typically very small in comparison to \left|E\right|.

**Scalar propagation**

The scalar propagation transformation tries to delete assignment
entries of the form x := y, where x and y are the same type, by
replacing all the uses of the value \( x \) with the value \( y \). If all the uses of \( x \) can be replaced and \( x \) is not live outside the block, the assignment entry may also be deleted. In effect, the constant folding transformation could be incorporated into this transformation making it more general, but for the purpose of this discussion, they are kept separate.

An example of scalar propagation is shown in Figure 4.5. The value \( x \) assigned in statement (1) is used in statement (2) and (3). Every time \( x \) appears in an expression in (2) or (3), it is replaced by \( y \). Since all the uses of \( x \) have been replaced by \( y \) and \( x \) is not live outside this block, statement (1) is no longer needed and may be deleted as shown in (b).

\[
\begin{align*}
(1) & \quad x := y \\
(2) & \quad z := x + z * b * x \\
(3) & \quad \text{do } i=1 \text{ to } x \\
(a) \text{ before } & \\
(2) & \quad z := y + z * b * y \\
(3) & \quad \text{do } i=1 \text{ to } y \\
(b) \text{ after }
\end{align*}
\]

Figure 4.5. Example of scalar propagation

The advantages of scalar propagation are the possible elimination of some assignment entries and the elimination of data dependencies.
The algorithm for scalar propagation is presented in Figure 4.6. A call to the algorithm is made with a statement of the form scalarpropagation(in(E,T),out(E,T)) where T is an interface or a procedure/function entry and E = E_1, ..., E_n is the set of entries in the body of T. The procedure scalarpropagation examines each entry in the body of the block to see if it is of the form x := y. If it is, the procedure propagate is called which replaces the uses of x with y, if possible, updating the data flow information. If all the values of x are replaced by y and x is not live outside the block, the entry is removed.

procedure scalarpropagation (in(E,T),out(E,T))
/*
E - set of entries in the body of T
T - interface entry or procedure/function entry
*/
o := Ø
/* for each entry in this block do */
for i=1 to |E| do
/* if the entry is of the form x := y, call propagate */
if TYPE(E_i) = assign and
  TREE(E_i) is of the form x := y and
  type of x = type of y
  then propagate(in(i,E),out(E,useless))
  else useless := false
  end if
/* if the entry is now useless, delete the entry */
if useless and not live (O(E_i),T)
  then o := o U O(E_i)
    st := def(I(E_i), E_i)(1)
    use(I(E_i),st)^i := use(I(E_i),st) - E_i
    E := E - E_i
  end if
end for
/* delete useless values from the interface entry */
if TYPE(T) ≠ (procedure or function)
  then O(T) := O(T) - o
end if
end procedure

Figure 4.6. Algorithm for scalar propagation
procedure propagate (in(i,E), out(E, useless))

/*
  i - index of entry to be propagated
  E - set of entries being considered
  useless - boolean flag indicating if entry E_i is now useless
*/

x := 0(E_i)
y := 1(E_i)
st := def(y,E_i)(l)

/* for all uses of the value x, do */
for a ∈ use(x,E_i) do
  if TYPE(a) ≠ {while or repeat or if or forall) then
    /* replace(in(x,y,TREE(a)),out(TREE(a))) */
    I(a) := I(a) - x U y
    use(y,st) := use(y,st) U a
    use(x,E_i) := use(x,E_i) - a
  end if
end for

/* if all uses of x have been replaced, set the flag */
useless := true
else useless := false
end if

end procedure

Figure 4.6. Continued

The work involved in performing the scalar propagation algorithm can be given by the following expression:

\[ W_{sp} = \sum_{i=1}^{\mid E \mid} \left( c_1 + \sum_{j=1}^{\mid u_i \mid} c_2 \right) \]

\( c_1 \) is the constant amount of work done by the body of the procedure scalarpropagation. \( c_2 \) is the constant amount of work done by the procedure propagate. \( \mid u_i \mid \) is the number of uses a given value which is to be propagated has and \( k_i \) is 1 if entry \( E_i \) can be propagated and 0
otherwise. It is anticipated that for most programs, \( W_{sp} \sim O(|E|) \) since \( |u_1| \) is small in comparison to \( |E| \).

**Strength reduction**

Computations of the form \( i \times 1 \) where \( i \) is the loop induction value and 1 is a loop invariant are candidates for strength reduction. A loop induction value is one whose value is changed in the loop only by instructions which increment an induction value by a constant amount; i.e., instructions of the form \( i := i + 1 \) or \( i := j + 3 \) where \( j \) is another induction value. The primary aim of this transformation is to remove multiplications of this form and replace it with an addition under the assumption that on most computers multiplications take longer to perform than do additions. If this assumption is not true for a particular machine, then the transformation is not applied. The transformation generally applies to array accessing when multi-dimensional arrays are linearized for accessing purposes.

The basic method is to define a temporary which holds the value of the multiplication. In order to maintain the correct value for the temporary, every time \( i \) is modified, the value of the temporary must also be modified; however, this can usually be done by a simple addition.

Consider the example in Figure 4.7. The multiplication of \( i \times 50 \) in statement (4) can be eliminated and replaced in the loop with an addition. This shown in (b) where the multiplication is replaced by \( t_1 \) and every time \( i \) is incremented in the loop, \( t_1 \) is also incremented.
(1) real array a(1:50,1:50)
   .
   .
(2) sum := 0.0
(3) do i=1 to 50
(4) sum := sum + a(i*50+3)
(5) end

(a) before

(b) after

Figure 4.7. Example of strength reduction after array subscripts have been linearized

Once strength reduction has been performed, some induction values are no longer needed except in the test for the condition of the loop. These induction values may be eliminated if the test can be replaced by a test on another induction value. This situation is shown in Figure 4.8 in which the induction value i in Figure 4.7 has been eliminated.

real array a(1:50,1:50)
   .
   .
   .
sum := 0.0
do tl=50 to 2500
    sum := sum + a(tl+3)
    tl := tl + 50
end

Figure 4.8. Induction value eliminated
The advantage of strength reduction is that a faster operation, addition (on most computers), replaces a slower one, multiplication, in a loop.

A high level summary of the algorithm for strength reduction developed by Cocke and Kennedy (Cocke and Kennedy 1977) is given in Figure 4.9 while the complete algorithm is given in Appendix B. A call to the algorithm is made with a statement of the form \texttt{reducestrength(in(E,T), out(E,T))} where \( T \) is an interface entry and \( E = E_1, \ldots, E_n \) is the set of entries in the body of \( T \). The procedure \texttt{reducestrength} calls the other procedures in an effort to reduce the strength in a given segment of code. The procedure \texttt{findrc} finds all loop constants in a given region forming the set \( RC \). The procedure \texttt{findiv} finds all the induction values in the region forming the set \( IV \). The procedure \texttt{findcands} forms the set \( CANDS \) which contains those subexpressions which are candidates for strength reduction. The procedure \texttt{computeffect} forms the array of sets \( AFCT \) which contains the names of all the induction values and region constants which can affect a given induction value. The procedure \texttt{reduce} reduces the strength of all possible subexpressions in \( CANDS \) by replacing a multiplication with an addition and introducing temporary values which hold the value of the multiplication.

\begin{verbatim}
procedure reducestrength (in(E,T),out(E,T))
    /*
    E - set of entries in the body of T
    T - interface entry for the loop
    */
    findrc(in(E,T),out(RC))
    findiv(in(E,RC),out(IV))
\end{verbatim}

Figure 4.9. Summary of algorithm for strength reduction
procedure findrc (in(E,T), out(RC))
    /*
    E - set of entries in the body of T
    T - interface entry for the loop
    RC - set of region constants
    */
    /* find all region constants */
    RC := I(T) - 0(T)
    for all entries
        RC := RC U constants in the entry
    end for
end procedure

procedure findiv (in(E,RC), out(IV))
    /*
    E - set of entries being considered
    RC - set of region constants
    IV - set of induction values
    */
    /* find all induction values */
    IV := ∅
    for all entries
        if TYPE(this entry) = assign and operation involved is +, - or :=
            then IV := IV U 0(this entry)
        end if
    end for
    while IV changes do
        for all entries
            if 0(this entry) ∈ IV
                then if operands ∉ (IV U RC)
                    then IV := IV - 0(this entry)
                end if
            end if
        end for
    end while
end procedure

procedure findcands (in(E,IV,RC), out(CANDS))
    /*
    E - set of entries being considered
    IV - set of induction values
    RC - set of region constants
    CANDS - set of instructions which are candidates for reduction
    */
end procedure
CANDS := ∅
for each entry
    if an entry has a subexpression involving a multiplication
        between an induction value and a region constant
        then CANDS := CANDS U subexpression
    end if
end for
end procedure

procedure computeaffect (in(E,IV,RC),out(AFCT))
/*
E - set of entries being considered
IV - set of induction values
RC - set of region constants
AFCT - set of all induction values and region constants which can affect the value of a given induction value
*/
for all i ∈ IV
    AFCT(i) := i
end for
for each entry
    if 0(this entry) ∈ IV
        then AFCT(this entry) := AFCT(this entry) U all input values and constant values for this entry
    end if
end for
while AFCT is changing do
    for iv ∈ IV
        AFCT(iv) := AFCT(iv) U U AFCT(j)
        j∈AFCT(iv) ∩ IV
    end for
end while
end procedure

procedure reduce (in (IV,RC,CANDS,AFCT,E,T),out(E,T))
/*
IV - set of induction values
RC - set of region constants
CANDS - set of reduction candidates
AFCT - set of values that can affect a given induction value
E - set of entries in the body T
T - interface entry for the loop
C(x) contains a list of constants for which temporaries must be maintained where x ∈ (IV U RC)
*/
for all x ∈ (IV U RC)
    C(x) := ∅
end for
Figure 4.9. Continued
for \( p \in \text{CANDS} \) with \( x \in \text{IV} \) and \( c \in \text{RC} \)
for \( y \in \text{AFCT}(x) \)
  \( C(y) := C(y) \cup c \)
end for
end for
for \( x \in (\text{IV} \cup \text{RC}) \) such that \( C(x) \neq \emptyset \)
  for \( c \in C(x) \)
    \( \text{ta}(x,c) := \text{newtemporary} \)
    insert initialization instruction of the form
    \( \text{ta}(x,c) := x \ast c \) before interface entry
  end for
end for
for each entry do
  /* insert an entry to update new induction value */
  if \( \text{TYPE}(\text{this entry}) = \text{assign} \) and \( 0(\text{this entry}) \in \text{IV} \) and
    \( C(0(\text{this entry})) \neq \emptyset \)
  then for \( c \in C(0(\text{this entry})) \)
    insert assignment entry after this entry with
    the new temporary and same operation and
    temporaries involved in \( C \)
  end for
  /* eliminate loop index value if possible */
  else if \( \text{TYPE}(\text{this entry}) = \text{condition} \) where \( x \in \text{IV} \) and \( k \in \text{RC} \)
  then if there is no temporary for \( \text{ta}(c,k) \) or \( \text{ta}(k,c) \)
    then \( \text{ta}(k,c) := \text{newtemporary} \)
    insert initialization instruction before
    interface entry
    end if
  replace condition with new temporary and new constant
  end if
end if
end for
/* replace instructions involving multiplication with an
appropriate temporary value */
for \( p \in \text{CANDS} \) let \( x \in \text{IV} \) and \( c \in \text{RC} \)
  replace right side of \( p \) with \( \text{ta}(x,c) \), the temporary holding the
  new value and update the data flow information
end for
end procedure

Figure 4.9. Continued

The work involved in performing the strength reduction algorithm
can be given by the following expression:
\[ W_{sr} = c_1 + \sum_{i=1}^{17} c_2 + \sum_{i=1}^{17} c_3 + \sum_{j=1}^{17} E + \sum_{j=1}^{17} (c_5 + \sum_{j=1}^{17} c_6) + \sum_{j=1}^{17} c_7 + \sum_{j=1}^{17} c_8 + \sum_{i=1}^{17} \sum_{j=1}^{17} c_9 + \sum_{i=1}^{17} c_{10} + \sum_{i=1}^{17} c_{11} + \sum_{j=1}^{17} (c_{12} + \sum_{j=1}^{17} c_{13}) + \sum_{i=1}^{17} \sum_{j=1}^{17} c_{14} + \sum_{i=1}^{17} (c_{15} + \sum_{j=1}^{17} c_{16}) + \sum_{i=1}^{17} c_{17} \]

c_i, for 1 \leq i \leq 17, are the constant time needed for the various procedures in the algorithm for strength reduction. This expression is a worst case because of the while loop in the procedures findiv and computeaffect. These while loops in the worst case have to search all other entries or induction values for every given entry or induction value. It is anticipated that for most programs, \( W_{sr} \sim O(|E|^2) \).

**Common subexpression elimination**

The common subexpression elimination transformation attempts to eliminate those subexpressions which compute the same value. The calculation of the common subexpression can be moved to a location in the IFT such that the value need only be computed once. Scalar propagation can be incorporated into this transformation (Hecht 1977) but is not in this discussion.

An example of common subexpression elimination is shown in Figure 4.10. The statement (2) is eliminated since the value calculated is also calculated by statement (1) even though the names are different. Statement (4) is simplified since the entire expression is already
calculated by statement (3). Statement (5) has the name \( u \) changed to an \( a \) since statement (1) is now calculating the value used by this statement. Thus the program segment appears as shown in (b).

(1) \( a := b + c \) 
    \[
    \begin{align*}
    \quad & \quad & \\
    \quad & \quad & \\
    \quad & \quad & \\
    \end{align*}
    \]

(2) \( u := c + b \) 
    \[
    \begin{align*}
    \quad & \quad & \\
    \quad & \quad & \\
    \quad & \quad & \\
    \end{align*}
    \]

(3) \( d := c \times a \) 
    \[
    \begin{align*}
    \quad & \quad & \\
    \quad & \quad & \\
    \quad & \quad & \\
    \end{align*}
    \]

(4) \( b := (b + c) \times c \) 
    \[
    \begin{align*}
    \quad & \quad & \\
    \quad & \quad & \\
    \quad & \quad & \\
    \end{align*}
    \]

(5) \( r := u \times f \) 
    \[
    \begin{align*}
    \quad & \quad & \\
    \quad & \quad & \\
    \quad & \quad & \\
    \end{align*}
    \]

(a) before

(b) after

Figure 4.10. Example of common subexpression elimination

The advantages of common subexpression elimination are that some redundant computations are eliminated and instruction space is saved by entries being deleted. This can be even more readily seen in Figure 4.11 where the common subexpression involves elements of a vector or data stream. This may save resources although the same execution time results if there are sufficient resources.
The disadvantage of common subexpression elimination is that the execution time on parallel machines of program segments which have had common subexpressions eliminated may never be decreased, but may, in fact, be increased due to added data dependencies. This is shown in Figure 4.12. Before the transformation is applied, the statements (1) and (2) can be executed in parallel in two time steps if statement (2) has its syntax tree balanced. After eliminating common subexpressions, three time steps are required instead of two. Therefore, in a parallel environment, common subexpression elimination needs to be applied with discretion.

Figure 4.12. Common subexpression elimination increases parallel execution time
The algorithm for the elimination of common subexpressions is presented in Figure 4.13. The general method employed is due to Loveman and Faneuf (Loveman and Faneuf 1975). A call to the algorithm is made with a statement of the form \texttt{commonsubexpression(in(E,T),out(E,T))} where \( T \) is an interface entry or a procedure/function entry and \( E = E_1, \ldots, E_n \) is the set of entries in the body of \( T \). The procedure \texttt{commonsubexpression} first looks at a given entry and extracts all subexpressions. Then for each subexpression, it finds where the first operand is defined and finds all uses of the first operand in the block. Each of these uses is checked to see if a subexpression with the same operation and same operands is involved. If one is found, the subexpression is redundant and is replaced. If the form of the entry is now \( x := y \), the value of \( y \) is propagated in order to find more common subexpressions. This procedure is done for all entries in the block.

```plaintext
procedure commonsubexpression (in(E,T),out(E,T))
/*
 E - set of entries in the body of T
 T - interface entry or procedure/function entry
 */
/* for each entry in the block do */
for i=1 to |E| do
  /* get all subexpressions in entry \( E_i \) */
  expr := getexpression(in(TREE(Ei)))
  /* for each subexpression in this entry do */
  for (op,opl,op2,subtree) e expr do
    cse := 0
    st := def(op1,Ei)(1)
    /* for each use of the first value of subexpression, 
    see if same subexpression is involved */
    for stt e (use(op1,st) - Ei) do
      /* get the second operand of an expression which involves 
      op1, op and op2 */
      pexpr := getoperations(in(op1,op,op2,stt))
  pexpr := getoperations(in(op1,op,op2,stt))
Figure 4.13. Algorithm for common subexpression removal
```
for a e pexpr do
  /* if it is a common subexpression, add to cse */
  if def(op2,E_)(l) = def(a,stt)(l)
    then cse := cse U stt
  end if
end for
end for

/* for the common subexpression, create a new entry and
   replace all the other expressions with this new
temporary value and update the data flow information */
if cse ≠ Ø
  then var := get temp
  stt := create entry(in('assign',var,(op1,op,op2),E_))
  /* for all uses of subexpression, replace it with a
     new temporary value */
  for st ∈ (cse U E_)
do
    if op1 appears in TREE(st) only once
      then d := def(op1,st)(l)
          use(op1,d) := use(op1,d) - st
          I(st) := I(st) - op1
      end if
    if op2 appears in TREE(st) only once
      then d := def(op2,st)(l)
          use(op2,d) := use(op2,d) - st
          I(st) := I(st) - op2
      end if
    replace(in(var,(op1,op,op2),TREE(st)),
        out(TREE(st)))
    use(var,stt) := use(var,stt) U st
    def(var,st)(l) := stt
  /* if entry is of the form x := y, then propagate
     the value of y otherwise see if there are new
     subexpressions */
  if TREE(st) is of the form x := y
    then propagate(in(st),out(E,useless)
      if useless and not live(0(st),T)
        then if TYPE(T) ≠ (procedure or function)
            then 0(T) := 0(T) - 0(st)
        end if
        use(var,stt) := use(var,stt) - st
        E := E - st
      end if
    end if
  else ex := get expression(in(TREE(st)))
    expr := expr U ex
  end if
end for
end if
end for
end for
end procedure
Figure 4.13. Continued
There is one procedure dealing with the syntax tree which has not been
previously described. The function "getoperations" finds subexpressions
in the tree involving the given operation and given operands returning
the second operand of those expressions. If an operation is commutative,
that is also taken into account by the procedure.

The work involved in the common subexpression elimination algorithm
can be given by the following expression:

\[ W_{cs} = \sum_{i=1}^{\left|E\right|} \left( c_1 + \sum_{j=1}^{\left|\text{expr}_i\right|} \left( c_2 + \sum_{k=1}^{\left|\text{use}_j\right|} c_3 + \sum_{k=1}^{\left|\text{cse}_j\right|} c_4 + (m_k \sum_{i=1}^{\left|u_k\right|}) \right) \right) \]

\( \left|\text{expr}_i\right| \) is the number of subexpressions found in the IFT entry \( E_i \).
\( \left|\text{use}_j\right| \) is the number of uses for the first value of each subexpression
in \( \text{expr}_i \). \( n_k \) is the number of times the common subexpression appears
in a given syntax tree. \( \left|\text{cse}_j\right| \) is the number of common subexpressions
found for a given subexpression. \( m_k \) is 1 if an entry may be propagated
and 0 otherwise. \( \left|u_k\right| \) is the number of uses an entry which can be
propagated has. \( c_1, c_2, c_3, c_4 \) and \( c_5 \) are constants representing the
time for the different sections in the procedure to execute. It is
anticipated that for most programs, \( W_{cs} = O(\left|E\right| \times \left|\text{expr}\right|) \), where \( \left|\text{expr}\right| \) is
the total number of expressions in the entries, since \( \left|\text{use}_j\right|, \left|\text{cse}_j\right| \)
and \( \left|u_k\right| \) are small in comparison with \( \left|E\right| \times \left|\text{expr}\right| \).

**Invariant code motion**

The invariant code motion transformation finds those subexpressions
in a loop which yield the same result independent of the number of
iterations of the loop (a loop invariant computation) and places them
outside the loop. The intent of the transformation is to move instructions from frequently executed areas of the program to areas which are less frequently executed.

An example of invariant code motion is shown in Figure 4.14. It is assumed that a and c used in statement (1) are not defined inside the loop. The subexpression is moved outside the loop causing the expression to be evaluated one time instead of n times as is the cause if it remained inside the loop.

\[
\begin{array}{ll}
\text{(a) before} & \text{(b) after} \\
\text{do } i=1 \text{ to } n & \text{do } i=1 \text{ to } n \\
\text{.} & \text{.} \\
\text{.} & \text{.} \\
\text{.} & \text{.} \\
\text{.} & \text{.} \\
\text{tl := a \times c} & \text{x := tl + x} \\
\text{end} & \text{end}
\end{array}
\]

Figure 4.14. Example of invariant code motion

The advantages of invariant code motion are that the number of instructions executed is decreased by moving subexpressions outside the loop. The subexpression that is calculated is also available earlier for use in the loop because the calculation is done prior to entering the loop. It is possible that calculating the subexpression would cause the loop to be entered at a later time step, but this might
be compensated for by a potential saving in time (if the subexpressions had been on the critical path of the body of the loop) and a savings in total number of operations performed. The resources used would also decrease by moving the subexpression outside the loop.

The algorithm for invariant code motion appears in Figure 4.15. A call to the algorithm is made with the statement invariantremoval(in (E,T),out(E,T)) where T is an interface entry and E = E_1, ..., E_n is the set of entries in the body of T. The procedure invariantmotion goes through each IFT entry in the block. For a given entry, it finds all the subexpressions and checks to see if both the operands are invariant to the loop. If they are, the procedure move is called which creates a new IFT entry outside the loop which calculates the invariant subexpression. The subexpression inside the loop is then replaced with the temporary value and the data flow information is updated.

procedure invariantremoval (in(E,T),out(E,T))
/*
  E - set of entries in the body of T
  T - interface entry
/* for each entry in the block do */
for i=1 to |E| do
  if TYPE(E_i) ≠ (if or repeat or while)
    then leaves := getexpression(in(TREE(E_i)))
    /* for all subexpressions in entry E_i do */
    for (op,opl, op2,subtree) e leaves do
      if op ≠ :=
        /* if subexpression is invariant, move 
         outside the loop */
        then if (opl \notin 0(T)) and (op2 \notin 0(T))
            then move(in(op,opl,op2,subtree,E_i,T),
                    out(E_i,T))
      end if
  end if
end if

Figure 4.15. Algorithm for invariant code motion
end for
end if
end for
end procedure

procedure move (in(op,op1,op2,subtree,E,T), out(E,T))
/*
op - operation of invariant subexpression
op1, op2 - operands of op
subtree - location in syntax where subexpression is found
E - entry from which subexpression is to be removed
T - interface entry
*/
/* create a new IFT entry and update the data flow information */
v := gettemp
st := createentry(in('assign',v,(op1 op op2),T))
flow(in(op1,st,E,T), out(st,E,T))
flow(in(op2,st,E,T), out(st,E,T))
place(in(v,subtree,TREE(E)), out(TREE(E)))
I(E) := I(E) U v
def(v,E)(1) := T
I(T) := I(T) U v
use(v,T) := E
def(v,T)(1) := st
use(v,st) := T
end procedure

procedure flow (in(opr,st,E,T), out(st,E,T))
/*
opr - operand whose data flow information is to be updated
st - entry where opr is now used
E - entry where opr used to reside
T - interface entry
*/
/* update data flow information for a given value and entry in the IFT */
stt := def(opr,T)(1)
def(opr,st)(1) := stt
use(opr,stt) := use(opr,stt) U st
if opr only appears once in TREE(E)
then I(E) := I(E) - opr
use(opr,T) := use(opr,T) - E
if use(opr,T) = Ø
then I(T) := I(T) - opr
use(opr,stt) := use(opr,stt) - T

Figure 4.15. Continued
All the procedures/functions involving the syntax tree have been described in previous sections.

The work involved in performing the invariant code motion algorithm is given by the following expression:

\[ W_{icm} = \sum_{i=1}^{|E|} \left( c_1 + \sum_{j=1}^{|\text{expr}_i|} (c_2 + k_j \cdot c_3) \right). \]

\( c_1 \) and \( c_2 \) are the work involved in the invariant removal procedure. \( c_3 \) is the work involved in the move and flow procedures. \(|\text{expr}_i|\) is the number of subexpressions in entry \( E_i \) and \( k_j \) is 1 if the \( j \)th subexpression of entry \( E_i \) is invariant and 0 otherwise. It is anticipated that for most programs, \( W_{icm} \sim O(|E| \cdot |\text{expr}|) \), where \(|\text{expr}|\) is the number of subexpressions in the entries.

**Induction value removal**

The induction value removal transformation attempts to remove most basic induction values which appear in a loop. A loop induction value is one whose value is changed in the loop only by instructions which increment an induction value by a constant amount; i.e., instructions of the form \( i := i + 1 \) or \( i := j + 3 \) where \( j \) is another induction value. Often these values are the ones used as counters in a loop.
In many cases, functions of these values are also computed, and it is possible to eliminate an induction value and calculate its value as a function of some other induction value.

An example of induction value removal is shown in Figure 4.16. k is an induction value and may be eliminated and its uses replaced with a function of i, another induction value.

```
  k := 101
  do i=1 to 100
     k := k - 1
     a(i) := b(k)
  end

(a) before

  do i=1 to 100
     a(i) := b(k)
  end

(b) after
```

Figure 4.16. Induction value removal

The advantages of induction value removal are that data dependencies are reduced and values and entries are eliminated. This can be especially important in parallel machines. In the example in Figure 4.16, the iterations of the loop are not independent of one another since k is computed in every iteration in (a). But once the induction value k has been removed as seen in (b), the iterations of the loop may be conceptually done in parallel.
The summary of the algorithm for induction value removal appears in Figure 4.17. A full algorithm appears in Appendix B. A call to the algorithm is made with a statement of the form inductionvalue(in(E,T), out(E,T)) where T is an interface entry and E = E₁, ..., Eₙ is the set of entries in the body of T. The procedure inductionvalue first finds all the constants in the block. It then finds the condition entry for the loop and sees if it is possible to have induction values. If it is, it finds the index of the loop and its initial value. If this can be done, it finds other induction values in the loop. Finally it eliminates those induction values which it has found putting them in terms of other induction values.

procedure inductionvalue (in(E,T),out(E,T))
/*
   E - set of entries in the body of T
   T - interface entry
*/
/* find loop constants */
findrc(in(E,T),out(RC)
findcondition(in(E,T,RC),out(cando,x,constant,oper))
if cando
   then findindex(in(E,x,RC,T),out(number,cando,cl,initx))
end if
if cando
   then findsothers(in(E,x,RC),out(IV,st))
end if
if cando
   then ridinduction(in(cl,IV,st,x,number,initx,E,T),
      out(E,T))
end if
end procedure

procedure findcondition (in(E,T,RC),out(cando,index,constant,oper))
/*
   E - set of entries in the body of T
   T - interface entry
Figure 4.17. Summary of algorithm for induction value removal
RC - set of region constants

cando - boolean flag indicating if enough information is available to perform induction value removal

index - the index value of loop

constant - the constant which is compared against the induction value in the condition entry

oper - the operation involved in the conditional test

/* find the condition entry and see if it is possible to eliminate induction values in this loop */
cando := true
for all entries
    if TYPE(this entry) = condition
        then get the expression for the conditional test
            set index to the index value and constant to the constant value
            if these cannot be determined
                then cando := false
            end if
    end if
end for
end procedure

procedure findindex (in(E,x,RC,T),out(number,cando,cl,initx)) /*

E - set of entries in the body of T
x - index value of loop
RC - set of region constants
T - interface entry

number - entry which increments the index value
cando - boolean flag indicating if enough information is available about the loop to eliminate induction values

cl - constant value by which x is incremented

initx - initial value of x, if it can be found

*/

/* find the IFT entry for the loop index value and its initial value if possible */
cando := true
for all entries
    if x € 0(this entry)
        then if x was assigned elsewhere or TYPE(this entry) is not assign
            then cando := false
        else set cl to be the value x is incremented by
            if cl can be obtained
                then number := this entry
    end if
end for

Figure 4.17. Continued
end if
find initial value of x
if the initial value cannot be found
then cando := false
end if
end if
end if
end for
end procedure

procedure findothers (in(E,x,RC),out(IV,st))
/*
E - set of entries being considered
x - loop index value
RC - set of region constants
IV - set of induction values
st - table indicating where induction values are assigned */
/* find other induction values */
findiv(in(E,RC),out(IV)) for each entry
if TYPE(this entry) = assign and O(this entry) ε IV
then find the value O(this entry) is incremented
by and set it equal to c2
st(O(this entry)):=(this entry, c2)
end if
end for
end procedure

procedure ridinduction (in(cl,IV,st,x,number,initx,E,T),
out(E,T))
/*
cl - increment of loop index
IV - set of induction values
st - table indicating where induction values are assigned
x - loop index
number - entry which increments the loop index
initx - initial value of x
E - set of entries in the body of T
T - interface entry */
/* eliminate the induction values replacing them with another
induction value */
for all iv ε IV
find iv's initial value
if it can be found

Figure 4.17. Continued
then for all uses of iv in the loop
if TYPE(this entry) ≠ (if or while or repeat or forall)
then find function which relates iv with x
replace iv with that function of x
update the data flow information
end if
if all uses of iv have been replaced
then delete this entry
end if
end if
end for
end procedure

Figure 4.17. Continued

The work involved in performing the induction value removal
algorithm is given by the following expression:

\[ W_{ivr} = \sum_{i=1}^{\mid E \mid} c_1 + \sum_{i=1}^{\mid E \mid} c_2 + k_1 \sum_{i=1}^{\mid E \mid} c_3 + k_2 \sum_{i=1}^{\mid E \mid} c_4 + k_3 \sum_{i=1}^{\mid E \mid} c_6. \]

\( c_1 \) is the constant work involved in the procedure findrc. \( c_2 \) is the
constant work involved in the procedure findconstant. \( c_3, c_4, c_5 \) and
\( c_6 \) are the constant work involved with the procedures findindex, find-
others, findiv and ridinduction. \( k_1, k_2 \) and \( k_3 \) are 1 if that particular
procedure is executed and 0 otherwise. \( \mid IV \mid \) is the number of induction
values that were found which could be removed. It is anticipated that
for most programs, \( W_{ivr} \sim O(\mid E \mid + \mid IV \mid). \)

Loop decomposition

A significant amount of work has been done in the analysis and
transformation of sequential loops for processing in a parallel environ-
ment. It is the purpose of this section to briefly review some of this
work and then to describe extensions to one particular method which appears particularly amenable to data flow processing.

One approach (Millstein and Muntz 1975, Presberg and Johnson 1975) is applicable only to tightly nested loops. This means that no statement can appear between adjacent nested loops and all loops can conceptually be ended with the same statement. The programmer may structure the program in such a form, but the compiler can also restructure the program most of the time so that it is in this form. Once the program has been placed in this form, loops can be analyzed and restructured, if necessary, using one of two methods. The first is the coordinate method (Lamport 1975) which is applicable to both singly and multiply nested loops. This method determines the parallelism which can be expressed in terms of the original indices of the loops. The sequential loop data dependencies are preserved by rearranging the statements so the iterations of the loop may be executed in parallel. The second method is the hyperplane method (Lamport 1974). This method is only applicable to nested loops. In the hyperplane method, the principal restructuring is performed on the subscript expressions of the array references. The ordering of the statements in the body of the loop is left unchanged. Conceptually, instead of passing through an array in a horizontal fashion, the array is passed through at some angle.

Another approach taken (Cohagen 1973, Schneck 1972, Schneck 1975, Wedel 1975) does not change the overall structure of the loop. The statements in the body are analyzed to find those which are implicitly vector or array operations. These statements are removed from the body
of the loop, if possible, and are changed into explicit vector or array operations.

Loop decomposition has also been proposed. Kuck et al. (1972) attempt to decompose the body of a loop into several smaller loops maintaining the data dependencies between the statements. These smaller loops are then analyzed in order that they might be transformed into parallel loops directly, through forward substitution, the use of temporary arrays, or by using the computational wavefront method which is similar to the hyperplane method mentioned above. Lo (1976) analyzes the entire loop to see if all the iterations of that loop are independent. If they are independent, the loop is transformed into a parallel loop. If they are not independent, the loop is examined to see if forward substitution or saving of values in a temporary array makes the iterations independent. If these transformations still do not allow the loop to be transformed into a parallel loop, the loop is partitioned into smaller loops. These smaller loops are analyzed to see if their iterations are independent and can be done in parallel or if the iterations can be made independent through the use of the above techniques.

An example of loop decomposition by Lo's method appears in Figure 4.18. The loop is analyzed and it is found that the iterations of the loop cannot be done in parallel because statement (2) has a data dependency on statement (1), and statement (1) has a data dependency on statement (2). If the values of the array b are stored in the temporary array b' before the new values of b are calculated, the iterations are independent as depicted in (b). The forall construct indicates that the
iterations of the loop are independent. Conceptually, the body of the loop can be performed in parallel for all iterations. The values of the index are indicated in the parentheses in the forall statement.

\[
\begin{align*}
\text{do } i &= 1 \text{ to } n \\
&\begin{array}{l}
(1) \quad b(i) := a(i) + c(i+1) \\
(2) \quad c(i) := b(i+1)
\end{array} \\
&\text{end}
\end{align*}
\]

\[
\begin{align*}
&\begin{array}{l}
\text{forall } i \in (1,n) \text{ do} \\
\quad b'(i) := b(i) \\
\quad \text{end}
\end{array} \\
&\begin{array}{l}
\text{forall } i \in (1,n) \text{ do} \\
\quad b(i) := a(i) + c(i+1) \\
\quad \text{end}
\end{array} \\
&\begin{array}{l}
\text{forall } i \in (1,n) \text{ do} \\
\quad c(i) := b'(i+1) \\
\quad \text{end}
\end{array}
\end{align*}
\]

(a) before (b) after

Figure 4.18. Example of loop decomposition

The advantage of loop decomposition is that a sequential execution of successive bodies of the loop may be conceptually done in parallel. The actual gain depends on the degree to which the architecture can support the forall construct. A disadvantage of loop decomposition is that redundant calculation of the loop index may be introduced since it is possible that one loop may be decomposed into many loops by this transformation.

The approaches mentioned above are applied to the innermost loop first. If an inner loop cannot be transformed into a parallel loop, the
approaches do not try anything else on the outer loops. The reason is the architectures of certain parallel machines are not designed to take advantage of parallel loops which have nested sequential loops. A data driven machine is unique in that transformations of the outer loops yield significant reductions in execution time even though the inner loops are performed sequentially. For this type of architecture, all loops must be analyzed regardless of the type of statements that appear in the body of the loop. The algorithm given below extends in two ways the method introduced by Lo (1976). First, Lo required that any array name appear on the left side of the assignment statement only once in the body of a loop. This restriction has been eliminated in order to facilitate the transformation of high level sequential languages to a data flow language. Second, he requires that the statements in the body of the loop be only assignment statements. This restriction has also been eliminated allowing any type of statement, including compound statements, to appear in the body of the loop.

The algorithm presented here does not specify a subscript analysis routine (Lamport 1973, Lamport 1974, Lo 1976, Presberg and Johnson 1975, Schneck 1972). The inclusion of a good subscript analysis routine is essential if a transformation like loop decomposition is to succeed, but it is not the purpose here to study these routines.

A summary of the algorithm appears in Figure 4.19 while the full algorithm appears in Appendix B. A call to the algorithm is made with a statement of the form loopdecomposition(in(E,T),out(E,T)) where T is the interface entry and E = E₁, ..., Eₙ is the set of entries in the body
of T. The transformation is performed in the following way. The body of the loop is analyzed as it stands to see if the iterations are independent. This is done by analyzing the subscripts and forming a matrix, called "order", which indicates the order in which two entries need to be executed. The order is determined by the data dependencies between the entries and the interference in the use of storage if the loop is executed as a forall type loop. This interference is possible only if array storage is statically allocated prior to the execution of the forall loop. For purposes of this discussion, it is assumed that such a storage management technique is used. This matrix is analyzed for cycles. If none are found, the iterations are independent and the loop is transformed directly into a forall type loop. If cycles are found, the loop is decomposed into several loops, one loop for each cycle. An individual entry, which is not part of a cycle, forms a loop by itself. An attempt is then made to transform into a parallel loop each minor loop, corresponding to a cycle, by forward substitution of expressions, saving of values in a temporary array or changing a scalar value to an array value. This involves the search of another matrix called "try" (which is formed at the same time the order matrix is formed) to see which technique is applicable to break the cycle. In any iteration, if a component of an array is defined and subsequentially used in right context, the data dependency between these two statements may be broken by forward substitution of the expression. If a component of an array is used in right context before being redefined, the problem of interference in the use of the same storage may be solved by saving
the values of the array in a temporary array. Any scalar value defined in the loop must be changed into an array if the loop is to be transformed into a forall type loop. If the application of these techniques breaks the cycle, the minor loop is transformed into a forall type loop. Otherwise the loop has to be executed sequentially. Any minor loop which contains a single entry without a data dependency on itself is transformed into a forall type loop. If loops are nested, the method transforms innermost loops first.

procedure loopdecomposition (in(E,T),out(E,T))
/*
E - set of entries in the body of T
T - interface entry for the loop
*/
findrc(in(E,T),out(RC))
findcondition(in(E,T,RC),out(cando,st,index,constant,oper))
if cando
  then findindex(in(E,index,RC,T),out(stt,cando,cl,initx))
end if
if cando
  then El := E - {st,stt}
      independentiterations(in(El,T,index),
      out(worthtrying,result,order,try,part,cycles))
end if
if cando and worthtrying
  then transform(in(order,index,initx,cl,constant,T,
      E,T),out(E,T))
else decompose(in(part,cycles,try,order),
      out(cycles,try,order,commands))
  for subpart e part do
    independentiterations(in(subpart,T,index),
    out(worthtrying,result,order,try,subpart,cycles))
  if result
    then transform(in(order,index,initx,cl,constant,E,T),
      out(E,T))
  end if
end for
end if
end procedure

Figure 4.19. Summary of algorithm for loop decomposition
procedure independentiterations(in(E,T,index),out(worthtrying,result,order,try,part,cycles))

/*
E - set of entries being examined
T - interface entry
index - the index value for the loop
worthtrying - boolean value indicating if loop decomposition
should be performed on the loop
result - boolean value indicating if the loop was decomposed
order - matrix indicating the data dependencies between entries of E
try - matrix indicating those techniques which would be useful in breaking a cycle which appears in order
part - set of partitions for the entries in E
cycles - set of cycles contained in each partition, if there are any
*/

for each entry, Ei, in the body of the loop do
  if Ei represents an if, while, repeat or forall entry
    then do this loop for all entries in the body of the entry
  else for each value, x, in the output set do
    for each use of x, called Ej, do
      analyze relation between Ei and Ej
      case of x
        (x is defined and used in the same iteration):
          order(Ei,Ej) := true
          try(Ei,Ej) := try(Ei,Ej) U (FS,x)
        (x is used in Ej before it is redefined in Ei):
          order(Ei,Ej) := true
          try(Ei,Ej) := try(Ei,Ej) U (TS,x)
        (x is a scalar in Ei used in Ej):
          order(Ei,Ej) := true
          try(Ei,Ej) := try(Ei,Ej) U (TM,x)
        (otherwise):
          order(Ei,Ej) := true
          try(Ei,Ej) := try(Ei,Ej) U (NH,x)
      end case
    end for
  end if
end for

/* using order matrix, find the cycles and return in part the set of partitions of this loop */
partition(in(order,try),out(part,cycles,ok))

/* if a partition was found, do the following */
if ok

Figure 4.19. Continued
then if cycles = ∅
    then result := true
    else result := false
end if
    worthtrying := true
else worthtrying := false
end if
end procedure

procedure transform (in(order,index,initx,cl,constant,u,E,T),
    out(E,T))
/*
order - matrix indicating the data dependencies between the entries
initx - initial value for index value
cl - increment of index value
constant - constant value used in the conditional test
u - entry before which the new loop is to be placed
E - set of entries under consideration
T - interface entry
*/
/* reorder the entries according to their data dependencies */
topologicalsort(in(order,E),out(E))
/* create a new loop and update the data flow information */
T1 := createentry(in('forall',nil,(i,initx,cl,constant),u))
for all entries in this loop
    if entry outputs a scalar value
        then change scalar value into an array value
        update all uses of the scalar value to be an array value
    end if
    update the data flow information to conform to the new loop
end for
end procedure

procedure decompose (in(part,cycles,try,order),
    out(cycles,try,order,commands))
/*
part - set of partitions for the set of entries E
cycles - set of cycles contained in each partition
try - matrix indicating those techniques which would be useful in breaking a cycle
order - matrix indicating the data dependencies between the entries
commands - set of transformations (forward substitution, creation of temporary arrays) to be applied to a given partition
*/

Figure 4.19. Continued
for all partitions, $p$, do
   if there is a cycle in $p$
      then attempt to break cycle with temporary savings technique
      if there is still a cycle in $p$
         then attempt to break cycle with forward substitution
      end if
   end if
end for
end procedure

Figure 4.19. Continued
Three examples are now presented which show the loop decomposition
process described by the above algorithm. Consider the program segment
in Figure 4.20.

1. do $i = 1$ to $n$
2. $x := e(i) / 2$
3. $b(i) := a(i) + c(i-1) + x$
4. $c(i) := b(i)$
5. $d(i) := b(i) + e(i+1)$
6. $e(i) := d(i+1)$
7. end

Figure 4.20. Example program segment

Each value defined in the loop is considered and analyzed to find
the relation it has with all uses of that value over all iterations of
the body of the loop. During this analysis, the order and try matrices,
shown in Figure 4.21, are formed. The order matrix indicates the
order in which two entries need to be executed. A $t$ in entry order($i,j$)
indicates that entry $i$ needs to precede entry $j$ in execution. Every
time a t appears in order(i, j), an entry also appears in try(i, j).

Try(i, j) consists of a list of techniques which might be applicable in changing the order in which entry i and entry j are executed. This is determined while the loop is being analyzed for data dependencies. Each member of the list is a 2-tuple, (a, b), where a is the technique to try and b is the value which determined the order of execution. The different techniques which are tried are forward substitution (FS), saving values in temporary arrays (TS), and changing a scalar value to an array (TM).

(NH) in try(i, j) indicates that there is no hope in changing the order of execution because of an irresolvable cycle. This could occur if order(i, i) = t, also if an entry represents any statement other than an assignment statement. If the entry does represent another type of statement, forward substitution and saving values in a temporary array are not possible. As a consequence, the statement must be executed in a sequential loop.

![Order and try matrices for Figure 4.20](image)

Figure 4.21. Order and try matrices for Figure 4.20
The order matrix is analyzed for cycles and the following cycles are found to exist, (3) - (4) - (3) and (5) - (6) - (5). The loop is decomposed into three minor loops consisting of the entries \{(2)\}, \{(3),(4)\}, and \{(5),(6)\}. An attempt is made to break the cycles in the two minor loops with cycles. The try matrix is consulted and it is found that the cycle (3) - (4) - (3) can be broken through forward substitution of an expression. This is done and the cycle is broken. The cycle (5) - (6) - (5) is then considered and the try matrix is again consulted finding that the saving of values in a temporary array can break this cycle. The loop that contains only the entry (2) assigns a scalar value and the scalar value is changed to an array value so the loop can be transformed into a forall type loop. The final form of the loop appears as is shown in Figure 4.22.

```plaintext
forall i in (1,n) do
    x(i) := e(i) / 2
end
forall i in (1,n) do
    c(i) := a(i) + c(i-1) + x(i)
end
forall i in (1,n) do
    b(i) := a(i) + c(i-1) + x(i)
end
forall i in (1,n) do
    d'(i) := d(i)
end
forall i in (1,n) do
    d(i) := b(i) + e(i+1)
end
forall i in (1,n) do
    e(i) := d'(i+1)
end
```

Figure 4.22. Loop after decomposition is performed
While the resulting speedup is obvious, the bottleneck of the computation remains the sequential loop involving the calculation of c(i).

To illustrate the process of decomposing loops with any type of statement in its body, consider the example in Figure 4.23.

```
(1) do i=1 to 100
(2) b(i) := e(i) + f(i-1)
(3) do j=1 to 100
(4) c(i,j) := c(i,j-1) + a(i,j)
(5) a(i,j) := c(i,j) + d(j)
(6) b(i) := a(i,j) + b(i)
(7) d(j) := d(j) + c(i,j)
(8) end
(9) if b(i) = 0
(10) then e(i+1) := 0
(11) else e(i+1) := e(i) / b(i)
(12) end
```

Figure 4.23. Example program segment

The loops are considered starting with the innermost loop, working outward until the outer loop has been considered. The inner loop, statements (3) - (8), is analyzed first forming the order and try matrices as shown in Figure 4.24.

```
(1) order matrix (b) try matrix

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>t</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>NH,c</td>
<td>FS,c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>FS,a</td>
<td>TS,d</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>NH,b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>NH,d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 4.24. Matrices formed for inner loop
The order matrix is searched for cycles and it is found that the 
only cycles are (4) - (4) and (6) - (6). Since these are cycles within 
a single entry, they cannot be broken. Any entry that is not part of a 
cycle may be transformed into a parallel loop. Those which have cycles 
which cannot be broken have to remain as sequential loops. The order in 
which the decomposed loops are placed depends only the data dependencies 
which exist among the entries. The inner loop is transformed in a 
straightforward manner giving the program segment in Figure 4.25.

(1) do i=1 to 100
(2) b(i) := e(i) + f(i-1)
(3) do j=1 to 100
(4) c(i,j) := c(i,j-1) + a(i,j)
(5) end
(6) forall j in (1,100) do
(7) a(i,j) := c(i,j) + d(j)
(8) end
(9) do j=1 to 100
(10) b(i) := a(i,j) + b(i)
(11) end
(12) forall j in (1,100) do
(13) d(j) := d(j) + c(i,j)
(14) end
(15) if b(i) = 0
(16) then e(i+1) := 0
(17) else e(i+1) := e(i) / b(i)
(18) end

Figure 4.25. After decomposing the inner loop

The outer loop is now analyzed. The entries for the outer loop are 
(2), (3), (6), (12) and (15). Whenever another loop or if entry is 
encountered during the analysis of the outer loop, all the entries in 
the body of that construct must be analyzed within the context of the 
outer loop. These statements are analyzed and the order and try matrices 
in Figure 4.26 are obtained.
The order matrix is again analyzed for cycles. It is found that the following cycles exist, (12) - (12), (15) - (15), and (2) - (9) - (15) - (2). The loop is decomposed into four minor loops with the bodies of the loops consisting of {(3)}, {(6)}, {(12)}, and {(2), (9), (15)}. An attempt is made to break the cycles which exist in each of the loops where one exists, but the attempt fails and the final form of the loop appears in Figure 4.27.

forall i in (1,100) do
do j=1 to 100
  c(i,j) := c(i,j-1) + a(i,j)
end
end
forall i in (1,100) do
  forall j in (1,100) do
    a(i,j) := c(i,j) + d(j)
  end
end
do i=1 to 100
  forall j in (1,100) do
  end
end

Figure 4.27. Loop after decomposition on inner and outer loops
\[ d(j) := d(j) + c(i,j) \]
end
end
do i=1 to 100
b(i) := e(i) + f(i-1)
do j=1 to 100
\[ b(i) := a(i,j) + b(i) \]
end
if b(i) =0
then e(i+1) := 0
else e(i+1) := e(i) / b(i)
end

Figure 4.27. Continued

Comparing the program segments in Figure 4.23 and Figure 4.27, yields the following results. Assume that both versions of the loop are executed on a data driven machine with memory statically allocated to arrays and all operations executing in unit time. Analyzing the number of time steps necessary to execute the loop given in Figure 4.23 it is found that the loop executes in 111,000 time steps. Analyzing the number of time steps necessary to execute the loop given in Figure 4.27 it is found that it can execute in 52,350 time steps. The speedup in transforming the loop is 2.1. It can readily be seen that the doubly nested loop which has to be executed sequentially is the bottleneck in the loop in Figure 4.27. But yet, the transformation yields a speedup of over two.

Consider the program segment in Figure 4.28 which is matrix multiplication. Assume that the array \( a \) is \( 1 \times n \), the array \( b \) is \( 1 \times m \) and the array \( c \) is \( m \times n \).
do i = 1 to 1
    do j = 1 to n
        a(i,j) := 0
        do k = 1 to m
            a(i,j) := a(i,j) + b(i,k) * c(k,j)
        end
    end
end

Figure 4.28. Matrix multiplication

This program segment is analyzed using the same technique given above and it is found that the innermost loop has to be executed sequentially, but the outer two loops may be transformed into forall loops. This is done giving the program segment in Figure 4.29.

forall i in (1,1) do
    forall j in (1,n) do
        a(i,j) := 0
        do k=1 to m
            a(i,j) := a(i,j) + b(i,k) * c(k,j)
        end
    end
end

Figure 4.29. Transformed matrix multiplication

If the loop in Figure 4.28 is analyzed under the same assumptions mentioned in the previous example, it is found that it executes in O(l*m*n) time on a parallel data flow machine while the loop in Figure 4.29 executes in O(l) + O(m) + O(n) time. The assumption is made that the indices are generated in linear time for a forall loop. There has been a reduction in the order of the computation even though the innermost loop had to be executed sequentially. The order went from
a multiplicative order to an additive order. Thus the potential for large gains exists through the transforming of sequential loops to parallel loops.

The work involved in the algorithm for loop decomposition is given by the following expression:

\[
W_{ld} = c_1 + \sum_{i=1}^{|E|} c_2 + \sum_{i=1}^{|E|} c_3 + k_1 \sum_{i=1}^{|E|} c_4 + k_2 \sum_{i=1}^{|E|} c_5 + \sum_{i=1}^{|E|} c_6 + \sum_{j=1}^{|E|} c_7 + \sum_{k=1}^{|E|} c_8 + \sum_{m=1}^{|E|} c_9 + c_{10} + \sum_{i=1}^{|part|} (c_{11} + k_4 \sum_{j=1}^{|cycles_i|} c_{12} + c_{13} + k_5 \sum_{j=1}^{|cycles_i|} c_{14} + \sum_{k=1}^{|cycles_j|} c_{15} + \sum_{j=1}^{|cycles_j|} c_{16} + \sum_{k=1}^{|cycles_j|} c_{17} + \sum_{m=1}^{|cycles_j|} c_{18} + \sum_{n=1}^{|command_i|} c_{19}) + c_{20} + k_7 \sum_{j=1}^{|command_i|} c_{21} + \sum_{k=1}^{|command_j|} c_{22} + \sum_{j=1}^{|E|} (c_{23} + \sum_{k=1}^{|E|} c_{24} + \sum_{j=1}^{|E|} c_{25}).
\]

\(c_i\), \(1 \leq i \leq 25\), are the constant times necessary to perform the different procedures of loop decomposition. \(|E|\) is the number of entries in the body of the loop. \(|E_i|\) and \(|E_k|\) are the number of entries in the body of the interface entry \(E_i\) or \(E_k\). \(|O_j|\) is the number of values in the output set for entry \(E_j\). \(|u_1|\) is the number of uses of a value in entry \(E_1\).
|part| is the number of partitions. |cyclesₐ| is the number of cycles in the ith partition. |cycleₐ| is the number of entries that are contained in the given cycle. |commandₐ| is the number of commands processed for a given partition. |Iₐ| is the number of values in the input set for entry Eᵢ. kᵢ, 1≤i≤7, is 1 if a particular part of the algorithm is performed and 0 otherwise. It is anticipated that for most programs, W₁d ~ O(|part|*|E|³) since |O_k| and |u_m| are very small in comparison with |E|.

Loop fusion

The loop fusion transformation takes two separate loops and determines if they can be combined into a single loop. The conditions for two loops to be combined are that each loop must be executed the same number of times, if one loop executes, the other must also execute and finally that computations in either loop do not depend on some computation in the other. These criteria can be relaxed for a particular situation.

An example of loop fusion appears in Figure 4.30. Since the two loops in (a) are independent of one another and are executed the same number of times, the loop are fused giving (b).

```
do i=1 to 100
  a(i) := 0.0
  c(i) := c(i) + y + z
end

b(i) := x(i) + y + z
```

(a) before

```
do i=1 to 100
  a(i) := 0.0
  c(i) := c(i) + y + z
end

b(i) := x(i) + y + z
```

(b) after

Figure 4.30. Example of loop fusion
The advantages of loop fusion are that loop overhead is reduced by eliminating one of the loops. The size of the body of the loop in increased thus increasing the possibility that instructions are able to execute in parallel. The space needed for instructions is reduced by the elimination of a loop and there are more instructions in the body of the loop available for local optimization. In the example in Figure 4.30, the fusion of the loops causes a common subexpression to be found that otherwise would not have been found.

The algorithm for loop fusion is presented in Figure 4.31. A call to the algorithm is made with a statement of the form loopfusion(in(E,T), out(E,T)) where T is an interface entry or procedure/function entry and E = E₁, ..., Eₙ is the set of entries in the body of T. The algorithm searches the set of entries, E, for two loops. When two loops are found, they are analyzed to see if they meet the criteria given above. If the two loops meet the criteria, they are fused changing the index value of one loop, if necessary, and updating the data flow information.

procedure loopfusion(in(E,T),out(E,T))

/*
 E - set of entries in the body of T
 T - interface entry or procedure/function entry */

for i=1 to |E| do
 /* find the first loop to be considered */
 if TYPE(Eᵢ) = (repeat or while)
  then F := F₁, ..., Fᵢ be the entries in the body of Eᵢ
     findrc(in(F,Eᵢ),out(RC))
     findcondition(in(F,Eᵢ,RC),out(cando,st,index,
        constant,oper))

Figure 4.31. Algorithm for loop fusion
if cando
    then findindex(in(F,index,constant,oper),
        out(cando,a))
end if
if cando
    then for j=i+1 to |E| do
        /* find the second loop to be considered */
        if TYPE(E.) = (repeat or while)
            then G := G, ..., G be the entries in the body of E.
            findrc(in(G,E.),out(RC))
            findcondition(in(G,E.,RC),out(cando,st,
                indexl,constant,oper))
            if cando
                then findindex(in(G,indexl,constant,oper) ,
                    out(cando,al))
            end if
            if cando
                /* check if loops are executed the same 
                   number of times and are independent 
                   of each other */
                then if (a = al) and (0(E.) - I(E.) = 0)
                    then fuse(in(F,E.,G,E.,index,indexl),
                        out(F,E.,G,i.))
                    E := E - E. j
                end if
            end if
        end if
    end for
end if
end for
end procedure

procedure fuse (in(F,E1,G,E2,index,indexl),out(F,E1,G,E2))
    /*
    F - set of entries in the 1st loop
    E1 - interface entry for the 1st loop
    G - set of entries in the 2nd loop
    E2 - interface entry for the 2nd loop
    index - index value for the 1st loop
    indexl - index value for the 2nd loop
    */
if index ≠ indexl
    /* if indices are different, modify second loop so it has 
       the same index as the first loop */
    then for g ∈ G do
        if indexl ∈ I(g)
            then I(g) := I(g) - indexl U index
    end if
end if
Figure 4.31. Continued
use(index,g) := use(index1,g)
modify(in(TREE(g),index,index1),out(TREE(g)))
end if
if index1 ∈ O(g)
then O(g) := O(g) - index1 U index
def(index,g)(1) := def(index1,g)(1)
end if
use(index,E1) := use(index,E1) U use(index1,E2)
O(E2) := O(E2) - index1
I(E2) := I(E2) - index1
end for
end if
/* fuse the two loops */
O(E1) := O(E1) U O(E2)
I(E1) := I(E1) U I(E2)
F := F U G
end procedure

Figure 4.31. Continued

The procedure "modify" changes all occurrences of index to index1 in
the specified syntax tree.

The work done by the algorithm for loop fusion is given by the
following expression:

\[ W_{lf} = \sum_{i=1}^{\vert E \vert} (c_1 + k_1 \sum_{j=i+1}^{\vert E \vert} (c_2 + k_2 \sum_{k=1}^{\vert E_j \vert} c_3)) \]

\(c_1, c_2\) and \(c_3\) are the constant amount of work done by the bodies of code
in the procedure and \(k_1\) and \(k_2\) are \(1\) if a particular section of code is
executed and \(0\) otherwise. \(\vert E \vert\) is the number of entries in the body being
considered and \(\vert E_j \vert\) is the number of entries in the second loop being
considered. It is anticipated that for most programs, \(W_{lf} \sim O(\vert E \vert^3)\).
Loop unrolling

The loop unrolling transformation takes a loop and replicates its instructions so that there are fewer iterations through the loop. A loop could be unrolled completely so that the loop was not necessary at all, but this might be impractical because of the amount of instruction space this would take and the number of iterations for a loop may not be known at compile time. A compromise is made in this case and the loop is partially unrolled.

An example of loop unrolling appears in Figure 4.32. The loop that is given in unrolled once thus doubling the number of instructions that appear in the body of the loop.

```
    do i=1 to 100
        a(i) := a(i-1) + b(i)
    end

    do i=1 to 100 by 2
        a(i) := a(i-1) + b(i)
        a(i+1) := a(i) + b(i+1)
    end

(a) before  (b) after
```

Figure 4.32. Example of loop unrolling

The advantages of loop unrolling are that the total number of instructions executed by the loop is decreased because there are fewer iterations of the loop and more instructions are exposed for parallel execution and local optimization. A disadvantage of loop unrolling is that there is an increase in the amount of instruction space that is needed.
The algorithm for loop unrolling is given in Figure 4.33. A call to the algorithm is made with a statement of the form loopunroll(in(E,T), out(E,T)) where T is an interface entry or procedure/function entry and E = E₁, ..., Eₙ is the set of entries in the body of T. The algorithm examines the loop to see if enough information is available in order to calculate the number of iterations. It is assumed that there is a constant set in the compiler indicating the maximum number of times a loop will be unrolled. After this information is gathered, the loop is unrolled and new entries are generated and the data flow information is updated to reflect the changes that have taken place.

procedure loopunroll (in(E,T),out(E,T))
/*
 E - set of entries in the body of T
 T - interface entry
*/
findrc(in(E,T),out(RC))
findcondition(in(E,T,RC),out(cando,st,index,constant,oper))
if cando
 then findindex(in(E,index,constant,oper,RC),out(cando,a))
end if
if cando
 then unroll(in(a,st,E,T),out(st,E,T))
end if
end procedure

procedure findindex (in(E,index,constant,oper,RC),out(cando,a))
/*
 E - set of entries under consideration
 index - index value of loop
 constant - constant the index value is compared against in the condition entry
 oper - relational operation of condition entry
 RC - set of region constants
 cando - boolean flag indicating if enough information is available to perform loop unrolling
 a - number of times the loop is to be unrolled
*/

Figure 4.33. Algorithm for loop unrolling
found := false
cando := true
for i=1 to |E| while (cando and not found) do
   /* if this entry increments the index, find increment */
   if (TYPE(E_i) = assign) and (index ∈ O(E_i))
      then found := true
      (op,opl,op2) := getoperation(in(TREE(E_i)))
      if op = '+'
          then if op2 ∈ RC
              then if op1 = index
                  then inc := op2
                  else cando := false
              end if
          else if op1 ∈ RC
              then if op2 = index
                  then inc := op1
                  else cando := false
              end if
          else cando := false
      end if
   else cando := false
end if
end for
if cando
   /* find initial value of the index value, the constant the index value is compared against in the condition entry and the increment of the index value */
   then x := findinit(index)
y := findinit(constant)
z := findinit(inc)
   if (x ≠ ∅) and (y ≠ ∅) and (z ≠ ∅)
      /* find the number of iterations of the loop */
      then case of oper
          (#,<=,>=): num := (abs(in(x-y)) + 1) / z
          (=,=<,>): num := abs(in(x-y)) / z
          else: cando := false
          end case
      end if
   else cando := false
end if
if cando
   /* find number of times to unroll loop */
   then a := number(num)
      if a > maximum
         then cando := false
      end if
   end if
end if
end procedure
Figure 4.33. Continued
procedure unroll (in(a,st,E,T),out(st,E,T))

a - number of times to unroll the loop
st - table indicating where values are assigned in the loop
E - set of entries in the body of T
T - interface entry

n := |E| - 1
for i=1 to a do
  for j=1 to n do
    /* if entry is not a condition entry, copy the entry and update the data flow information, otherwise just update the data flow information */
    if TYPE(E_i) ≠ condition then
      u^j := copyentry(in(E_n,E_i))
      for x ∈ I(u) do
        def(x,u)(1) := st(x)
        use(x,st(x)) := use(x,st(x)) U u
      end for
      for x ∈ 0(u) do
        st(x) := u
      end for
    else for x ∈ I(E_i) do
      if def(x,E_i)(1) ≠ T then
        stt := def(x,E_i)(1)
        def(x,E_i)(1) := st(x)
        use(x,st(x)) := use(x,st(x)) U E_j
        use(x,stt) := use(x,stt) - E_j
      end if
    end if
  end for
end for
end procedure

Figure 4.33. Continued

All the procedures dealing only with the syntax tree have been previously defined.

The work done in performing the algorithm for loop unrolling can be given by the following expression:
\[ W_{1u} = \sum_{i=1}^{|E|} c_i + \sum_{i=1}^{|E|} c_2 + k_1 \sum_{i=1}^{|E|} c_3 + k_2 \sum_{i=1}^{|E|} \sum_{j=1}^{|I_j|} (c_4 + \sum_{k=1}^{|E|} c_5 + \sum_{k=1}^{|E|} c_6). \]

\( c_i, 1 \leq i \leq 6, \) is the constant amount of work involved in the different sections of the algorithm. \( |E| \) is the number of entries in the body of the loop. \( a \) is the number of times a loop is unrolled. \( |I_j| \) and \( |O_j| \) are the number of values in the input set and the output set in entry \( E_j \), \( k_1 \) and \( k_2 \) are 1 if a particular section of code is executed and 0 otherwise. It is anticipated that for most programs, \( W_{1u} \sim O(|E|^a) \) since \( |I_j| \) and \( |O_j| \) are small in comparison with \( |E| \).

**Forward substitution**

The forward substitution transformation looks for entries whose syntax trees have the form \( x := \text{expression} \) where \( x \) is a scalar or array value. When such an entry is found, the expression is substituted for the uses of \( x \) whenever possible.

An example of forward substitution appears in Figure 4.34. The expression for \( a \) is substituted into its use in (2). The expression for \( e \) is then substituted into it use in (3). Thus (b) is obtained.

(1) \( a := b * c * d \) \hspace{1cm} (a) before \hspace{1cm} a := b * c * d

(2) \( e := f * a \) \hspace{1cm} (b) after \hspace{1cm} e := f * b * c * d

(3) \( g := e + h \) \hspace{1cm} g := f * b * c * d + h

Figure 4.34. Example of forward substitution
The advantage of forward substitution is that the execution time may be decreased. This can be seen in the above example. Suppose that all the expressions could be evaluated in parallel. The expressions in (a) would take four time steps to be evaluated while the expressions in (b) would only take three time steps. The number of data dependencies is also decreased by this transformation. The disadvantages of this transformation are that there is a possible increase in the instruction space because of the substituting of the expressions. There is also the possibility that there are more redundant computations performed because of forward substitution.

The algorithm for forward substitution is given in Figure 4.35. A call to the algorithm is made with a statement of the form forward-substitution(in(E,T),out(E,T)) where T is an interface or procedure/function entry and E = E₁, ..., Eₙ is the set of entries in the body. When an assignment entry is found, if possible the expression is substituted into all the uses of the value assigned by that entry. If there are no more uses of that value and the value is not live outside that block, the entry is deleted and the value is removed from the output set of the interface entry.

```
procedure forwardsubstitution (in(E,T),out(E,T))
/*
E - set of entries in the body of T
T - interface entry or procedure/function entry
*/
o := Ø
/* for each entry in the body do */
for i=1 to |E| do
  /* if forward substitution is possible, perform the substitution */
Figure 4.35. Algorithm for forward substitution
if \( \text{TYPE}(E_i) = \text{assign} \)
then substitute\((\text{in}(i,E),\text{out}(E,\text{useless}))\)
else useless := false
end if

/* if the entry is now useless, remove it */
if useless and not live\((O(E_i),T)\)
then \(o := o \cup O(E_i)\)
\(E := E - E_i\)
end if
end for

/* if \(T\) is an interface entry, remove the useless values */
if \(\text{TYPE}(T) \neq \text{procedure or function}\)
then \(O(T) := O(T) - o\)
end if
end procedure

procedure substitute \((\text{in}(i,E),\text{out}(E,\text{useless}))\)

/*
\(i\) - index of entry to forward substitute
\(E\) - set of entries under consideration
\text{useless} - flag indicating if \(E_i\) is now useless
*/

\(x := O(E_i)\)
\(e := \text{expression on right hand side of TREE}(E_i)\)

/* for all uses of \(x\), see if it can be replaced by the expression */
for \(a \in \text{use}(x,E_i)\) do
if \(\text{TYPE}(a) \neq \text{while or repeat or if or forall}\)
then genreplace\((\text{in}(\text{name},e,\text{TREE}(a)),\text{out}(\text{TREE}(a),\text{sub},\text{left}))\)
if sub
then if type of \(x \neq \text{array}\)
then \(I(a) := I(a) - x \cup I(E_i)\)
all := true
else if left
then \(I(a) := I(a) \cup I(E_i)\)
all := false
else \(I(a) := I(a) - x \cup I(E_i)\)
all := true
end if
end if
else all := false
end if
else all := false
end if

/* if all uses in entry \(a\) were replaced, remove the use of \(a\) */
if all
then \(\text{use}(x,E_i) := \text{use}(x,E_i) - a\)
end if

Figure 4.35. Continued
end for
/* if all the uses of x have been replaced, set useless to true
otherwise set useless to false */
if use(x,E) = 0
    then useless := true
    else useless := false
end if
end procedure

Figure 4.35. Continued

The procedure "genreplace" replaces a name with an expression. There are two flags returned by the procedure. The first tells whether a substitution took place and the second flag tells whether any occurrence of x still appears in the tree. If there are still occurrences of x, x cannot be deleted from the input set.

The work involved in performing the forward substitution algorithm can be given by the following expression:

\[ W_{fs} = \sum_{i=1}^{\mid E \mid} (c_1 + k_1 \sum_{j=1}^{u_i} c_2). \]

\( c_1 \) is the constant amount of work done by the body of the procedure forwardsubstitution. \( c_2 \) is the constant amount of work done by the procedure substitute. \( \mid E \mid \) is the number of entries in the body under consideration. \( \mid u_i \mid \) is the number of uses a given value which is to be substituted has. \( k_1 \) is 1 if entry \( E_i \) can be substituted and 0 otherwise. It is anticipated that for most programs, \( W_{fc} \sim O(\mid E \mid) \).

Tree balancing

Much has also been done in the area of balancing syntax trees so they may be evaluated in parallel in fewer time steps (Baer and
Bovet 1968, Cheung 1976, Deb 1976, Kuck et al. 1972). The syntax tree is balanced to minimize the height of the tree which exposes more sub-expressions for parallel execution.

An example of tree balancing is shown in Figure 4.36. The expression shown in (a) takes three time steps to be evaluated whereas the expression shown in (b) only takes two time steps.

\[ e := (((a+b)+c)+d) \quad e := (a+b) + (c+d) \]

(a) before \hspace{1cm} (b) after

Figure 4.36. Example of tree balancing

The advantage of tree balancing is that the execution time of the balanced tree is decreased. The disadvantages are that redundant computations may be introduced and there might also be an increase in the amount of instruction space needed.

Distribution can also decrease the execution time, but if not applied with discretion, the execution time could also increase. This is shown in the example in Figure 4.37. The expression in (a) can be evaluated in four time steps but after distribution has occurred it can be evaluated in three time steps as seen in (b). But the expression in (c) can be evaluated in two time steps before distribution and it takes three time steps after distribution as seen in (d). A detailed discussion of distribution is given by Kuck (1972).

\[ f := a * (b * c * d + e) \quad f := a * b * c * d + a * e \]

(a) before \hspace{1cm} (b) after

\[ f := a * b * (c + d) \quad f := a * b * c + a * b * d \]

(c) before \hspace{1cm} (d) after

Figure 4.37. Examples of distribution and tree balancing
The algorithm for tree balancing is given in Figure 4.38 and is due to Cheung (1976). Distribution is not included in this algorithm. A call to the algorithm is made with a statement of the form \texttt{treebalance\,(in(E,T),out(E,T))} where \(T\) is an interface entry or procedure/function entry and \(E = E_1, \ldots, E_n\) is the set of entries in the body of \(T\). The algorithm considers each syntax tree and tries to balance the tree to achieve the minimum height without distribution. Two stacks are involved in this algorithm. One stack is used to maintain operators, \(o\), and the other is used to maintain operands, \(v\). Each entry in the \(o\) stack has an operator, the number of operands to which the operator is to be applied and a flag indicating if the operator represents a \(-\) or \(./\). Each entry in the \(v\) stack has an operand, a time when the operand is available (every time a subexpression is calculated, the time the subexpression is available is increased to be the maximum of the two operands) and a flag indicating if it is an operand for a \(-\) or \(./\) operator.

```plaintext
procedure treebalance\,(in(E,T),out(E,T))

/\*
  E - set of entries under consideration
  T - interface entry or procedure/function entry
/\*

for i=1 to |E| do
  /* if the entry has a syntax tree, balance it */
  if \(\text{TYPE}(E_i) \neq \text{(repeat or while or if or forall)}\)
    then balance\,(in(E_i),out(E_i))
  end if
end for
end procedure
```

```plaintext
procedure balance\,(in(E),out(E))

/\*
  E - entry whose syntax tree is to be balanced \* /
so := ""
SY := getsymbol\,(in\,(\text{TREE\,(E)}),out\,(\text{subtree}))
```

Figure 4.38. Algorithm for tree balancing
for sy ∈ SY do
    case type(sy) do
        (operator):
            flag := true
            while (not empty(v) and (top(o) ≠ sy) and flag) do
                if empty(o) or (top(o) = '()' or
                        ((top(o) ≠ '()') and (prec(sy) > prec(top(o)))))
                    then push(in((sy, l, nil), o), out(o))
                    flag := false
                else pop(in(o), out(o, (op, n, fl)))
                nterms(in(op, n, v), out(t, avail, fl, v))
                push(in((t, avail, fl), v), out(v))
            end if
        end while
        so := sy
    (operand):
        fl := check(in(so))
        push(in((sy, o, fl), v), out(v))
        if top(o) ≠ (empty or '()')
            then pop(in(o), out(o, (op, n, fl)))
            push(in((op, n+1, fl), o), out(o))
        end if
        so := '('
    (')'):
        if top(o) ≠ '('
            then pop(in(o), out(o, (op, n, fl)))
            push(in((op, n+1, fl), o), out(o))
        end if
        fl := check(in(so))
        push(in((sy, 0, fl), o), out(o))
        so := '('
    (')'):
        generate(in(')'), o, v), out(o, v))
    (nil):
        generate(in(null, o, v), out(o, v))
        pop(in(v), out(v, (t, avail, fl)))
        replace(in(t, subtree, TREE(E)), out(TREE(E)))
    end case
end for
end procedure

function check (in(so))
    /*
    so - symbol to be checked to see if it needs to be flagged
    */
    if so = '/'
        then check := '/'
    else if so = '-'
        then check := '-'
    else check := ''
    Figure 4.38. Continued
procedure generate (in(sy,o,v), out(o,v))
    /*
    sy - symbol indicating when the code has been generated
    o - operator stack
    v - operand stack
    */
    /* generate code until sy is encountered on the stack */
    while (top(o) \neq sy) do
        pop(in(o), out(o, (op, n, fl)))
        nterms(in(op, n, v), out(t, avail, fl, v))
        push(in((t, avail, fl), v), out(v))
    end while
    if sy = '('
        then pop(in(o), out(o, (op, n, fl)))
        if fl \neq ''
            then pop(in(v), out(v, (t, avail, flag)))
            push(in((t, avail, fl), v), out(v))
        end if
    end if
end procedure

procedure nterms (in(op, n, v), out(t, avail, flag, v))
    /*
    op - operator under consideration
    n - number of terms to be taken off the stack
    v - operand stack
    t - new expression tree generated
    avail - time expression is available
    flag - flag indicating - or /
    */
    /* procedure takes n terms off the v stack and generates a new
    syntax tree for the given expression also calculating
    the time when the expression is available */
    if op = '+'
        then opr := '-'
        else opr := '/'
    end if
    for i = 1 to n do
        pop(in(v), out(v, (list(i), av(i), fl(i))))
    end for
    for i = 1 to n-1 do
        sort(in(list, av, fl), out(list, av, fl))
        opl := list(1)
        op2 := list(2)
    end for
Figure 4.38. Continued
flag := ' '
if fl(1) = opr
  then if fl(2) = opr
    then x := gentree(in(op,op1,op2))
        flag := opr
    else x := gentree(in(opr,op2,op1))
    end if
  else if fl(2) = opr
    then x := gentree(in(opr,op1,op2))
    else x := gentree(in(op,op1,op2))
  end if
end if
list(l) := x
av(l) := max(av(l),av(2)) + 1
fl(l) := flag
list(2) := Ø
end for
t := list(l)
avail := av(l)
flag := fl(l)
end procedure

Figure 4.38. Continued

The work involved in performing the algorithm for tree balancing is

given by the following expression:

\[ W_{tb} = \sum_{i=1}^{|E|} (c_1 + k_1 \sum_{j=1}^{|\text{expr}_i|} \sum_{k=1}^{n_j} c_3). \]

c_1, c_2 and c_3 are the constant amount of time needed to performed the
code in the procedures for tree balancing. |E| is the number of entries
in the body under consideration. |\text{expr}_i| is the number of expressions
in entry E_i and |n_j| is the number of operands associated with the
operand in \text{expr}_i. k_1 is 1 if entry i has a syntax tree and 0 otherwise.
It is anticipated that for most programs, \( W_{tb} \sim O(|E|*|\text{expr}|) \), where
|\text{expr}| is the number of expressions in the entries, since |n_j| is small
in comparison with |\text{expr}|.
Orderings of the Transformations

The primary goal in introducing the transformations in the previous section was to reduce the execution time for a given computation. In this section, a relation is derived for the set of transformations which suggests an order in which some of the transformations should be applied to further reduce the execution time for a computation. If no relation can be found to exist between two transformations, it still may be possible to order them according to some secondary criteria. The secondary criteria of redundant computations introduced into the computation and amount of instruction space needed for the computation are derived as relations on the set of transformations. If two transformations are not related with respect to execution time, then the secondary criteria may suggest an ordering.

The assumption is made that the transformations are applied to a program in a highly structured manner. The transformations are applied only to blocks of statements in the program. If a block of statements contains other nested blocks of statements (while-do, repeat-until, if-then-else), the transformations are applied on the nested blocks first and then on the enclosing block. It is realized that by applying the transformations in such a structured manner, the optimizations may only have a local effect and some of the global usefulness of the transformations may be lost. But it is also realized that a local optimization applied on a critical path in the program could have a
global effect on the execution of the program. Once these transformations have been applied to a given block, that block becomes a single statement with respect to the enclosing block and the application of the transformations. There are only two exceptions where the transformations may affect an enclosing block. These are the transformations of invariant code motion and strength reduction. Invariant code motion moves an invariant subexpression in an inner block to the block which encloses it. Strength reduction replaces a more expensive operation, multiplication, with a less expensive operation, addition. In performing this transformation, it is necessary to initialize values which are used inside the loop, and these values have to be initialized outside the given loop thus potentially affecting the enclosing block.

The set of transformations on which the relations will be defined is given in Figure 4.39.

\[ \phi_1 \] - Constant Folding  
\[ \phi_2 \] - Scalar Propagation  
\[ \phi_3 \] - Strength Reduction  
\[ \phi_4 \] - Common Subexpression Elimination  
\[ \phi_5 \] - Invariant Code Motion  
\[ \phi_6 \] - Induction Value Removal  
\[ \phi_7 \] - Loop Decomposition  
\[ \phi_8 \] - Loop Fusion  
\[ \phi_9 \] - Loop Unrolling  
\[ \phi_{10} \] - Forward Substitution  
\[ \phi_{11} \] - Tree Balancing

Figure 4.39. Notation used for derived orderings

Let \( R \) be a real valued performance measure of interest and \( B \) denote a block of code on which the transformations are to be performed. \( \phi_1 \)
is related to \( \emptyset_j(\emptyset_i < \emptyset_j) \) if and only if there exists a \( B \) such that 
\[ R(\emptyset_j[\emptyset_i[B]]) < R(\emptyset_i[\emptyset_j[B]]) \]. Note that \( < \) is not reflexive or symmetric and even transitivity is not guaranteed.

The relations which are derived below do not take into account any interacting effects that exist among more than two of the transformations. It may be that there are relations such as \( \emptyset_i[\emptyset_j[B]] < \emptyset_k[B] \) or
\[ \emptyset_k[B] < \emptyset_i[\emptyset_j[B]] \], but these have not been considered in this discussion. Thus the results might potentially be less than optimal.

The derivation of these relations in subsequent sections is based strictly on an intuitive understanding of the effect of the transformations on data dependencies in the program. They are supported only with examples and informal discussion. The formal proofs are not attempted and are left as an area for further research.

**Partial ordering with respect to execution time**

Let \( t(B) \) be a function which returns the time needed to execute \( B \), a block of code. If there exists a \( B \) such that \( t(\emptyset_j[\emptyset_i[B]]) < t(\emptyset_i[\emptyset_j[B]]) \) then \( \emptyset_i < \emptyset_j \). This indicates that if transformation \( \emptyset_i \) is applied to the block of code \( B \) before transformation \( \emptyset_j \), the execution time is less than if transformation \( \emptyset_j \) is applied before transformation \( \emptyset_i \).

In deriving the relation, the number of processors used and the amount of memory expended is disregarded. Because of this, the transformation for common subexpression elimination is disregarded since this transformation can never reduce the execution time of a program. On the
other hand, the transformation may increase the execution time of the program by introducing another data dependency. Therefore, $\Phi_4$ is not included in this discussion.

The argument to establish the relation with respect to execution time is presented in subsequent paragraphs. Let $E$ be the set of entries in the IFT corresponding to a block $B$ to which the transformations are to be applied.

$\Phi_1$ and $\Phi_2$ are mutually unrelated. Suppose $\Phi_1$ precedes $\Phi_2$. $\Phi_1$ may replace values with their constant value. If an expression becomes a constant, this value is also folded. Thus if a constant value is folded into an entry of the form $x := y$, to which $\Phi_2$ applies, the constant value may be propagated further to all the uses of $x$. If this happens, $\Phi_2$ is no longer applicable to that entry. Suppose $\Phi_2$ precedes $\Phi_1$. $\Phi_2$ propagates scalar values to their uses. $\Phi_1$ then replaces the scalar value with a constant value. The net result is the same since $t(\Phi_1[E]) = t(\Phi_2[E])$.

$\Phi_1$ and $\Phi_3$ are mutually unrelated. $\Phi_3$ only applies to the body of a loop on expressions of the form $a \times b$ where $a$ is an induction value and $b$ is an expression of constant values. Suppose $\Phi_1$ precedes $\Phi_3$. $\Phi_1$ can only affect $\Phi_3$ through the folding of a constant value into the $b$ expression or performing some constant calculation in that expression, either of which leaves $b$ a constant expression. Suppose $\Phi_3$ precedes $\Phi_1$. $\Phi_3$ uses the constant expression $b$ as the initial value of a temporary value and as an increment for the same temporary value. When $\Phi_1$ is
performed on $\phi_3[E]$, the constant calculation is still performed. The result is $t(\phi_1[\phi_3[E]]) = t(\phi_3[\phi_1[E]])$.

$\phi_1$ and $\phi_5$ are mutually unrelated. Suppose $\phi_1$ precedes $\phi_5$. $\phi_1$ folds constants and does any constant calculation in the body of the loop. The results of $\phi_1$ are invariant to the loop. Therefore $\phi_1$ does not affect the results of $\phi_5$. Suppose $\phi_5$ precedes $\phi_1$. $\phi_5$ moves expressions which are invariant to a loop outside the loop. $\phi_5$ may move some constant expressions out of the loop since they are invariant. $\phi_1$ will perform the calculation of those constant expressions, and fold them if possible, when the enclosing block has the transformations performed on it. The result is $t(\phi_1[\phi_5[E]]) = t(\phi_5[\phi_1[E]])$.

$\phi_1$ and $\phi_6$ are mutually unrelated. $\phi_6$ requires entries of the form $y := x + c$ to appear in the body of a loop where $x$ is an induction value and $c$ is a loop invariant expression. Suppose $\phi_1$ precedes $\phi_6$. $\phi_1$ can fold values and/or perform constant calculation in the $c$ expression resulting in an expression which is still loop invariant. Suppose $\phi_6$ precedes $\phi_1$. $\phi_6$ replaces induction values with other induction values and some function of $c$, $f(c)$. $\phi_1$ is still able to fold values into $f(c)$ and/or perform constant calculation. The result being $t(\phi_1[\phi_6[E]]) = t(\phi_6[\phi_1[E]])$.

$\phi_1$ and $\phi_7$ are mutually unrelated. Suppose $\phi_1$ precedes $\phi_7$. $\phi_1$ does nothing to change the data dependencies which exist between different iterations of a loop. Therefore $\phi_1$ does not affect $\phi_7$. Suppose $\phi_7$ precedes $\phi_1$. $\phi_7$ attempts to transform sequential loops into forall type
loops. Any constant calculation that can be done before the loop is transformed can still be done once the loop is transformed. The result is $t(\phi_1[\phi_7[E]]) = t(\phi_7[\phi_1[E]])$.

$\phi_1$ and $\phi_8$ are mutually unrelated. $\phi_8$ applies only to loops inside another block. These loops appear as single entries in the body of the block. Suppose $\phi_1$ precedes $\phi_8$. $\phi_1$ does not affect any loops which appear in the body of a block. Suppose $\phi_8$ precedes $\phi_1$. $\phi_8$ only affects loops inside the body and does not affect entries to which $\phi_1$ is applicable. The result is $t(\phi_1[\phi_8[E]]) = t(\phi_8[\phi_1[E]])$.

$\phi_1$ and $\phi_9$ are mutually unrelated. Suppose $\phi_1$ precedes $\phi_9$. $\phi_1$ folds and calculates all constants in the body of a loop before the loop is unrolled. $\phi_1$ does not affect any conditions which allow a loop be to unrolled. Suppose $\phi_9$ precedes $\phi_1$. $\phi_9$ unrolls the body of a loop a given number of times. When unrolling the body of a loop, all data flow information is maintained. Thus $\phi_1$ is still able to fold and calculate constants. The result is $t(\phi_1[\phi_9[E]]) = t(\phi_9[\phi_1[E]])$.

$\phi_1$ and $\phi_{10}$ are mutually unrelated. Suppose $\phi_1$ precedes $\phi_{10}$. $\phi_1$ does not affect any expression that can be forward substituted by making it impossible to forward substitute. Suppose $\phi_{10}$ precedes $\phi_1$. $\phi_{10}$ does not affect any constant expressions. Any constant expressions can still be calculated by $\phi_1$. The result is $t(\phi_1[\phi_{10}[E]]) = t(\phi_{10}[\phi_1[E]])$.

$\phi_1 < \phi_{11}$. $\phi_1$ can affect the final form of an expression because of the constant arithmetic that may be done at compile time. $\phi_{11}$ is then
be able to balance the final form of the expression if it is applied after $\phi_1$. Consider the following example in Figure 4.40.

\begin{verbatim}
  b := 6
  x := 2 + b + c + d + e

  x := (8 + (c + d)) + e
\end{verbatim}

(a) original

(b) $\phi_{11}$ precedes $\phi_1$

\begin{verbatim}
  x := (8 + c) + (d + e)
\end{verbatim}

(c) $\phi_1$ precedes $\phi_{11}$

Figure 4.40. $\phi_1 \prec \phi_{11}$

Suppose these expressions are evaluated on a parallel machine. Let $E$ be the entries in (a). $t(\phi_1[\phi_{11}[E]]) = 3$ as shown in (b) while $t(\phi_{11}
[\phi_1[E]]) = 2$ as shown in (c). Therefore $t(\phi_{11}[\phi_1[E]]) < t(\phi_1[\phi_{11}[E]])$.

$\phi_2$ and $\phi_3$ are mutually unrelated. Suppose $\phi_2$ precedes $\phi_3$. $\phi_2$ propagates scalar values which cannot affect $\phi_3$. The scalar values can be induction values or loop invariant values, but still $\phi_3$ is unaffected. Suppose $\phi_3$ precedes $\phi_2$. $\phi_3$ can introduce entries of the form $x := y$, but these do not affect the execution time of a loop (if executed on a data flow machine). The result is $t(\phi_2[\phi_3[E]]) = t(\phi_3[\phi_2[E]])$.

$\phi_2$ and $\phi_5$ are mutually unrelated. Suppose $\phi_2$ precedes $\phi_5$. $\phi_2$ propagates values to their uses. If these values are loop invariant, they remain loop invariant after $\phi_2$ has been applied. Suppose $\phi_5$ precedes $\phi_2$. $\phi_5$ moves invariant expressions out of a loop. This can in no way affect $\phi_2$. The result is $t(\phi_2[\phi_5[E]]) = t(\phi_5[\phi_2[E]])$. 

\( \phi_2 \) and \( \phi_6 \) are mutually unrelated. \( \phi_6 \) deals with entries of the form \( x := y + c \) where \( y \) is an induction value and \( c \) is a loop invariant expression. Suppose \( \phi_2 \) precedes \( \phi_6 \). \( \phi_2 \) may propagate induction values or loop invariant values, but neither affects the form of the entry. Thus \( \phi_6 \) is still applicable to the same entries. Suppose \( \phi_6 \) precedes \( \phi_2 \). If \( \phi_6 \) eliminates an induction value, \( \phi_2 \) may not take place if it propagates an induction value. If \( \phi_2 \) propagates a loop invariant, it is able to still replace the loop invariant value in the expression where the induction value was removed. The result is \( t(\phi_2[\phi_6[E]]) = t(\phi_6[\phi_2[E]]) \).

\( \phi_2 \) and \( \phi_7 \) are mutually unrelated. Suppose \( \phi_2 \) precedes \( \phi_7 \). \( \phi_2 \) can propagate scalar values and remove those values thus allowing the loop to be immediately transformed into a forall loop. Suppose \( \phi_7 \) precedes \( \phi_2 \). \( \phi_7 \) also rids the loop of an assignment of a scalar value through forward substitution. The net result is \( t(\phi_2[\phi_7[E]]) = t(\phi_7[\phi_2[E]]) \).

\( \phi_2 \) and \( \phi_8 \) are mutually unrelated. Suppose \( \phi_2 \) precedes \( \phi_8 \). \( \phi_2 \) does not affect any entries which deal with loops. Suppose \( \phi_8 \) precedes \( \phi_2 \). \( \phi_8 \) only affects entries which deal with the loops. The result is \( t(\phi_2[\phi_8[E]]) = t(\phi_8[\phi_2[E]]) \).

\( \phi_2 \) and \( \phi_9 \) are mutually unrelated. Suppose \( \phi_2 \) precedes \( \phi_9 \). \( \phi_2 \) propagates values in the body of a loop before it is unrolled. \( \phi_2 \) cannot affect any of the conditions which allow a loop to be unrolled. Suppose \( \phi_9 \) precedes \( \phi_2 \). \( \phi_9 \) unrolls the body of a loop maintaining the data flow information. Thus \( \phi_2 \) is still able to propagate its values. The result is \( t(\phi_2[\phi_9[E]]) = t(\phi_9[\phi_2[E]]) \).
\( \varphi_2 \) and \( \varphi_{10} \) are mutually unrelated. Suppose \( \varphi_2 \) precedes \( \varphi_{10} \). \( \varphi_2 \) cannot affect the conditions which allow an expression to be forward substituted. Suppose \( \varphi_{10} \) precedes \( \varphi_2 \). \( \varphi_{10} \) does not affect the entries which \( \varphi_2 \) propagates. The result is \( t(\varphi_2[\varphi_{10}[E]]) = t(\varphi_{10}[\varphi_2[E]]) \).

\( \varphi_2 \) and \( \varphi_{11} \) are mutually unrelated. Suppose \( \varphi_2 \) precedes \( \varphi_{11} \). \( \varphi_2 \) does not affect the final form of an expression since it is substituting a named value for another named value. Suppose \( \varphi_{11} \) precedes \( \varphi_2 \). \( \varphi_{11} \) does not affect entries in any way which would not allow them to be propagated. The result is \( t(\varphi_2[\varphi_{11}[E]]) = t(\varphi_{11}[\varphi_2[E]]) \).

\( \varphi_3 \) and \( \varphi_5 \) are mutually unrelated. Suppose \( \varphi_3 \) precedes \( \varphi_5 \). \( \varphi_3 \) reduces the strength of the multiplication operator and this does not affect any expression which is loop invariant. Suppose \( \varphi_5 \) precedes \( \varphi_3 \). \( \varphi_5 \) moves expressions which are loop invariant out of the loop replacing it with a temporary value which is loop invariant. This does not affect the ability of \( \varphi_3 \) to perform strength reduction. The result is \( t(\varphi_3[\varphi_5[E]]) = t(\varphi_5[\varphi_3[E]]) \).

\( \varphi_6 \prec \varphi_3 \). The purpose of \( \varphi_3 \) is to eliminate some multiplications and replace them with additions under the assumption that multiplication is more expensive than addition. If \( \varphi_3 \) precedes \( \varphi_6 \), \( \varphi_6 \) undoes what \( \varphi_3 \) has accomplished by reintroducing multiplications. Consider the example in Figure 4.41. Assume that a is an array of 100 x 100 elements and b is an array of 200 x 100 elements.
Consider only the inner loop. Assume all operations are unit time except multiplication which takes five time units. Assume the body is executed on a parallel machine. Let E be the entries in the body of the inner loop in (a). \( t(\varnothing_3[E]) = 3 \) as shown in (b) and \( t(\varnothing_6[\varnothing_3[E]]) = 8 \) as shown in (c). Therefore \( t(\varnothing_3[E]) < t(\varnothing_6[\varnothing_3[E]]) \).

\( \varnothing_7 < \varnothing_3 \). \( \varnothing_3 \) is applied only to sequential loops since it introduces new induction values in the body of a loop. These new induction values add data dependencies to the body of a loop which cannot be removed by \( \varnothing_7 \). This indicates that \( \varnothing_7 \) should precede \( \varnothing_3 \) in order to find whether the loop needs to be executed sequentially or whether it may be executed in parallel. Consider the following example in Figure 4.42.
real array a(1:50)

\[
\begin{align*}
\text{do } i &= 1 \text{ to } 50 \\
\text{a}(i) &= i \times 4 \\
\text{end}
\end{align*}
\]

(a) Original

\[
\begin{align*}
\text{real array a(1:50)} \\
&\vdots \\
\text{do } i &= 1 \text{ to } 50 \\
\text{a}(i) &= t_1 \\
t_1 &= t_1 + 4 \\
\text{end}
\end{align*}
\]

(b) $\emptyset_3$ precedes $\emptyset_7$

\[
\begin{align*}
\text{forall } i \in (1,50) \\
a(i) &= i \times 4 \\
\text{end}
\end{align*}
\]

(c) $\emptyset_7$ precedes $\emptyset_3$

Figure 4.42. $\emptyset_7 < \emptyset_3$

Had $\emptyset_7$ been applied to (a) before $\emptyset_3$, the loop would have been transformed into a forall loop as shown in (c). But if $\emptyset_3$ is applied first, the result is (b) and it cannot be transformed into a forall loop. It is obvious that the loop as it appears in (c) executes faster than the loop as it appears in (b). Therefore $t(\emptyset_3[\emptyset_7[E]]) < t(\emptyset_7[\emptyset_3[E]])$.

$\emptyset_3$ and $\emptyset_8$ are mutually unrelated. Suppose $\emptyset_3$ precedes $\emptyset_8$. $\emptyset_3$ is concerned only with entries in the body of the loop which have a syntax tree. Interface entries for nested loops do not have a syntax tree, therefore, $\emptyset_3$ does nothing which does not allow loops to be fused. Suppose $\emptyset_8$ precedes $\emptyset_3$. $\emptyset_8$ deals with different types of entries than does $\emptyset_3$. Therefore $\emptyset_8$ does not affect $\emptyset_3$. Thus $t(\emptyset_3[\emptyset_8[E]]) = t(\emptyset_8[\emptyset_3[E]])$.

$\emptyset_3$ and $\emptyset_9$ are mutually unrelated. Suppose $\emptyset_3$ precedes $\emptyset_9$. $\emptyset_3$ reduces the strength of some operators but does not affect any conditions which allow a loop to be unrolled. Suppose $\emptyset_9$ precedes $\emptyset_3$. $\emptyset_9$ replicates the body of a loop a given number of times. $\emptyset_9$ updates the data flow information, thus does not affect $\emptyset_3$. Therefore $t(\emptyset_3[\emptyset_9[E]]) = t(\emptyset_9[\emptyset_3[E]])$. 
\( \varphi_3 \) and \( \varphi_{10} \) are mutually unrelated. Suppose \( \varphi_3 \) precedes \( \varphi_{10} \). \( \varphi_3 \) does nothing that inhibits an expression from being forward substituted since \( \varphi_3 \) updates all the data flow information. Suppose \( \varphi_{10} \) precedes \( \varphi_3 \). \( \varphi_{10} \) can only affect \( \varphi_3 \) by forward substituting induction values or loop invariant values. Neither of these affect the ability of \( \varphi_3 \) to reduce the strength of operators. Thus \( t(\varphi_3[\varphi_{10}[E]]) = t(\varphi_{10}[\varphi_3[E]]) \).

\( \varphi_3 < \varphi_{11} \). \( \varphi_3 \) can add new statements to the body of a loop that is being transformed. As such, \( \varphi_3 \) needs to be performed before \( \varphi_{11} \) so that all the expressions are in their final form so \( \varphi_{11} \) can balance the syntax tree. Consider the example in Figure 4.43.

\begin{verbatim}
  do i = 1 to 50
    a(i) := i*(4*a+c)
  end

(a) Original

  t1 := 4 * a + c
  do i = 1 to 50
    a(i) := t1
    t1 := (t1 + (4 * a)) + c
  end

(b) \( \varphi_3 \) precedes \( \varphi_{11} \)

\end{verbatim}

Figure 4.43. \( \varphi_3 < \varphi_{11} \)

If \( \varphi_3 \) is applied after \( \varphi_{11} \), the syntax trees for the temporary value \( t1 \), assigned in part (b), never has its syntax tree balanced. Assume \( E \) is the code as it appears in the body of the loop in (a). Assume that \( E \) is executed on a parallel machine with all operators except multiplication executing in unit time. Multiplication takes five units of time.
\( t(\varnothing_{11}[\varnothing_3[E]]) = 6 \) as shown in (b) while \( t(\varnothing_3[\varnothing_{11}[E]]) = 7 \) as shown in (c). Therefore \( t(\varnothing_{11}[\varnothing_3[E]]) < t(\varnothing_3[\varnothing_{11}[E]]) \).

\( \varnothing_5 \) and \( \varnothing_6 \) are mutually unrelated. Suppose \( \varnothing_5 \) precedes \( \varnothing_6 \). \( \varnothing_5 \) removes expressions which are invariant to the loop replacing the expression with a temporary value which is also invariant. Therefore \( \varnothing_5 \) does nothing to affect the removal of induction values. Suppose \( \varnothing_6 \) precedes \( \varnothing_5 \). \( \varnothing_6 \) attempts to remove induction values by making their uses a function of some other induction value. This does not affect whether an expression is invariant to the loop. Therefore \( t(\varnothing_5[\varnothing_6[E]]) = t(\varnothing_6[\varnothing_5[E]]) \).

\( \varnothing_5 \) and \( \varnothing_7 \) are mutually unrelated. Suppose \( \varnothing_5 \) precedes \( \varnothing_7 \). \( \varnothing_5 \) removes invariant expressions but their removal does not affect the independence of the iterations of the loop. Suppose \( \varnothing_7 \) precedes \( \varnothing_5 \). \( \varnothing_7 \) transforms sequential loops into forall loops. This transformation does not affect expressions which are invariant to the loop. Thus \( t(\varnothing_5[\varnothing_7[E]]) = t(\varnothing_7[\varnothing_5[E]]) \).

\( \varnothing_5 \) and \( \varnothing_8 \) are mutually unrelated. Suppose \( \varnothing_5 \) precedes \( \varnothing_8 \). \( \varnothing_5 \) deals with entries which have a syntax tree and \( \varnothing_8 \) deals with entries which have no syntax trees (interface entries). Suppose \( \varnothing_8 \) precedes \( \varnothing_5 \). \( \varnothing_8 \) does not deal with the same type of entries as does \( \varnothing_5 \). Thus \( t(\varnothing_5[\varnothing_8[E]]) = t(\varnothing_8[\varnothing_5[E]]) \).

\( \varnothing_5 \) and \( \varnothing_9 \) are mutually unrelated. Suppose \( \varnothing_5 \) precedes \( \varnothing_9 \). \( \varnothing_5 \) removes invariant expressions replacing them with an invariant temporary value. This does not affect the conditions for unrolling a loop.
\( \varnothing \) precedes \( \varnothing _5 \). \( \varnothing \) replicates code and this replication does not change the invariance of an expression. Therefore \( t(\varnothing _5[\varnothing _9[E]]) = t(\varnothing _9[\varnothing _5[E]]) \).

\( \varnothing _5 \) and \( \varnothing _{10} \) are mutually unrelated. Suppose \( \varnothing _5 \) precedes \( \varnothing _{10} \). \( \varnothing _5 \) does nothing to affect expressions which can be forward substituted.

Suppose \( \varnothing _{10} \) precedes \( \varnothing _5 \). \( \varnothing _{10} \) does nothing with invariant expressions which changes their invariance. Thus \( t(\varnothing _5[\varnothing _{10}[E]]) = t(\varnothing _{10}[\varnothing _5[E]]) \).

\( \varnothing _5 \prec \varnothing _{11} \). \( \varnothing _5 \) affects the final form of an expression by moving subexpressions outside the body of the loop replacing the subexpression with a temporary value. This decreases the length of the expression. Therefore to realize the full benefit of \( \varnothing _{11} \), \( \varnothing _5 \) should be applied before \( \varnothing _{11} \).

\( \varnothing _6 \prec \varnothing _7 \). \( \varnothing _6 \) can eliminate data dependencies from the body of a loop by eliminating some induction values. Consider the example in Figure 4.44.

\[
k := 101 \\
do \ i = 1 \ to \ 100 \\
\quad k := k - 1 \\
\quad v(i) := u(k) \\
end
\]

(a) Original

\[
k := 101 \\
do \ i = 1 \ to \ 100 \\
\quad v(i) := u(101 - i) \\
end
\]

(b) \( \varnothing _6 \) precedes \( \varnothing _7 \)

\[
do \ i = 1 \ to \ 100 \\
\quad v(i) := u(101 - i) \\
end
\]

(c) \( \varnothing _7 \) precedes \( \varnothing _6 \)

Figure 4.44. \( \varnothing _6 \prec \varnothing _7 \)
If $\phi_7$ is applied before $\phi_6$, the loop cannot be transformed into a forall loop because of the data dependency of the induction value between iterations. By applying $\phi_6$ before $\phi_7$, an induction value and its data dependency are eliminated allowing the loop to be transformed into a forall loop. It is obvious that the loop as it appears in (b) executes faster than the loop as it appears in (c). Thus $t(\phi_7[\phi_6[E]]) < t(\phi_6[\phi_7[E]])$.

$\phi_6$ and $\phi_8$ are mutually unrelated. Suppose $\phi_6$ precedes $\phi_8$. $\phi_6$ deals with assignment entries while $\phi_8$ deals with loop interface entries. $\phi_6$ does not affect $\phi_8$. Suppose $\phi_8$ precedes $\phi_6$. $\phi_8$ does nothing which affects the removal of induction values. Thus $t(\phi_6[\phi_8[E]]) = t(\phi_8[\phi_6[E]])$.

$\phi_6$ and $\phi_9$ are mutually unrelated. Suppose $\phi_6$ precedes $\phi_9$. $\phi_6$ eliminates induction values which does not affect the conditions allowing a loop to be unrolled. Suppose $\phi_9$ precedes $\phi_6$. $\phi_9$ replicates code in the body of the loop but does not introduce any new induction values. Thus $t(\phi_6[\phi_9[E]]) = t(\phi_9[\phi_6[E]])$.

$\phi_6$ and $\phi_{10}$ are mutually unrelated. Suppose $\phi_6$ precedes $\phi_{10}$. $\phi_6$ cannot affect whether or not an expression can be forward substituted. Suppose $\phi_{10}$ precedes $\phi_6$. $\phi_{10}$ can only affect $\phi_6$ by forward substituting loop invariant values or induction values, and this does not affects the removal of induction values. Thus $t(\phi_6[\phi_{10}[E]]) = t(\phi_{10}[\phi_6[E]])$.

$\phi_6 < \phi_{11}$. $\phi_6$ can affect the final form of an expression. In order to obtain the greatest benefit from $\phi_{11}$, $\phi_6$ should be performed first.

Consider the example in Figure 4.45.
\begin{align*}
  k & := 101 - a + b \\
do i & = 1 \text{ to } 100 \\
k & := k - 1 \\
v(i) & := u(k) \\
\end{align*}
(a) Original

\begin{align*}
do i & = 1 \text{ to } 100 \\
v(i) & := u(((101 - a) + b) - i) \\
\end{align*}
(c) \( \varnothing_{11} \) precedes \( \varnothing_{6} \)

\begin{align*}
do i & = 1 \text{ to } 100 \\
v(i) & := u((101 - a) + b + (b - i)) \\
\end{align*}
(b) \( \varnothing_{6} \) precedes \( \varnothing_{11} \)

Figure 4.45. \( \varnothing_{6} < \varnothing_{11} \)

If \( \varnothing_{11} \) is performed before \( \varnothing_{6} \), the syntax tree for the subscript expression in (c) is not balanced. Let \( E \) be the body of code in (a). Assume that \( E \) is executed on a parallel machine with all operators taking unit time to execute. \( t(\varnothing_{11}[\varnothing_{6}[E]]) = 4 \) as shown in (b) while \( t(\varnothing_{6}[\varnothing_{11}[E]]) = 5 \) as shown in (c). Thus \( t(\varnothing_{11}[\varnothing_{6}[E]]) < t(\varnothing_{6}[\varnothing_{11}[E]]) \).

\( \varnothing_{7} \) and \( \varnothing_{8} \) are mutually unrelated. Suppose \( \varnothing_{7} \) precedes \( \varnothing_{8} \). \( \varnothing_{7} \) transforms loops into forall loops which are not fused. Suppose \( \varnothing_{8} \) precedes \( \varnothing_{7} \). Any loops that are fused by \( \varnothing_{8} \) can be analyzed in order to determine if they can be transformed into forall loops. Thus \( t(\varnothing_{7}[\varnothing_{8}[E]]) = t(\varnothing_{8}[\varnothing_{7}[E]]) \).

\( \varnothing_{7} < \varnothing_{9} \). \( \varnothing_{9} \) is only applied to loops which are sequential. Therefore, \( \varnothing_{7} \) is applied before \( \varnothing_{9} \) to find those loops which really have to be executed sequentially.

\( \varnothing_{7} \) and \( \varnothing_{10} \) are mutually unrelated. Suppose \( \varnothing_{7} \) precedes \( \varnothing_{10} \). \( \varnothing_{7} \) transforms loops into forall loops. One of the techniques used by \( \varnothing_{7} \) is forward substitution. Suppose \( \varnothing_{10} \) precedes \( \varnothing_{7} \). \( \varnothing_{10} \) forward
substitutes values which may help in transforming a loop. But
\[ t(\varnothing_7[^7]0_10[7][E]]) = t(0_10[^7]0_7[7][E]]) \]

\[ \varnothing_7 < \varnothing_{11}. \] \( \varnothing_{11} \) has nothing to do with reducing or adding data dependencies in the body of a loop and therefore does not affect \( \varnothing_7 \). \( \varnothing_7 \) can change the form of an expression in trying to transform a loop into a forall loop. \( \varnothing_7 \) is applied before \( \varnothing_{11} \) in order to have the expressions in their final form. Consider the example in Figure 4.46.

```
do i = 1 to 100
  b(i) := c(i-1) + x + y
  c(i) := b(i)
end
```

(a) Original

```
do i = 1 to 100
  c(i) := (c(i-1) + x) + y
end
forall i in (1,100) do
  b(i) := (c(i-1) + x) + y
end
```

(c) \( \varnothing_{11} \) precedes \( \varnothing_7 \)

Figure 4.46. \( \varnothing_7 < \varnothing_{11} \)

Comparing the time to execute the body of forall loops, assuming unit execution time and execution on a parallel machine, results in
\[ t(\varnothing_{11}[\varnothing_7[E]]) = 4 \] as shown in (b) and \[ t(\varnothing_7[\varnothing_{11}[E]]) = 5 \] as shown in (c). Thus \[ t(\varnothing_{11}[\varnothing_7[E]]) < t(\varnothing_7[\varnothing_{11}[E]]) \].

\[ \varnothing_9 < \varnothing_8. \] \( \varnothing_9 \) replicates the code in the body of a loop. If a nested loop appears in the body, the loop is also replicated. Thus \( \varnothing_8 \) is able to fuse those loops. Consider the example in Figure 4.47.
do i = 1 to 100
  do j = 1 to 100
    x := b(j) / 3
    a(i) := a(i) + x * y
  end
end

(a) Original

do i = 1 to 100 by 2
  do j = 1 to 100
    x := b(j) / 3
    a(i) := a(i) + x * y
  end
end

do j = 1 to 100
  x := b(j) / 3
  a(i+1) := a(i+1) + x * y
end
end

(b) $\varnothing_9$ precedes $\varnothing_8$

(c) $\varnothing_8$ precedes $\varnothing_9$

Figure 4.47. $\varnothing_9 \prec \varnothing_8$

When $\varnothing_9$ precedes $\varnothing_8$, the loops are fused thus allowing more parallel operations to occur. Thus $t(\varnothing_8[\varnothing_9[E]]) < t(\varnothing_9[\varnothing_8[E]])$.

$\varnothing_8$ and $\varnothing_{10}$ are mutually unrelated. Suppose $\varnothing_8$ precedes $\varnothing_{10}$. $\varnothing_8$ does not affect any entries which can be forward substituted. Suppose $\varnothing_{10}$ precedes $\varnothing_8$. $\varnothing_{10}$ does not change any conditions which allow for the fusion of loops. Thus $t(\varnothing_8[\varnothing_{10}[E]]) = t(\varnothing_{10}[\varnothing_8[E]])$.

$\varnothing_8$ and $\varnothing_{11}$ are mutually unrelated. $\varnothing_8$ and $\varnothing_{11}$ do not deal with the same type of entries. They cannot affect each other. Therefore $t(\varnothing_8[\varnothing_{11}[E]]) = t(\varnothing_{11}[\varnothing_8[E]])$.

$\varnothing_9 \prec \varnothing_{10}$. Forward substitution cannot affect whether a loop is unrolled or not, but unrolling a loop may create more opportunities for
forward substitution. Consider the example in Figure 4.48.

\[
\begin{align*}
\text{do } & k = 1 \text{ to } 100 \\
& a(i,j) := b(i,k) \ast c(k,j) + a(i,j) \quad \text{end} \\
\text{(a) Original} \\
\text{do } & k = 1 \text{ to } 100 \text{ by } 2 \\
& a(i,j) := b(i,k) \ast c(k,j) + a(i,j) \\
& a(i,j) := b(i,k+1) \ast c(k+1,j) + a(i,j) \quad \text{end} \\
\text{(b) } & \varnothing_{10} \text{ precedes } \varnothing_9 \\
\text{do } & k = 1 \text{ to } 100 \text{ by } 2 \\
& a(i,j) := b(i,k+1) \ast c(k+1,j) + b(i,k) \ast c(k,j) + a(i,j) \quad \text{end} \\
\text{(c) } & \varnothing_9 \text{ precedes } \varnothing_{10}
\end{align*}
\]

Figure 4.48. \( \varnothing_9 < \varnothing_{10} \)

The original loop (a) is unrolled by two resulting in (b). If \( \varnothing_{10} \) has already been applied, there is a possible forward substitution missed. If \( \varnothing_{10} \) is done after \( \varnothing_9 \), the result is (c). Suppose \( E \) is the entry in the body of the loop. Assume the body is executed on a parallel machine with all operations executing in unit time. \( t(\varnothing_{10}[\varnothing_9[E]]) = 8 \) as shown in (c) while \( t(\varnothing_9[\varnothing_{10}[E]]) = 11 \) as shown in (b). Thus \( t(\varnothing_{10}[\varnothing_9[E]]) < t(\varnothing_9[\varnothing_{10}[E]]) \).

\( \varnothing_9 \) and \( \varnothing_{11} \) are mutually unrelated. Suppose \( \varnothing_9 \) precedes \( \varnothing_{11} \). \( \varnothing_9 \) replicates code after which \( \varnothing_{11} \) balances all the syntax trees. Suppose \( \varnothing_{11} \) precedes \( \varnothing_9 \). \( \varnothing_{11} \) balances the syntax trees after which \( \varnothing_9 \) replicates the code. The net result is \( t(\varnothing_9[\varnothing_{11}[E]]) = t(\varnothing_{11}[\varnothing_9[E]]) \).
\( \vartheta_{10} \prec \vartheta_{11} \). \( \vartheta_{10} \) affects the final form of an expression and therefore \( \vartheta_{10} \) needs to precede \( \vartheta_{11} \). Consider the example in Figure 4.49.

\[
x := y + q \\
\vdots \\
\vdots \\
z := ((x + a) + b) \\
\]

(a) Original

\[
(\text{b) } \vartheta_{10} \text{ precedes } \vartheta_{11}
\]

\[
(\text{c) } \vartheta_{11} \text{ precedes } \vartheta_{10}
\]

Figure 4.49. \( \vartheta_{10} \prec \vartheta_{11} \)

If in (a), \( \vartheta_{11} \) is applied before \( \vartheta_{10} \), the resulting expression is not balanced and as a consequence can take more time to execute than if the syntax tree is balanced. Suppose \( E \) is the code in (a). \( t(\vartheta_{11}[\vartheta_{10}[E]]) = 2 \) as shown in (b) while \( t(\vartheta_{10}[\vartheta_{11}[E]]) = 3 \) as shown in (c). Thus \( t(\vartheta_{11}[\vartheta_{10}[E]]) < t(\vartheta_{10}[\vartheta_{11}[E]]) \).

Figure 4.50 shows the graph of the relation with respect to execution time. It is clear from the above discussion that the relation could be extended to a partial ordering by imposing transitivity where necessary.

**Partial ordering with respect to redundant computation**

Let \( \vartheta_i \) and \( \vartheta_j \) be two transformations and \( B \) be a block of code on which the transformations are to be applied. Let \( r(B) \) be a function which returns the amount of redundant computation introduced by the application of transformations. If there exists a \( B \) such that
Figure 4.50. Relation with respect to execution time
The argument to establish the relation with respect to redundant computation is presented in subsequent paragraphs.

In the following list, the members of each pair of transformations are mutually unrelated: $(0_1,0_2), (0_1,0_3), (0_1,0_5), (0_1,0_6), (0_1,0_8), (0_1,0_9), (0_1,0_{11}), (0_2,0_3), (0_2,0_5), (0_2,0_6), (0_2,0_8), (0_2,0_9), (0_2,0_{11}), (0_3,0_4), (0_3,0_5), (0_3,0_6), (0_3,0_8), (0_3,0_9), (0_3,0_{11}), (0_4,0_5), (0_4,0_6), (0_4,0_{11}), (0_5,0_6), (0_5,0_9), (0_5,0_{11}), (0_6,0_9), (0_6,0_{11})$.
\((\emptyset_5, \emptyset_6), (\emptyset_5, \emptyset_8), (\emptyset_5, \emptyset_9), (\emptyset_5, \emptyset_{11}), (\emptyset_6, \emptyset_8), (\emptyset_6, \emptyset_9), (\emptyset_6, \emptyset_{11}),
(\emptyset_8, \emptyset_9), (\emptyset_8, \emptyset_{11}),\) and \((\emptyset_9, \emptyset_{11})\). For \((\emptyset_i, \emptyset_j)\) in the above list, it immediately follows that \(r(\emptyset_i[\emptyset_j[E]]) = r(\emptyset_j[\emptyset_i[E]])\) for all \(E\).

\(\emptyset_1 \prec \emptyset_4\). \(\emptyset_1\) can reveal more common subexpressions thus reducing the amount of redundant computation in a block of code. Consider the example in Figure 4.51.

\begin{align*}
x &:= 3 \\
y &:= a + x + c \\
z &:= a + 3 + b \\
\text{(a) Original} &
\end{align*}

\begin{align*}
tl &:= a + 3 \\
y &:= tl + c \\
z &:= tl + b \\
\text{(b) } \emptyset_1 \text{ precedes } \emptyset_4 &
\end{align*}

Figure 4.51. \(\emptyset_1 \prec \emptyset_4\)

If \(\emptyset_4\) is applied to (a) before \(\emptyset_1\), no common subexpressions are found. But if \(\emptyset_1\) is applied before \(\emptyset_4\), a common subexpression is found thus reducing the number of adds done from four to three. Therefore \(r(\emptyset_4[\emptyset_1[E]]) < r(\emptyset_1[\emptyset_4[E]])\).

\(\emptyset_2 \prec \emptyset_4\). \(\emptyset_2\) can reveal more common subexpressions thus reducing the number of redundant computations. Consider the example in Figure 4.52.

\begin{align*}
x &:= r \\
y &:= a + x + c \\
z &:= a + r + b \\
\text{(a) Original} &
\end{align*}

\begin{align*}
tl &:= a + r \\
y &:= tl + c \\
z &:= tl + b \\
\text{(b) } \emptyset_2 \text{ precedes } \emptyset_4 &
\end{align*}

Figure 4.52. \(\emptyset_2 \prec \emptyset_4\)

If \(\emptyset_4\) is applied to (a) before \(\emptyset_2\), no common subexpressions are found. By applying \(\emptyset_2\) before \(\emptyset_4\), more opportunities for common subexpression elimination are found. In the above example, a common subexpression is found and the number of additions are reduced from four to three. Therefore \(r(\emptyset_4[\emptyset_2[E]]) < r(\emptyset_2[\emptyset_4[E]])\).
\( \emptyset_8 \prec \emptyset_4 \). \( \emptyset_8 \) combines loops and thus increases the possibility of finding more common subexpressions. Consider the example in Figure 4.53.

\[
\begin{align*}
\text{do } & i = 1 \text{ to } 50 \\
& a(i) := x + y + a(i-1) \\
\text{end} \\
\text{do } & i = 1 \text{ to } 50 \\
& b(i) := x + y + a(i) + b(i-1) \\
\text{end}
\end{align*}
\]

(a) Original

\[
\begin{align*}
\text{do } & i = 1 \text{ to } 50 \\
& t_1 := x + y \\
& a(i) := t_1 + a(i) + b(i) \\
\text{end}
\end{align*}
\]

(b) \( \emptyset_8 \) precedes \( \emptyset_4 \)

Figure 4.53. \( \emptyset_8 \prec \emptyset_4 \)

If \( \emptyset_4 \) is performed before \( \emptyset_8 \), no common subexpressions are found. By combining the loops, a common subexpression is found. The only common subexpressions that are found are subexpressions which are invariant to both loops. Otherwise the loops could not have been fused because of data dependencies between the two loops. Thus \( r(\emptyset_4[\emptyset_8[E]]) < r(\emptyset_8[\emptyset_4[E]]) \).

\( \emptyset_9 \prec \emptyset_4 \). \( \emptyset_9 \) duplicates code and increases the possibility for finding more common subexpressions. Consider the example in Figure 4.54.

\[
\begin{align*}
\text{do } & i = 1 \text{ to } 100 \\
& a(i) := x + y + a(i-1) \\
\text{end} \\
\text{do } & i = 1 \text{ to } 100 \text{ by } 2 \\
& t_1 := x + y \\
& a(i) := t_1 + a(i-1) \\
& a(i+1) := t_1 + a(i) \\
\text{end}
\end{align*}
\]

(a) Original

(b) \( \emptyset_9 \) precedes \( \emptyset_4 \)

Figure 4.54. \( \emptyset_9 \prec \emptyset_4 \)
If \( \phi_4 \) is applied before \( \phi_9 \), no common subexpressions are found. Since \( \phi_9 \) is applied first in the above example, a common subexpression is found thus reducing the total number of additions done in the body of the loop. Therefore \( r(\phi_4[\phi_9[E]]) < r(\phi_9[\phi_4[E]]) \).

\( \phi_{11}' \prec \phi_4 \). \( \phi_{11}' \) can reassociate the values which appear on the leaves of a syntax tree. Consider the example in Figure 4.55.

\[
\begin{align*}
x &:= c * f \\
y &:= (a * (b * (c * ((d + e) * f))))
\end{align*}
\]

(a) Original

\[
\begin{align*}
x &:= c * f \\
y &:= a * b * x * (d + e)
\end{align*}
\]

(b) \( \phi_{11}' \) precedes \( \phi_4 \)

Figure 4.55. \( \phi_{11}' \prec \phi_4 \)

If \( \phi_4 \) is performed before \( \phi_{11}' \), no common subexpressions are found since c and f are not adjacent in the syntax tree. If \( \phi_{11}' \) is applied before \( \phi_4 \), a common subexpression is found because c and f are now adjacent in the syntax tree. Thus \( r(\phi_4[\phi_{11}'[E]]) < r(\phi_{11}'[\phi_4[E]]) \).

\( \phi_4 \prec \phi_{11}' \). Subexpressions which are adjacent in the original syntax tree may not be adjacent after the tree is balanced. Consider the example in Figure 4.56.
\[ \begin{align*}
    w & := ((b \ast c) + e) + a \\
    x & := b \ast c + e \\
    y & := x + z \\
    \text{(a) Original} \\
    tl & := b \ast c \\
    t2 & := tl + e \\
    w & := t2 + a \\
    x & := t2 \\
    y & := x + z \\
    \text{(b) } \emptyset_4 \text{ precedes } \emptyset_{11} \\
    tl & := b \ast c \\
    w & := tl + (e + a) \\
    x & := tl + e \\
    y & := x + z \\
    \text{(c) } \emptyset_{11} \text{, precedes } \emptyset_4 \\
\end{align*} \]

Figure 4.56. \( \emptyset_4 \prec \emptyset_{11} \)

If \( \emptyset_{11} \) is performed before \( \emptyset_4 \), \( e \) and \( a \) are associated in the syntax tree and the only common subexpression found is \( b \ast c \). If \( \emptyset_4 \) is applied before \( \emptyset_{11} \), not only is the common subexpression \( b \ast c \) found, but also \( tl + e \). Therefore \( r(\emptyset_{11}[\emptyset_4[E]]) < r(\emptyset_4[\emptyset_{11}[E]]) \).

Figure 4.57 shows the graph of the relation according to redundant computation.

It is clear from the above discussion, involving \( \emptyset_4 \) and \( \emptyset_{11} \), that the relation cannot be extended to a partial order. However, it appears that \( \emptyset_4 \prec \emptyset_{11} \) is preferred. Common subexpressions as in the first example, Figure 4.55, are believed to be less frequent than the common subexpressions in the second example, Figure 4.56. In \( \emptyset_{11} \), values are associated with each other in order to reduce the height of the syntax tree thereby broading the base of the tree. Associations are made which did not exist in the original syntax tree. It seems that since there are
Figure 4.57. Relation with respect to redundant computation
fewer associations in the unbalanced tree, \( \phi_4 \lessdot \phi_{11} \), is preferred.

Imposing \( \phi_4 \lessdot \phi_{11} \), the new relation is readily extended to a partial ordering.

**Partial ordering with respect to instruction space**

Let \( \phi_1 \) and \( \phi_j \) be two transformations and \( B \) be a block of code on which the transformations are to be applied. Let \( s(B) \) be a function which returns the amount of instruction space required by a given block of code, \( B \). If there exists a \( B \) such that \( s(\phi_j[B]) < s(\phi_1[B]) \) then \( \phi_1 \lessdot \phi_j \). This indicates that the amount of instruction space required by the computation is less if transformation \( \phi_1 \) is applied before transformation \( \phi_j \) than if the transformations are applied in the reverse order.

Any transformation which adds more instructions is not considered in this partial ordering. There are four transformations which can increase the instruction space. They are loop decomposition, loop unrolling, forward substitution and tree balancing. Loop decomposition, loop unrolling and forward substitution are not considered. As long as distribution is not performed in balancing syntax trees, \( \phi_{11} \) can be considered. As in the previous section, it is denoted by \( \phi_{11} \).

The argument to establish the relation with respect to instruction space is presented in subsequent paragraphs.

In the following list, the members of each pair of transformations are mutually unrelated: \((\phi_1, \phi_2), (\phi_1, \phi_3), (\phi_1, \phi_5), (\phi_1, \phi_6), (\phi_1, \phi_8), (\phi_2, \phi_3), (\phi_2, \phi_6), (\phi_2, \phi_8), (\phi_2, \phi_{11}), (\phi_3, \phi_4), (\phi_3, \phi_5), (\phi_3, \phi_6), (\phi_3, \phi_8)\).
(0, 0, 11), (0, 0, 5), (0, 0, 6), (0, 0, 5), (0, 0, 6), (0, 0, 11), (0, 0, 8), 
(0, 0, 11), and (0, 0, 11). For (0, 0) in the above list, it immediately 
follows that \( s(0, 0, 11, 11, E) = s(0, 0, 11, 11, E) \) for all \( E \).

\( 0, 1 \ll 0, 4 \). If \( 0, 1 \) is applied before \( 0, 4 \), more common subexpressions can 
be found reducing the number of instructions. Consider the example in 
Figure 4.58.

\[
\begin{align*}
x &:= 3 \\
y &:= a + x + c \\
z &:= a + 3 + b \\
tl &:= a + 3 \\
y &:= tl + c \\
z &:= tl + b
\end{align*}
\]

(a) Original \hspace{1cm} (b) \( 0, 1 \) precedes \( 0, 4 \)

Figure 4.58. \( 0, 1 \ll 0, 4 \)

If \( 0, 4 \) is applied before \( 0, 1 \), no common subexpressions are found and four 
additions are performed. If \( 0, 1 \) is applied before \( 0, 4 \), only three additions 
are performed. Therefore \( s(0, 4, 0, 1, E) < s(0, 1, 0, 4, E) \).

\( 0, 11, 11 \ll 0, 1 \). \( 0, 11 \), can reassociate values in the syntax tree and thus 
create the opportunity for more constant calculation. Consider the 
example in Figure 4.59.

\[
\begin{align*}
c &:= 6 \\
y &:= a * b * c * (d + e) * 3 \\
y &:= a * b * 18 * (d + e)
\end{align*}
\]

(a) Original \hspace{1cm} (b) \( 0, 11 \), precedes \( 0, 1 \)

Figure 4.59. \( 0, 11 \ll 0, 1 \)
If $\phi_1$ is applied before $\phi_{11}$, the constant cannot be calculated since $c$ and $3$ are not adjacent in the syntax tree. If $\phi_{11}$ is applied before $\phi_1$, $c$ and $3$ are associated with each other and the constant arithmetic is performed. Thus $s(\phi_1[\phi_{11}[E]]) < s(\phi_{11}[\phi_1[E]])$.

$\phi_1 <\cdot \phi_{11}$. $\phi_1$ may not be able to perform some constant calculation after the syntax tree is balanced because some of the constants may be reassigned. Consider the example in Figure 4.60.

$$y := ((2 \times 3) + 4) + a$$

(a) Original

$$y := 10 + a$$

(b) $\phi_1$ precedes $\phi_{11}$

$$y := 6 + (4 + a)$$

(c) $\phi_{11}$ precedes $\phi_1$

Figure 4.60. $\phi_1 <\cdot \phi_{11}$.

If $\phi_{11}$ is performed before $\phi_1$, the constant values are reassigned and not as much constant calculation can be performed as if $\phi_1$ is performed before $\phi_{11}$. Thus $s(\phi_{11}[\phi_1[E]]) < s(\phi_1[\phi_{11}[E]])$.

$\phi_3 <\cdot \phi_2$. $\phi_3$ can introduce statements of the form $x := ti$ which $\phi_2$ can remove. Consider the example in Figure 4.61. As seen by the following example, an instruction of the form $x := ti$ is introduced by $\phi_3$ and is not removed if $\phi_3$ follows $\phi_2$. Thus $s(\phi_2[\phi_3[E]]) < s(\phi_3[\phi_2[E]])$.

$\phi_2 <\cdot \phi_4$. $\phi_2$ can expose more common subexpressions. Consider the example in Figure 4.62.
\begin{align*}
\text{do } i & = 1 \text{ to } 100 \\
x & := j * 5 \\
\text{.} \\
a & := x + 7 \\
\text{.} \\
j & := j + 1 \\
\text{end}
\end{align*}

\begin{align*}
\text{tl} & := j * 5 \\
\text{do } i & = 1 \text{ to } 100 \\
x & := \text{tl} \\
\text{.} \\
a & := x + 7 \\
\text{.} \\
j & := j + 1 \\
tl & := tl + 5 \\
\text{end}
\end{align*}

(a) Original 
(b) $\emptyset_2$ precedes $\emptyset_3$ 
(c) $\emptyset_3$ precedes $\emptyset_2$

Figure 4.61. $\emptyset_3 \prec \emptyset_2$

\begin{align*}
x & := r \\
y & := a + x + c \\
z & := a + r + c \\
\text{(a) Original} \\
\text{tl} & := a + r \\
y & := tl + c \\
z & := tl + b \\
\text{(b) } \emptyset_2 \prec \emptyset_4
\end{align*}

Figure 4.62. $\emptyset_2 \prec \emptyset_4$

If $\emptyset_4$ is applied before $\emptyset_2$, no common subexpressions are found and four additions are performed. If $\emptyset_2$ is applied before $\emptyset_4$, a common subexpression is uncovered and only three additions are performed. Therefore $s(\emptyset_4[\emptyset_2[E]]) < s(\emptyset_2[\emptyset_4[E]])$.

$\emptyset_8 \prec \emptyset_4$. The fusion of loops allows for the possibility of more common subexpressions. These common subexpressions between the loops have to be loop invariant or the loops cannot be fused. Consider the example in Figure 4.63.
do i = 1 to 100
a(i) := x + y + a(i-1)
end

do i = 1 to 100
b(i) := x + y + b(i-1) + a(i)
end

(a) Original

Figure 4.63. $\phi_8 \prec \phi_4$

If $\phi_4$ is applied before $\phi_8$, there are no common subexpressions and five additions are performed in every iteration of the loop. Since $\phi_8$ is applied before $\phi_4$, a common subexpression is found and there are only four additions in each iteration of the loop. Therefore $s(\phi_4[\phi_8[E]]) < s(\phi_8[\phi_4[E]])$.

$\phi_{11}' \prec \phi_4$. $\phi_{11}'$ can reassociate values which appear on the leaves of the syntax tree. Consider the example in Figure 4.64.

x := c * f
y := a * b * c * (d + e) * f

(a) Original

Figure 4.64. $\phi_{11}' \prec \phi_4$

If $\phi_4$ is applied before $\phi_{11}'$, no common subexpressions are found and five multiplications are performed. If $\phi_{11}'$ is applied before $\phi_4$, a common subexpression is found because $c$ and $f$ are associated and there are only four multiplications performed. Thus $s(\phi_4[\phi_{11}'[E]]) < s(\phi_{11}'[\phi_4[E]])$. 

\[ \text{do } i = 1 \text{ to } 100 \]
\[ a(i) := x + y + a(i-1) \]
\[ \text{end} \]

\[ \text{do } i = 1 \text{ to } 100 \]
\[ b(i) := x + y + b(i-1) + a(i) \]
\[ \text{end} \]
Balancing trees can also reassociate values thus eliminating some common subexpressions which previously existed. Consider the example in Figure 4.65.

\[
\begin{align*}
    w & := b \times c + e + a \\
    x & := b \times c + e \\
    y & := x + z
\end{align*}
\]

(a) Original

\[
\begin{align*}
    t1 & := b \times c \\
    t2 & := t1 + e \\
    w & := t1 + a \\
    x & := t2 \\
    y & := x + z
\end{align*}
\]

(b) $\phi_4$ precedes $\phi_{11}$

Figure 4.65. $\phi_4 \prec \phi_{11}$

If $\phi_{11}$ is performed before $\phi_4$, e and a are associated and the only common subexpression that is found is $b \times c$. By applying $\phi_4$ before $\phi_{11}$, an extra common subexpression is found reducing the number of additions from four to three. Thus $s(\phi_{11}[E]) < s(\phi_4[\phi_{11},E])$.

The graph of the relation with respect to instruction space is shown in Figure 4.66.

Figure 4.66. Relation with respect to instruction space
It is clear from the above discussion involving $\varnothing_1$ and $\varnothing_{11}'$, and also $\varnothing_4$ and $\varnothing_{11}'$, that the relation cannot be extended to a partial order. However, for practical purposes, it is imposed that $\varnothing_1 <' \varnothing_{11}'$. This is because it is not often that constant values would be reassigned in a syntax tree causing $\varnothing_1$ not to be able to perform constant calculation. It is much more likely that $\varnothing_1$ would add constant values to the tree and enable more constant calculation to be performed. Similarly, it is imposed that $\varnothing_4 <' \varnothing_{11}'$. It is probably more likely to find longer common subexpressions than to find more common subexpressions by the reassociation of the values in the syntax tree. The result is a relation which can be extended to a partial order by imposing transitivity.

Inclusion of Partial Orderings in a Compiler

The partial orderings with respect to execution time, redundant computation and instruction space are useful in determining the order the transformations are to be applied to a program. These partial orderings attempt to minimize the effect of the criterion on which they are based.

These partial orderings can be incorporated into an optimizing compiler in the following way. The main criterion for ordering the transformations is minimizing the execution time. The transformations can first be ordered according to that particular partial order. There are several transformations, though, which have no relation with respect to the partial order based on execution time. These transformations
are ordered according to one of the secondary criteria, redundant computations or instruction space. The programmer is given the option of specifying the secondary criterion he deems most important. Those transformations which were not ordered according to execution time are now ordered according to the secondary criterion chosen by the programmer. If some of the transformations still remain unordered, the last partial ordering is chosen to find a relation. Any transformations which have no relations with respect to any of the three partial orderings can be applied in any order.

Language Extensions for the Exposure of Parallelism

In this section, extensions to the high level language are discussed which allow for higher utilization of a parallel machine, especially a data flow machine. The concepts of the forall statement (Ackerman and Dennis 1978), the stream data type (Morris and Treleaven 1975, Weng 1975) and array to scalar functions are introduced to allow more efficient execution. The flow analysis, described in Chapter III, remains basically the same for these additions to the language.

The previous sections have discussed ways in which the compiler can enhance the execution of the program through the application of transformations. With the exception of loop decomposition, these transformations speed the execution of a program by about fifty percent on the average.

Loop decomposition, on the other hand, can greatly speed the execution of the program as has been shown. But there is a drawback
with loop decomposition. This drawback arises because of the need for loop decomposition to analyze subscripts to determine if the iterations of the loop are independent. The success of the loop decomposition algorithm is determined by the subscript analysis technique. Subscript analysis techniques typically only allow simple expressions to appear as subscripts. It is not unusual to encounter in the body of a loop subscript expressions which cannot be analyzed. When this situation arises, loop decomposition cannot continue. The problem really is that the compiler does not have sufficient information available to determine if the iterations of the loop are independent. Significant parallelism can be exposed in computer programs by restructuring the algorithm through loop decomposition but the requirements for restructuring often exceed the capabilities of known optimization techniques.

Since the compiler often cannot restructure the algorithm, the programmer needs to have available in the high level language, constructs to explicitly indicate parallelism to the compiler. Three ways are discussed which allow the programmer to indicate this parallelism.

The forall statement, depending on its implementation, allows for significant reduction in the order of the computation. The intent of the forall is that the invocations of the body are independent so that, in theory, all may execute in parallel. The syntax of the forall statement and the corresponding IFT entries are shown in Figure 4.67. The input and output sets are calculated in the same manner as the iterative for statement. It is assumed that the body of the forall statement obeys the single assignment rule which states that a value
Figure 4.67. Forall statement and IFT entries

<table>
<thead>
<tr>
<th>High level statement</th>
<th>Entries in the IFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall forall cond do statement list end</td>
<td>forall forall condition entries for statement list close</td>
</tr>
</tbody>
</table>

may be assigned only once during the execution of the program. Thus, any value used on a right hand side within the body of the forall must be computed outside the forall statement. In addition, no data dependencies are allowed between the statements in the body of the forall statement. The only values that can be output from a forall statement are array values.

The implementation of the forall statement depends on the underlying architecture of the parallel machine. Possible implementations on a data flow machine include unwinding of loops by the architecture (Arvind and Gostelow 1976), recursion, or the use of special hardware functions such as compose or decompose (Rumbaugh 1977).

As a second technique, streams appear to offer some advantages when a forall statement is not applicable. The data flow analysis technique, described in Chapter III, establishes data dependencies on the array name and not on the components within the array. This is due to the complexity which arises in the analysis of subscripts in trying to determine if one computation of an array value depends on some other computation of an array value. When any component of the array is
modified, subsequent computations, which may be independent of the component being modified, cannot begin their execution until that component of the array has been modified. This is a consequence of treating all the components of an array as one entity rather than treating each individual component as a separate entity.

In order to improve this situation, a new data type is introduced in the language called streams. A stream is a sequence of scalar values which are allowed to flow through the program as scalar values. A stream must be used in a very regular fashion since there is no way to randomly access individual elements of the stream, as is the case with arrays. The arithmetic operators in the high level language are extended to operate on streams. There is also a need for the addition of functions which allow the programmer to manipulate parts of the stream or add elements on the end of the stream.

The value of a stream data type is two-fold. First, as soon as the first element of the stream has been calculated, that element is allowed to pass on to subsequent portions of the program and those portions of the program may begin to execute. This gives the program a pipelined effect because the values of the stream are piped through the body of the program. Second, when a forall statement is not applicable, streams may be used to reduce the coefficient of the order of the computation. Streams, combined with recursion, can result in a reduction in the order of the computation.

It may be of some concern if an entire program does not stream and the need arises to convert between streams and arrays. This is of
no concern since the changing of streams to arrays and arrays to streams can be done in parallel with the execution of the body of the loop. As successive values of the array are selected to form the stream, the body of the loop may begin to execute while the rest of the elements of the array are selected. When a stream value has come through the loop and needs to be changed into an array, it is appended to the array while successive elements of the stream are being calculated by the body of the loop. Very little time consumed is the transition from streams to arrays and arrays to streams.

A third technique for the higher utilization of a parallel machine is the introduction of certain functions (e.g., sum, product) which map an array or stream to a scalar value. A recursive implementation may be used to reduce the order of a computation from $O(n)$ to $O(\log_2 n)$, where $n$ is the length of the array or stream.
Methods for finding and reducing data dependencies in high level programs have been presented. These methods are applicable to a large class of parallel processors, but are especially important for data driven computers. A technique for data flow analysis was presented. This technique is useful in two ways. First, it is essential in order to be able to perform any optimizing transformations. Second, it is necessary in order to generate code for a data driven machine. This technique for data flow analysis differs from techniques previously presented because of the extra information (specific entries are specified for the uses and the definition of a value) maintained by the technique. This information is necessary for the generation of code for a highly parallel machine like a data driven machine. Eleven optimizing transformations which are typically found in optimizing compilers were then presented. These transformations helped reduce the time needed to execute a program and reduce the resources needed by the program. Algorithms were presented for each of the transformations showing that the data flow technique presented generated sufficient information to perform these transformations. The algorithm for loop decomposition was extended to allow any type of statement to appear in the body of the loop. This is important for a data driven machine because the time necessary to execute a loop may be greatly decreased by transforming a loop into a forall loop no matter the type of statements that appear in the body of the loop. An example of matrix multiplication was presented. The inner
loop of the example had to be executed sequentially but the outer two 
loops could be done as forall loops. As a result, the time necessary to 
execute the loop changed from $O(l \times m \times n)$, where $l$, $m$ and $n$ are the 
dimensions of the arrays, to $O(l) + O(m) + O(n)$. Even though the inner 
loop was sequential, the time to execute the loop went from multiplica-
tive order to additive order. Partial orderings were then derived for 
the eleven transformations studied. The main criterion was execution 
time while the secondary criteria were redundant computation and 
instruction space. The intent of the partial orderings was to minimize 
the effect of each of the different criteria. These partial orderings 
were derived with an informal argument based on an intuitive understanding 
of each of the transformations. An area for further research is to 
formulate proofs, for the relations of the partial orderings. Finally, 
three extensions to the sequential high level language were specified 
which allow the programmer to pass to the compiler information about 
the parallelism which exists in the program.

Areas for further research include the following. Some of the 
transformations presented were not especially applicable to a highly 
parallel machine like a data driven machine. There were a number of 
transformations not chosen as part of this study which may be applicable 
to highly parallel machines. The study of these transformations and their 
usefulness to data driven machines would be interesting.

This study assumed a sequential von Neumann type high level language. 
This type of language was chosen because it is typical of the type of
language that is in current use today. A study to determine the type of language needed to exploit a highly parallel machine would be useful. The type of language needed is one that tells the compiler what needs to be done, but does not specify how to do it. The compiler should make the decision of how best it might be done.
REFERENCES


Kuck, D. J. 1975. Parallel Processing of Ordinary Programs. Report UIUCDS-R-75-767, Department of Computer Science, University of Illinois at Urbana-Champaign.


ACKNOWLEDGEMENTS

I thank Professor Arthur E. Oldehoeft for the many hours spent with me in directing my research, for his encouragement and for the many insights which he has shared with me. I thank my committee members, Professors Roy J. Zingg, Robert M. Stewart, Dennis G. Kafura and Arthur V. Pohm for their help and encouragement. I also thank the data flow group for their help.

My wife Vicki has been most loving and supportive throughout this research; I thank her deeply. Finally, I thank all of my friends and colleagues for being so understanding.
APPENDIX A. GRAMMAR FOR SEQUENTIAL HIGH LEVEL LANGUAGE

The grammar for the high level language is presented below. "[" and "]" are used as metasymbols to denote at most one occurrence of what is enclosed. "{" and "}" are used as metasymbols to denote zero or more occurrences of what is enclosed.

program ::= procedure ident [ ( prmlist ) ] begin block

block ::= real decl ; block
        | integer decl ; block
        | boolean decl ; block
        | file decl ; block
        | procedure procdcl ; block
        | statement end

decl ::= array ident ( sublist ) { , ident ( sublist ) }
       | procedure procdcl
       | idlist

procdcl ::= ident ( prmlist ) begin block

sublist ::= expr : expr { , sublist }

prmlist ::= in ( idlist ) { , prmlist }
          | out ( idlist ) { , prmlist }

idlist ::= ident { , idlist }

outlist ::= ( impout { , outlist }
          | expr { , outlist }

statement ::= state { ; state }

state ::= input inlist file = ident format = format
        | output outlist file = ident format = format
        | if expr then state [ else state ]
        | repeat statement until expr
        | while expr do statement end
        | begin statement end
        | var := expr
        | procall

inlist ::= ( impin { , inlist }
          | var { , inlist }

expr ::= conjun { | expr }
conjun ::= neg { & conjun }

neg ::= [ ~ ] rel

rel ::= sum
     | sum = sum
     | sum = sum
     | sum > sum
     | sum < sum
     | sum >= sum
     | sum <= sum
     | sum < sum
     | sum <= sum

sum ::= - sum
      | term [ + term | - term ]

term ::= factor [ * factor | / factor ]

factor ::= factorl [ ** factor ]

factorl ::= ( expr )
          | number
          | var
          | funcall
          | gen

gen ::= abs ( expr )
      | sin ( expr )
      | cos ( expr )
      | tan ( expr )
      | asin ( expr )
      | acos ( expr )
      | atan ( expr )
      | sinh ( expr )
      | cosh ( expr )
      | tanh ( expr )
      | sqrt ( expr )
      | log ( expr )
      | true
      | false

var ::= ident [ ( sscrip t ) ]

sscript ::= sum { , sum}

procall ::= ident ( prglst )

funcall ::= ident ( frglst )
prglist ::= in (xlist) [], prglist
    | out (vlist) [], prglist
frglist ::= in (xlist) [], frglist
xlist ::= expr [], expr
vlist ::= var [], var
impin ::= (impin impbody
    | vlist impbody
impout ::= (impout impbody
    | xlist impbody
impbody ::= do ident = expr to expr [by expr]
format ::= F(integer, integer) [], format
    | E(integer, integer) [], format
    | I(integer) [], format
    | X(integer) [], format
    | B(integer) [], format
    | SKIP(integer) [], format
    | PAGE(integer) [], format
number ::= integer
    | real
integer ::= digit [digit]
real ::= integer .[integer]
    | . integer
digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 0
ident ::= letter [letter | digit]
letter ::= A | B | C ... X | Y | Z
APPENDIX B. ALGORITHMS FOR TRANSFORMATIONS

Strength Reduction

procedure reducestrength (in(E,T),out(E,T))
/*
E - set of entries in the body of T
T - interface entry for the loop
*/
  findrc(in(E,T),out(RC))
  findiv(in(E,RC),out(IV))
  findcands(in(E,IV,RC),out(CANDS))
  computeaffect(in(E,IV,RC),out(AFCT))
  reduce(in(IV,RC,CANDS,AFCT,E,T),out(E,T))
end procedure

procedure findrc (in(E,T),out(RC))
/*
E - set of entries in the body of T
T - interface entry for the loop
RC - set of region constants
*/
  RC := I(T) - 0(T) /* for all entries, find their constant values */
  for i = 1 to |E| do
    if TYPE(E) # (repeat or while or if or then or else)
      then c := constants(in(TREE(E)))
    end if
  end for
end procedure

procedure findiv (in(E,RC),out(IV))
/*
E - set of entries being considered
RC - set of region constants
IV - set of induction values
*/
  IV := Ø /* for all entries, find the induction values */
  for i = 1 to |E| do
    if TYPE(E) = assign
      then if operation(in(TREE(E))) = ('+' or '-' or ':=')
        then IV := IV U 0(E)
      end if
    end if
  end for
end procedure
end if
end for
oldIV := ∅
/* find other induction values */
while (oldIV ≠ IV) do
oldIV := IV
for i = 1 to |E| do
if (O(E_i) - IV) = ∅
then if (I(E_i) - (IV U RC)) ≠ ∅
then IV := IV - O(E_i)
end if
end if
end for
end while
end procedure

procedure findcands (in(E,IV,RC),out(CANDS))
/*
E - set of entries being considered
IV - set of induction values
RC - set of region constants
CANDS - set of instructions which are candidates for reduction.
    The set has the form (entry, expression) */
CANDS := ∅
/* for all entries, find entries which are candidates
for strength reduction */
for i = 1 to |E| do
if TYPE(E_i) = (repeat or while or if or then or else)
then expr := getexpression(in(TREE(E_i)))
for (op,a1,a2,subtree) ∈ expr do
if op = '*'
then ivandrc(in(a1,a2,RC,IV),out(x,c))
if x ≠ null
then CANDS := CANDS U (E_i,subtree)
end if
end if
end for
end if
end for
end procedure

procedure computeaffect (in(E,IV,RC),out(AFCT))
/*
E - set of entries being considered
IV - set of induction values
RC - set of region constants
AFCT - the set of all induction values and region constants which
    can affect the value of a given induction value */
/* find the induction values a given induction value affects */
for i ∈ IV do
    AFCT(i) := i
end for
for i = 1 to |E| do
    if TYPE(E_i) = assign
        then if O(E_i) ∈ IV
            then AFCT(O(E_i)) := AFCT(O(E_i)) U I(E_i)
            U constants(in(TREE(E_i)))
        end if
    end if
end for
change := true
while (change) do
    change := false
    for i ∈ IV do
        oafct := AFCT(i)
        for j ∈ (AFCT(i) ∩ IV) do
            AFCT(i) := AFCT(i) U AFCT(j)
        end for
        if oafct ≠ AFCT(i)
            then change := true
        end if
    end for
end while
end procedure

procedure reduce (in(IV,RC,CANDS,AFCT,E,T),out(E,T))
/*
IV - set of induction values
RC - set of region constants
CANDS - set of reduction candidates. The set has the form
    (entry, expression)
AFCT - set of values that can affect the given induction
    value
E - set of entries in the body of T
T - interface entry for the loop
*/
/*
C - a list of constants c for which a temporary value must
    be maintained
*/
for x ∈ (IV U RC) do
    C(x) := ∅
end for
/*
ta - table where temporaries assigned to a given induction
    value or a constant value is kept
*/
sta - table telling the entry where a given temporary was assigned

```plaintext
ta := ∅
sta := ∅
```

/*
add c to C(y) for each y ∈ AFCT(x) where x is an induction value in an operation which might be reduced

```plaintext
for (El,subtree) ∈ CANDS do
(op,al,a2,stree) := getexpression(in(subtree))
ivandrc(in(al,a2,RC,IV),out(x,c))
/* associate the proper constant with the induction value */
for y ∈ AFCT(x) do
C(y) := C(y) U c
end for
end for
/* create temporary values for all induction values and constants and insert an initialization entry prior to T */
for x ∈ (IV U RC) do
for c ∈ C(x) do
ta(x,c) := gettemp
insert(in('*,ta(x,c),x,c,T,null,sta),out(T,sta))
end for
end for
for i = 1 to |E| do
if TYPE(E_i) = assign
then z := 0(E_i)
/* insert entry to update temporary holding the value of multiplication */
if (z ∈ IV) and (C(z) ≠ ∅)
then (op,al,a2,stree) := getexpression(in(TREE(E_i)))
for c ∈ C(z) do
insert(in(op,ta(z,c),ta(al,c),ta(a2,c),E_i+1,T,sta),
out(E_i,T,sta))
end for
end if
/* see if condition can be replaced by another induction value */
else if TYPE(E_i) = condition
then expr := getexpression(in(TREE(E_i)))
for (op,al,a2,stree) ∈ expr do
ivandrc(in(al,a2,RC,IV),out(x,k))
if (x ≠ null) and (C(x) ≠ ∅)
then c ∈ C(x)
if ta(c,k) = ∅
then if ta(k,c) = ∅
then ta(k,c) := gettemp
insert(in(op,ta(k,c),
k,c,E_i,T,sta),
out(E_i,T,sta))
```

end if
  \( t := t_a(k,c) \)
else \( t := t_a(c,k) \)
end if
/* update the data flow information and replace induction value */
treeflow(in(ta(x,c),al,sta,E,T), out(E,T))
place(in(ta(x,c),al,TREE(E)), out(TREE(E)))
treeflow(in(t,a2,sta,E,T), out(E,T))
place(in(t,a2,TREE(E)), out(TREE(E)))
end if
end if
end for
end if
end if
end for
/* update data flow information */
for (El,subtree) \( \in \) CANDS do
  (op,al,a2,stree) := getexpression(in(subtree))
  ivandrc(in(al,a2,RC,IV),out(x,c))
treeflow(in(ta(x,c),subtree,sta,El,T), out(El,T))
place(in(ta(x,c),subtree,TREE(El)), out(TREE(El)))
end for
end procedure

procedure insert (in(op,v,opl,op2,E,T,sta),
  out(E,T,sta))
/*
  op - operator of expression to be placed in syntax tree
  v - value on left side of assignment operator
  opl, op2 - two operands for op
  E - entry after which newly created entry is to be placed
  T - interface entry
  sta - table indicating the entry a given temporary was assigned,
  if it has been assigned
*/
/* create a given entry and update data flow information */
st := createentry(in('assign',v,(opl op op2), E))
if T \( \neq \) \( \emptyset \) and (TYPE(T) \( \neq \) (procedure or function ))
  then use(v,sta) := E
      \( I(E) := I(E) \cup v \)
      \( \text{def}(v,E)(1) := st \)
  else sta(v) := st
end if
if opl \( \neq \) constant
then operandflow(in(sta,op1,st,T),out(st,T))
end if
if op2 ≠ constant
then operandflow(in(sta,op2,st,T),out(st,T))
end if
end procedure

procedure operandflow (in(sta,opr,E,T),out(E,T))
/*
sta - table indicating the entry where a given temporary was assigned
opr - value whose data flow information is to be updated
E - entry where opr is used in right context
T - interface entry of loop */
/* update data flow information */
if TYPE(T) ≠ (procedure or function)
then if sta(opr) = 0
then def(opr,E)(1) := T
use(opr,T) := E
else def(opr,E)(1) := sta(opr)
use(opr,sta(opr)) := use(opr,sta(opr)) U E
end if
else stt := def(opr,E)(1)
def(opr,E)(1) := stt
use(opr,stt) := use(opr,stt) U E
end if
end procedure

procedure treeflow (in(v,subtree,sta,E,T),out(E,T))
/*
v - temporary value which has been added to entry
subtree - instruction where v was placed
sta - table indicating where v was assigned
E - entry whose tree has been modified
T - interface entry of loop */
/* update data flow information if x only appears once in
the given tree */
stree := subtree
operand := getoperand(in(stree))
for x ∈ operand do
if x only appears once in TREE(E)
then st := def(x,E)(1)
I(E) := I(E) - x
use(x,st) := use(x,st) - E
if (use(x,st) = Ø) and (st = T)
then I(T) := I(T) - x.
end if
end if
end for
I(E) := I(E) U v
if sta(v) = ∅
    then def(v,E)(l) := T
        use(v,T) := use(v,T) U E
    else def(v,E)(l) := sta(v)
        use(v,sta(v)) := use(v,sta(v)) U E
end if
end procedure

procedure ivandrc (in(a,b,RC,IV),out(x,c))
   /*
   a,b - values which are to be checked to see which is an
      induction value and which is a constant value
   RC - set of region constants
   IV - set of induction values
   x - the induction value
   c - the constant value
   */
   /* find whether a or b is an induction value or constant */
   if (a ∈ IV) and (b ∈ RC)
      then x := a
          c := b
   else if (b ∈ IV) and (a ∈ RC)
      then x := b
          c := a
   else x := null
   end if
end if
end procedure

There are a number of procedures used in the algorithm presented
which have not been described previously and they manipulate the syntax
tree for an entry. The function "constants" finds all the constant
values in a tree and returns that set. The function "operands" returns
the first operation on the right hand side of the assignment operator,
or the assignment operator itself if there are no other operations.
The function "getexpression" returns the set of expressions which appear
at the leaves of the syntax tree. The function "operand" returns the
set of operands found in the syntax tree. The function "gettemp"
returns a new temporary name which can be used in forming a new entry. The function "createentry" creates an IFT entry of the given type placing it in the IFT at the specified location.

Induction Value Removal

procedure inductionvalue (in(E,T),out(E,T))
/*
   E - set of entries in the body of T
   T - interface entry
*/
/* find loop constants */
findrc(in(E,T),out(RC))
findcondition(in(E,T,RC),out(cando,x,constant,oper))
if cando
   then findindex(in(E,x,RC,T),out(number,cando,cl,initx))
end if
if cando
   then findothers(in(E,x,RC),out(IV,st))
end if
if cando
   then ridinduction(in(cl,IV,st,x,number,initx,E,T),
      out(E,T))
end if
end procedure

procedure findcondition (in(E,T,RC),out(cando,index,constant,oper))
/*
   E - set of entries in the body of T
   T - interface entry
   RC - set of region constants
   cando - boolean flag indicating if enough information is available
to perform induction value removal
   index - the index value of loop
   constant - the constant which is compared against the induction
   value in the condition entry
   oper - the operation involved in the conditional test
*/
/* find the condition entry and see if it is possible to eliminate
induction values in this loop */
cando := true
for i=1 to |E| do
   if TYPE(E_i) = condition
      then (oper,op1,op2) := getoperation(in(TREE(E_i)))
if op2 \in RC
then if op1 is a value
then index := op1
constant := op2
else cando := false
end if
else if op1 \in RC
then if op2 is a value
then index := op2
constant := op1
else cando := false
end if
else cando := false
end if
end if
end if
end for
end procedure

procedure findindex (in(E,x,RC,T),out(number,cando,cl,initx))

/*
E - set of entries in the body of T
x - index value of loop
RC - set of region constants
T - interface entry
number - entry which increments the index value

cando - boolean flag indicating if enough information is available about the loop to eliminate induction values
cl - constant value by which x is incremented
initx - initial value of x, if it can be found
*/

/* find the IFT entry for the loop index value and its initial value if possible */
cando := true
found := false

/* for each entry in the IFT, find the entry which assigns a value to the index value */
for i=1 to |E| do
if x \in O(E_i)
then if found or (TYPE(E_i) \neq assign)
then cando := false
else found := true
getinfo(in(E_i,RC),out(index,cl,cando))
if cando
then number := E_i
end if
/* find initial value for loop index if possible */
initx := findinit(in(x,T))

end if
procedure findothers (in(E,x,RC),out(IV,st))

/*
E - set of entries being considered
x - loop index value
RC - set of region constants
IV - set of induction values
st - table indicating where induction values are assigned
*/

/* find other induction values */
findiv(in(E,RC),out(IV))
for i=1 to |E| do
  if TYPE(E.) = assign and 0(E.) ε IV
    then getinfo(in(E.,RC),out(y,c2,cando))
      if cando
        then st(y) := (E.,c2)
      end if
  end if
end for
end procedure

procedure getinfo (in(E,RC),out(index,constant,cando))

/*
E - entry where induction values resides
RC - set of region constants
index - induction value
c2 - constant value
cando - boolean flag indicating if enough information is available to perform induction value removal
*/

/* find which operand is the index value and which is the constant value */
geoperation(in(TREE(E.i)),out(oper,op1,op2))
cando := true
if oper = ('+' or '-' )
  then if op2 ε RC
    then if op 1 is a value
      then index := op1
      constant := op2
      else cando := false
    end if
    else if op1 ε RC
      then index := op2
      constant := op1
      else cando := false
      end if
then if op2 is a value
then index := op2
constant := op 1
else cando := false
end if
else cando := false
end if
end if
if oper = '-'
then constant := - constant
end if
end if

procedure ridinduction (in(cl,IV,st,x,number,initx,E,T),
out(E,T))
/
cl - increment of loop index
IV - set of induction values
st - table indicating where induction values are assigned
x - loop index
number - entry which increments the loop index
initx - initial value of x
E - set of entries in the body of T
T - interface entry
/ * eliminate the induction values replacing them with another
induction value */
for y \in IV do
(E1,c2) := st(y)
a := cl / c2
inity := findinit(in(y,T))
if inity ≠ null
then for stt \in (use(y,T) U use(y,E1)) do
if number > stt
then b := initx
else b := initx + cl
end if
if E1 > stt
then c := inity
else c := inity + c2
end if
if TYPE(stt) ≠ (if or while or repeat or forall)
then replace(in(y,a*x-a*b+c,TREE(stt)),
out(TREE(stt)))
I(stt) := I(stt) U x - y
use(y,T) := use(y,T) - stt
use(y,E1) := use(y,E1) - stt
end if
end for
end procedure
if \((\text{use}(y,T) = E_1)\) and \((\text{use}(y,E_1) = \emptyset)\) and
\((\text{not live}(y,T))\)
then delete(in(E_1,E,T),out(E,T))
end if

end if
end for
end procedure

The procedure "delete" deletes the given entry along with its initialization and updates any data flow information necessary. All other procedures have been defined.

Loop Decomposition

procedure loopdecomposition (in(E,T),out(E,T))
/*
E - set of entries in the body of T
T - interface entry for the loop
*/
findrc(in(E,T),out(RC))
findcondition(in(E,T,RC),out(cando,st,index,constant,oper))
if cando
then findindex(in(E,index,RC,T),out(stt,cando,cl,initx))
end if
if cando
then \(E_1 := E - \{st,stt\}\)
    independentiterations(in(E_1,T,index),
                         out(worthtrying,result,order,try,part,cycles))
end if
if cando and worthtrying
then transform(in(order,index,initx,cl,constant,T,\(E,T\),out(E,T))
else decompose(in(part,cycles,try,order),
               out(cycles,try,order,commands))
for subpart \(\epsilon\) part do
    independentiterations(in(subpart,T,index),
                         out(worthtrying,result,order,try,subpart,cycles))
if result
then processcommands(in(commands,i,subpart),
                    out(subpart))
    transform(in(order,index,initx,cl,constant,E,T),
              out(E,T))
end if
end for
end if
end procedure
procedure independentiterations(in(E,T,index),out('worthtrying,result,order,try,part,cycles))

/*
E - set of entries being examined
T - interface entry
index - the index value for the loop
worthtrying - boolean value indicating if loop decomposition can be performed on the loop
result - boolean value indicating if the loop was decomposed
order - matrix indicating the data dependencies between entries of E
try - matrix indicating those techniques which would be useful in breaking a cycle which appears in order
part - set of partitions for the entries in E
cycles - set of cycles contained in each partition, if there are any
*/
worthtrying := true
result := true
/* initialize order and try matrices */
for i = 1 to |E| do
    for j = 1 to |E| do
        order(E,i,E,j) := false
        try(E,i,E,j) := φ
    end for
end for
/* for each entry, find all uses of a value in left context to construct the order matrix */
for i = 1 to |E| do
    if TYPE(E,i) = (repeat or while or if or forall)
        then st := entries in body of E,i
           while (st ≠ φ) do
              stt := next(st)
              st := st - stt
              /* if stt is an interface entry, add all entries in the body of stt to st so they may be checked */
              if TYPE(stt) = (repeat or while or if or forall)
                 then st := st U entries in body of stt
              else finduses(in(true,E,i,stt,order,try),out(order,try))
           end if
        end while
    else finduses(in(false,E,i,E,i,order,try),out(order,try))
    end if
end for
/* using order matrix, find the cycles and return in part the set of partitions of this loop */
partition(in(order,try),out(part,cycles,ok))
/* if a partition was found, do the following */
if ok
    then if cycles = ∅
        then result := true
        else result := false
    end if
    worthtrying := true
else worthtrying := false
end if
end procedure

procedure finduses (in(flag,E.,stt,order,try),out(order,try))
/*
flag - indicates whether it is looking for a use in the same statement the value is found.
true indicates the entry is in a nested block
false indicates the entry is not in a nested block
E. - indicates the entry being considered at the present time
stt - the entry whose output set is being examined
order - matrix that indicates the precedence ordering between entries in the body of a statement
try - matrix that indicates ways the data dependencies may be broken between entries in the body of the statement

/* check the uses of every value in the output set of stt */
for x ∈ O(stt) do
    /* get the subscript of x used in left context */
    exprl := getsub(in(x,stt,left))
    if flag
        /* if x is defined in a nested block of entry E., need only consider the uses of x in E.
        then for E. ∈ use(x,stt) do
            if TYPE(E.) = (repeat or while or if or forall)
                /* analyze uses of x in body of E. */
                then body(in(x,E.,exprl,right,order,try),
                    out(order,try))
                /* analyze uses of x in entry E. */
                else if x ∈ I(E.)
                    then subexpr2 := getsub(in(x,E.,right))
                        for expr2 ∈ subexpr2 do
                            analyze(in(x,expr2,E.,right,exprl,E.,
                                index,order,try),out(order,try))
                        end for
                    end if
            end if
        end for
    else for E. ∈ use(x,stt) do
        /* if x is defined also by this entry, show a data dependency */
end for
end for
*/
if x ∈ O(E_n)
    then order(E_i, E_n) := true
    try(E_i, E_n) := try(E_i, E_n) U (NH, x)
end if
if TYPE(E_i) = (repeat or while or if or forall)
    /* analyze uses of x in body of E_i */
    then body(in(x, E_i, exprl, left, order, try), out(order, try))
    /* analyze uses of x in E_i */
else subexpr2 := getsub(in(x, E_i, right))
    for expr2 ∈ subexpr2 do
        analyze(in(x, expr2, E_i, right, exprl, E_i, index, order, try), out(order, try))
    end for
end if
end for
end for
end procedure

procedure body (in(x, E_i, E_i, exprl, side, order, try), out(order, try))
/*
   x - value being search for data dependencies
   E_i, E_i - the two entries being analyzed to see if they
   are independent of one another
   exprl - the subscript expression for the value being defined
   by E_i
   side - boolean value indicating where E_i appears in the body
   of the statement
       left indicates that E_i appears before E_i
       right indicates that E_i appears before E_i
   order - matrix indicating the precedence relation that exists
   between entries in the body of the statement
   try - matrix indicating different alternatives to try if two
   entries are found not to be independent
*/
    u := use(x, E_i)
    while (u ≠ Ø) do
        us := next(u)
        u := u - us
        if TYPE(us) = (repeat or while or if or forall)
            /* if us is an interface entry, add uses of x in us to u
               to be analyzed */
            then u := u U use(x, us)
            /* analyze uses of x in entry us */
        else if x ∈ I(us)
            then subexpr2 := getsub(in(x, us, right))
            if side = left
                then for expr2 ∈ subexpr2 do

analyze(in(x,expr1,Ei,left,expr2,Ej,index, order,try),out(order,try))

end for

else for expr2 e subexpr2 do
  analyze(in(x,expr2,Ej,right,expr1,Ei,index, order,try),out(order,try))
end for

end if

end while

end procedure


procedure analyze(in(x,expr1,E1,side,expr2,E2,index,order,try),
     out(order,try))

/*
x - value being analyzed
expr1 - subscript expression for first value
E1 - entry in which expr1 appears
side - context in which x appears in E1
expr2 - subscript expression for second value
E2 - entry in which expr2 appears
index - index value for loop
order - matrix indicating the data dependencies in a loop
try - matrix indicating those techniques to try to break
cycles
*/

/* analyze the subscripts */
result := sub(in(expr1,expr2,index))
/* if enough information about subscripts cannot be obtained
to analyze the subscripts, assume a data dependency
between the two entries */
if result = nil
  then order(E1,E2) := true
      try(E1,E2) := try(E1,E2) U (NH,x)
      order(E2,E1) := true
      try(E2,E1) := try(E2,E1) U (NH,x)
/*
E1 = E2 - indicates a use and a definition appear in the
same entry
side = left - indicates the definition appears before its
use
side = right - indicates the use appears before the
definition
*/
else case of result
  (result=0):
    case of
(* do nothing *)
(order(El,E2) := true
(try(El,E2) := try(El,E2) U (FS,x))
(side=right): order(El,E2) := true
(try(El,E2) := try(El,E2) U (TS,x))
end case

(result=0):
case of
(El=E2): /* do nothing */
(side=left): order(E2,El) := true
(try(E2,El) := try(E2,El) U (TS,x))
(side=right): order(E2,E1) := true
(try(E2,El) := try(E2,El) U (NH,x))
end case

(result<0):
case of
(El=E2): order(El,E1) := true
(side=left): order(El,E2) := true
(try(El,E2) := try(El,E2) U (NH,x))
(side=right): order(El,E2) := true
(try(El,E2) := try(El,E2) U (TS,x))
end case

(result=scalar):
case of
(El=E2 or
(side=right): order(El,E2) := true
(try(E2,E1) := try(E2,E1) U (NH,x))
(order(E2,E1) := true
(try(E2,E1) := try(E2,E1) U (NH,x))
(side=left): order(El,E2) := true
(try(E2,El) := try(E2,El) U (TM,x))
end case
end case
end if
end case

procedure transform (in(order,index,initx,cl,constant,u,E,T),
out(E,T))

order - matrix indicating the data dependencies between the entries
initx - initial value for index value
cl - increment of index value
constant - constant value used in the conditional test
u - entry before which the new loop is to be placed
E - set of entries under consideration
T - interface entry

/* reorder the entries according to their data dependencies */
topologicalsort(in(order,E),out(E))
/* create a new loop and update the data flow information */
T1 := createentry(in('forallnil,(i,initx,cl,constant),u))
ot := Ø
for i=1 to |E| do
  I(T1) := I(T1) U (I(Ei) - ot)
  for x e I(Ei) do
    if (def(x,Ei)(1) = T) or (def(x,Ei)(1) ≠ E)
      then def(x,Ei)(1) := T1
      use(x,T1) := Ei
    end if
  end for
  ot := ot U 0(Ei)
end for
O(T1) := ot
for x e I(T1) do
  st := finddef(in(x,T1))
  def(x,T1)(1) := st
  use(x,st) := use(x,st) U T1
end for
end procedure

procedure decompose (in(part,cycles,try,order),
  out(cycles,try,order,commands))
/*
part - set of partitions for the set of entries E
cycles - set of cycles contained in each partition
try - matrix indicating those techniques which would
  be useful in breaking a cycle
order - matrix indicating the data dependencies between
  the entries
commands - set of transformations (forward substitution,
  creation of temporary arrays) to be applied to
  a given partition
*/
for i=1 to |part| do
  commands(i) := Ø
  /* if there are cycles in this partition */
  if cycles(i) ≠ Ø
    /* then try and break cycles by saving values in a temporary array */
    then dotted(in(cycles,i,order,try,commands),
      out(order,try,commands,scycles))
    /* if there are still cycles in the partition, see if they can be broken with forward substitution */
    if scycles ≠ Ø
      then solid(in(scycles,i,order,cycles,commands),
        out(order,cycles,commands))
    end if
  end if
end if
procedure dotted (in(cycles, i, order, try, commands),
              out(order, try, commands, scycles))
/*
cycles - set of cycles contained in each partition
i - partition being considered
order - matrix indicating the data dependencies between the
        entries
try - matrix indicating those techniques which would be useful
      in breaking a cycle
commands - set of transformations to be applied to a given
           partition
scycles - set of data dependencies in partition i which cannot
           be broken by temporary savings of values
*/
/* procedure attempts to break data dependencies involving
   temporary saving of values */
/* initialize c to those data dependencies which temporary
   savings would help */
for j = 1 to |cycles(i)| do
  for (x, y) ∈ cycles(i, j) do
    if (TS, *) ∈ try(x, y)
      then c((x, y), j) := 1
    end if
  end for
end for
/* find column of c with the minimum number of 1's */
findmin(in(c), out(col, scycles))
if col ≠ 0
/* break the cycle at the position given by col */
then for (x, y) ∈ cycles(i, col) do
  (ty, var) ∈ try(x, y) such that ty = TS
  stt := gettempentry
  order(x, y) := false
  order(stt, x) := true
  try(stt, x) := (NH, var)
  order(stt, y) := true
  try(stt, y) := (TS, var)
  commands(i) := commands(i) U (TS, var, x, y, stt)
end for
end if
end procedure

procedure solid (in(scycles, i, order, cycles, commands),
              out(order, cycles, commands))
/*
scycles - set of cycles still existing which are to be broken using forward substitution
i - partition being considered
order - matrix indicating data dependencies in the partition
cycle - set of cycles contained in each partition
commands - set of transformations to be applied to a given partition

/* procedure attempts to break a cycle using forward substitution. If there are n entries in the cycle, n-1 forward substitutions must be made to break the cycle */
for c € scycles do
  ok := true
  first := 0
  for j = 1 to |cycle(i,c)| while ok do
    (x,y) € cycles(i,c)
    (ty, var) := try(x,y) such that ty = FS
    if ty = null
      then if first > 1
        then ok := false
        else first := first + 1
      end if
    else z := findmate(in(x))
      order(x,y) := false
      order(z,y) := true
      cycle(i,c) := cycle(i,c) \ {(x,y),(z,x)} U (z,y)
      commands(i) := commands(i) \ (FS, var, x, y, null)
    end if
  end for
end for
end procedure

procedure processcommands (in(commands,i,index,part),
                          out(part))
/*
commands - set of transformations to be applied to a given partition
i - partition being considered
index - index of loop
part - entries in a given partition
*/
for i = 1 to |part| do
  for x € 0(parti) do
    /* if a scalar is output, change it into a temporary array */
    if type of x is scalar
      then for El € (Ei U use(x,Ei)) do
        changetype(in(index,x,El),out(El))
      end for
end if
end for
end for

/* process the commands for each partition */
for (type, var, El, E2, st) ∈ commands(i) do
  case of type
  (FS): I(E2) := I(E2) ∪ I(El)
    for x ∈ (I(E2) ∩ I(El)) do
      E3 := def(x, El)(1)
      def(x, E2)(1) := E3
      use(x, E3) := use(x, E3) ∪ E2
    end for
    replace(in(var, expr(in(El)), TREE(E2)), out(TREE(E2)))
  (TS): varl := gettemp
    stt := createentry(in('assign', varl, (var, index), El))
    I(stt) := {var, index}
    O(stt) := varl
    use(varl, stt) := El
    replace(in(varl, var, TREE(El)), out(TREE(El)))
    def(varl, st)(1) := stt
  end case
end for
end procedure

The procedures "findrc", "gettemp", "findindex", "createentry", "replace" and "findcondition" have all been described in previous sections. The function "getsub" (referred to in Figure 4.19) returns all the subscripts in the syntax tree for a given value in the specified context (i.e., left or right). The "partition" procedure divides the set of entries in the order matrix according to the data dependencies which exist. The set of cycles which are found for each partition are also returned from the procedure. An algorithm like that of Johnson (1975) could be used to specify the partition procedure. The procedure "topologicalsort" takes the order matrix and sorts the entries according to the data dependencies that exist in the matrix. An algorithm like that of Knuth's (1973) could be used. The "sub" function is the subscript
analysis procedure. It analyzes the two subscript expressions given to
determine the relationship between the expressions returning a positive,
negative or zero value. If the relationship between the subscript
expressions cannot be found with the information available, a nil value
is returned. If the subscript expressions do not involve the loop index,
the value returned indicates that these values should be treated as
scalars. The procedure "finddef" finds where a given value was last
defined. The "findmin" procedure finds the column in the given matrix
which has the minimum number of ones in it. The function "findmate" looks
for the cycles which have a specified first component.